Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

SELF-TIMED ARITHMETIC STRUCTURES IN CMOS DIFFERENTIAL LOGIC

Shih-Lien Lu

August 1991
CSD-910059

UNIVERSITY OF CALIFORNIA

Los Angeles

Self–Timed Arithmetic Structures in CMOS Differential Logic

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Sciences
by

Shih–Lien Lu

1991

DEDICATION

I dedicate this dissertation to my personal Savior, Jesus Christ.

Table of Contents

Chapter 1 Introduction 1
- 1.1 Contributions 2
- 1.2 Motivation 2
- 1.3 Thesis Approach and Related Work 3
- 1.4 Scope of Dissertation 3
- 1.5 Outline of Dissertation 4

Chapter 2 CMOS Technology : Different Implementations of Logic . 5
- 2.1. Introduction 5
- 2.2 Single–Ended CMOS Logic 7
- 2.2.1 Conventional Static CMOS 7
- 2.2.2 Domino CMOS 10
- 2.2.3 N–P Dynamic CMOS or NORA CMOS 12
- 2.3 Differential CMOS Logic 15
- 2.3.1 Cascoded Voltage Switch Logic Family 15
- 2.3.2 Sample Set Differential Logic 18
- 2.3.3 Split–Level Differential 19
- 2.3.4 Enable/Disable CMOS Differential 20
- 2.4 Comparisons of Different Logic Structures 21
- 2.5 Conclusion 21

Chapter 3 Basic Principles of Enable/Disable CMOS Differential Logic 29
- 3.1 Introduction. 29
- 3.2 Enable/Disable CMOS Differential Logic Operation Principle 29
- 3.3 Electrical Properties of Enable/Disable CMOS Differential Logic 31
- 3.4 Design Trade–offs 37
- 3.5 Conclusion 40

Chapter 4 Self–Timed Networks for Iterative Functions **41**
- 4.1 Introduction 41
- 4.2 ECDL Iterative Networks 41
- 4.3 Examples of ECDL Iterative Networks 44
- 4.4 Array Network in ECDL 50
- 4.5 Conclusions 53

Chapter 5 The Self–Timed Synchronization in ECDL **54**
- 5.1 Introduction 54
- 5.2 Self–Timed Signaling 55
- 5.3 Using ECDL to Implement Self–Time Elements and Event Controls 57
- 5.4 Using ECDL to Implement Bundled Data Convention 64
- 5.5 Conclusion 66

Chapter 6 Design and Implementation of Two–Operand Adders ... **67**
- 6.1 Introduction 67

6.2 Modeling Methodology 67
6.2.1 Delay Model 67
6.2.2 Area model 68
6.3 ECDL Adders 68
6.3.1 Carry–Ripple Adder (CRA) 69
6.3.2 Carry–Completion–Sensing Adder (CCA) 74
6.3.3 Carry–Skip Adder (CSA) 78
6.3.4 Carry–Lookahead Adder (CLA) 83
6.4. Measurement and Comparison 87
6.4.1 Delay Comparison 87
6.4.2 Area Comparison 87
6.5. Conclusion 93
7.1 Introduction 94
7.2 Array Multiplier 94
7.2.1 Unsigned Array Multiplier 95
7.2.2 Radix–4 Iterative Array Multiplier 102
7.2.3 Array Multiplier using 5–Counters 103
7.2.4 Comparison 106
7.3 Array Divider 109
7.4 Automatic Layout Generation 110
7.5 Conclusion 113

Chapter 8 Clocking Schemes with Differential Logic 114
8.1 Introduction 114
8.2 Circuit Design of a D–type DET–FF 115
8.3 Conclusion 121

Chapter 9 Conclusion 122
9.1 Summary 122

References 124

Appendix A SPICE parameters 129
The corner parameters used in Chpater 2 129

Acknowledgements

First and foremost, I would like to thank my adviser, professor Milos Ercegovac, for his insight, patience, technical guidance and inspiration. He has been and will continue to be my role model in life.

I thank all the members of my reading committee, professors: Abidi, Baker, McNamee, Rennels, and Tyree, for their helpful comments and constructive suggestions.

I am also fortunate to be working at MOSIS, which provides me with the opportunity to prove concepts through silicon fabrication. In particular, Dr. George Lewicki and Dr. Ron Ayres have provided encouragement. Mr. Jen–I Pi and Mr. Tzyh–Yung Wuu have enriched my understanding through many good discussions. Mr. Rod Van Meter has read and corrected many mistakes of the manuscript.

During this period of graduate study, my family has been extremely supportive. My wife, Jenny, to whom I am deeply indebted, has made many sacrifices. I thank our two children, Joshua and Caleb, for providing me much happiness with their laughter. My nephew, David, has proofread this manuscript. I also want to thank my parents for raising me and for facing a new culture by migrating from Taiwan to the States in order to provide us with a better education.

Last, but not least, I want to thank God for giving me eternal life through His only Son, Jesus Christ.

Vita

Bachelor of Science in EECS, 1980
University of California, Berkeley

Master of Science in Computer Science, 1984
University of California, Los Angeles
Thesis title: A Compiler for Functional Programming Language

Publications

S. Lu and M. Ercegovac, "Evaluation of Two Summand Adders in Differential CMOS", accepted by IEEE JSSC for publication, August 1991

S. Lu and M. Ercegovac, "A Novel CMOS Implementation of Double–Edge–Triggered Flip–Flops", IEEE JSSC, August 1990, pp. 1008–1010

S. Lu, "Incremental versus Conventional ASIC Design Cycle", Proceedings of the 3rd IEEE ASIC Seminar, Rochester NY, 1990

T. Wuu, J. Pi and S. Lu, "A Netlist Based Language for ASIC, Prototyping", Proceedings of the 3rd IEEE ASIC Seminar, Rochester NY, 1990

S. Lu, "A Single–Phase Clocked NOR/NOR CMOS Programmable Sequential Array Structure", ISI Research Report–89–230, April 1989

C. P. Wan, B. J. Sheu, and S. Lu, "Device and Circuit Simulation Interface for an Integrated VLSI Design Environment", IEEE Trans. on CAD, Sept., 1988

S. Lu, "Implementation of Iterative Networks with Differential CMOS", IEEE Journal of Solid State and Circuits, Aug. 1988

S. Lu, "A Safe Single Phase Clocking Scheme for CMOS", IEEE Journal of Solid State and Circuits, Feb. 1988

S. Lu and J. Ousterhout, "Magic Technology Manual #2: Scalable CMOS", a section in *1986 VLSI Tools – Still More Works by the Original Artists,* Edited by W. S. Scott, R. N. Mayo, G. Hamachi, and J. K. Ousterhout, UCB Techincal Report, 1986.

Milos, Ercegovac and S. Lu, "A Functional Language Approach for High Speed Digital Simulation", 1983 Summer Computer Simulation Conference, July 1983.

K. Huang and S. Lu, "Some Theoretical Problems and Algorithms for Identification of Distributed Parameter Systems", March 1983, Scientific Report, Shan Dun University, China

ABSTRACT OF THE DISSERTATION

Self–Timed Arithmetic Structures in CMOS Differential Logic

by

Shih–Lien Lu,

Doctor of Philosophy in Computer Sciences
University of California, Los Angeles, 1991
Professor Miloš D. Ercegovac, Chair

The design, implementation and evaluation of self–timed computer subsystems in CMOS technology is presented. More specifically, arithmetic structures such as adders, array multipliers, and array dividers are investigated.

First, a new circuit technique – Enable/Disable CMOS Differential Logic (ECDL), is proposed and studied. This new circuit is used to actualize the design of self–timed modules. Several self–timed arithmetic structures which includes adders, array multipliers and array dividers were designed and implemented using ECDL. A model is suggested to predict the area and speed of different arithmetic algorithms when implemented in ECDL. Several chips are fabricated through MOSIS and measured. Comparisons are made between the measured results, SPICE simulation and this prediction model. This new CMOS differential logic family gives a new dimension in the design of arithmetic structures as well as computer systems.

Self–Timed Arithmetic Structures in CMOS Differential Logic

by

Shih–Lien Lu

1991

Chapter 1

Introduction

"... Now, if you only kept on good terms with him, he'd do almost anything you liked with the clock." ... Lewis Carroll in *Alice in Wonderland*

Clocked logic has been the main synchronization discipline for digital computing systems. In a clocked system time is divided into quantized periods. Only with each "beat" of time (clock pulse), may a new state or event occur. All sequences of computation are controlled by the clock. Thus the computing power or the execution speed of instructions in a clocked system is bounded by the speed of its clock. That is, in order to design faster clocked digital systems, faster clocks must be used. However, in synchronous systems controlled by a central clock, the clock period must be greater than the sum of the longest combinational logic delay, the state register cell delay and the setup time of the state register. That is, the clock period must accommodate the worst case delay of the system [Ung 86]. However the actual delay of a digital system may be much shorter than the worst case delay. As a result digital systems with a global clock run slower than what the actual delay may provide. Over the past several years, remarkable technical progress has been made in the manufacture of semiconductor integrated circuits. By 1988, advances in photo–lithography have allowed very large–scale integrated (VLSI) circuits with transistor effective channel lengths of one micron in volume production. Further scaling down to 0.5 micron and below is expected to be realized in the near future. In particular, Complementary Metal Oxide Silicon (CMOS) technology has played an increasingly important role. The potential for high speed with almost no static power consumption, while still allowing very high densities, permitted CMOS to be accepted as the technology to build compact fast digital systems. However, with the down scaling of feature size and the increase in chip area, not only has the complexity of chips increased, but the timing relationship between wiring and circuit elements has also been altered. Synchronization of elements in a system with a global clock is becoming increasingly problematic [Sei 79]. While the technology offers gates with delay in the fraction of nano–second range, very few synchronous systems can achieve a global system clock speed of 50 MHZ or above. An alternative is to have a *self–timed* system. A *self–timed* [Sei 80] system is a interconnection of self–timed parts – called *self–timed elements*. Elements initiate the computation with the arrival of *start* signals. Upon the completion of the computation by the elements *done* signals are generated. These done signals become the start signals of elements whose inputs depend on the outputs of the elements generating the done signals. The sequencing of computational steps is determined by the way elements are connected. Therefore in a self–timed system, there is no need for a global clock. The computation is governed by the sequence of actions of each module's dependence on one another. As a result the speed of the self–timed system is not determined by the worst case propagation delay, and there is no waiting, hence, no time wasted by any of the sub–modules. Replacing the elements of the critical path with faster modules will improve the overall system performance without fine tuning the system clock.

The aim of the work presented in this dissertation is to address the problem of design, implementation and evaluation of self–timed computer subsystems in CMOS technology. More specifically, we concentrate on arithmetic structures such as adders and multipliers, which are essential in the design of digital systems. As a new implementation approach, a CMOS differential logic family is introduced [Lu 88a] and used in the development and implementation of these arithmetic structures. We show that this new CMOS differential logic family gives a new useful alternative in the design of arithmetic structures as well as computer systems. It can be used not only for building self–timed digital subsystems, but

also serves to realize a safe single phase clocking methodology for synchronous digital systems [Lu 88] [Lu 90].

1.1 Contributions

The results of this dissertation are not intended to be panaceas for all system timing problems nor will they be used to manage the entire VLSI complexity issue. However, this dissertation contributes in the following areas:

D A new CMOS differential logic family – Enable/Disable CMOS Differential Logic (ECDL), with propagation delays smaller than the standard static CMOS.
D A transition scheme for controlling events using ECDL.
D Realization of iterative networks with ECDL circuits.
D Several arithmetic structures using ECDL circuits:
(1) Two–summands adders
(2) Array multipliers
(3) Iterative divider.
D Evaluation and measurement of arithmetic ECDL structures.
D Alternative clocking schemes using ECDL.

1.2 Motivation

Circuit structures for efficiently performing arithmetic functions are often required for high speed computing. The design process of an arithmetic structure consists of two phases. The first is to develop efficient algorithms and the second is to realize the logic implementations of these algorithms in a suitable technology. Although the development of efficient algorithms for an arithmetic function requires more creative thinking and theoretical background, efficient implementation of an appropriate algorithm determines the overall system performance. Moreover, in many cases the implementation techniques as well as the medium used will affect the decisions made on the level of algorithmic design. Therefore, we must not underestimate the importance of implementation and should consider these two phases as one inseparable entity. With the rapid development in very large scale integration comes a need for further research in suitable fast and efficient arithmetic algorithms and implementations to take full advantage of the advancement in technology. In many arithmetic–intensive applications the use of traditional arithmetic algorithms and implementations, primarily developed for small scale integration or medium scale integration (SSI/MSI) gate–level technology, not only is inefficient but is also becoming unsatisfactory in performance. First, scaling down feature size and scaling up chip area increases the complexity and makes timing become more difficult to solve. Global clock distribution and long distance communication required by synchronous systems will become more and more problematic [Sei 79]. Second, as the chip area is scaling up, there is an increasing demand for higher integration. This increasing complexity demands modularity. For example, in [Sans 81], it is shown that as the number of cells increases, the size of individual cells decreases, and more area is taken up by interconnections. In particular cases, total chip area may increase, even though the devices are scaled down, which is mainly caused by the amount of routing area required. Moreover, the design time for these individual cells increases dramatically as well. Third, algorithmic timing analysis based on gate delays becomes an unreliable reflection of how an implemented chip will perform. Delays contributed by wiring parasitics, fan–ins, and fan–outs

are not fully observed nor accounted for. To address the difficulty of having a global clock we propose to use self–timed circuits.

1.3 Thesis Approach and Related Work

Over the last few years, CMOS technology has been accepted as one of the viable technologies for VLSI systems. It has several major advantages: low static power dissipation, high noise margin and single power supply requirement. Since the output of a CMOS gate is guaranteed to settle at either the power supply or the ground level, the designing of CMOS circuits is easier than that of nMOS. No delicate balancing of the load and driver transistors is needed. The conventional static CMOS gate, however, is intrinsically slow and area consuming because the logic function is duplicated by both n and p channel devices. Moreover, since each signal must drive both an n–channel and a p–channel device, the load capacitance is the sum of both which amounts to approximately three times the load capacitance of an nMOS implementation. Several methods have been proposed to improve the static CMOS performance. In this dissertation, we proposed a new way of implementing logic with an improved differential CMOS logic family – enable/disable CMOS differential logic (ECDL). Other differential logic families reported in the literature include: cascode voltage switch logic (CVSL) [Hel 84], dynamic cascode voltage switch (DCVSL) [Hel 84], complementary set–reset logic (CSRL) [Mead 85], sample–set differential logic (SSDL) [Grot 86] and split–level differential logic (SLL) [Pfen 85]. In short, ECDL has the following characteristics:

- it can be either static or dynamic
- it has a good combined time/speed/power product
- it has little overhead circuitry
- it can be asynchronous or synchronous
- it lends itself to automatic generation of geometry

By using this ECDL to implement some self–timed structures we hope to demonstrate practical performance improvements over clocked CMOS circuits. Work reported by Jacobs [Jacob 88] and Meng [Meng 88] both use the dynamic Cascoded Voltage Switch Logic (CVSL) [Hel 84] structures as the implementation circuitry. Our implementation of self–timed structure differs from work done by Jacobs [Jacob 88] in two ways. First, ECDL is static. There is no charge sharing problem in comparison with the dynamic CVSL approach. Second, since some event control can be implemented with ECDL directly, the amount of control circuit is reduced. This idea of combining logic for event control and information processing is the main difference that sets apart this work from other self–timed approaches proposed [Meng 88, Suth 89]. While previous work done by Jacobs [Jacob 90] and Meng [Meng 88] uses bit–slice (ripple) and Booth algorithm to implement their computation modules, this work explores other arithmetic algorithms.

1.4 Scope of Dissertation

This thesis will concentrate on the design and implementation of submodeles used in a digital system. We will discuss and compare different CMOS logic structures as an implementation medium. A detailed circuit design analysis of ECDL will be performed. We will evaluate the effectiveness of using ECDL to implement different arithmetic computation modules. Actual structures will be designed and fabricated through MOSIS which in-

cludes several two summand adders, array multipliers and other logic functions. An analytical model predicting the performance is proposed. Appraisal of different arithmetic algorithms on the silicon level, instead of on the gate level as previously done, will be performed by comparing the measured results with modeled data.

1.5 Outline of Dissertation

As mentioned, the focal point of this thesis is to develop a new differential logic circuit called Enable/Disable CMOS Differential Logic (ECDL). We begin in Chapter 2 by comparing different CMOS logic families as an implementation medium of designs. Tradeoffs in area and speed of several CMOS logic techniques are compared through SPICE simulation for a particular logic. In Chapter 3, the detailed circuit design of this new differential logic family is explained. The basic circuit structure is extended to implement the completion circuitry for self–timed operation. We also explain how event control can be performed from the circuit structure itself. In Chapter 4, implementation of iterative networks and array networks with ECDL is discussed. Two ways to improve the performance of iterative networks are shown. In Chapter 5, we examine the approach of implementing a data flow graph using ECDL. Firing of tokens is regulated by the transition signals. These signals are processed with ECDL. Chapter 6 and 7 present self–timed iterative networks implementation for several arithmetic operation. Chapter 6 covers four different adders and Chapter 7 examines three array multipliers and an array divider. A first–order modeling technique is proposed to predict the performance of arithmetic structures designed using ECDL. Chapter 8 demonstrates a single–phased clocking scheme which utilizes both clock transition edges as an additional capacity of ECDL circuits. Chapter 9 summarizes the contribution and open research problems.

Chapter 2

CMOS Technology : Different Implementations of Logic

2.1. Introduction

The behavior of the logic implementation definitely affects high level architectural decisions. In this chapter we summarize some different styles of CMOS logic structures and discuss their speed, are and power properties. First, different styles of CMOS logic techniques can be categorized into two groups according to their timing properties. Logic structures which have no minimum clocking speed requirement are labeled as *static logic*. Logic structures which operate on charge storage and require a lower bound on the speed of the clock to prevent incorrect behavior and malfunction are called *dynamic logic*. Conventional CMOS with both N–channel devices and P–channel devices is a static logic. Domino logic [Kra 82] is an example of a dynamic CMOS logic. There is another way to group logic structures. This grouping depends on the way logic is realized. When a single logic network of either N–channel MOS devices or P–channel MOS devices or a two logic structures of both N–channel and P–channel is used to implement the logic function or its complement (not both), these logic structures are called *single–ended* logic. Logic structures using a single network or two logic networks to implement a given logic equation and produce both the function and its complement are named *differential* logic (or *double–railed* logic). In Figure 2.1, we summarize the classification of CMOS logic structures discussed in this chapter.

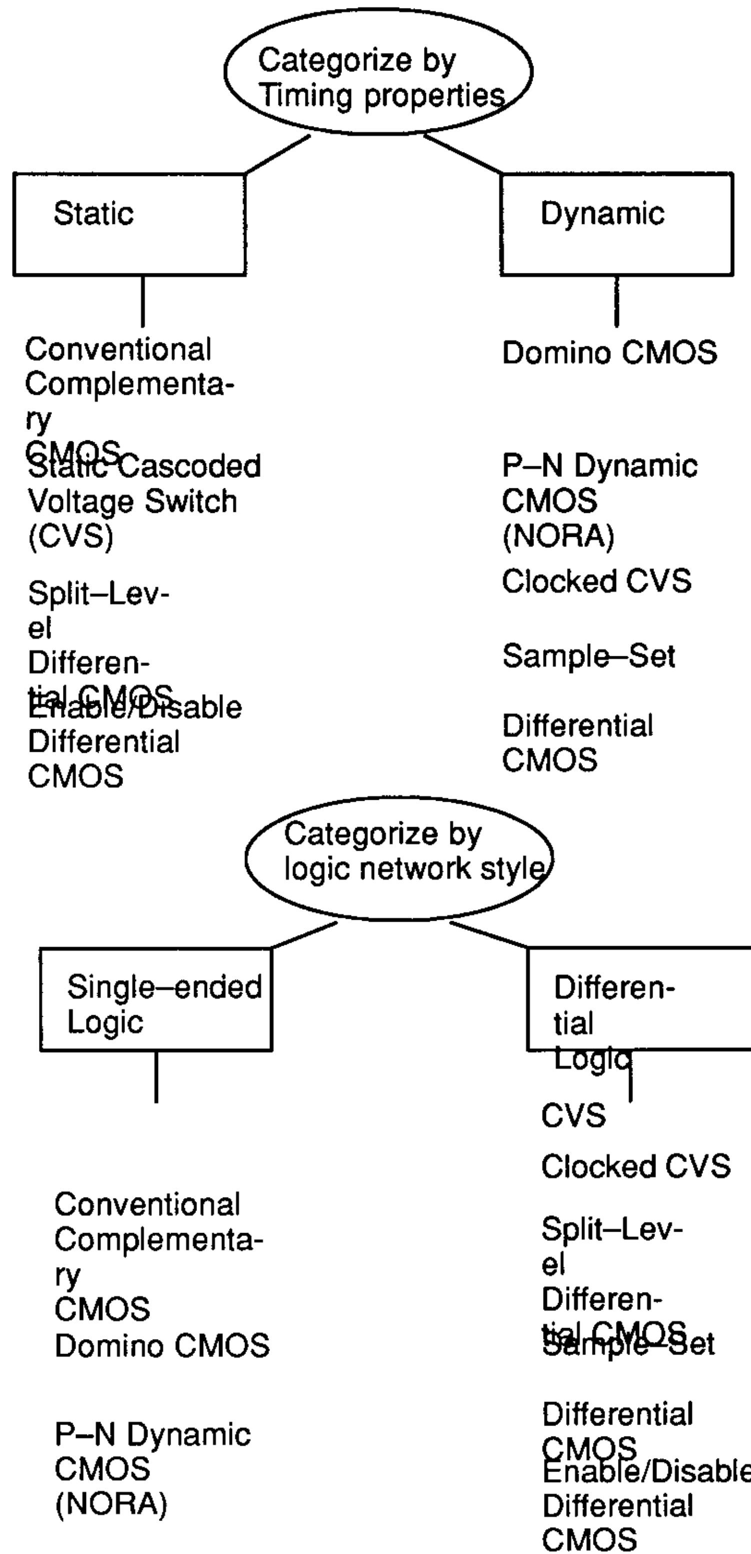


Figure 2.1 Different CMOS Logic Approaches

2.2 Single–Ended CMOS Logic

2.2.1 Conventional Static CMOS

Static CMOS implementation of a suitable function is considered redundant. The function is duplicated by both n and p channel devices. Moreover, each signal must drive both an n–device and a p–device (Figure 2.2).

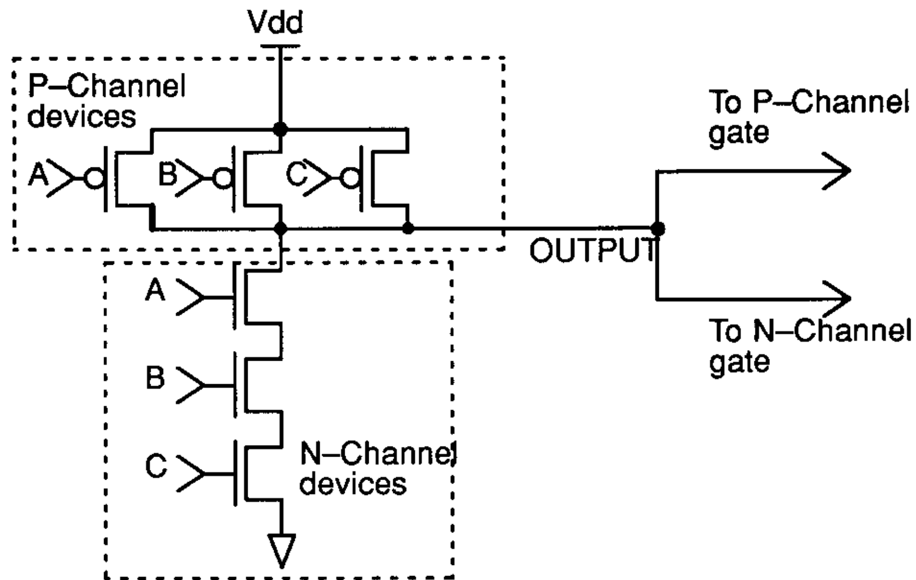


Figure 2.2 Static CMOS NAND of 3 inputs

We denote C_{load} as the load capacitance, C_{ox} as the gate capacitance per unit area, W_n and L_n as the width and length of the n–channel device, and W_p and L_p as the width and length of the p–channel device. Then, the load capacitance is expressed as:

$$C_{load} = C_{ox}\ (W_n L_n + W_p L_p) \tag{2.1}$$

which amounts to about three times that of an nMOS implementation, for $W_p = 2W_n$. This load capacitance contributes to the slowness of static CMOS logic compared to nMOS. Furthermore, the redundancy of the logic network contributes to a larger implementation area.

In static CMOS, NAND and NOR functions are easier to implement than AND and OR functions, since n–channel devices pass the GND signal without voltage degradation while p–channel devices pass the Vdd signal without any voltage drop. Moreover, CMOS gates in NAND form are used more often than the NOR form of gates. The major factor contributing to this selection is the difference in the mobility of electrons and holes. For particu-

lar values of V_{gs} and V_{ds}, the MOSFET current is directly proportional to k, which is the transconductance parameter. We denote k_p and k_n as the transconductance parameter of p–channel device and n–channel device, respectively. Let μ_n and μ_p be the mobility of carrier in n–channel and p–channel devices, respectively. Therefore, we have:

$$k_n = \frac{W_n}{L_n}\mu_n C_{ox} \quad (2.2)$$

$$k_p = \frac{W_p}{L_p}\mu_p C_{ox} \quad (2.3)$$

There are two characteristics of interest: First, since μ_n and μ_p are different, in order to provide a similar fall and rise times, width of P–channel devices need to be sized. Second, since μ_n is larger than μ_p, it is better to chain the higher performance n–channel devices in series to form NAND gates than to chain P–channel devices in the NOR form of gates. Otherwise, the larger–sized P–channel devices not only consume much circuit area, but also add extra load capacitance to the gate. Figure 2.3 illustrates the tabulated results of SPICE simulation for two gates, a 5 input NAND and a 5 input NOR. The technology used in simulation is a 3 micron technology. However, this is not the most advanced technology available. For comparison purposes, we use this technology throughout the thesis except where it is explicitly stated otherwise. The fast and slow cases are simulated with fast and slow SPICE level 2 model parameters provide by MOSIS [MOSIS 84]. The model's parameters used for the simulation are included in Appendix A. Conventional static CMOS can be clocked. A general clocked CMOS gate is illustrated in Figure 2.4. Originally, this form of logic was developed to build low power CMOS circuits. The effect of reducing the power is mainly a result of having metal gates. We can use this kind of logic structure to enable the design in interfacing with other dynamic forms of logic circuits. The clocked gates have the same input gate capacitance as the static CMOS gates. However, the clocking transistors will cause longer rise and fall times.

Since conventional static CMOS tends to be slow in speed and large in area, the Domino logic method was proposed to reduce the transistor count and to increase the operation speed [Kra 83].

5–input NAND (W/L=3/4 all transistors)

	rise time (ns)	fall time (ns)
Typical case	2.2	8.2
Fast case	1.6	5.8
Slow case	2.7	9.8

5–input NOR (W/L=3/4 all transistors)

	rise time (ns)	fall time (ns)
Typical case	28.5	1.4
Fast case	20.0	1.2
Slow case	32.4	1.5

Figure 2.3 Rise and fall time for 5–input NAND and NOR gates

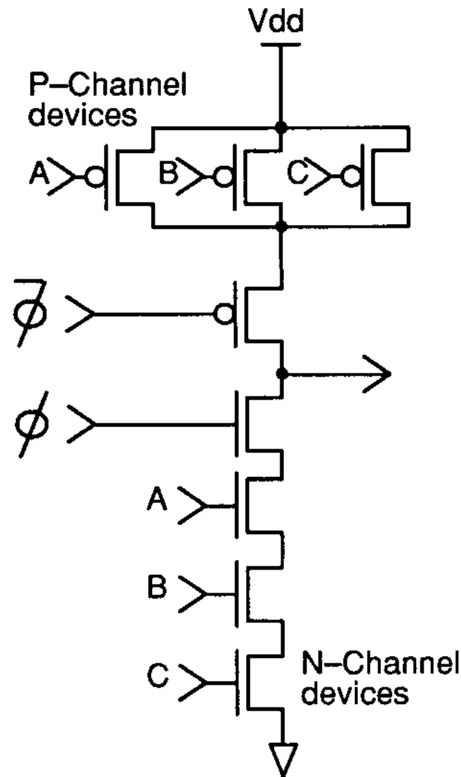


Figure 2.4 Clocked CMOS 3–input NAND

2.2.2 Domino CMOS

A basic circuit is shown in Figure 2.5. The Domino logic family operates with a single–phase clock in two modes. During the first operation mode (ϕ = 0), the output node (node X in Figure 2.5) is pre–charged to Vdd. The clock signal (f) then goes to one, disabling the pre–charge p–type transistor and enabling the bottom n–type transistor to provide a path to GND. If the logic network provides a path from node X to ground, charge accumulated during the pre–charge period will be discharged through the path. The output will be obtained from an inverter. This inverter is necessary to prevent charge redistribution as well as spikes. With this inverter, AND and OR functions are readily obtained. In contrast to conventional static CMOS logic, the practical gate preferred is an OR gate instead of an NAND. Simulation has been performed to demonstrate a typical speed. Results are shown in Figure 2.6. Again this is done for the MOSIS' 3 micron CMOS technology.

There are two major disadvantages with Domino logic family. First, this logic family is not complete. It does not provide negation. Second, the usual method of using transmission gates, as illustrated in Figure 2.7, to do pipelined implementation leads to race conditions, if clocks are not completely non–overlapping. When both ϕ and $\bar{\phi}$ are 1 or 0, there exists a path from input to output in Figure 2.7, which causes compromised (not the power rail voltages) logic levels . An extension of the Domino logic was proposed by Goncalves/DeMan [Gon 83]. This modified method is called NORA. A similar approach is also reported by Friedman and Liu [Frie 84].

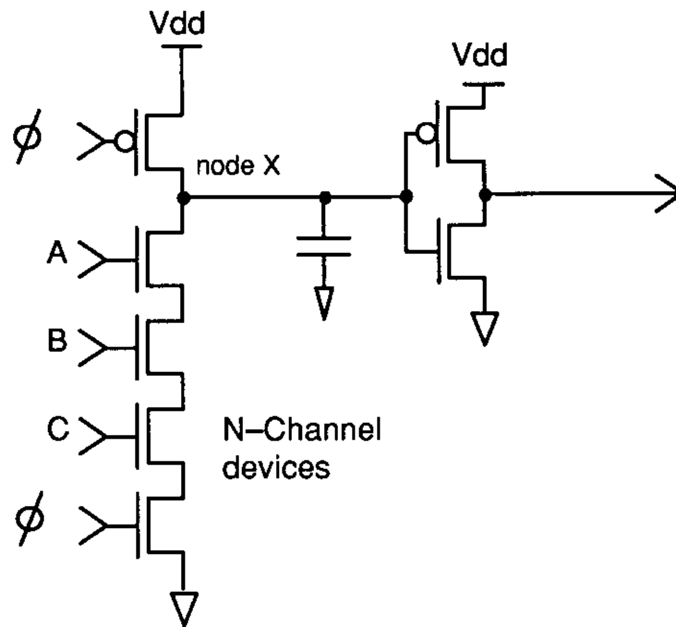


Figure 2.5 CMOS Domino NAND of 3 inputs

Number of inputs	rise time (ns)
2	7.5
5	13.5
9	20.0

Figure 2.6 Rise times of several Domino AND gates

2.2.3 N–P Dynamic CMOS or NORA CMOS

The new Domino logic extension utilizes both the n–core and the p–core logic network gates in alternating stages. A general circuit is shown in Figure 2.8. In this setup, n–core networks are clocked by ϕ and the p–core networks are clocked by $\overline{\phi}$. Both the n–core and the p–core stages are pre–charged when ϕ = 0 and $\overline{\phi}$ = 1. In the n–core stage, the output node is pre–charged to Vdd. In the p–core network stage, the output node is pre–charged to 0. Consequently, when an n–core network stage is in evaluation phase, output node is discharged. As it is discharged, it turns on the p–channel devices of the next p–core network, which in turn, will charge up the output node. By using this method, inverters are eliminated and the logic family is complete. However, outputs of n–core gates cannot be used for the inputs of other n–core gates, and outputs of P–core gates cannot be used for other P–core gates. Goncalves and De Man [Gon 83] also proposed a dynamic CMOS method for pipelined logic structures. Instead of using the CMOS transmission gates, clocked CMOS latches, as illustrated in Figure 2.9, are used to couple pipeline stages. Each stage may be composed of an arbitrary number of the n–core and the p–core network gates. In all odd stages, the period when ϕ is 0 and $\overline{\phi}$ is 1 is used as the precharge phase and the period when ϕ = 1 and $\overline{\phi}$ = 0 is used as evaluation phase. Conversely, in all even stages, the period of time when ϕ is 1 and $\overline{\phi}$ is 0 is used as precharge phase and the period of time when ϕ = 0 and $\overline{\phi}$ = 1 is used as evaluation phase. Figure 2.10 illustrates the alternating phases. Since both ϕ and $\overline{\phi}$ are used as precharge phase as well as evaluation phase, it requires an even split between the two phases.

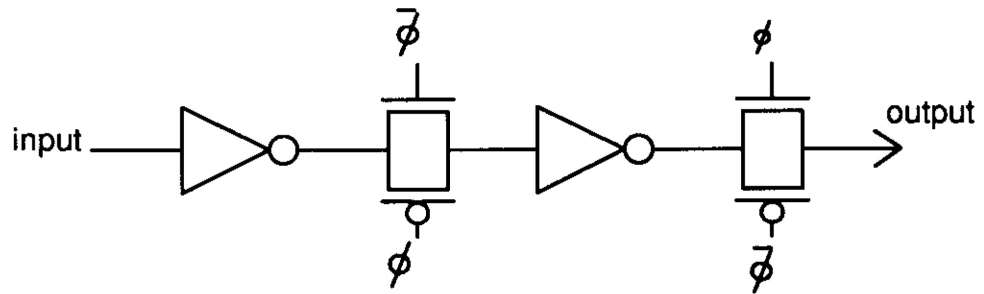


Figure 2.7 A CMOS Shift Register Using Transmission Gates

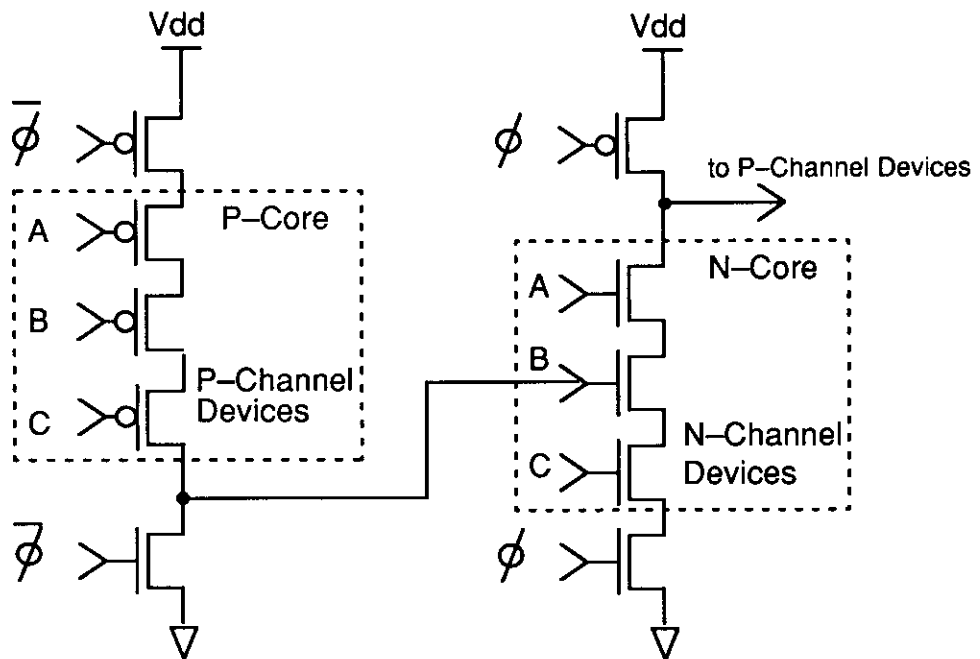


Figure 2.8 A General Structure of an N–P Dynamic CMOS

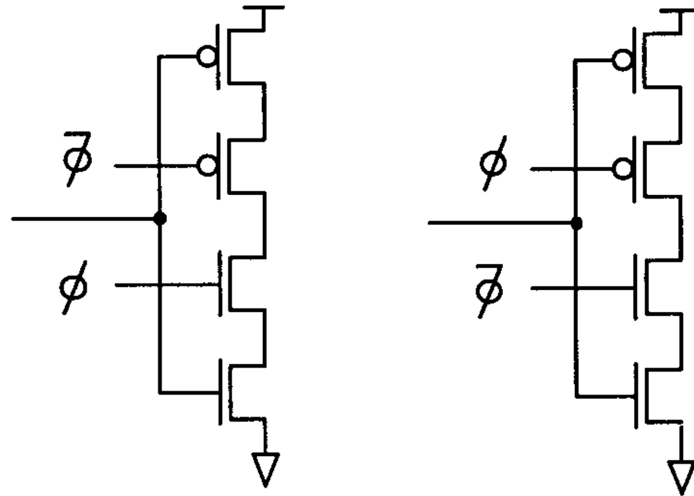


Figure 2.9 Clocked CMOS Latch

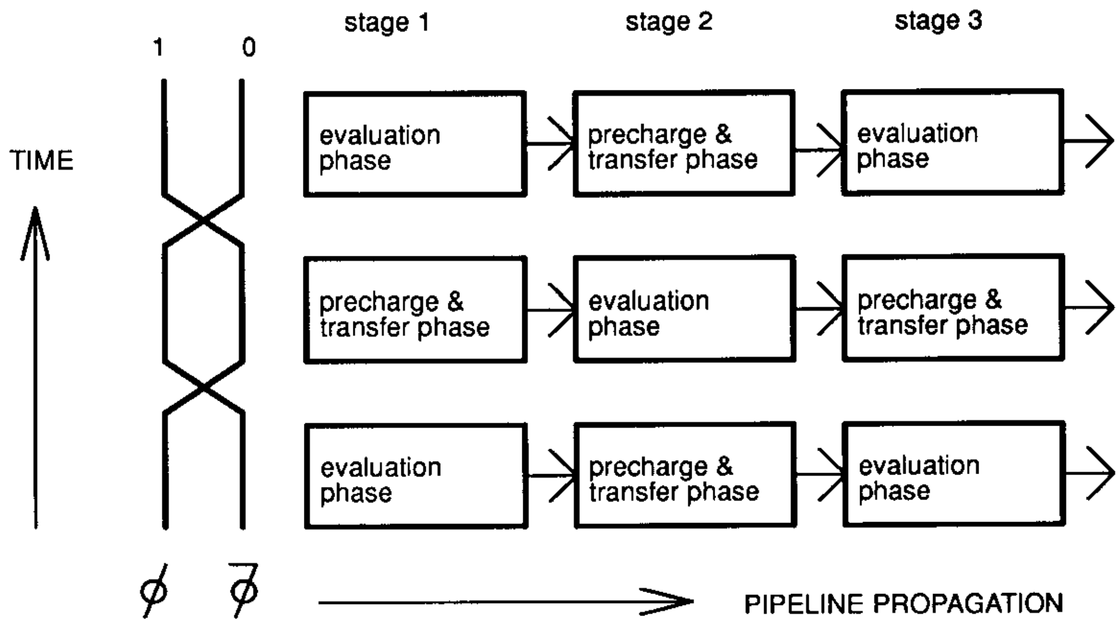


Figure 2.10 Alternating Stages – Pipelined

2.3 Differential CMOS Logic

2.3.1 Cascoded Voltage Switch Logic Family

Another approach in improving the speed and the density of static CMOS logic is proposed by Heller et al. [Hel 84]. This logic family named Cascode Voltage Switch Logic (CVSL) is based on a cascoded differential pairs of MOS devices forming a complex differential network tree. A basic circuit is illustrated in Figure 2.11. Depending on the input values, one of the node (*OUTPUT* or $\overline{OUTPUT}$) will be pulled down to ground by the combinational network. Two P–type transistors connected in a cross coupled fashion provide regenerative action to pull one node to Vdd and pull the other to Ground. There is no static power dissipation. Once the result is set, it remains unchanged. Since each output only drives N–type gates, the gate loading capacitance is generally three times smaller than that of static CMOS implementation. Moreover, since both output and its complement are available, most of inverters are eliminated. Designing the differential logic network can be formalized as proposed by Chu and Pulfrey [Chu 86], which lends to automation and logic minimization. An example is shown in Figure 2.12 which illustrates the compactness of CVSL. The idea of Domino logic can be applied to CVSL as well. Figure 2.13 shows a high performance clocked CVSL. During the precharge period, when f is 0, both N and $\overline{N}$ are pre–charged high which brings both *OUTPUT* and $\overline{OUTPUT}$ nodes to Vdd. As f goes high, two P–channel transistors used to precharge are turned off and the circuit is left in the evaluation state. Depending on the differential network logic tree one of the nodes (N or $\overline{N}$) is discharged to ground while the other stays at Vdd. Feedback through M1 and M2 improves the noise margin.

There is a major disadvantage in both clocked CVSL and Domino logic. Since the delay of the gate built with the clocked CVSL and Domino logic is linearly proportion to the number of stacked N–channel transistors in the logic tree, this limits the complexity of function which can be built with these circuits. A practical limit of only 4 – 5 N–channel transistors in the combinational logic tree can be stacked in series.

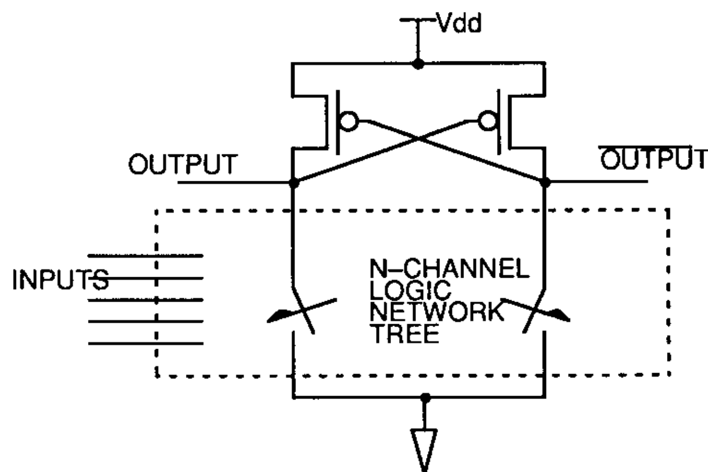


Figure 2.11 A Basic CVSL Circuit

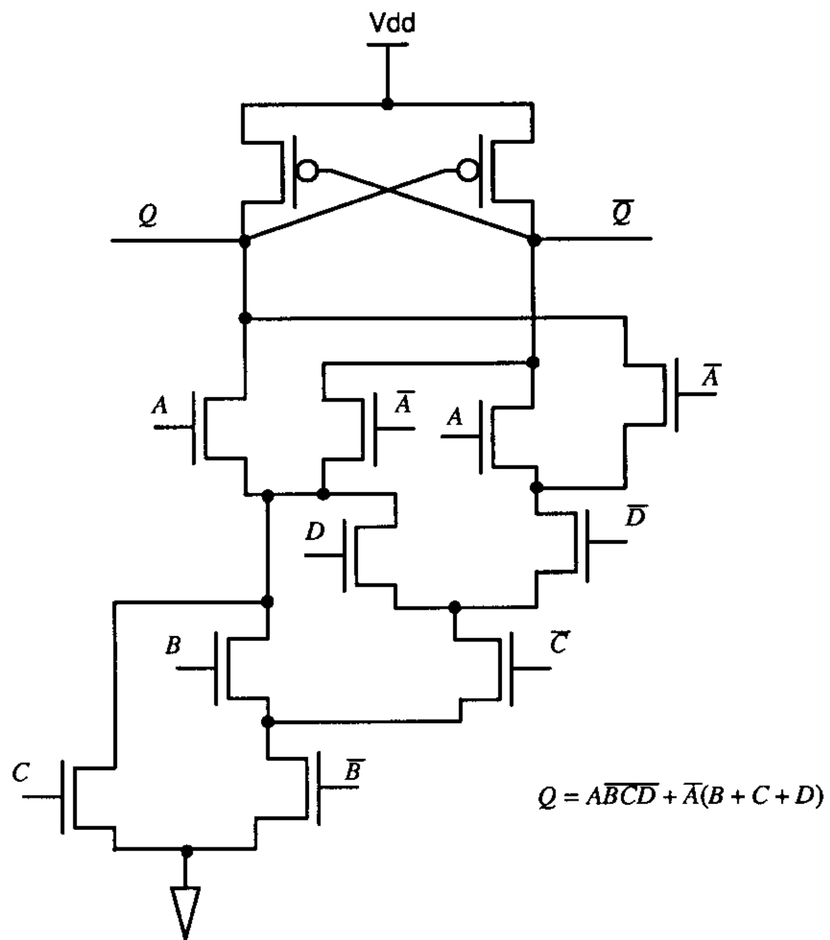


Figure 2.12 An Example of CVSL

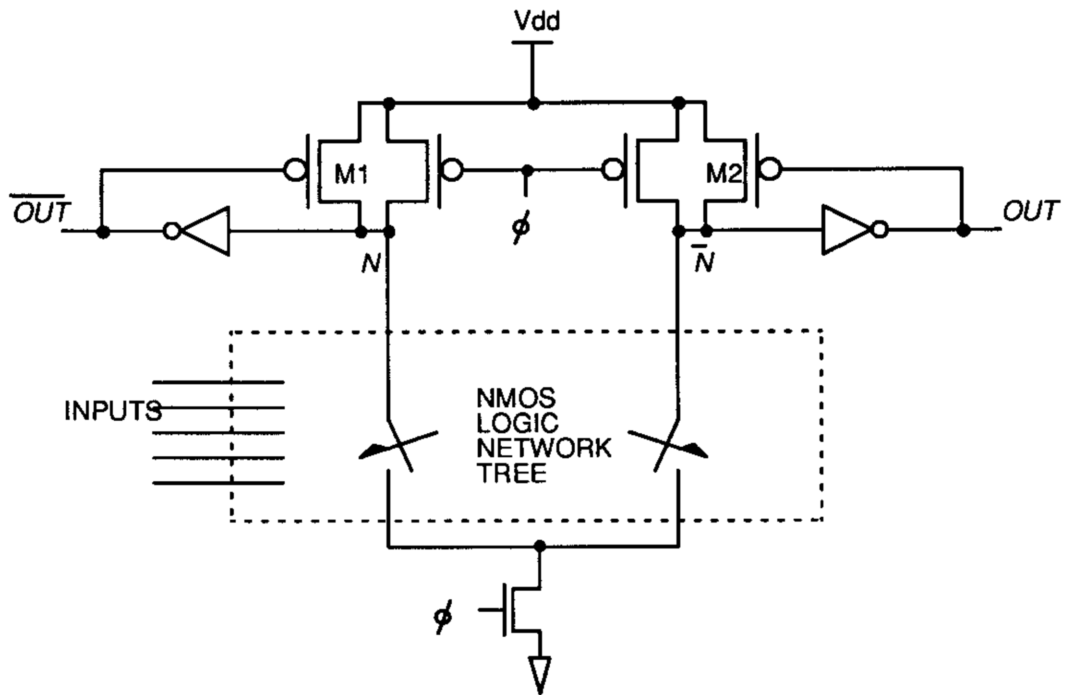


FIGURE 2.13 Clocked CVSL

2.3.2 Sample Set Differential Logic

Grotjohn and Hoefflinger improved the clocked CSVL and called it Sample–Set Differential Logic (SSDL) [Gro 86]. A basic circuit is shown in Figure 2.14. In this set up a

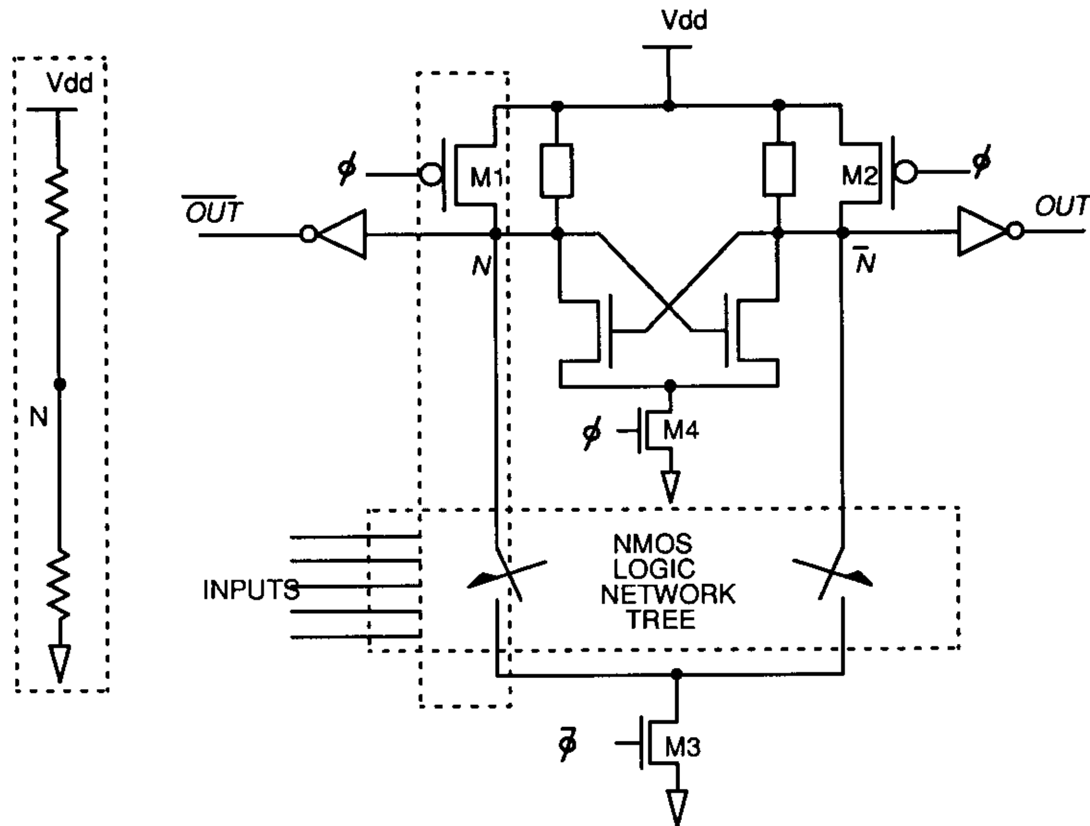


FIGURE 2.14 Sample Set Differential Logic

single phase clock is used. Both clock and the complement of the clock are used to control the operation, though. When the clock is low, the circuit is in a sample stage. Transistors M1, M2 and M3 are all turned on by the clock and its complement. Input values and the differential network tree will provide a path from ground to one of the nodes, either N or $\overline{N}$. As a result that particular node has a voltage lower than Vdd set by the voltage divider, while the other node remains at Vdd. As the clock swings high, the circuit is in the set stage. In this stage M1, M2 and M3 are turned off but transistor M4 is turned on enabling the sense amplifier. At this time the node with a voltage lower than Vdd will continue to drop and feeding back to enable the other node to remain at Vdd. The discharging is done by the nMOS transistor of the sense amplifier. This contributes to the major advantage of the Sample Set Differential Logic. The speed performance is independent of the number of nMOS transistors in series of the differential network tree. There are several drawbacks. First, during the sample stage both M1 and M2 are on, there exists a direct path from Vdd to ground. Static power is consumed because of this path. Second, while there is no speed penalty for large numbers of nMOS transistors in

series, a single transistor differential tree may cause the node N or $\overline{N}$ to drop past the logic threshold voltage, Vinv, during the sample stage feeding the next input with wrong results. Third, although it needs only one clock signal, however both clock and its complement are necessary in order to control the circuit operation. Any overlapping of the clock and its complement due to skewing will cause logic level to compromise.

2.3.3 Split–Level Differential

Another static differential logic family is proposed by Pfennings et al. [Pfe 85]. A basic DSL circuit is shown in Figure 2.15. This circuit differs from CVSL by adding two extra N–channel devices, M4 and M5. These two N–channel devices are gated by a reference voltage. The reference voltage is set to be one half of Vdd plus the threshold voltage of an N–channel device to give the optimal speed performance. Assume, initially, N is low and $\overline{N}$ is high. The P–channel transistor, M2, is turned on while M1 is turned off. The voltage value for N and $\overline{N}$ is 100mV and half of Vdd, respectively. If the differential logic tree changes its input signal and a path exists from $\overline{N}$ to GND, only half of Vdd times loading capacitance of charge needs to be discharged. This gives a speedup of at least 2 with respect to static CMOS circuits. While the speed is increased, there will be static power produces in this logic circuit. This is caused by transistors M3 and M4 being turned on constantly which provides a path from either N or $\overline{N}$ to ground.

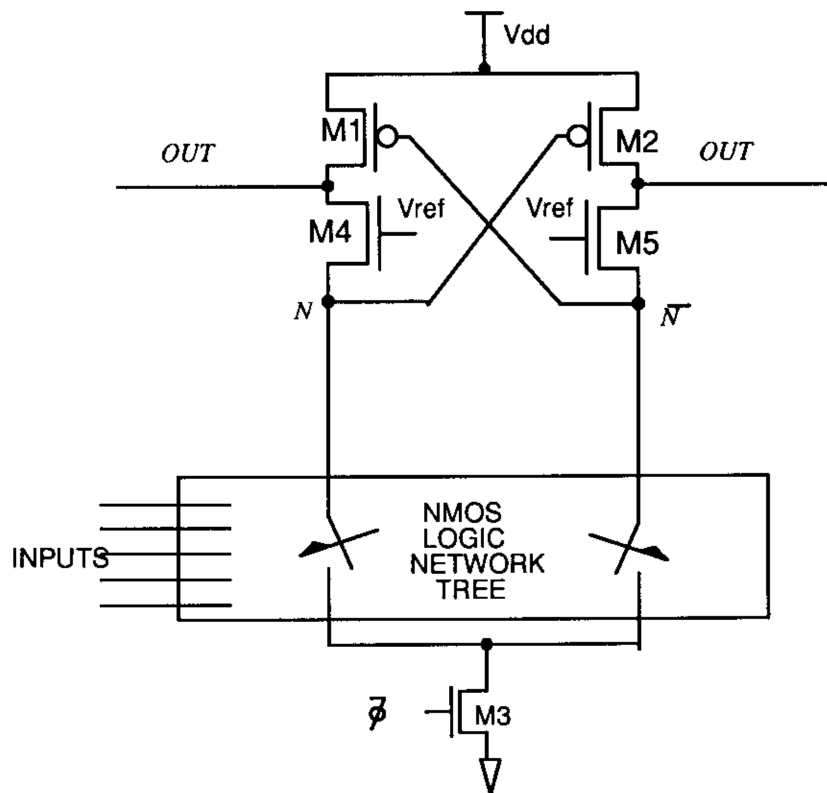


FIGURE 2.15 Split Level Differential CMOS

2.3.4 Enable/Disable CMOS Differential

[Lu 88a] proposed a new static CMOS differential logic named Enabled/Disabled CMOS Differential Logic(ECDL). ECDL is designed for implementation of high speed CMOS circuit. Unlike the previous two logic families –SSDL and DSL, this method consumes no static power. The logic noise margin is also higher, since there are no voltage dividers. Moreover, a true one–phase clock is used, which eliminates the overlapping clock problem caused by clock skews. ECDL can be further modified to implement pipelined logic as well as asynchronous iterative logic circuits. We will present the operation principle and the detailed electrical properties of ECDL in the next chapter. Figure 2.16 shows a basic ECDL circuit. While clocked CVS logic belongs to dynamic logic family, SSDL and ECDL and CVSL are all in the static logic family.

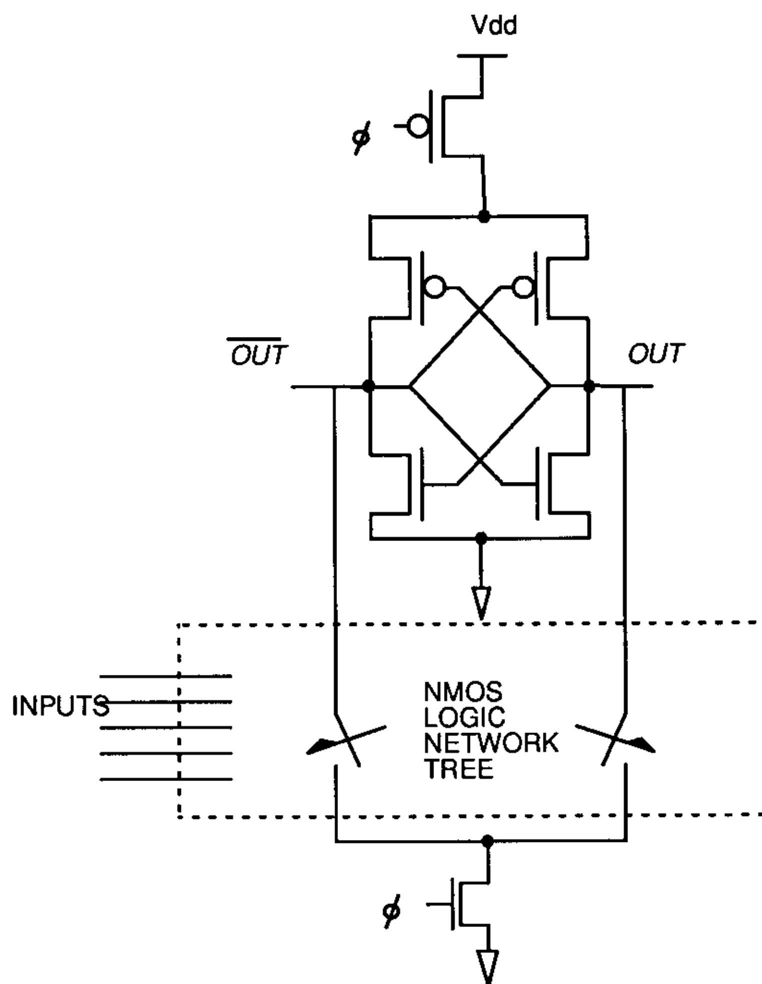


FIGURE 2.16 Basic Enable Disable Differential CMOS Circuit

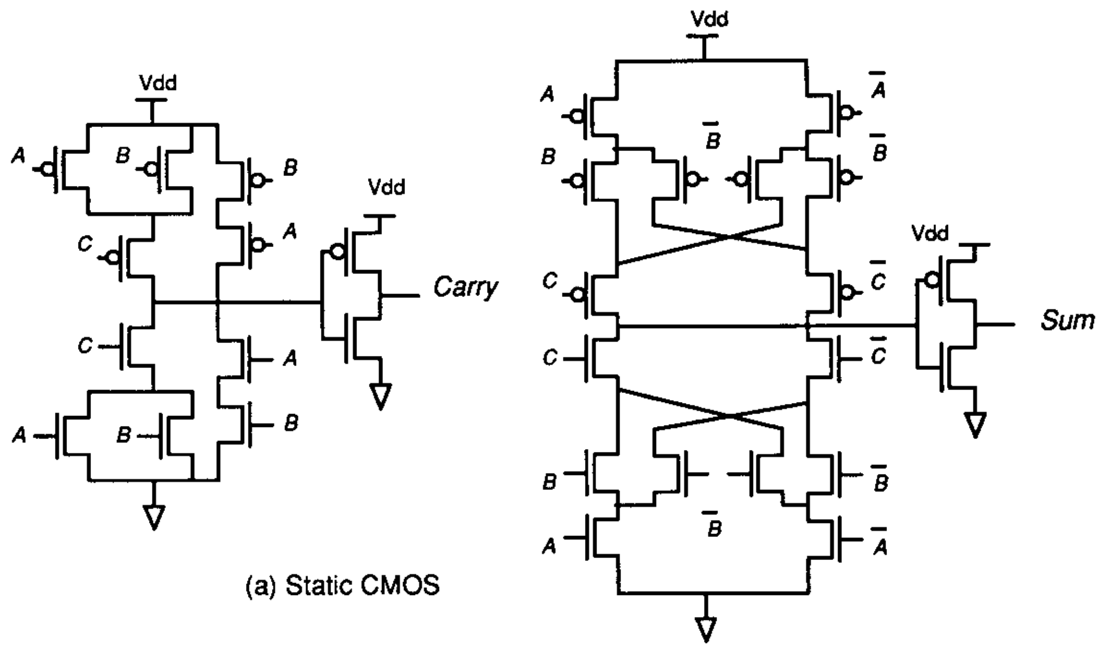
2.4 Comparisons of Different Logic Structures

To compare the performance of the various form of logic structures, a full adder is implemented with all the logic techniques discussed in this chapter. Figure 2.17 shows the various forms of full adder cells in different logic families. A total of eight different adders are simulated with SPICE using typical parameters listed in Appendix A. Three of them are single–ended logic structures and the other five use differential logic structures. In order to have meaningful speed comparison, all transistors are sized as L/W = 3.0m / 4.0m. Each of the full adders is chained linearly to form an 8–bit carry–ripple adder. A small wiring load capacitance is included with output nodes of each bit. The worst case input patterns of A=(11111111), B=(00000000) is applied to these adders together with the initial carry set to 1. The delay per–bit is calculated by dividing the total dealy of an 8–bit carry–ripple adder by 8. Power dissipation is also simulated using SPICE with a technique described by Kang [Kang 86].

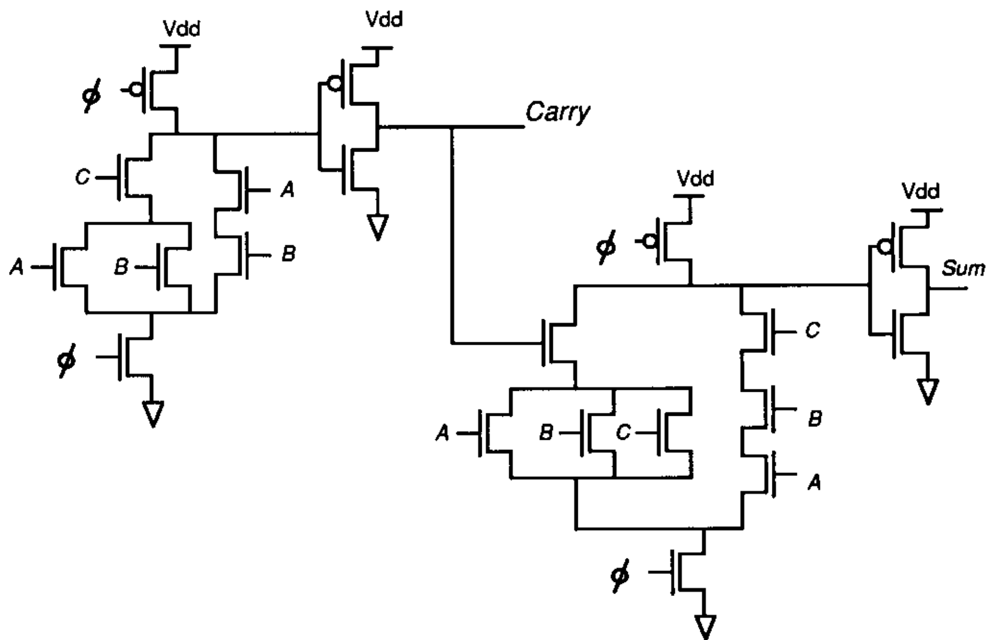
A tabulated comparison is shown in Figure 2.18. We observe that the Differential Split–Level (DSL) circuit yields the best performance. This is explained in the previous section due to the need to only swing one–half of the rail–to–rail power voltage. However it also consumes much more power. Sample Set Differential Logic and Clocked CVSL have the worst delay–power product. ECDL has the best power–delay product.

2.5 Conclusion

Differential logic can be faster than conventional static. The fastest static differential logic is the differential split–level logic (DSL). However, DSL consumes static power. It also has problems with noise margin and number of input signals. In contrast, ECDL is static. It consumes no static power. It is not as fast as DSL but gives better performance in comparison with other differential logic families. It does not have practical problems with number of input signals. In the next section, ECDL is studied in more detail.

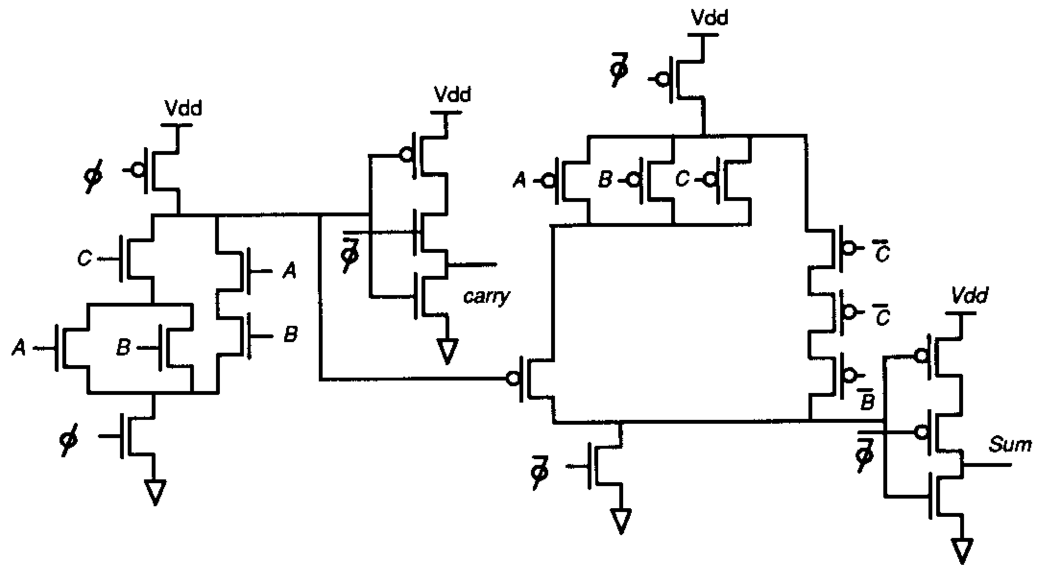


(a) Static CMOS

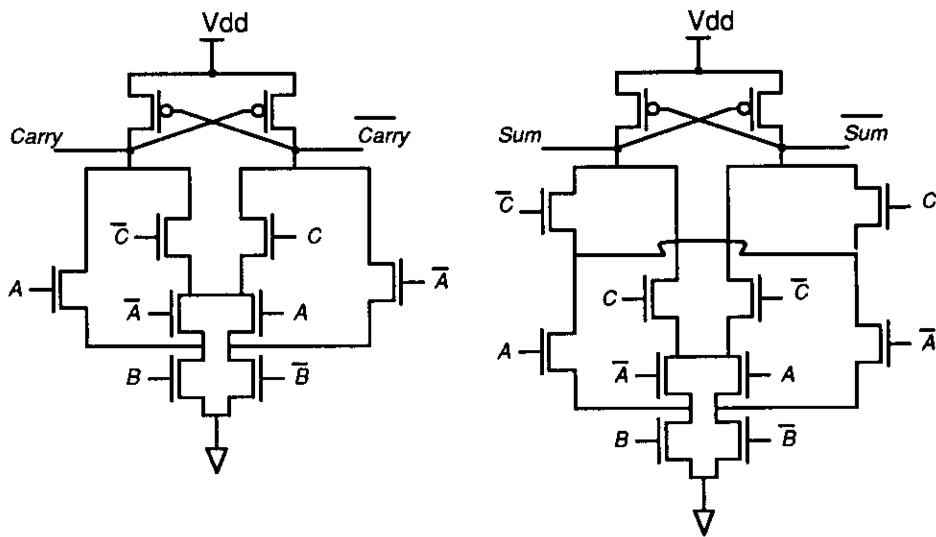


(b) Domino CMOS

FIGURE 2.17 Full Adder Implemented with Static CMOSN and Dominal CMOS

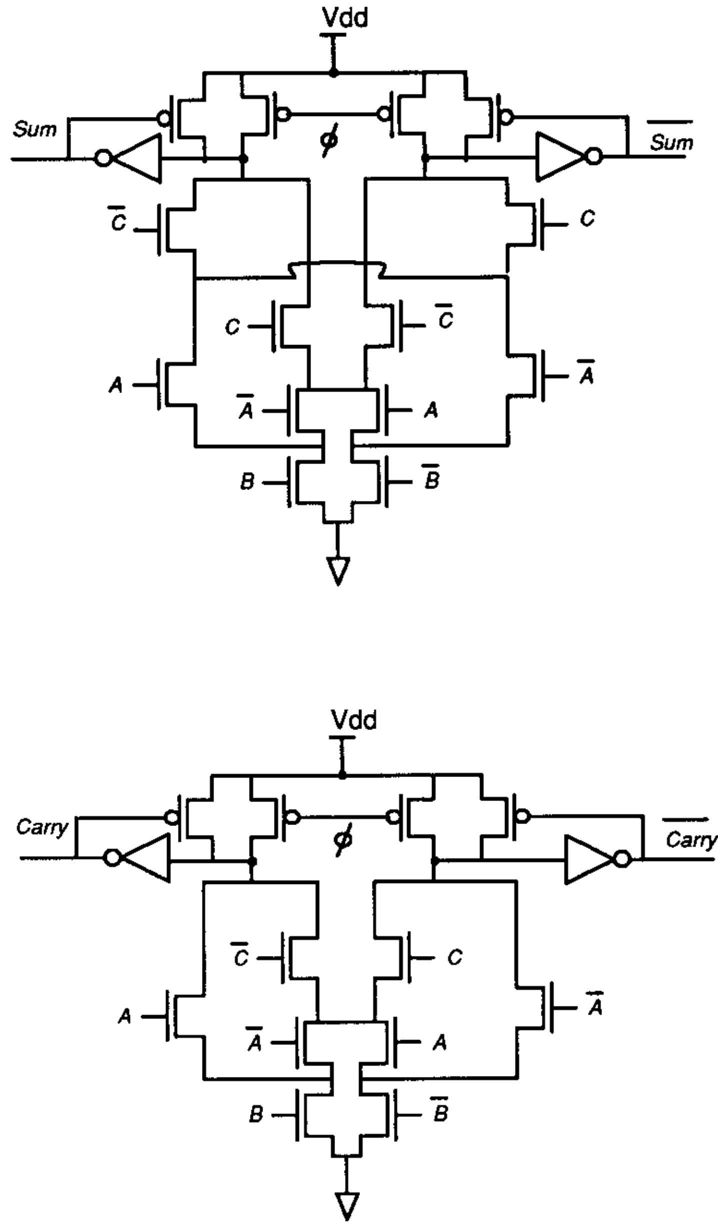


(c) P–N Dynamic CMOS (NORA)



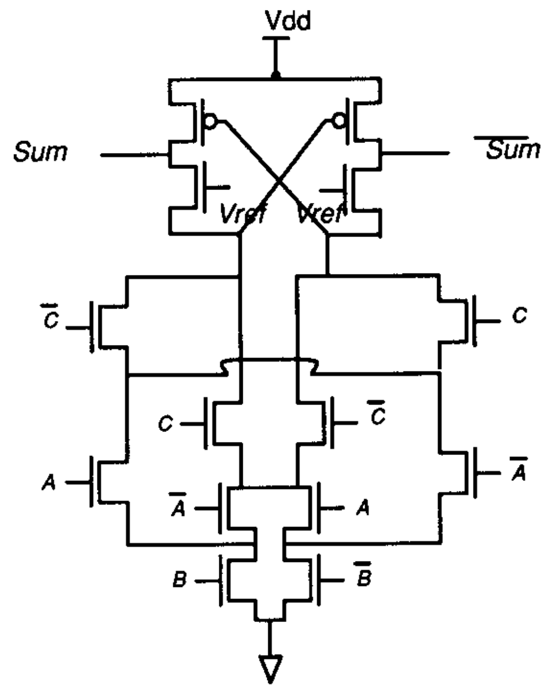
(d) CVSL CMOS

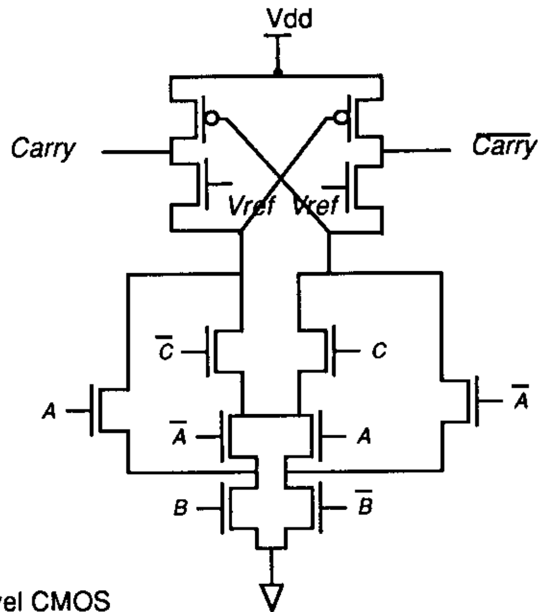
FIGURE 2.17 Full Adder Implemented with NORA and CVSL CMOS



(e) Clocked CVS CMOS (Domino CVS)

FIGURE 2.17 Full Adder Implemented with clock CVS CMOS





(f) Differential Split–Level CMOS

Figure 2.17 Full Adder Implemented with Split–Level CMOS

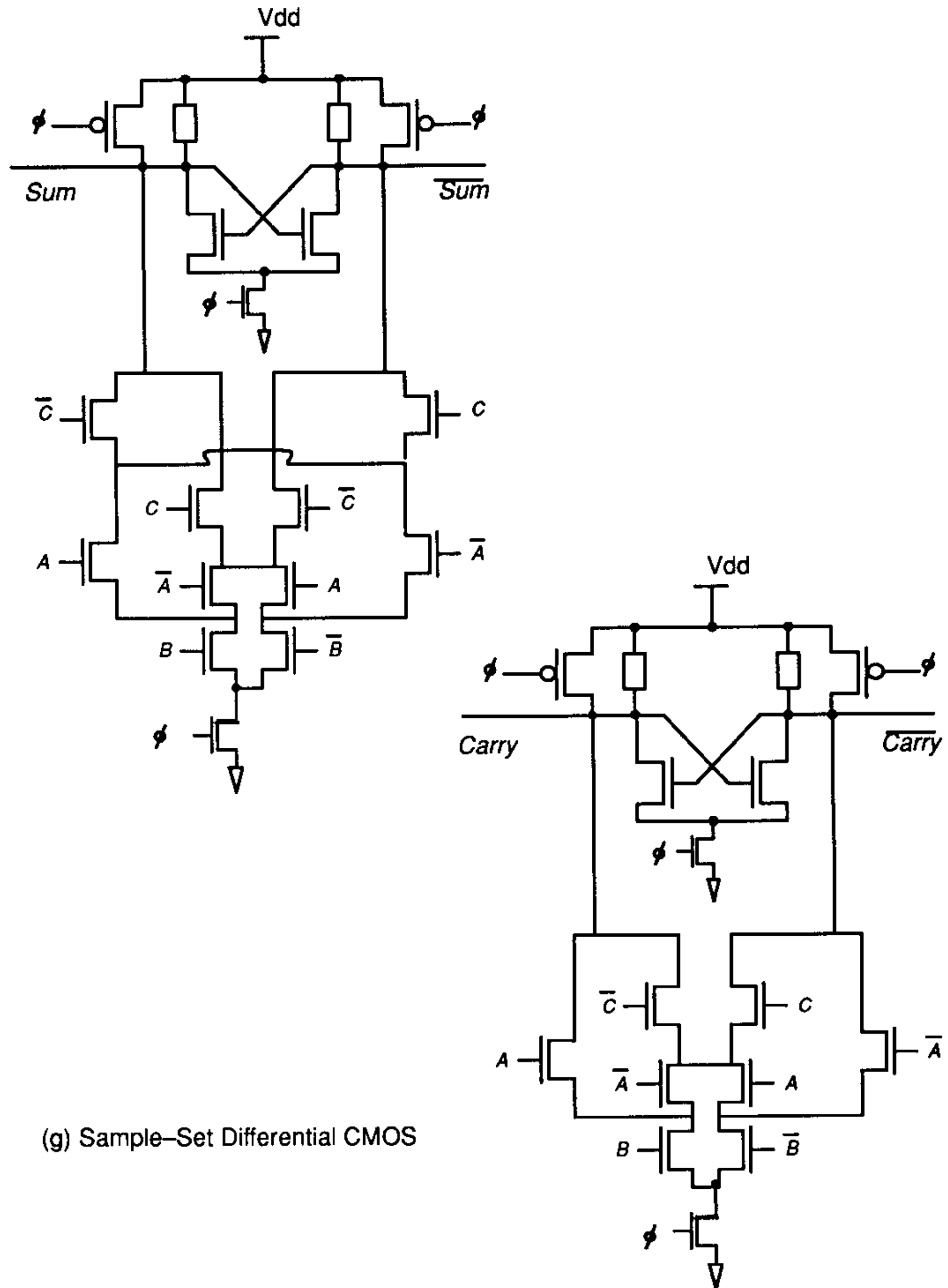
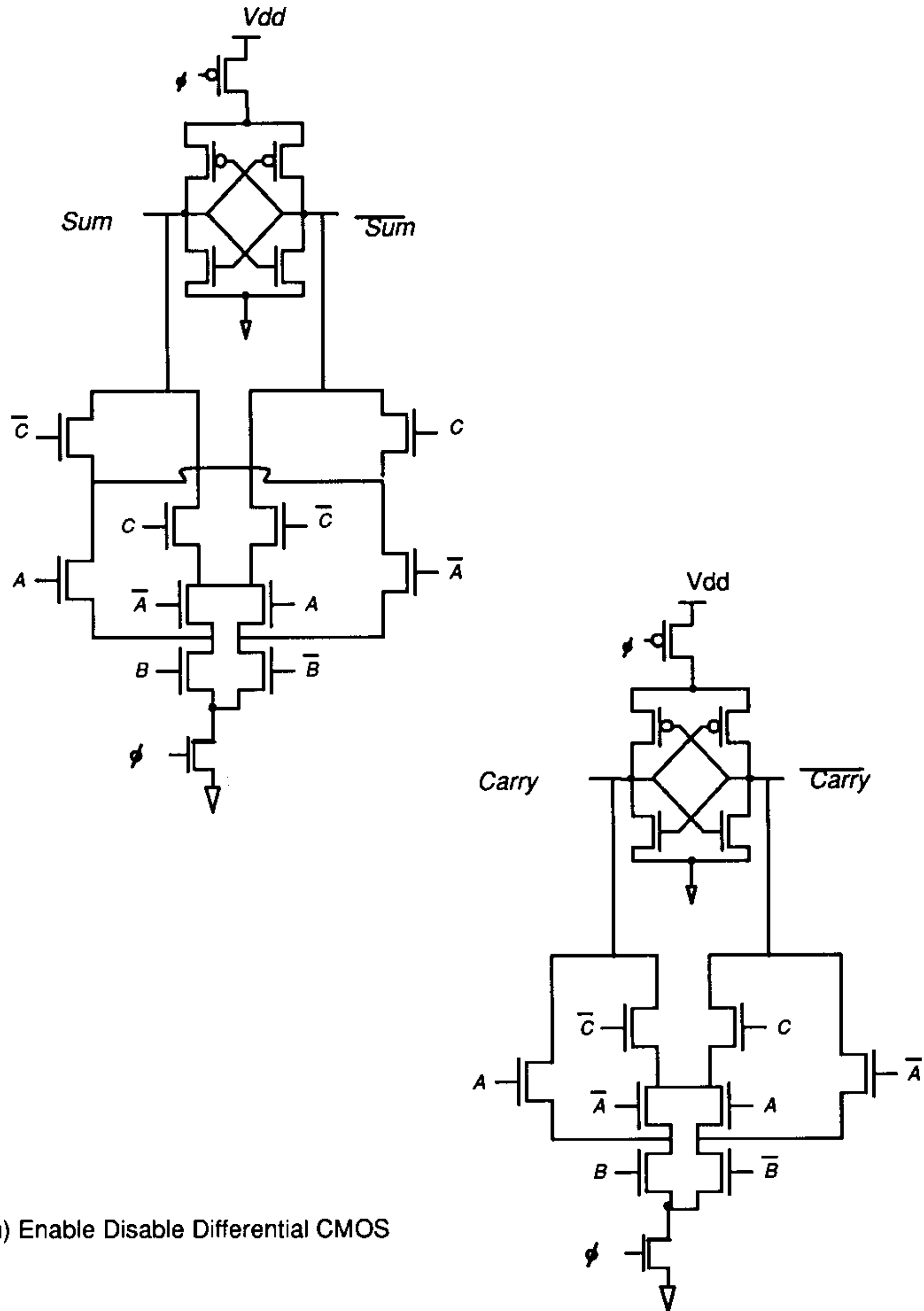


Figure 2.17 Full Adder Implemented with Sample–Set Differential CMOS



(h) Enable Disable Differential CMOS

Figure 2.17 Full Adder Implemented with ECDL

	# of N devices	# of P devices	Total #	carry propagation time (ns/bit)	power dissipation (mW at 10Mhz)
Static	15	15	30	6	0.35
Domino	16	4	20	7	0.52
NORA	11	13	24	7.5	0.37
CVSL	18	4	22	12	0.56
Clocked CVSL	22	12	34	7	1.31
Sample Set	30	12	42	4	2.67
Split Level	22	4	26	2	1.82
Enable Disable	28	7	35	3.5	0.34

Figure 2.18 Comparison of Different Full Adder Implemented with Different Logic Structures

Chapter 3

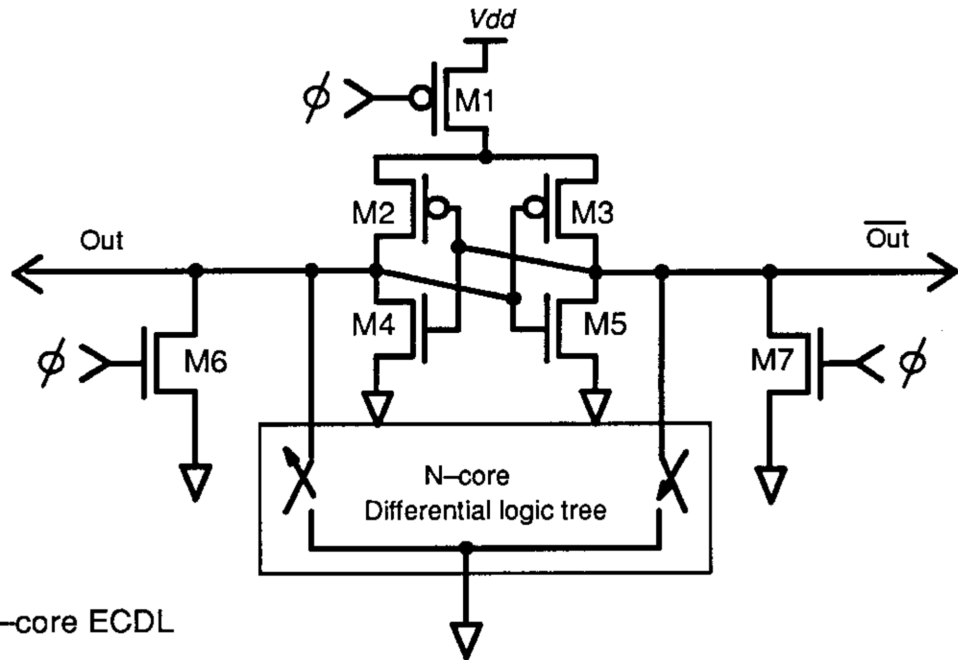
Basic Principles of Enable/Disable CMOS Differential Logic

3.1 Introduction.

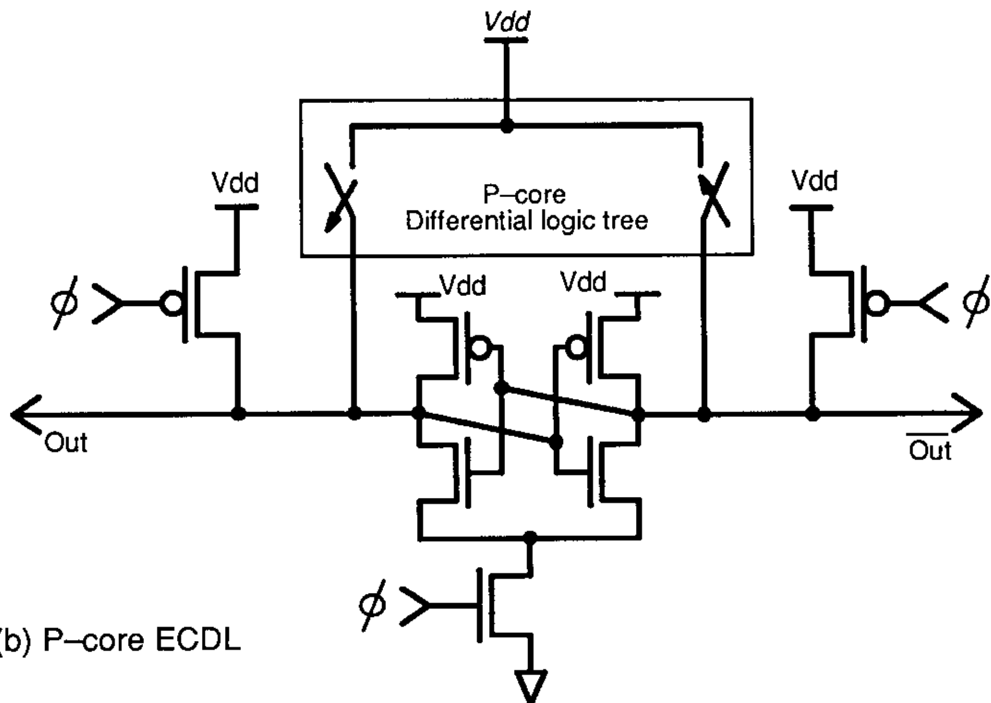
There are several CMOS differential logic families discussed in the previous chapter. In this chapter we will discuss the detailed circuit design of Enable/Disable CMOS Differential Logic circuits(ECDL). We will explain the reason why ECDL implementaiton of a boolean function is faster in comparison with other differential circuits implementation. We will also discuss why its fall time and rise time does not depend on the number of transistors stacked in series used to realized the boolean function.

3.2 Enable/Disable CMOS Differential Logic Operation Principle

ECDL is based on principles similar to the complementary set–reset logic (CSRL) proposed by Mead and Wawrzynek [Mead 85]. General schematics for both the N–core and P–core ECDL are depicted in Figure 3.1. Each ECDL gate has two main functional blocks. They are the latch block and the logic tree block. The latch block consists of an enable/disable transistor, two preset devices and two inverters connected as a cross–couple latch. The logic tree block consists of the differential network tree logic. There are two operational states for the latch used in EDCL. One is the unpowered state and the other is the powered state. While the latch is un–powered, both output nodes are initialized to ground or *Vdd* for N–core type and P–core type ECDL, respectively. For an N–core ECDL gate, the latch is powered up (or turned on) by the trailing edge of the clock signal. As the clock signal goes low, the enable/disable transistor, M1 (Figure 3.1) will turn on and supply current to the cross coupled inverters. Since both outputs are preset to low by the clock signal, M2 and M3 will also turn on. However, since one of the output nodes has a path connected to ground while the other path is cutoff, the latch will be set to either (high, low) or (low, high). The differential network tree determines the way the latch will set. Instead of using only the differential network tree to discharge / charge the output, the transistors in the latch are used to charge and discharge. This is the main reason why ECDL can have more N–devices in series without performance degradation. In an N–core type EDCL gate, the P–channel transistors of the cross–coupled latch are used to charge up one of the output nodes. The N–transistors in the latch are used to discharge the node in P–core type EDCL. The feedback action of the latch also aids the charging and discharging once the latch decides which way it will be set.



(a) N–core ECDL



(b) P–core ECDL

Figure 3.1 General Schematics of ECDL Gate

The feedback action also helps to speed up the operation of ECDL gates. Similarly, the P–core ECDL gate uses an N–channel device to enable / disable the crossed coupled inverter latch. However, the P–core ECDL uses two P–devices to preset the output nodes to high. The sizing of the transistors in the cross–coupled inverter latch as well as the enable/disable transistor will affect the performance of the ECDL gate. We discuss this in more detail in the next section.

3.3 Electrical Properties of Enable/Disable CMOS Differential Logic

In Figure 3.2, a simple N–core ECDL inverter is illustrated. We use this simple ECDL inverter first to examine the details of the behavior of each device. In the following discussion, we denote the voltage at a particular node i as V_i , the gate to source voltage of a transistor Mi as $V_{gs}(Mi)$, the drain to source current of a transistor Mi as $I_{ds}(Mi)$ and the current charging node i as I_i .

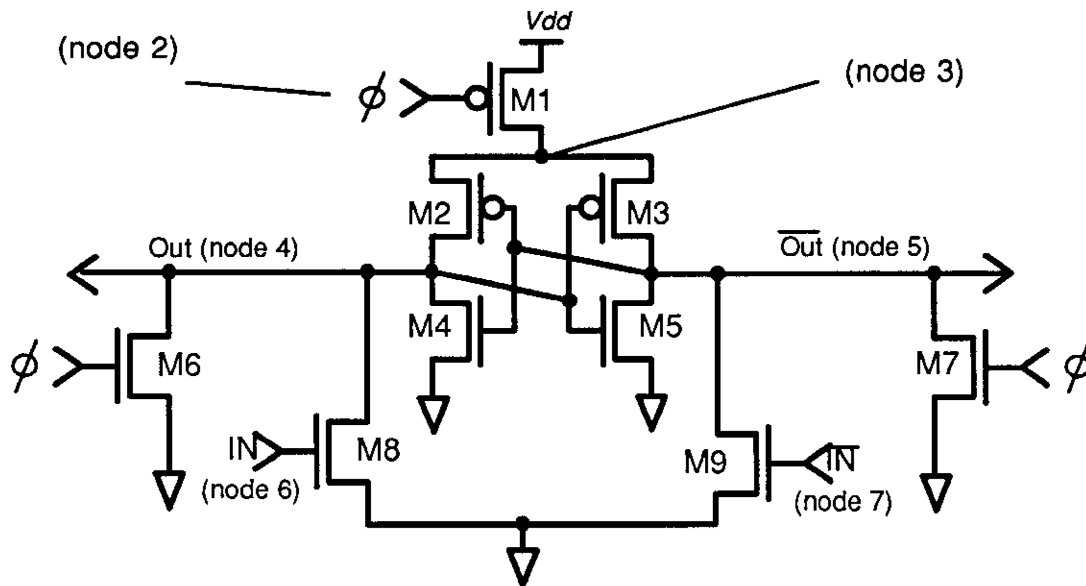


Figure 3.2 A Simple ECDL Inverter

Let us assume initially at time zero that the clock signal (f) is high (5 Volts). Transistor M1 is in the cutoff region. There is no current path from *Vdd* to node 3. Transistors M6 and M7 are both in the saturation region, bringing both outputs to ground (nodes 4 and 5 are at 0 volts). Since node 3, node 4 and node 5 are at 0 volts, the gate to source voltages (*Vgs*) of devices M2 and M3 are both at 0 volts and are all in the cutoff region. Similarly, devices M4 and M5 are also in the cutoff region, since their *Vgs* are also at 0 volts. Devices M8 and M9 are also in the cutoff region, since both the inputs are low. At time T, the clock signal starts to drop. Let's assume it goes down to 4 volts. Device M1 has *Vgs* equal to minus one volt which is less than the threshold voltage of a P–channel device.

$$V_{gs}(M1) = V_2 - Vdd = 4 - 5 = -1 \quad (3.1)$$

Therefore M1 is in the saturated region. Its source drain current charges up node 3. Sometime

later, voltage at node 3 goes up from zero to a voltage larger than the threshold voltage. This causes the gate to source voltages of M2 and M3 to drop below the threshold voltage.

$$V_{gs}(M2) = V_{gs}(M3) = V_5 - V_3 = 0 - V_3 = -V_3 \qquad (3.2)$$

As a result both M2 and M3 are in the saturation region. However at the same time, the input signal and its complement are being applied to M8 and M9, respectively. Let us assume that the input is a logic 1. The voltage at node 6 is going up from zero to be greater than the threshold voltage, while the voltage at node 7 remains at zero.

$$V_{gs}(M8) = V_6 - GND = V_6 \qquad and \qquad V_{gs}(M9) = V_7 - GND = 0 \qquad (3.3)$$

We conclude that M8 is also in the saturation while device M9 remains in the cutoff region. Since both M2 and M8 are on, the current which is used to charge up node 4 is the difference of the drain to source current of these two devices minus the source–drain current of the device M6.

$$I_4 = I_{ds}(M2) - I_{ds}(M8) - I_{ds}(M6) \qquad (3.4)$$

At the same time, M3 is turned on but transistor M9 is turned off. The current charging node 5 is equal to the total source to drain current of the device M3 minus the source–drain current of M7.

$$I_5 = I_{ds}(M3) - I_{ds}(M9) - I_{ds}(M7) = I_{ds}(M3) - I_{ds}(M7) \qquad (3.5)$$

This current is definitely larger than the current charging up the capacitance at node 4, since M2 and M3 are of the same size and initially their gate–source voltages are the same and their drain–source currents are the same. Moreover, M6 and M7 are of the same size and their gate–source voltages are the same, which means their drain–source currents are the same also. We obtain the following:

$$\Delta I = I_4 - I_5 = I_{ds}(M8) > 0 \qquad (3.6)$$

This positive current difference will create a difference in voltage between node 4 and 5. By the time the voltage at node 5 is greater than the threshold voltage, device M4 is also in the saturation and is adding more path from node 4 to ground and making sure node 4 is holding at zero volts. As the voltage at node 5 continues to rise, M2 will go from the saturation region to the linear region, then to the cutoff region, further enforcing the current difference between node 4 and node 5. At the same time, the clock has dropped from high to low making the gate–source voltages of M6 and M7 zero. This puts M6 and M7 into the cutoff mode, further increasing the total current charging node 4. It is because of this push–pull effect that a gate in ECDL is faster than CVSL. Moreover, since the logic network is only used to create a difference in current charging the output nodes, having several N–channel devices in series does not affect the performance in a significant manner. Figure 3.3 illustrates the SPICE simulation of an ECDL inverter gate. Notice that the input switches at the same time as the clock signal. There is no setup time for inputs. This fact is used to produce the self–timed circuitry presented in chapter 4. Figure 3.4 illustrates the simulation of a 20 input NAND gate in ECDL. There are 20 N–channel

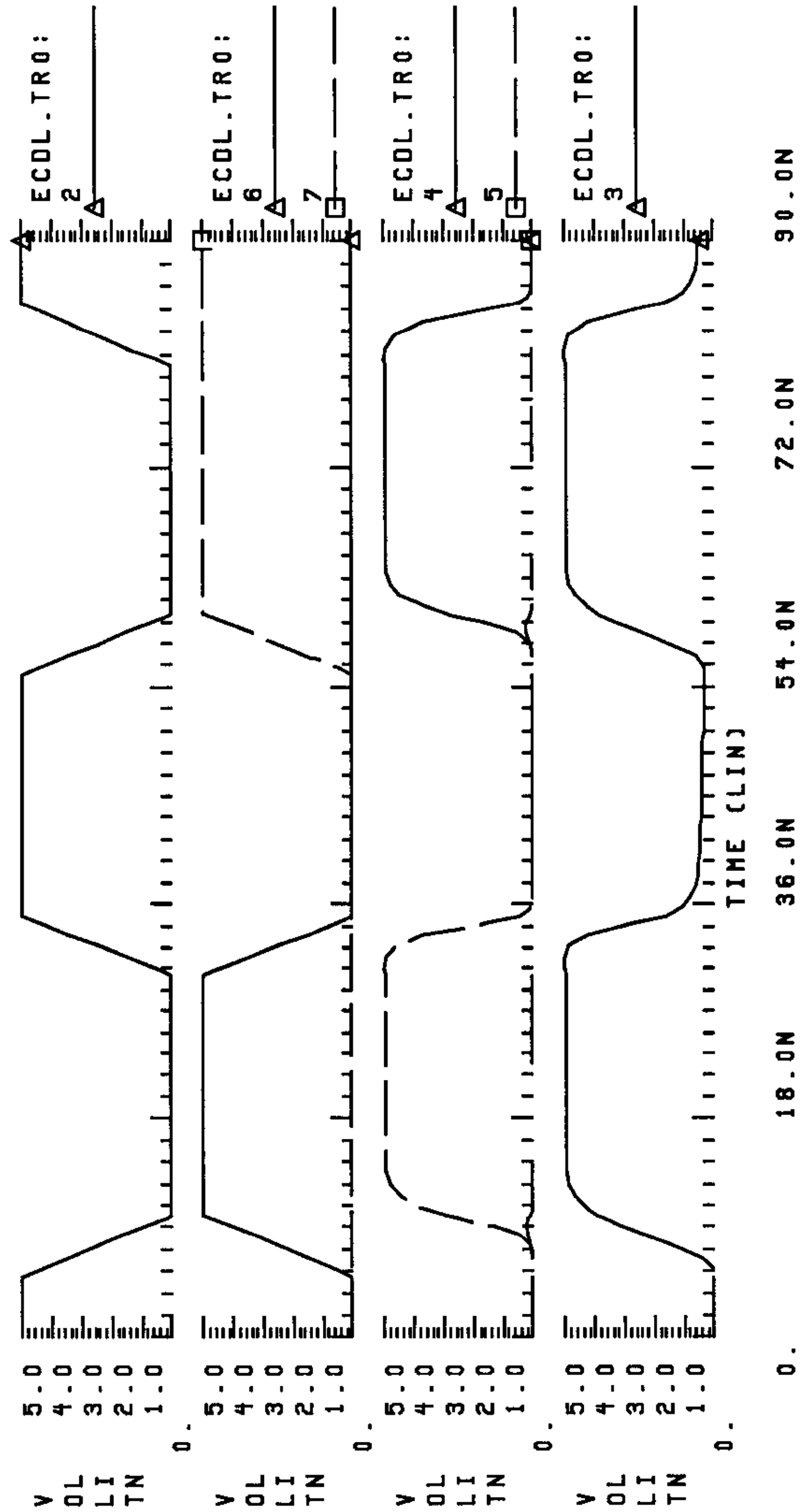


Figure 3.3 HSPICE Simulation of an ECDL Inverter (Figure 3.2)

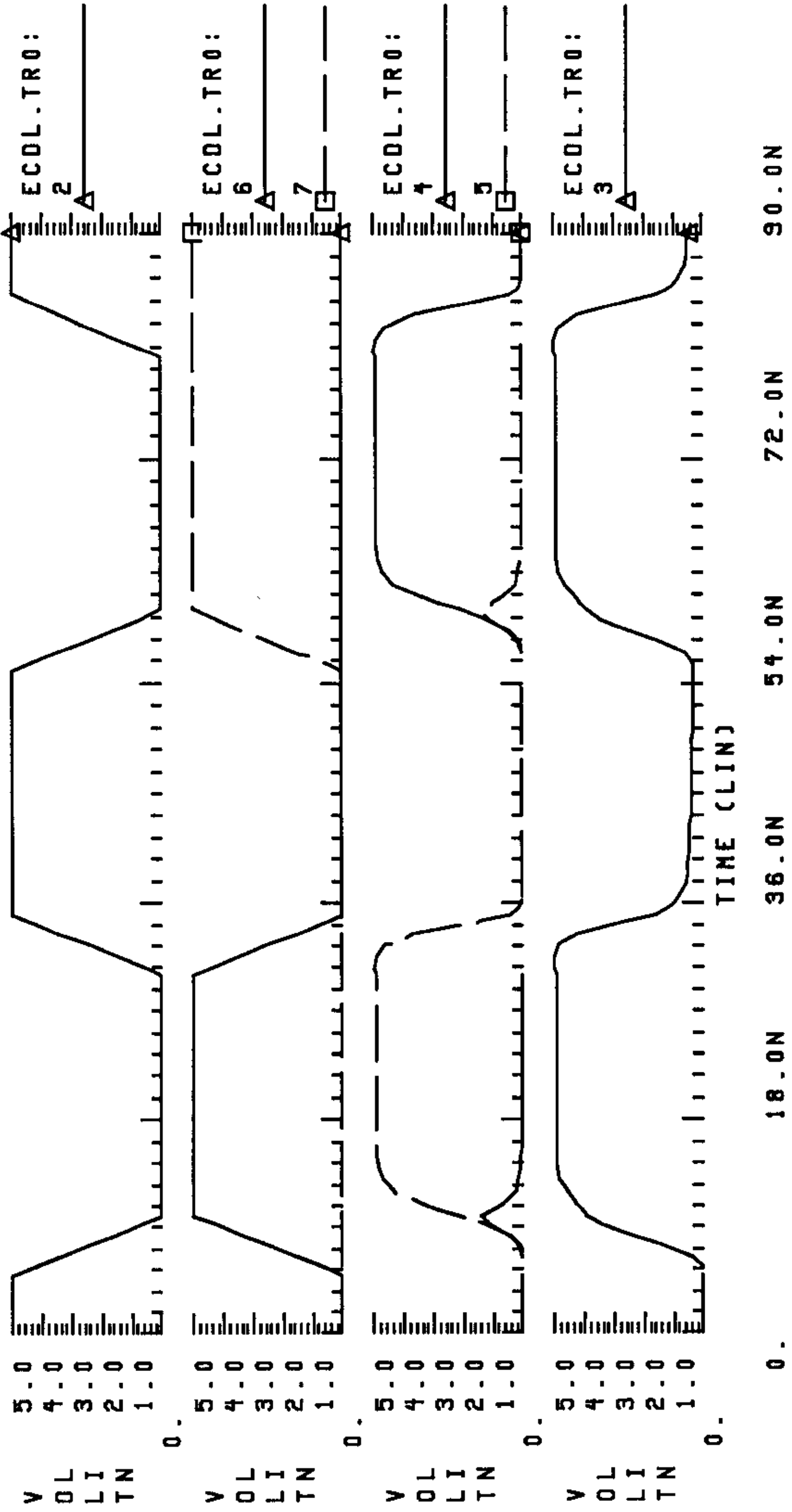


Figure 3.4 HSPICE Simulation of a 20–input ECDL NAND gate

devices in series and the output result is still correct. We observe that as the number of devices in series increases, the noise of the output also increases. This is because the difference in the current initially used to charge up the output and its complement depends on the number of transistors in series. With noise and mismatching caused by fabrication, the practical limit of the number of N–channels in series allowed would be around 10 to 15. Increasing the size of the P–channel devices of the cross coupled inverter latch does not contribute to the current difference. It merely increase the total current available to charge the output nodes. In order to increase the total number of N–channel devices allowed in series, the size of the N–channels used to build logic networks needs to be increased. However, with increase of their size, the gate capacitances also increase which will degrade the overall performance. The other factor which affects the performance of an ECDL gate is parasitic capacitance of the output nodes. This capacitance is contributed mainly by the number of drains connected to the output nodes. In the NAND gate example, the number of drain area of a MOSFET transistor connected to the output nodes are independent of the number of inputs. In a NOR gate, the number of drain of transistors connect to the output is linearly proportional to the number of inputs. With a complex logic tree, the speed will be reduced compared to a simple inverter.

ECDL is static. There is no static path from power to ground, hence no static power consumption. This is different from the Differential Split–Level CMOS Logic (DSLCL) which trades power consumption for speed gain. Figure 3.5 illustrates the current measured at the power supply of an ECDL inverter. Since ECDL is fully static, there is no minimum clocking frequency requirement. An N–core ECDL's ouptut will remain unchanged as long as the clock signal is low.

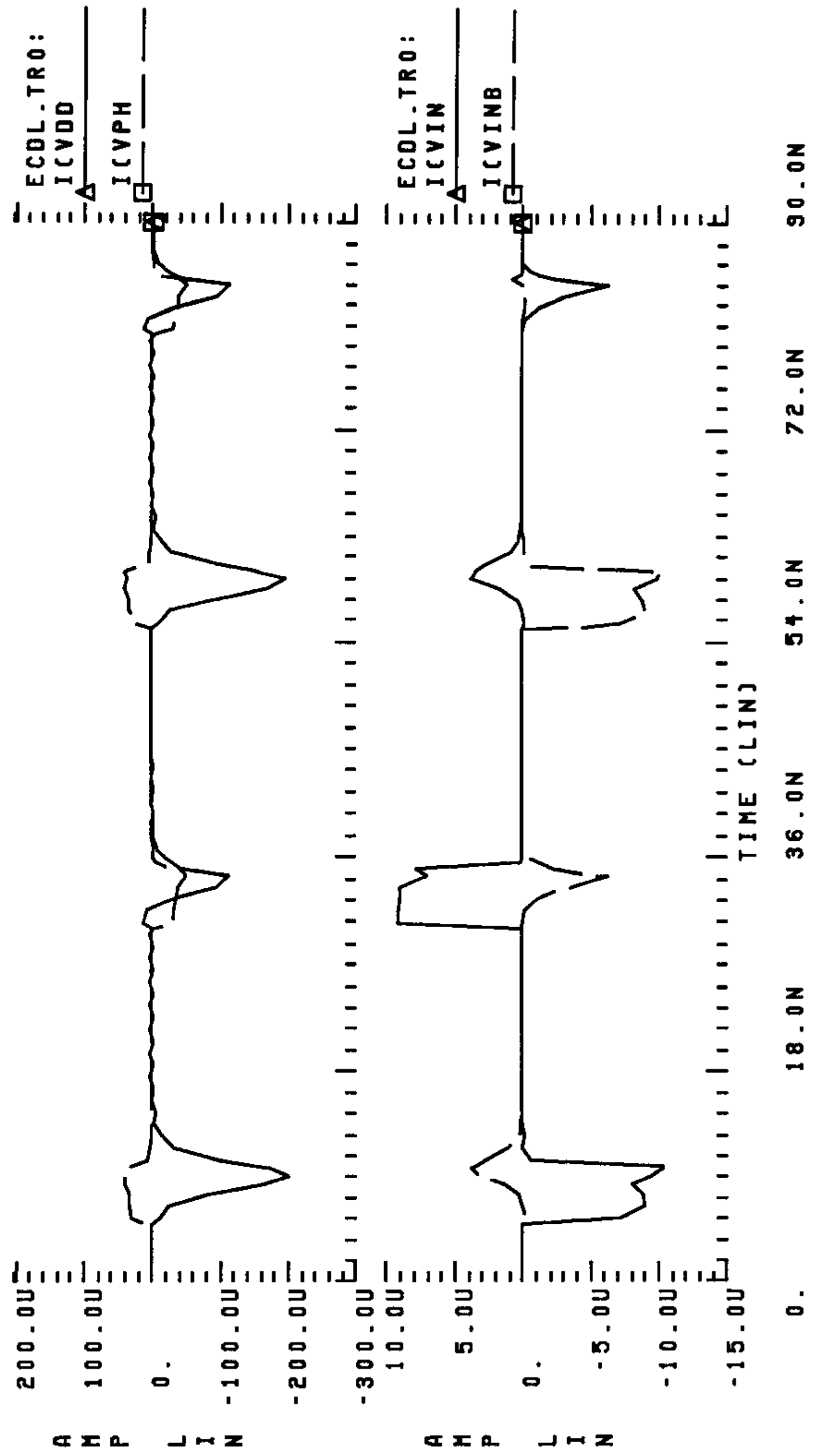


Figure 3.5 Current Consumption of ECDL Inverter

3.4 Design Trade–offs

Speed, area, power consumption and noise margin are some of the more important trade–off criteria in evaluating digital circuit designs. Let us examine an ECDL inverter as illustrated in Figure 3.2. If we treat each transistor as a voltage controlled current source, Figure 3.6 illustrates the large signal equivalent circuit of an ECDL inverter.

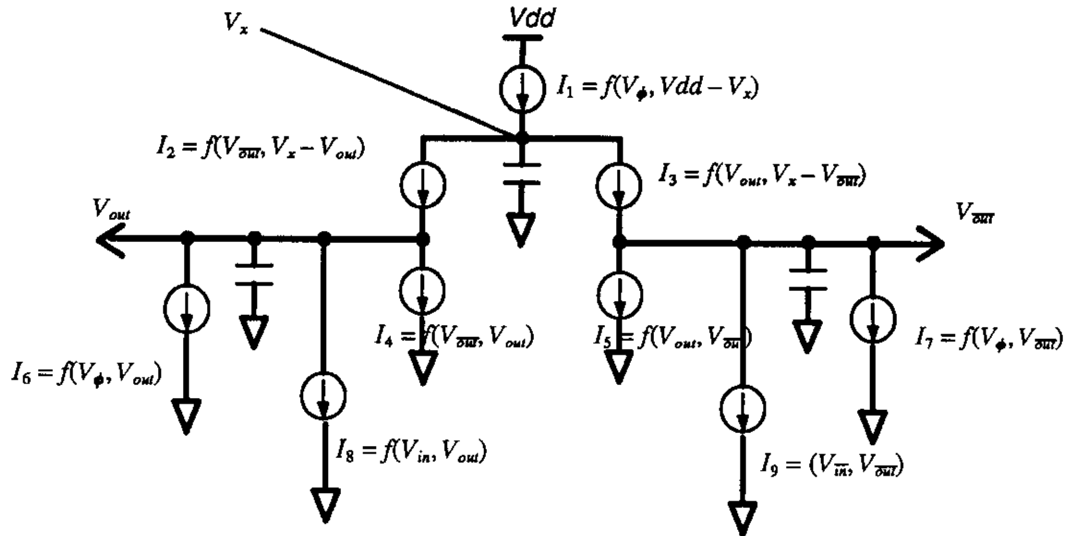


Figure 3.6 ECDL Inverter Analysis

Before the clock makes a transition from high to low at t=0, the voltages at node "x", output and its complement are all equal to zero. As the clock makes its transition from high to low, the following equation defines the voltage at node "x".

$$\frac{d(C_x V_x)}{dt} = I_1 \qquad (3.7)$$

To simplify the analysis, we assume Cx is a lumped capacitive value instead of a function of *Vx*. After integrating both sides of (3.7) we obtain:

$$V_x = \frac{1}{C_x}\int I_1 dt \qquad (3.8)$$

The first order current equations describing the behavior of a MOSFET is expressed as:

$$V_x = 0 \qquad \text{(cutoff)}$$

$$V_x = \frac{\beta}{2C_x}\int (V_{dd} - V_\phi - V_{thp})^2 \; dt \qquad \text{(saturated)} \qquad (3.9)$$

$$V_x = \frac{\beta}{C_x}\int ((V_{dd} - V_\phi - V_{thp})(V_{dd} - V_x) - \frac{(V_{dd} - V_x)^2}{2})dt \qquad \text{(linear)}$$

Here the threshold voltage of the P–channel device is taken to be a positive value (absolute value) incontrast to the usual definition[Wes 85]. From (3.9) we can solve the time needed to charge node "x" to the voltage necessary to turn on M2 and M3. This delay can be reduced either by increasing the W/L value or by reducing the capacitance at node "x". When the voltage of node "x" is larger than the threshold voltage of the P–channel device, M2 and M3 are in the saturation region. Output voltage is expressed as:

$$V_{out} = \frac{1}{C_x}\int I_2 dt \qquad (3.10)$$

At this point the transistors M2 and M3 are in the saturation also, so both output and its complement have voltage as:

$$V_{out} = \frac{1}{C_{out}}\int (I_2 - I_6 - I_8)dt$$

$$V_{\overline{out}} = \frac{1}{C_{\overline{out}}}\int (I_3 - I_7 - I_9)dt \qquad (3.11)$$

Initially, output and its complement are both low (0 volts). Devices M6 and M7 are in the saturation. In order to get positive current to charge up the capacitance, M6 and M7 must be smaller than M2 and M3. Moreover the mobility difference in N–device and P–device, M6 and M7 should be less than half the size of M2 and M3. However, M6 and M7 are only in the saturation for a short period of time. As the clock input continues to drop and output continues to rise, M6 and M7 quickly goes from the saturation, to the linear, then, to the cutoff region. As both output and its complement rise to greater voltage than the threshold voltage of N–devices, either M8 or M9 will be on depending on the input value. The rise time of output or its complement again depends on the size of M2 and M3 as well as on the size of M6 and M7. Assume *Vin* is high, therefore $I_8 > I_9 = 0$. The voltage at output and its complement can be expressed by the following equations:

$$V_{out} = \frac{\beta}{2C_{out}}\int (V\overline{out} - V_x - V_{thp})^2 - (V_\phi - V_{thn})^2 - (V_{in} - V_{thn})^2 dt$$

$$V_{\overline{out}} = \frac{\beta}{2C_{\overline{out}}}\int (V_{out} - V_x - V_{thp})^2 - (V_\phi - V_{thn})^2 dt \qquad (3.12)$$

If the current I_8 is too small in comparison with I_2, then the inverter will give wrong results, since the mis–match caused by the cross–coupled transistors may be more than the difference caused by the current discharging. However, if the current I_8 is too large, it fights with I_2 and makes the rise time very long.

Figure 3.7 summarizes the timing of this ECDL inverter. There are three areas that contribute to the delay. From time 0 to t_1, the delay is the fall time of the clock signal from *Vdd* to *Vdd–Vthp*. The delay time from t_1 to t_2 is caused by the charging of node "x" to

Vthp. Time from t_2 to t_3 is contributed by charging up both outputs to *Vthn*. Finally, the time from t_3 to t_4 is the rise time of the output. From the analysis discussed in this section, the P–channel devices should be large to gain speed. The only devices needing careful balancing are the input N–channel devices. Their size also depends on the number of transistors in series.

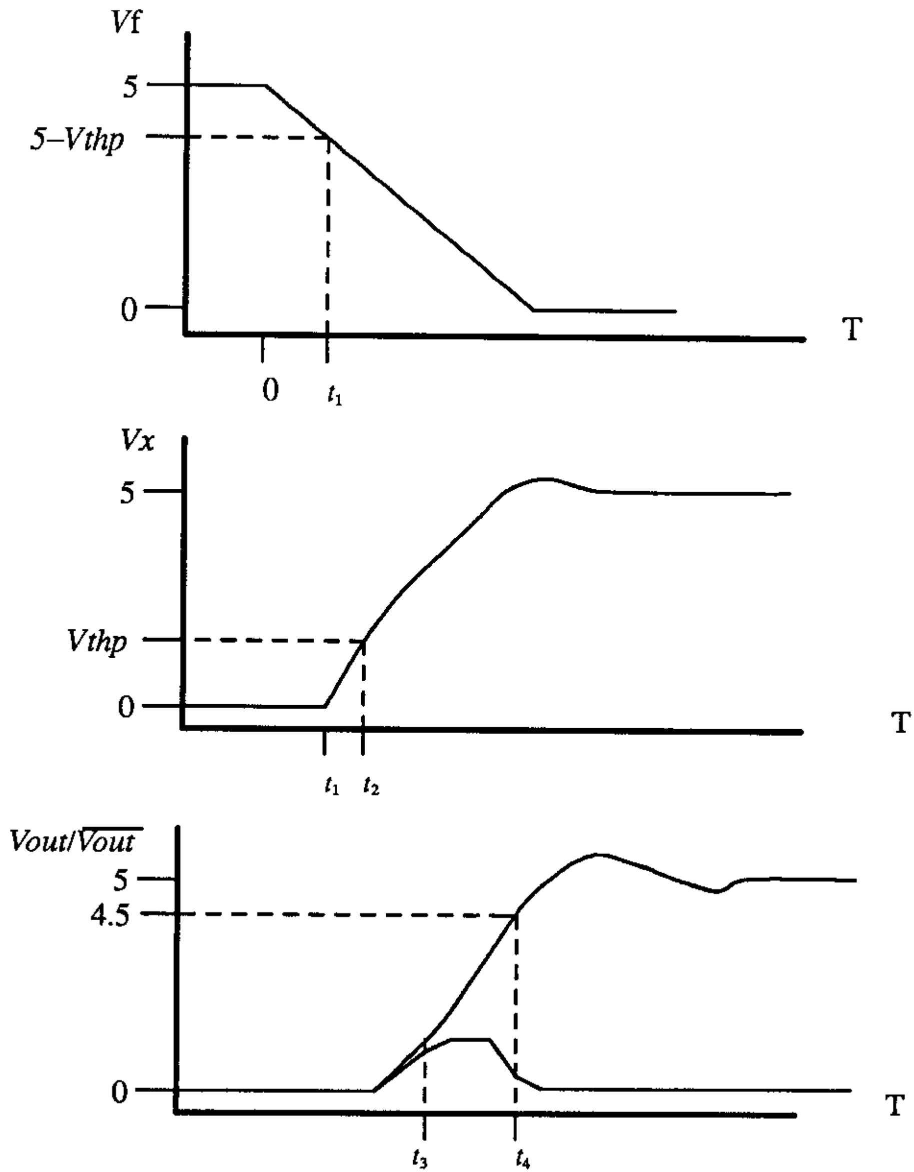


Figure 3.7 Timing diagram

3.5 Conclusion

ECDL has the same main advantage as static CVSL, which is that there is no static power consumption. It is faster and has no pratical limitation in terms of number of N–channel devices in series allowed in the network tree. It has several advantages over Sample Set Differential Logic (SSDL). First, in SSDL, during the sample phase there exists a path from power to ground. Because of this path, SSDL dissipates power half of the time. Second, SSDL needs two extra inverters to convert the outputs to right polarity before they can be fetched into the next gate. These two inverters increase the area needed and add delays to the swtiching time. Third, SSDL's output level is not fully to the power supply level. This decreases noise margin. ECDL has none of these disadvantages.

Chapter 4

Self–Timed Networks for Iterative Functions

4.1 Introduction

An *iterative network* is a circuit network constructed by interconnecting a set of identical modules or submodules in a one–dimensional regular fashion[Erc 85]. This concept was introduced by Keister et al. [Kei 51]. The best known example of an iterative network is a binary carry–ripple adder. This is a unidirectional iterative network. Iterative networks are widely used in implementing some logic functions which have a lar ge number of inputs. Implementing these functions with a single logic network is usually not cost effective. However, many times these functions can be partitioned and implemented with iterative networks. The advantages of this implementation approach over conventional combination circuit realizations of the same functions are [Ung 77]:

(1) Limited number of elements to design: instead of designing an *n*–input network, a much smaller network is designed.

(2) Simple connections: only nearest neighbor modules are connected.

(3) Regular organization.

(4) The total number of modules grows linearly with the number of inputs instead of nearly exponentially.

(5) Testing, fault detection and repair are more regular.

All these advantages contribute to the suitability for very large scale integration implementation. The main disadvantage of using iterative networks to implement logic functions is its delay, because signals must propagate through all stages of the network sequentially. However, there are general methods which can be used to speed up the implementation of iterative networks.

In order to reduce the delay of an iterative network, we may reduce the number of cells. The most straight forward way to reduce the number of cells is by combining two or more cells into one complex cell. This is the *merged cell method* proposed by Unger [Ung 77]. Take for example, the binary ripple–carry adder. By combining two full adders into one 2–bit adder, we are effectively reducing the delay by half if the new module has the same gate delay as the original gate.

Another method to reduce the delay problem of iterative networks is to transform a linear array into a tree array. [Ung 77] also studied this problem and outlined the conditions under which an iterative network can be transformed into a tree network.

4.2 ECDL Iterative Networks

As mentioned in Chapter 3, there are two states for each of the general ECDL circuits – the *disabled* state and the *enabled* state. The rising or the falling of the clock signal enables or disables the cell. Figure 4.1(a) illustrates a straight forward way to generate a *Complete (Done)* signal for its outputs. During the disable stage, both output and its comple-

ment are at logic 0. The output of *Done* signal is at logic 1. As the ECDL gate is enabled and one of the output nodes is charged to logic 1, the *Done* signal will become 0. However, this is not a very efficient circuit.

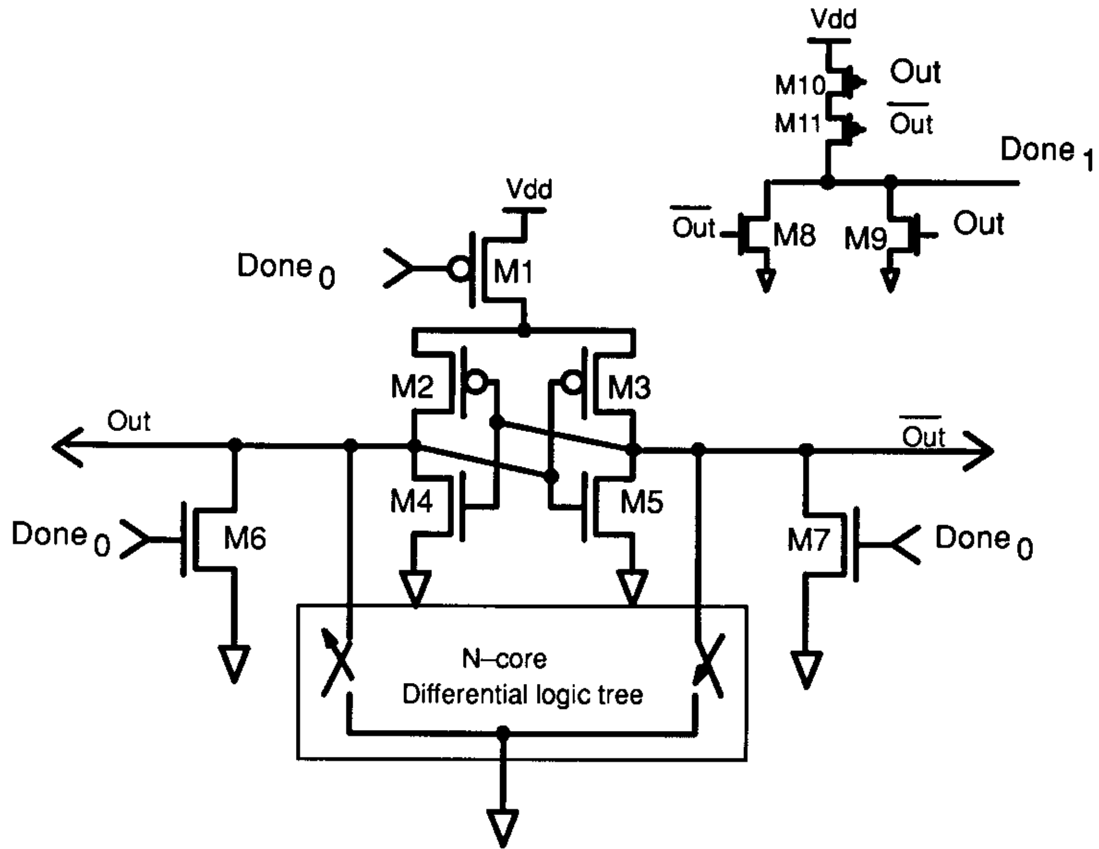


Figure 4.1(a) N–Core ECDL Gate with the Completion Circuit

Figure 4.1(b) shows a modified circuit of Figure 4.1(a) where a local clock signal can be generated with the reduction of one transistor per cell. No matter which method is used to generate the local *Done* signal, an iterative array network can be built by organizing several identical ECDL cells in an array fashion. An initial enabling signal is fed into one end of the array. As the cell finishes its evaluation, a completion signal is generated and sent to the neighboring cells. This completion signal acts as the initiation signal for the following cells. A "Domino" effect causes this completion signal to propagate through all the cells to the other end. All outputs are stable after the final completion signal is generated.

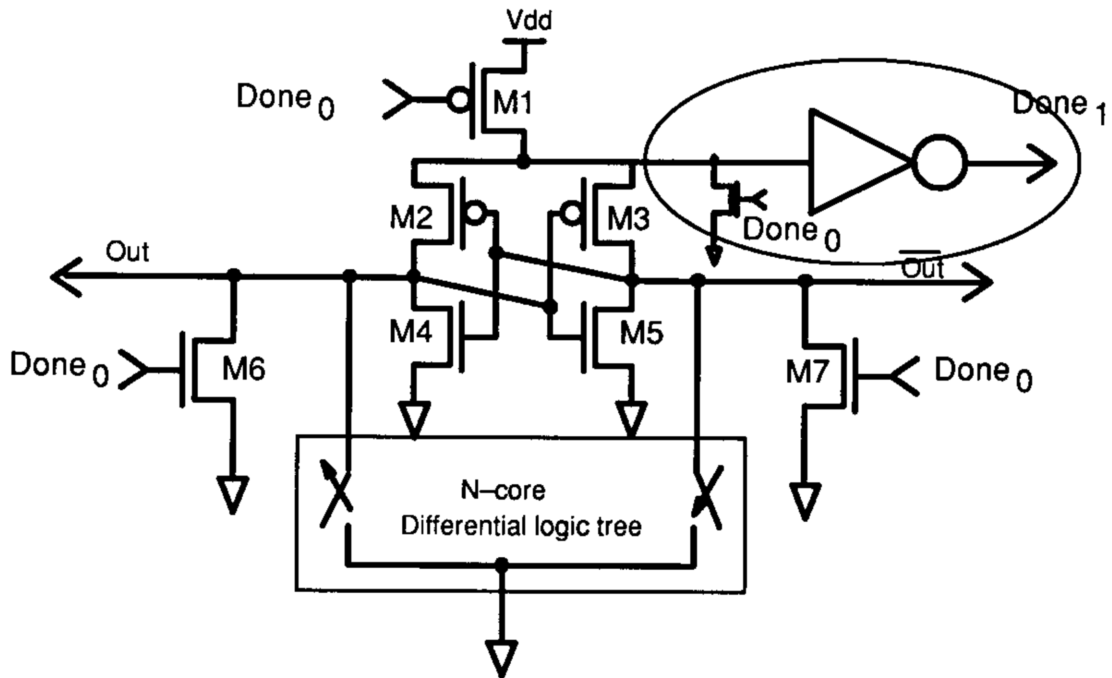


Figure 4.1(b) Modified N–Core ECDL Gate

A one–dimensional iterative network consists of linear array of several identical modules [Erc 85]. Each module is connected only to its nearest neighbors. A given logic function F of n variables can be implemented with an iterative networks if a new logic function G may be derived which satisfies the following condition:

$$F(X_n, X_{n-1}, X_{n-2}, \ldots X_1) = G(X_n, G(X_{n-1}, G(X_{n-2}, G(\ldots, G(X_1, C_0)))))$$

$$where X_n,\ X_{n-1},\ X_{n-2},\ \ldots\ X_1,\ C_0 \text{ are either scalars or vectors} \tag{4.1}$$

In comparison, logic function G is a much simpler function to realize than F. Moreover, intermediate results, $G(x_1, c_0)$, $G(x_2, G(x_1, c_0)$, ... are also available as outputs. We may design the logic function G with an ECDL cell. Chu and Pulfrey [Chu 88] proposed two methods for realizing differential network trees. With these two methods, each individual cell with the minimal transistor count can be implemented either manually or automatically.

Given that there is a method to implement individual self–timed modules, a hierarchical system is readily available. The final completion signal of a particular function may serve as the completion signal for this entire functional block. This block completion signal may then be used to initiate operation in another functional block.

4.3 Examples of ECDL Iterative Networks

In some digital applications, it is desirable to use cyclic codes to represent data. A particular cyclic code, Gray code, is quite important. Conversions from Gray code to binary code are needed. This conversion can be implemented by iterative arrays with cells characterized by simple equations. We obtain each individual bit of the binary code as Gray bits by the following expressions:

$$b_n = g_n$$

$$\vdots$$

$$b_i = b_{i+1} \oplus g_i \qquad 0 \le i \le n-1$$

$$\vdots$$

$$b_0 = G(g_0,\ G(g_1,\ ..\ .G(g_{n-2},\ G(g_{n-1},\ g_n))))\ ..\ .))$$

$$\text{where} \quad G \text{ is } \oplus \qquad (4.2)$$

Besides the most significant bit, each binary bit depends on the previously coded binary bit and the current Gray bit. With this characteristic, an iterative array network can be specified to implement this conversion. Figure 4.2(a) illustrates the connections between cells.

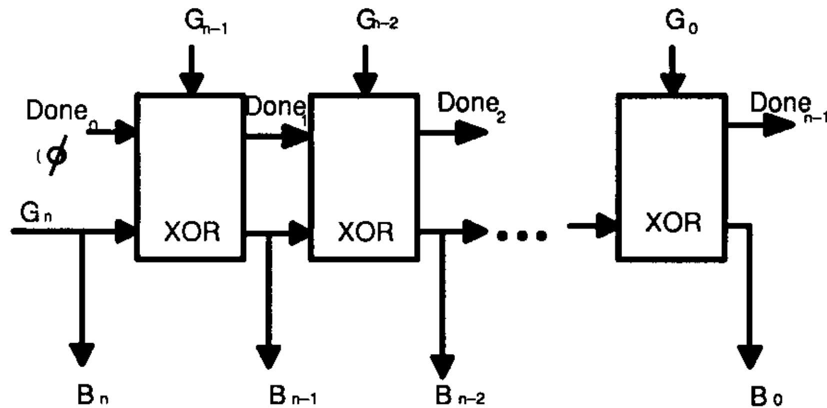


Figure 4.2 (a) Connection of ECDL Cells

Figure 4.2(b) shows the unique cell used for conversion. Other code conversion algorithms can also be implemented with iterative arrays. An example is given by Nicoud [Nic 71] who presented a general algorithm for radix conversion using iterative arrays. To speed up the conversion, we can group k–bits together and use a single function to realize the function of a k–bit converter. Instead of having to wait n stages for the final results to propagate, the total delay is n/k. The next example will illustrate this speed up. This is the *merge cell speedup method* mentioned by Unger [Unger 77].

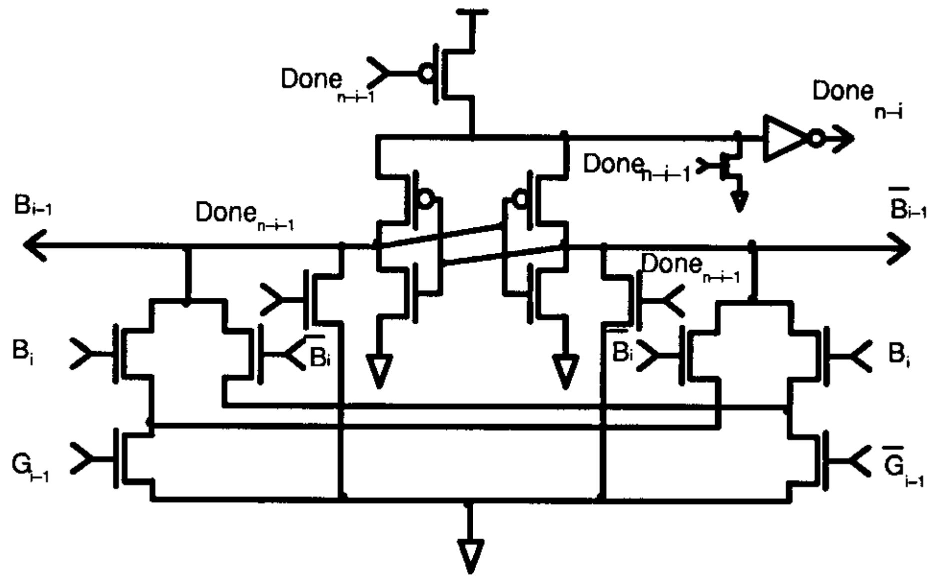


Figure 4.2 (b) XOR ECDL Gate

Magnitude comparators are useful in many subsystems. In Figure 4.3(a) we illustrate a single cell which compares two one–bit numbers and determines if one is greater than the other. Again this cell can be chained in an iterative array to form an *n*–bit comparator. Figure 4.3(b) shows how they are connected. For a faster comparison we can group several bits together. In Figure 4.4, we illustrate a 4–bit comparator done with a single ECDL cell. It is clear that we are trading overall speed with single cell complexity. An *n*–bit comparator implemented using *k*–bit cell can perform a comparison in *n/k* delays. Since the single stage delays changes with the complexity of the single cell, an optimal k value with a given n can be calculated. The major contribution to the increase of a single stage delay comes from the output node capacitance. There are 6 transistors drain/source connected to the output node of the ECDL gate shown in Figure 4.4. Instead, there are only 3 transistors have their drain/source connected to the output node. Through simulation we obtained that an eight bit comparator implemented using 4–bit cells gives a speedup of approximately 2.5 instead of 4. Another interesting observation is that because of the sharing of terms is more possible with complex functions, the total number of transistors used in implementing ECDL comparators with cells depicted in Figure 4.4 is less than the implementation using cells depicted in Figure 4.3. Assume we use *k*–bit comparator cells to implement an *n*–bit comparator, the total transistors counts is:

$$\frac{n}{k}(6k+12) \tag{4.3}$$

The total number of transisitors used in the single–bit style of implementation is :

$$18n. \tag{4.4}$$

Another way to speed up the one–dimensional network is to convert it into a tree network. For example, an *n*–bit parity generator can be implemented in ECDL circuit using tree–network connection. The basic cell of an XOR gate is similar to Figure 4.2 (b) with the

exception of having two P–channel enabling transistors. Figure 4.5 illustrate the between cell connection with self–timed signals.

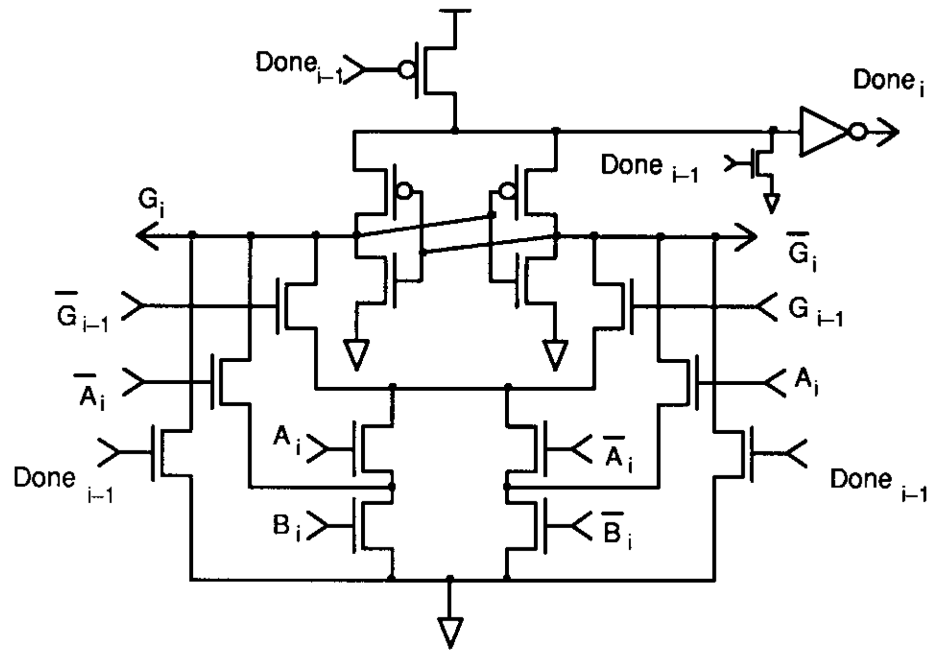
Using self–timed style of design to implement iterative array also gives speed advantage. The delay of a conventional iterative network is:

$$T = (\frac{n}{k} - 1)\Delta C + \max(\Delta C, \Delta Z) \qquad (4.5)$$

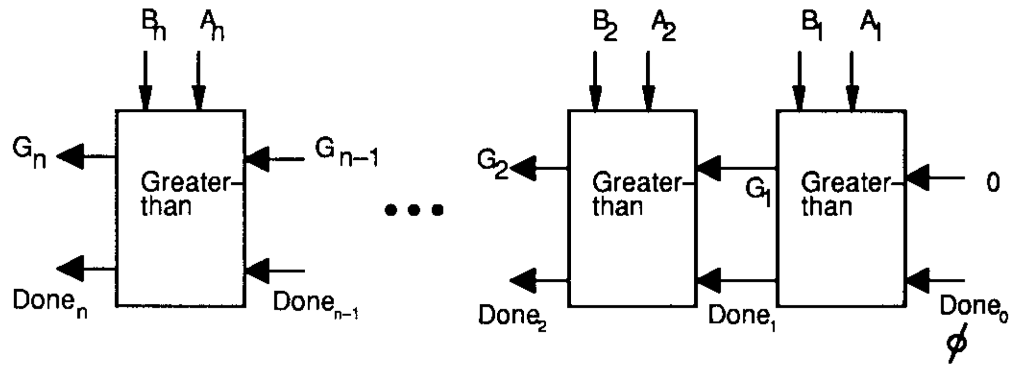
where DC is the delay from the inputs of the cell to its internal outputs, and DZ is the delay of the cell to external outputs. The network delay is calculated as the worst case delay. The self–timed implementation of the same iterative network exhibits the actual delay:

$$T_{ST} = \sum_{i=0}^{\frac{n}{k}-2} \Delta C_i + \max(\Delta C_{\frac{n}{k}-1}\ ,\ \Delta Z_{\frac{n}{k}-1}) \qquad (4.5)$$

where ΔC_i and ΔZ_i are the delay from the inputs of the ith cell to its internal outputs and the delay from the ith cell to external outputs, respectively. Values of ΔC_i and ΔZ_i vary depending on the inputs. In the later chapters we give examples of delays and delay improvements due to self–timed modules.



(a) ECDL Implementation of Single–Bit Comparator



(b) Connection Between Cells

Figure 4.3 N–bit Magnitude Comparator in ECDL using 1–bit Comparators

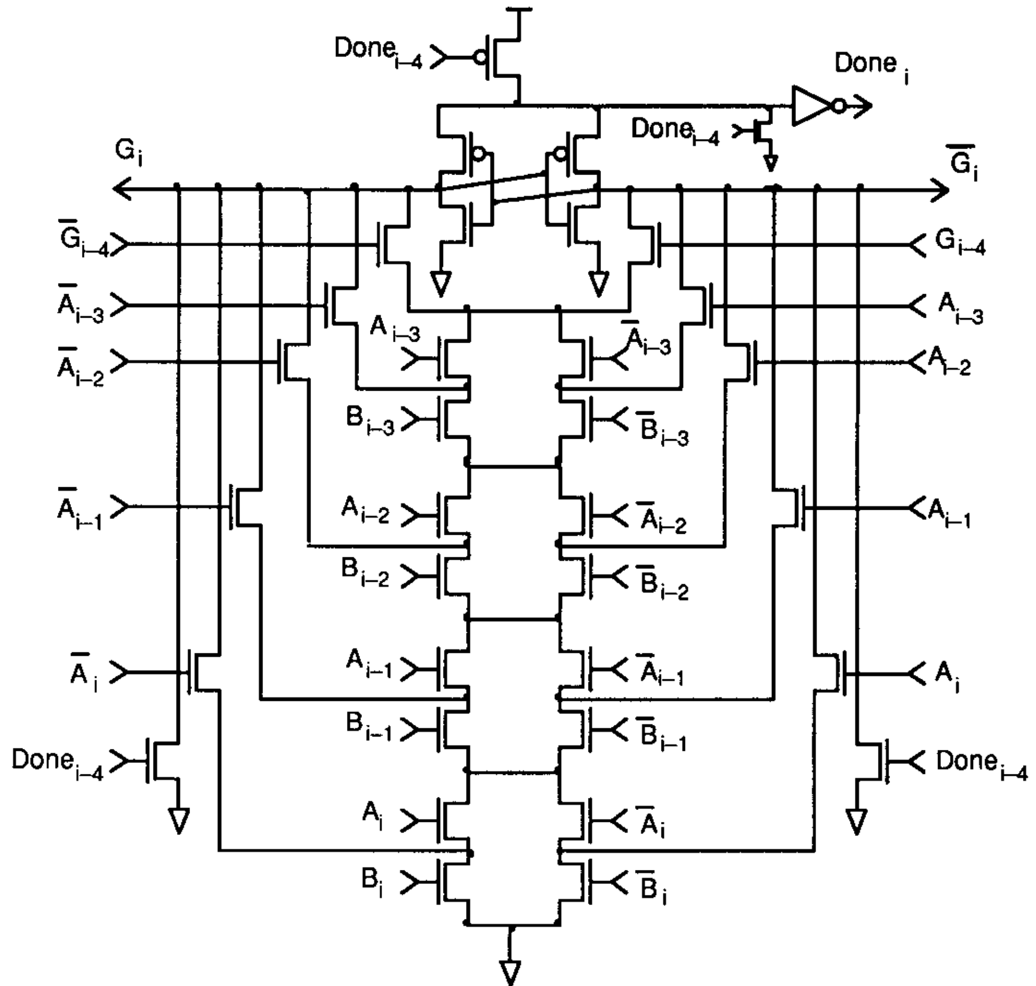


Figure 4.4 4–bit ECDL Comparator Cell

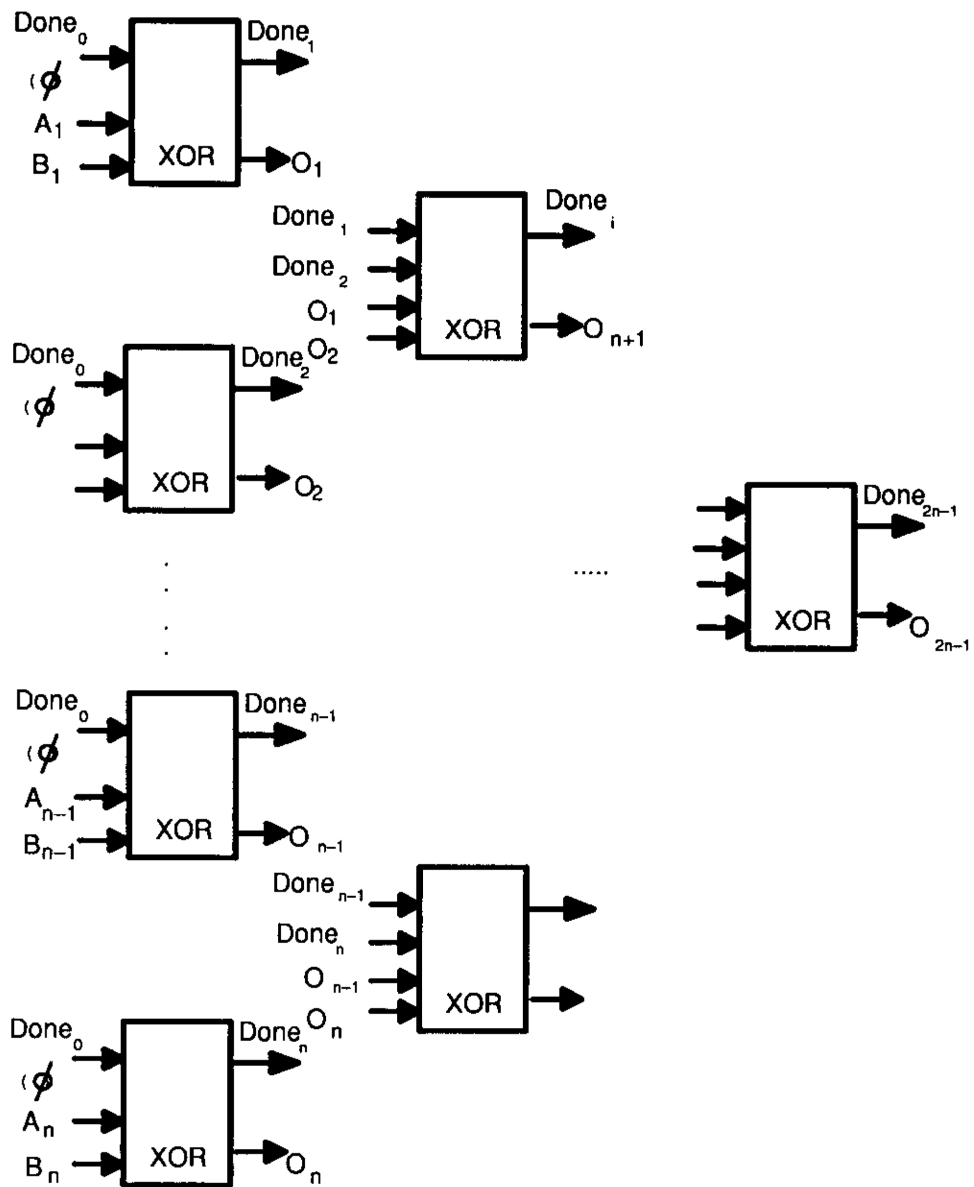


Figure 4.5 Tree Network ECDL Parity Circuit

4.4 Array Network in ECDL

ECDL circuits can also be used to implement two–dimensional networks, called array networks. The only modification necessary is on the enable/disable portion of the cell. Figure 4.6 (a) and (b) illustrates the two possible logic functions of the control signals. When the transistors are in series, all signals must be present to enable the cell logic. This gives the AND function of several events. When the enable/disable transistors are in parallel, any one of the incoming control signals may enable the cell. This provides the OR function of the event control.

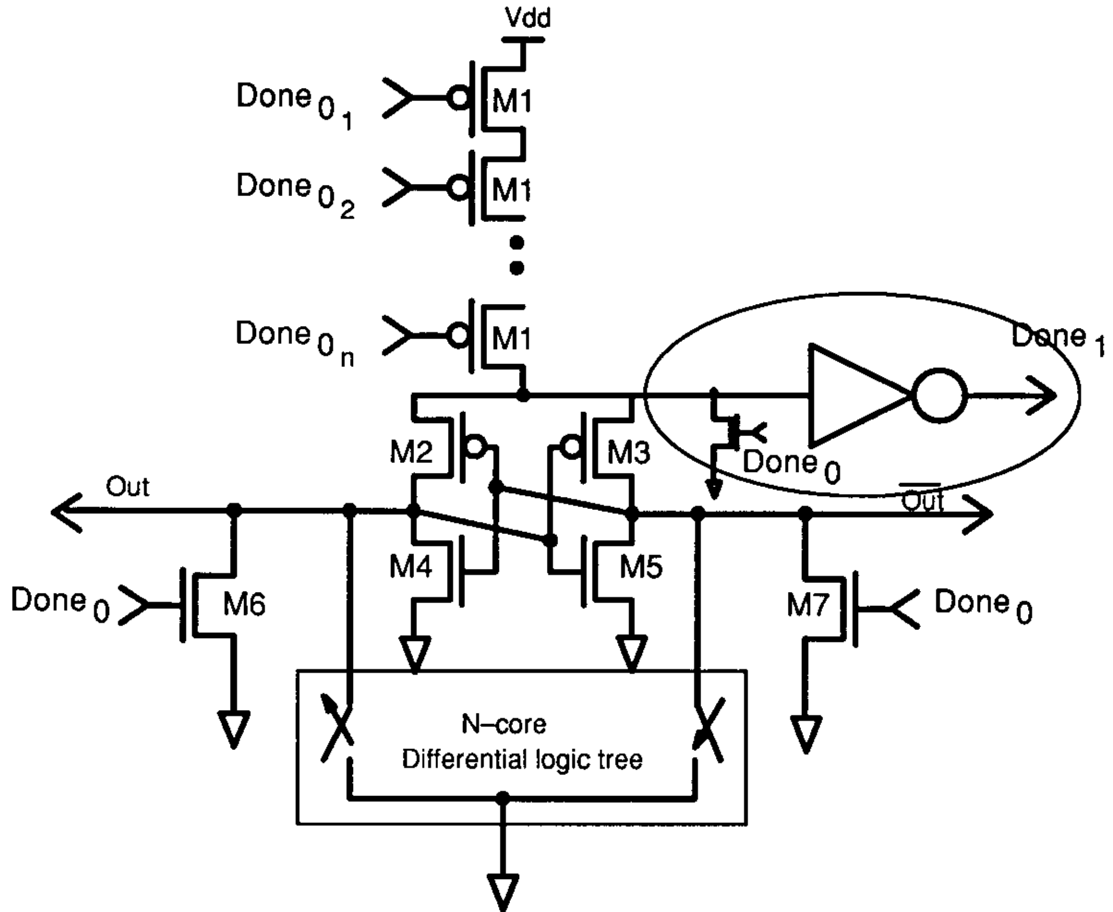


Figure 4.6 (a) AND function of events

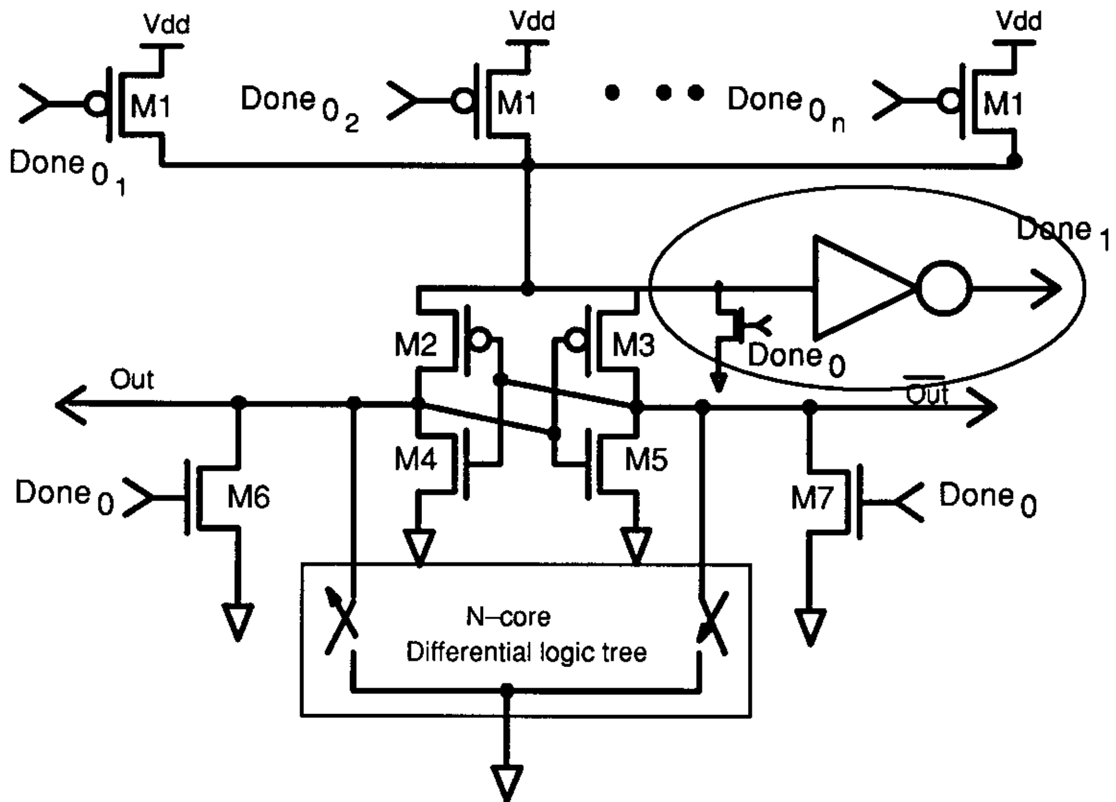
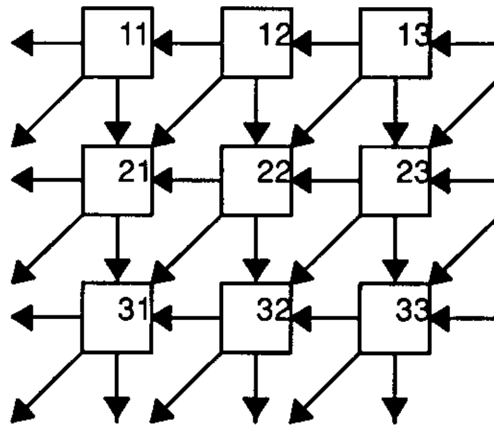


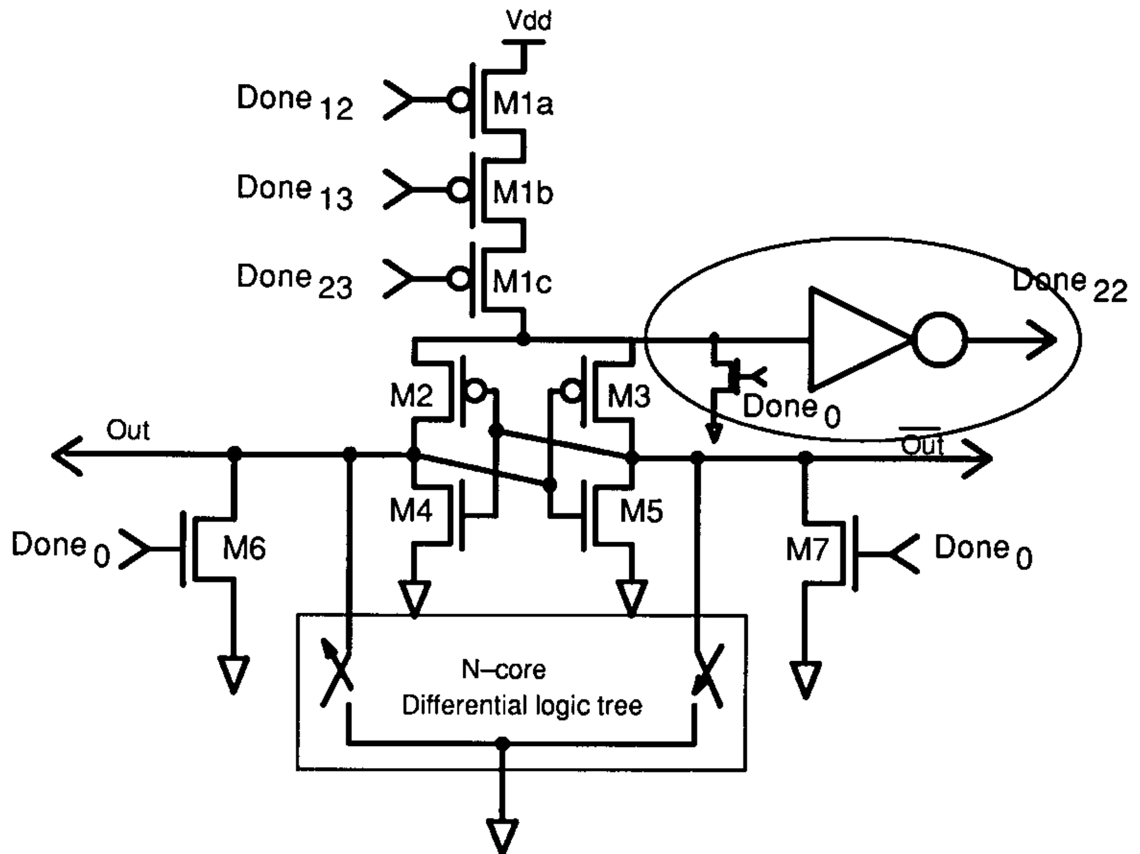
Figure 4.6 (b) OR function of events

For example, if an array network uses outputs from three of its neighboring cells' outputs as its input signals. The "done" signals from these three signals should be ANDed by using the setup as illustrated in Figure 4.6(a). Figure 4.7 shows this example graphically. With these two implementations of event functions, we can also implement other networks such as tree networks and lookahead networks.

In Chapters 6 and 7 we use self–timed array networks to implement array multipliers and dividers.



(a) An example Array Network



(b) Cell 22's ECDL Implementation

Figure 4.7 An Example of ECDL Array Network

4.5 Conclusions

Using iterative networks to implement logic functions has many advantages. First, it is simple. Only a few cells need to be designed. Second, it is regular. This reduces the interconnection and increases density. Third, they are easily tested. In this chapter we discussed the use of ECDL circuits to implement iterative one–dimension and two–dimension networks. This methodology is attractive in providing the overall speed which is lacking in the conventional implementation of iterative networks and array networks. Simple examples have been given to illustrate the methodology. In Chapter 6 and Chapter 7, we will give more examples using this methodology and analyse them more detailly. In the following chapter, we discuss how ECDL can be used to realize general computation sequencing.

Chapter 5

The Self–Timed Synchronization in ECDL

5.1 Introduction

Traditionally, the approach to designing digital computers has been as synchronous systems based on a global clock. Some of the important factors which influenced the methodology of digital system design to take this course were:

(1) Major components of a computer system, like the memory and the arithmetic logic unit were built synchronously.

(2) With global synchronization, the circuit transients do not affect the proper operation of the whole system.

(3) The step–by–step nature of the synchronous systems made it easy to design and trace the sequence of actions in the systems.

(4) The synchronous systems were favored because they required fewer gates which meant a lower cost if the systems were built from gates.

With the advancements in semiconductor technology and the increase in the system size and sophistication of computers most of the factors that influence the choice of synchronous approach have gradually been changed. Moreover, the speed of operation for a digital system on a VLSI chip has increased to a point where the signal propagation delays contributed by wires in a system have grown to the same order of magnitude of delays consumed by switching elements. In fact, these wiring propagation delays cause the same global clock pulse to represent different time instances on two distinct places of a chip. This clock skewing problem causes the synchronous approach of system design to become problematic.

At the same time the theory and methodology of asynchronous system design has been maturing. There are several reasons this approach may prove useful in designing high performance special purpose computing systems. First, asynchronous designs are algorithmic. It is easier to convert an algorithm to a wiring list for asynchronous modules than translating the algorithm into step by step procedures. Second, speed independent asynchronous modules allow the system to perform correctly. There is no need to adjust the pulse width and clock period to fit all modules timing requirements. It will avoid the clock skewing problem and allow the system to perform correctly. Third, speed is taken to be as fast as the problem or algorithm will allow. Fourth, composition of asynchronous module into asynchronous systems is readily available. Building systems hierarchically is inherent. Fifth, building each individual asynchronous modules on a single chip enable the testing and verification of each chip to be performed independently. With each module or chip verified to be functional correct, they can be assembled together again on a single chip if area permits with no extra timing constraint to satisfy. This ability to verify module independently is becoming more and more desirable since testing compass larger and larger portions of the development cycle with each passing day. Last, as Sutherland [Suth 89] points out, incremental performance gains are easier to come by.

However asynchronous systems have disadvantages also. Firstly, since some part of a system will continue to remain synchronous, interfacing between asynchronous parts and synchronous parts are both costly and slow. Secondly, asynchronous designs are very difficult to test. Since there is no required arrival time for data, it is hard to distinguish

faults from very slow arriving results. Lastly, the overhead circuitry caused by signaling convention is more than synchronous systems.

5.2 Self–Timed Signaling

The self–timed scheme is a design discipline where the sequencing of events is controlled by the internal delays of system elements rather than by an external clock. Pictorially, a synchronous systems works like a scheduled train line. At every designated interval, there will be a train taking off from the station whether there is a full load of passenger or none at all. A particular passenger has to synchronize her travel itinerary in the schedule of the train. However, an asynchronous system is like traveling with your own car. There is no fear of missing a scheduled departure time. There is no waiting in a depot for a train to arrive. You may visit a new place whenever you have finished visiting an old location. It is a sequence of events and we are interested only in the ordering them.

To accomplish the scheduling of events, a protocol discipline has to be enforced. This protocol strategy governs the proper relationship between events. In digital systems, this protocol is implemented with signaling. The most elementary signaling strategy which can be used to compose self–timed signaling conventions is a transition instead of a pulse. Seitz [Mead 80] has provided an excellent explanation on the signal convention of self–timed systems, which we summarize briefly here. The most energy efficient and least time consuming signaling scheme is called 2–cycle, or non–return–to–zero (NRZ) signaling. Figure 5.1 illustrates the working of this signaling scheme. Stabilized input signals create a "request" transition. After a certain time, depending on the delay of the element, output data become stable and an "acknowledge" transition is initiated. This transition will de–stabilize input data. When input data becomes stabilized again, a transition is made to the "request" signal. This transition de–stabilizes the output data. After a delay, output data become stable again which causes the "acknowledge" signal to have a transition. This transition de–stabilizes the input data completing the signaling scheme. We observe that there are two transitions made in one data transfer. It takes two transfers to go back to the original state. An alternative to 2–cycle signaling is named Muller signaling . It is so named because it was first proposed by Muller [Mul 59]. It is also named 4–cycle or return–to–zero signaling because there are total of four transitions made to the "request" and "acknowledge" signal in one transfer. Figure 5.2 illustrates the working of 4–cycle signaling. Control circuitry for transition signalings is necessary to perform logical combination of events besides the self–timed elements. Many works have been performed on the design of self–timed elements and their corresponding signaling control circuits. Recently, implementation of these elements and control circuits in CMOS were reported in [Suth 89] [Jacobs 88] and [Meng 89]. In the following section we present how ECDL can be used to implement self–timed elements. We also show that some of the event control circuits can be included in the same ECDL cell for an asynchronous system.

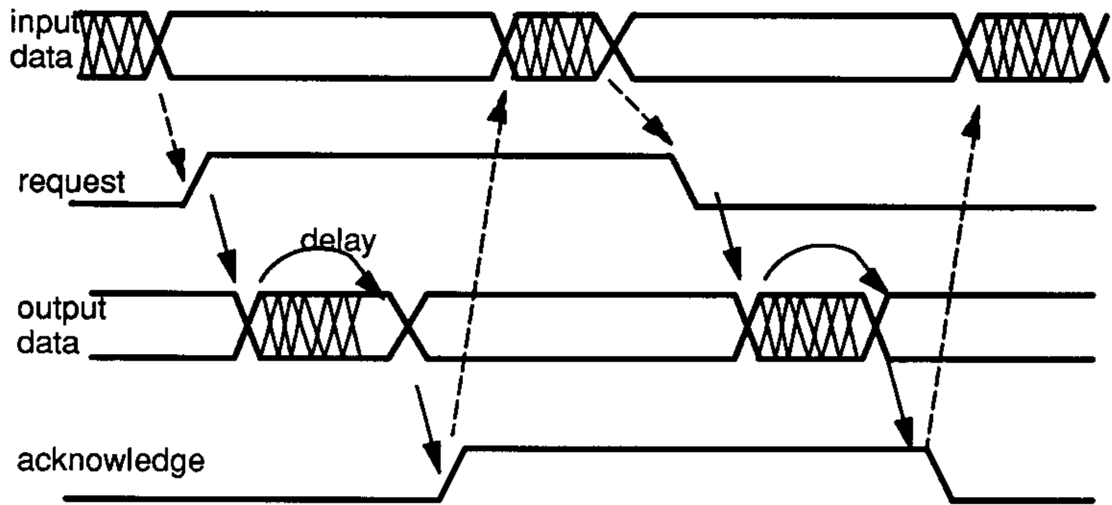


Figure 5.1 2–cycle Signaling

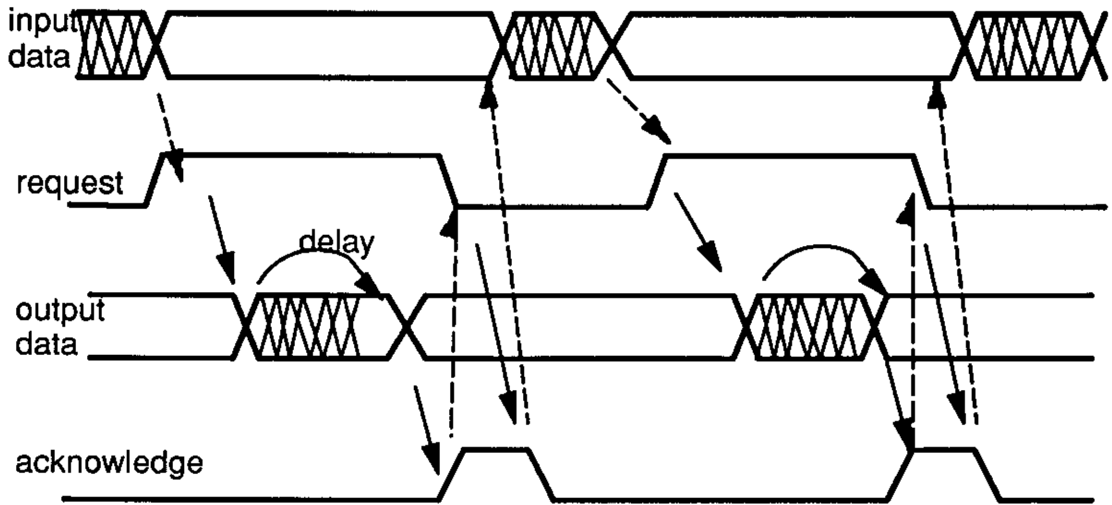
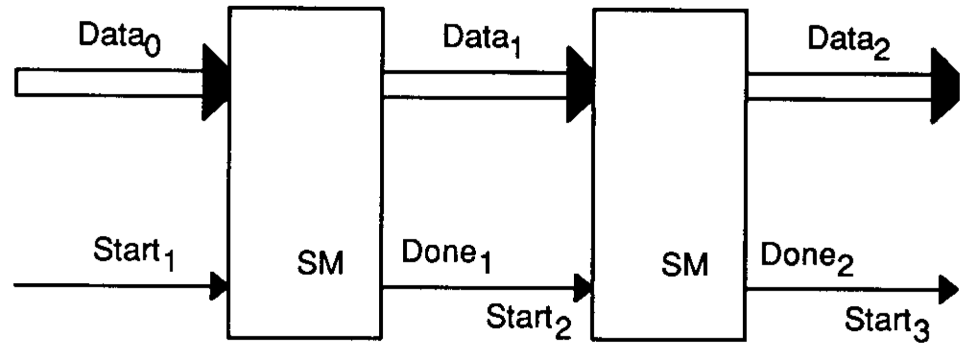


Figure 5.2 4–cycle Signaling

5.3 Using ECDL to Implement Self–Time Elements and Event Controls

Using ECDL to implement self–timed elements and control circuits has one major difference in comparison with works that have been reported. Previously, the control circuits and self–timed elements were two separate modules. Similar to synchronous system, the design of asynchronous system involves two separate paths – the control path and the data path. In an asynchronous system, control paths for transition signals requires circuits to form various combination of events. The Muller–C element is used for ANDing events. XOR gate is used for ORing events. In [Suth 89], detailed implementation of these logic combination of events have been reported. However, in ECDL, the controlling circuits are part of the self–timed elements. This is accomplished because of the unique property of ECDL's having two states – the enabled and disabled states. The signals which controls the ECDL states can also be used as the event controlling signals. As mentioned in Chapter 4 Section 5, ANDing and ORing of events can be achieved either by connecting enabling transistors of ECDL in series or in parallel. Figure 5.3 illustrates the controlling signals of two ECDL modules connected in series. As depicted, there are two controlling signals with each module, the incoming and the outgoing control signals, besides the data bus. We named the incoming control signal "$Start_i$" and the outgoing signal "$Done_i$" of the module *i*. These two signals are the $Done_i$ and $Done_{i-1}$ signals as illustrated in Figure 4.1. $Start_i$ is used to enable the ECDL module and $Done_i$ is used to tell the next module that the output is available now. We observe that $Start_{i+1}$ is equivalent to $Done_i$. It is denoted as $Start_{i+1} \Rightarrow Done_i$. Therefore, any data flow graph is a directed graph with nodes as the computation modules and directed arcs as the data connections. With every datum arc we can add a control arc.Which means any data flow graph can be implemented using ECDL. However, we have not explained how cycles in a digraph can be resolved using this signaling methodology.



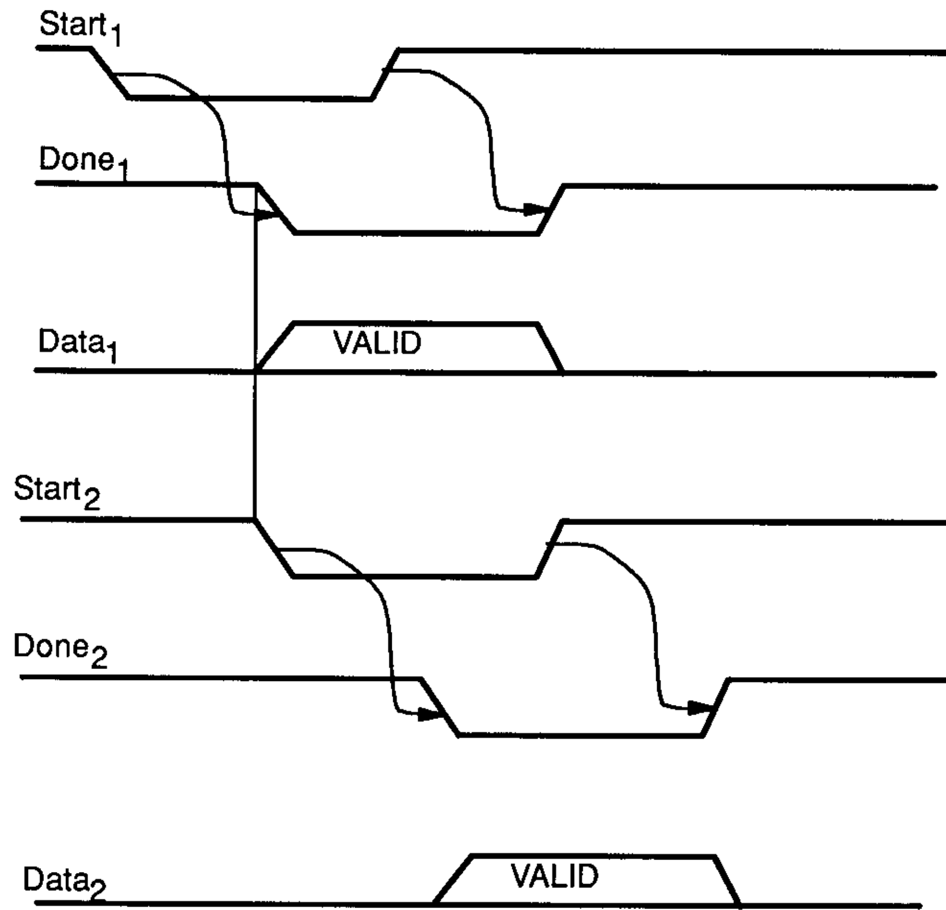


Figure 5.3 ECDL's 4–cycle Signaling of a 2 node pipeline

Let us concentrate on acyclic graphs first. In the next theorem we show that any acyclic digraph can be modified and be implemented using ECDL.

Theorem 1

Given any acyclic digraph G = (V,E), where V = { $v_1, v_2, v_3, \ldots v_n$ } are the vertices of G and E = { e_{ij} } and i,j is a directed arc from node i to node j and both i,j e { $1, 2, 3, \ldots n$ }, there is an implementation of this graph using ECDL.

Proof:

In Figure 5.3 we see that if G is a line of any length, we can implement G. If G is a tree or any general acyclic digraph, we must prove that ECDL can implement nodes with multiple inputting arcs and outputting arcs. Let us look at multiple input arcs first. If a node with multiple input arcs, this means that this node will not start its "computation" until all inputs are ready. This is clearly the ANDing function. In chapter 4 section 5 we have already shown that ANDing function of events can be implemented with ECDL. Now let us look at nodes with multiple output arcs. There are two situation during which this may happen. First, there are multiple output data signals and all of them go to more than one node. This means that there are multiple ECDL modules in the node and each module is implemented with an ECDL, where they all have the same incoming enable signals. All the outputting enable signals are ORed together to generate the "Done"signal of this node. Second, there are multiple output data lines and part of them go to one or more nodes while the other part may also go to one of more nodes. This situation can be replaced with one or more duplicated nodes. These duplicated nodes have the same "Start" signals as the original node. However these output are the part of the original output, that results in a new digraph which we have already shown that can be implemented by ECDL.

Q.E.D.

So far we have not discussed how recurrence or cycles can be implemented with ECDL. In order to implement cycles in a data flow graph, a new control signal must be introduced. This signal is named "Acknowledge" or "Ack" for short. This signal is an output controlling signal. It is used to tell the modules which are precedents of the current module that its outputs have been determined.Therefore there is no need for the proceeding modules to hold their outputs valid anymore for the current module. This signal is implemented with an inverter. The input of the inverter is the "Done" signal. The output of the inverter is the "Ack"signal. ECDL needs to be modified to accommodate this new signal "Ack". Figure 5.4 shows the modified ECDL gate with "Ack" signal added. We observed that there are two transitions in a signal. In the two–cycle signaling methodology, both transitions are "useful" in controlling events. In the four–cycle signaling methodology, any one of the transition is"useful". ECDL's signaling methodology is also an modified 4–cycle signaling methodology. Only one of the transition (edge) of the signal is "useful"in controlling events. However there exists a side effect in ECDL' signaling methodology. While only one transition is used to control the events the other transition will affect the validity of the data. This is the main reason for adding the "Acknowledge" signal. With the arrival of the "Ack" signal, the ECDL module will enter into the "disabled" state where its outputs are invalid. Figure 5.5 illustrates the timing diagram of four modules connected in pipelined fashion. If we look at the digraph it represents, we conclude that the transition of signals is really depicting the firing of tokens. Figure 5.6 (a) illustrates the initial marking of the four module pipeline. Figure 5.6 (b) illus-

trates the state where a transition has been made– or a "firing" has occurred. Figure 5.6 (c) illustrates the state where the second transition has been made. We observe that after the first transition node 1 is "not" ready to "fire" again even if a new token would arrive at its input arc because there is a token still at the output arc of the node.

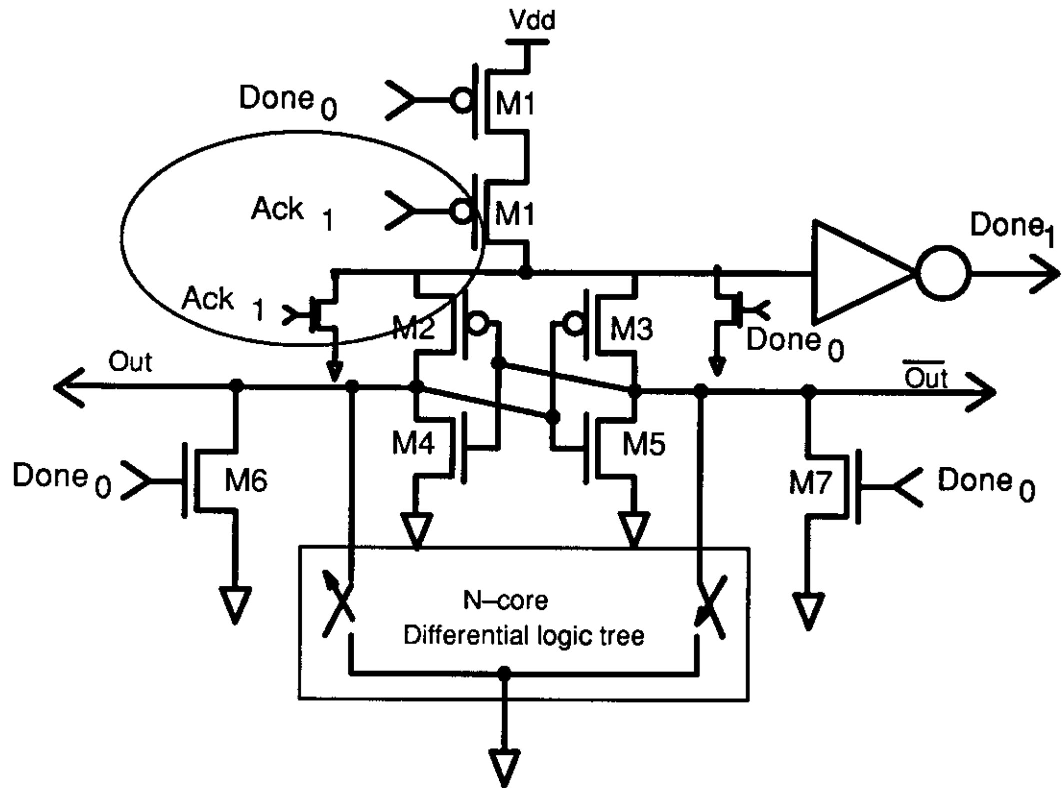
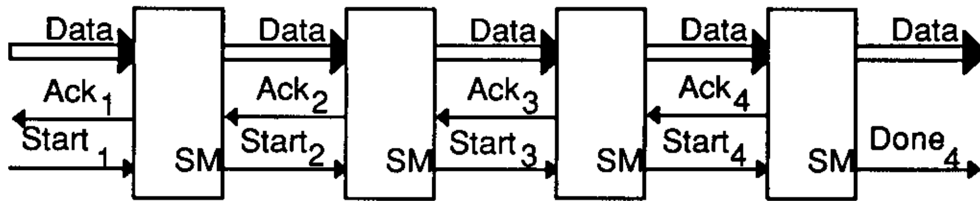


Figure 5.4 Modified N–Core ECDL Gate with Ack Signal



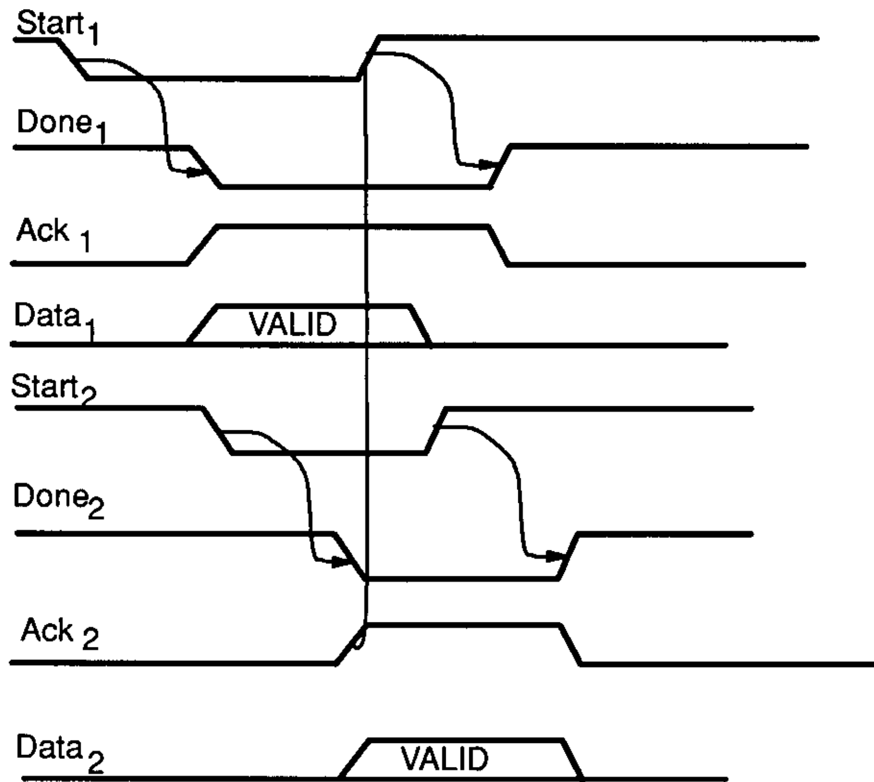


Figure 5.5 ECDL's 4–cycle Signaling of a 4 node pipeline with Ack

However, after the second transition, the first node is ready to fire again whenever a new token arrives at its input arc. Therefore there are in fact "four" states for each node in a data flow graph. These four states are depicted in Figure 5.7 (a) to (d). Only the state as in Figure 5.7 (b) is ready to "fire". In fact, the "Ack" signal of an ECDL module informs the previous node that the token does not exists on its outputting arc any more. It is free to "fire" if there should come another token which, in ECDL, means that the enabling signal has returned to the "high" state and is ready for the next "useful" transition. This leads to the following theorem.

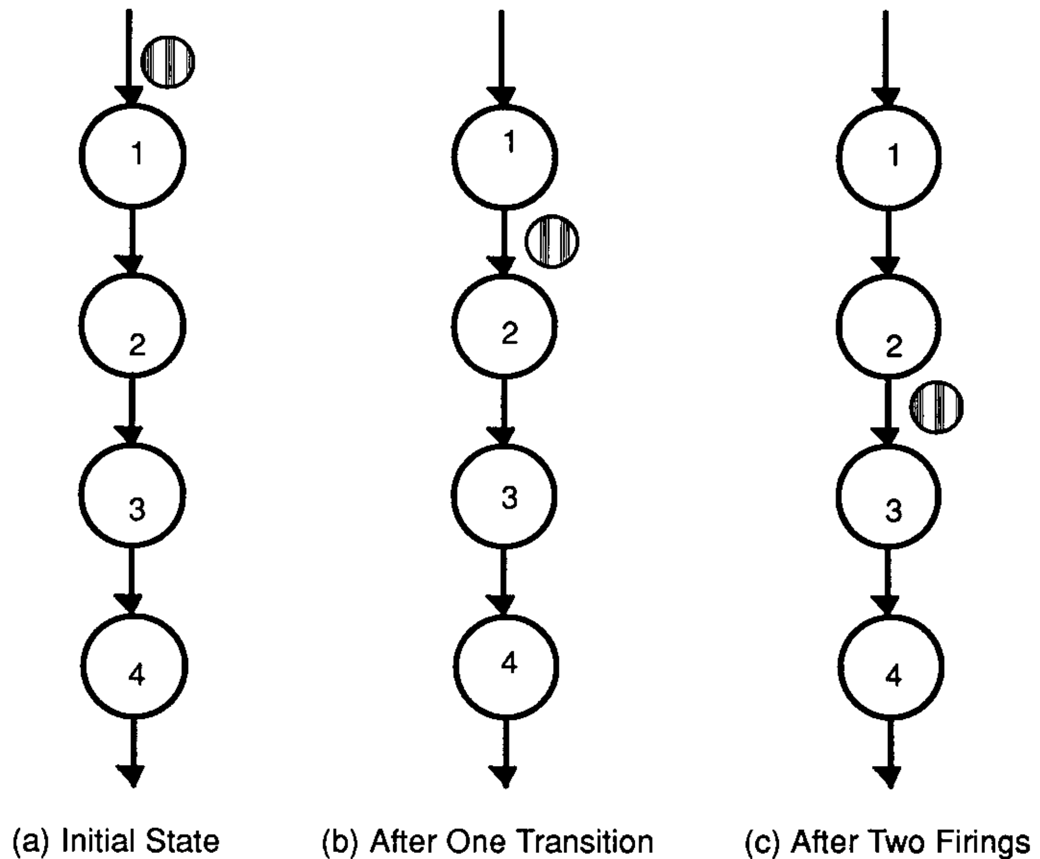


Figure 5.6 Data Flow Graphs of a Four Nodes Pipeline with Tokens

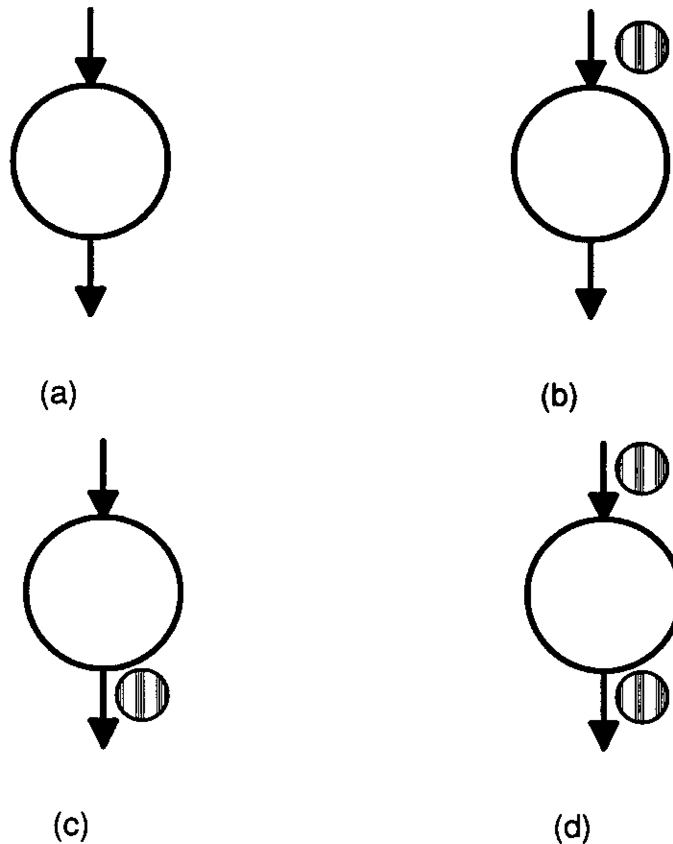


Figure 5.7 Four possible states of a node in a pipeline data flow graph

Theorem 2:

Given any digraph G = (V,E), where V = { $v_1, v_2, v_3, \ldots, v_n$ } are the vertices of G and E = { e_{ij} } and i,j is a directed arc from node i to node j and both $i,j \in \{1, 2, 3, \ldots n\}$, there is an implementation of this graph using ECDL.

Proof:

This proof is similar to the proof of theorem 1. In Figure 5.5 we see that a pipeline of any length can be implemented. If there is a node with multiple inputting arcs, then with each inputting arc we add a back arc representing the "Ack" signal. Whenever the current multiple inputting arc node has finished its computing, all proceeding nodes do not have to hold its output anymore. Now let us look at node with multiple output arcs. There are two situations during which this may happen. First, there are multiple output data signals and all of them go to more than one nodes. With each fanout of output there will be an incoming "Ack" signal.

This node will have to hold its output until all its successors have completed its computation. Therefore we need an AND gate to make sure the last arrival of "Ack" triggers the "disabling" effect. Second, there are multiple output data lines and parts of them go to one or more nodes which the other parts may also go to one of more nodes. This situation can be replaced with one or more duplicated nodes. These duplicated nodes will have its successors to generate a back arc representing the "Ack" of nodes connected to its outputting arc. This again results in a new digraph which we have already shown can be implemented by ECDL.

Q.E.D.

5.4 Using ECDL to Implement Bundled Data Convention

ECDL can also be used with the two–phase bundle data convention proposed by Sutherland [Suth 89]. In fact, by using ECDL to implement the logic used by the micropipeline with processing, the delay element can be eliminated. Instead, an additional XNOR gate and an edge trigger flip/flop is used to warrant the requirement that the control signal must arrive after the data processing is completed. Figure 5.8 illustrates the design of a stage of a micropipeline with processing using ECDL. The registers used in Figure 5.8 is the same as ones used in [Suth 89] which have two control ports (C and P). Cd and Pd are "delayed signals of C and P caused by the inputloads of the registers. Figure 5.9 shows the timing diagram for nodes (including internal nodes u, w, y, and z) denoted in Figure 5.8. Since the ECDL logic provides a completion signal, there is no need to calibrate the delay element to ensure the timing requirement is satisfied. This elimination of delay elements from the original micropipeline will help to ease the capability of auto–synthesis of self–timed designs using this bundled–data convention.

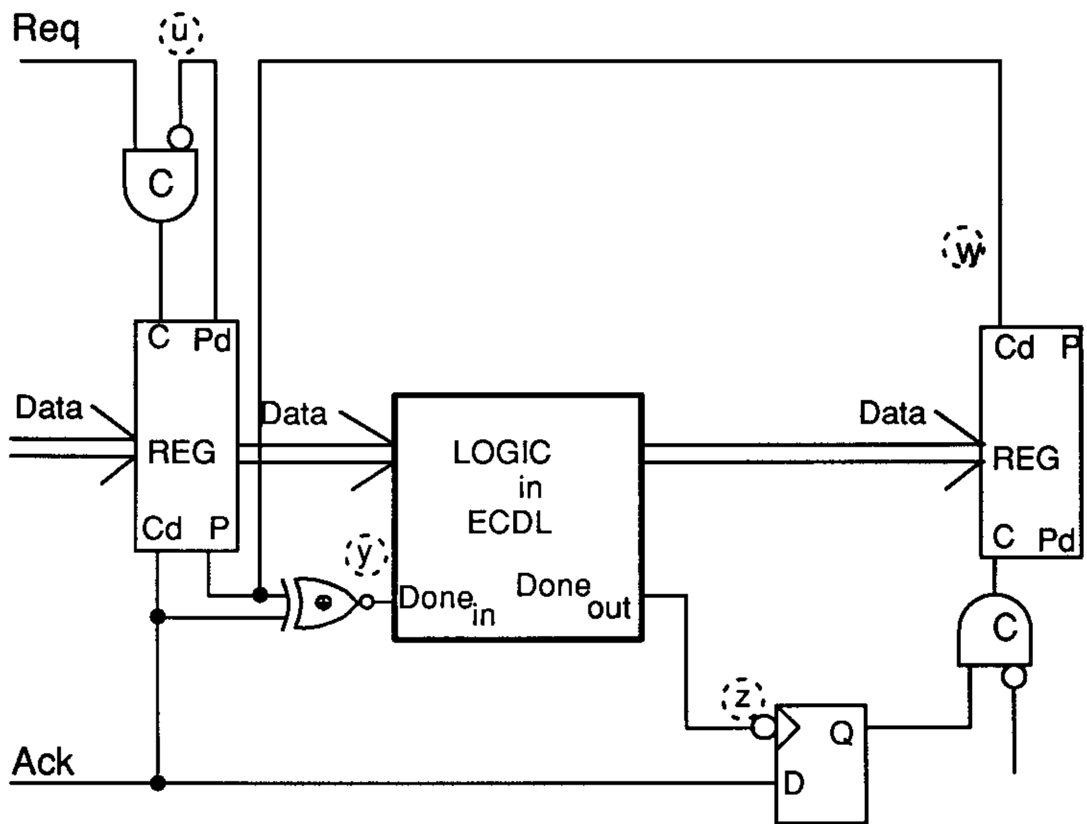


Figure 5.8 2–phase Bundled Data using ECDL

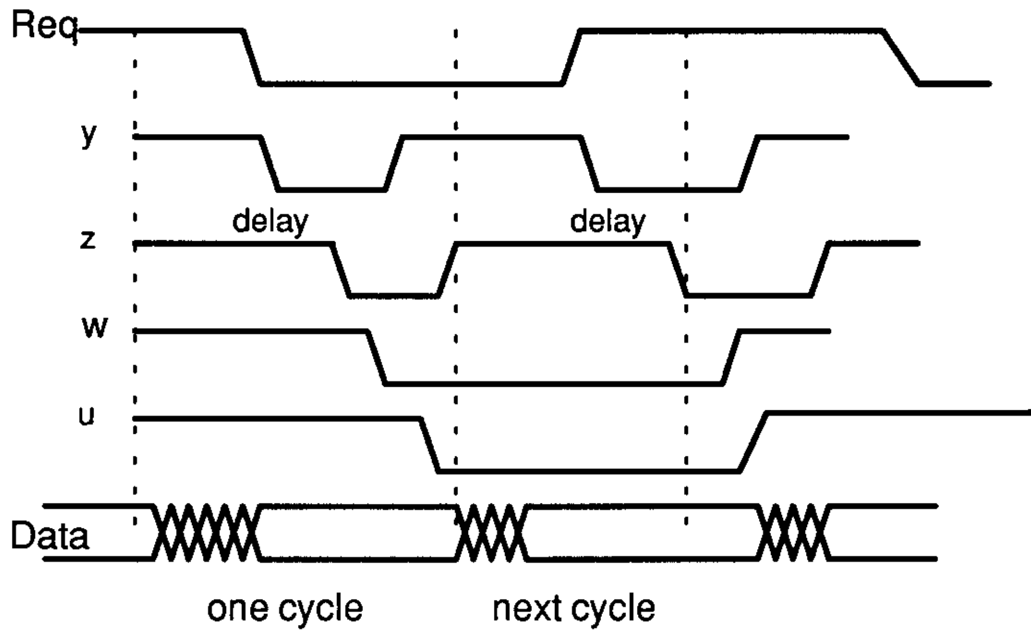


Figure 5.9 Timing Diagram for the Circuit in Fig. 5.8

5.5 Conclusion

ECDL provides a way to implement a data flow graph. Each node of the data flow graph can be a gate, an ALU or even a full processor. Control structures which previously required additional hardware circuits can now be implemented within the self–timed elements using ECDL. In the next two chapter, several arithmetic modules are implemented using ECDL. These modules can serve as a node of a dataflow graph as discussed in this chapter.

Chapter 6

Design and Implementation of Two–Operand Adders

6.1 Introduction

Many arithmetic algorithms base their operation on adders. As a result, performance of a processor depends heavily on the speed of adders. There are various way to implement adders in a particular technology. There are as many algorithms to achieve two–operand addition. These algorithms, implemented with the same technology and logic family, give different performance results and cost. Traditional evaluation methodology uses a model based on a gate level methodology independent of the technology used [Skl 60], for which gate count and delay are key factors. With the advancement in technology and the innovation in logic methods, this model is no longer sufficient to reflect the true measure of merits associated with each addition algorithm. Guyot et al. [Guy 87] showed how to build efficient carry–skip adders in VLSI context. Wei et al. [Wei 85] presented a systematic approach in constructing a fast parallel adder, again in the context of VLSI technology. Both works based their designs on the conventional CMOS logic family.

In this chapter we will utilize the proposed CMOS logic methodology – ECDL to implement several two–operand addition algorithms on silicon. These adders have the self–timed characteristic, discussed in the previous chapter where a completion signal is generated at the completion of the addition. An analytical modeling method is formulated to predict the performance and cost of each algorithms implemented in ECDL. This modeling strategy is comparable to Chan and Schlag's work [Cha 89] where a RC timing model is used to determine optimal block sizes for carry paths. Here a current (or effective impedance) and capacitance is used to predict the performance of adders in ECDL logic. This evaluation method provides feedback to designers in using ECDL to realize different algorithms into logic. Actual measurements of the implementation are used to compare the model and SPICE simulation results.

6.2 Modeling Methodology

6.2.1 Delay Model

In [Cha 89] an analytical method was used to evaluate dynamic CMOS Manchester adders with variable carry–skip length. The evaluation is based on the RC timing model. In this section a current–capacitance model is used to evaluate different adders implemented in ECDL. The model is based on the equations:

$$\frac{\delta Q}{\delta t} = I \qquad \text{and} \qquad CV = Q$$

From the above equations we can estimate the time required to charge up a capacitor to a voltage V to be:

$$\Delta t = \frac{CV}{I}$$

In the following section we will explain in more detail the model used in the chapter. In chapter 3 we discussed the basic operation principles of the ECDL. In Figure 3.7, we observe that in a self–timed ECDL gate there are three major capacitance loads. One of the loads contributes the delay of the control signal, the other two loads are associated with the output and its complement. The intrinsic delay (without external loading) of an ECDL gate is the sum of three delays: T_1 – the time needed to switch the enabling transistor, T_2 – the time needed to create a difference between the output and its complement, and T_3 – the rise time needed to bring one of the output to logic 1. Assuming that the current charging the capacitive loads is linear and a linear function of the effective size, the the total delay can be approximated by:

$$T = T_1 + T_2 + T_3 = (2x + 2yz + 8xy)\frac{C}{I} = (x + xyz + 4xy)\,\tau$$

where $x = number\ of\ P-devices\ in\ series\ for\ enabling$
$y = number\ of\ nodes\ at\ output\ contributed\ by\ the\ N-network\ tree$
$z = number\ of\ N-devices\ in\ series\ in\ the\ N-network\ tree$
$\tau = intrinsic\ switching\ time\ of\ a\ min.\ sized\ N-device$ (6.1)

In section 4 of this chapter, the result of measurements is compared with this model.

6.2.2 Area model

Instead of counting the number of 2–input gates, the total number of transistors required to implement a ECDL cell plus the *wiring equivalent number* is used as the measurement for area. This equivalent wiring number is determined by the number of non–local signals needed for a particular cell. The summation of all non–local signals gives the equivalent wiring number.

6.3 ECDL Adders

In this section we discuss design, implementation and relative performance of several n–bit adders – all realized with the proposed ECDL self–timed circuit family. The adders include a carry–ripple adder (as a reference design), a carry–completion sensing adder[Gil 55], a fixed group carry–skip adder[Leh 61], and a carry–lookahead adder[Wei 56].

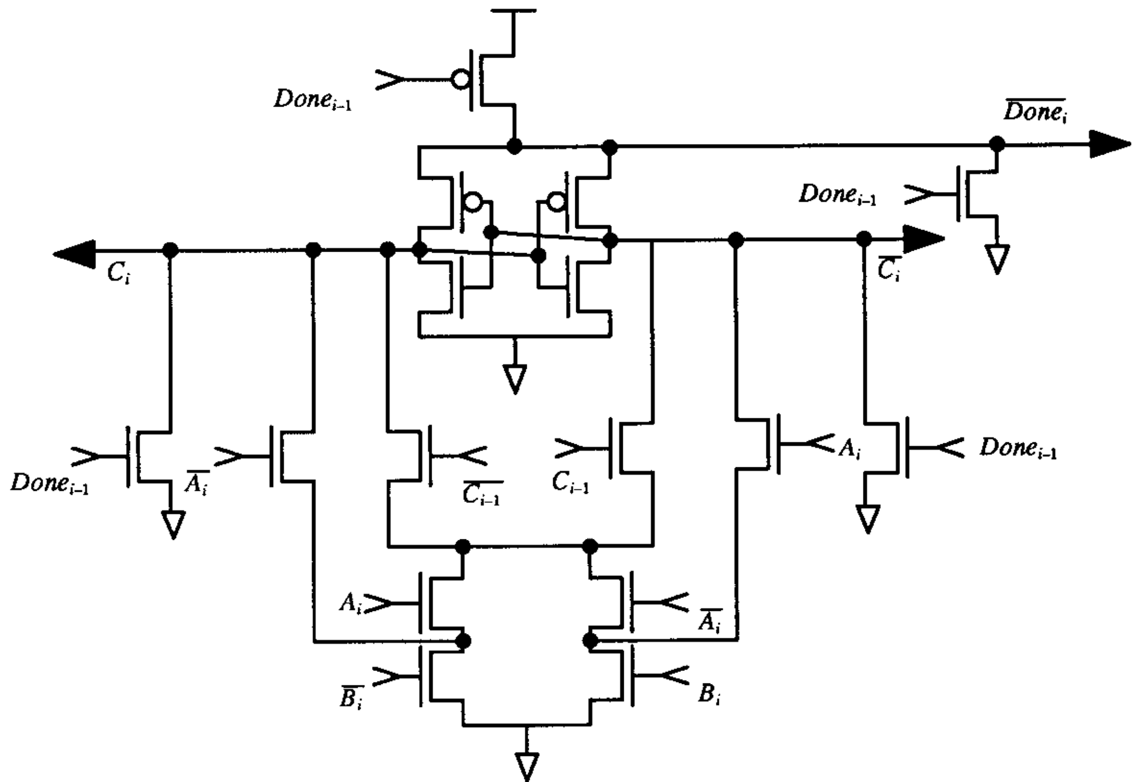
6.3.1 Carry–Ripple Adder (CRA)

One bit full adder is implemented with ECDL as shown in Figure 6.1. A full adder has three data inputs *(A, B, C)* and two data outputs – *(S, C)*. In addition, there are two self–timing control signals: one input control and one output control signal, which are used to control the propagation of results. The logic equation of a full adder used as the *i*th bit of a ripple adder can be expressed as:

$$
\begin{aligned}
S_i &= (A_iB_i + \overline{A_i}\overline{B_i})C_{i-1} + (\overline{A_i}B_i + A_i\overline{B_i})\overline{C_{i-1}} \\
C_i &= A_iB_i + (\overline{A_i}B_i + A_i\overline{B_i})C_{i-1}
\end{aligned}
\qquad (6.2)
$$

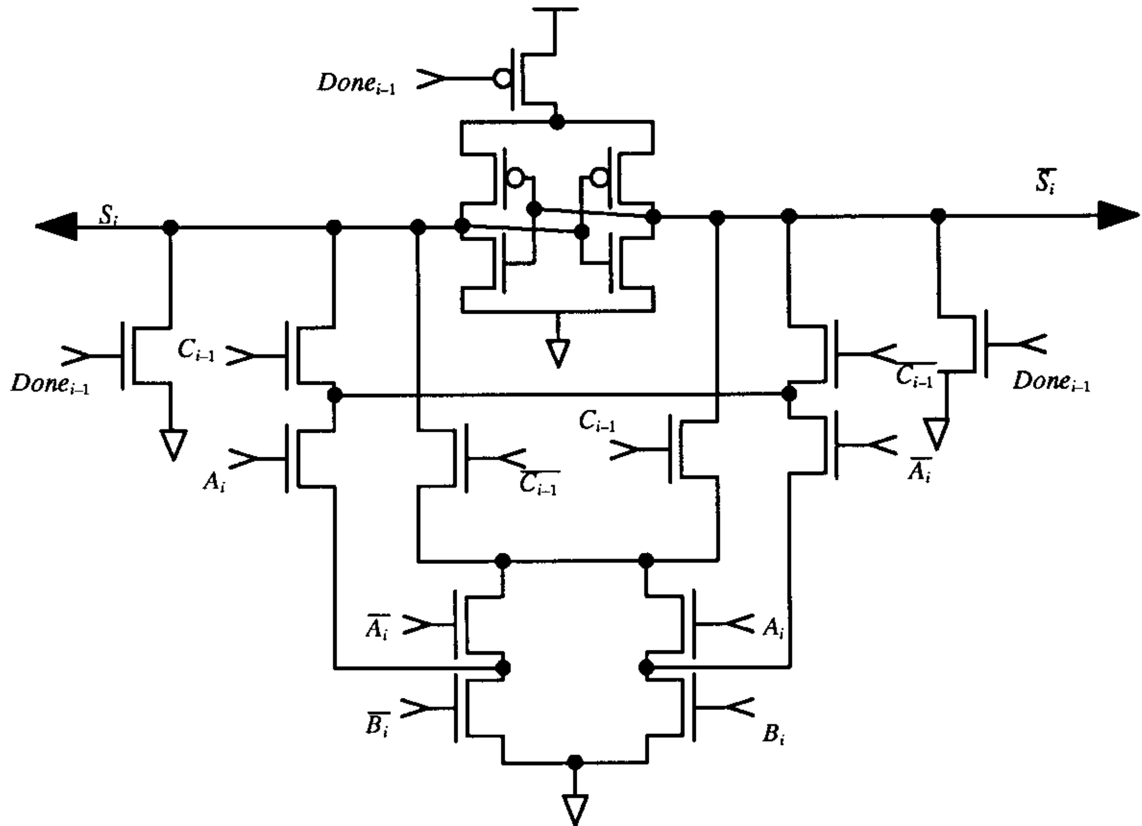
In ECDL both the function and its complement are needed since ECDL, like other differential circuits, is a double–ended logic. The above logic equation is chosen to maximize the sharing of terms. The complements of the two outputs are expressed as:

$$
\begin{aligned}
\overline{S_i} &= (A_iB_i + \overline{A_i}\overline{B_i})\overline{C_{i-1}} + (\overline{A_i}B_i + A_i\overline{B_i})C_{i-1} \\
\overline{C_i} &= \overline{A_i}\overline{B_i} + (\overline{A_i}B_{i+}A_i\overline{B_i})\overline{C_{i-1}}
\end{aligned}
\qquad (6.3)
$$



(a) The Carry Cell

Figure 6.1 The ECDL Implementation of a Full Adder



(b) The Sum Cell

Figure 6.1 The ECDL Implementation of a Full Adder

Using expressions (6.2) and (6.3), partial terms are shared when implemented with the differential network tree. In [Chu 86], two methods are presented for realizing optimal differential network tree of a particular logic function. Because of this optimization and sharing of terms, a function often uses comparable number of devices when implemented with ECDL in comparison with the same function implemented with conventional static CMOS logic. After minimization and sharing of terms, our ECDL implementation of a full–adder uses 35 devices including the circuit overhead for self–timed control. However only 7 of the 35 are P–channel devices, which are usually twice the size of an n–channel device. In comparison, the conventional CMOS implementation has 15 of the total 30 transistors being p–channel devices. The ECDL CRA adder has been implemented with MOSIS' 3 micron CMOS process

[Tom 88] [Lu 90]. Please refer back to Figure 2.18 which tabulates the comparison of various implementation of carry–ripple adders in terms of total number of devices, average per bit delay and dynamic power dissipation. For each logic approach, an 8–bit carry–ripple adder is implemented and layouted. Figure 6.2 gives the layout of an 8–bit ECDL CRA. This layout is extracted and simulated using SPICE with level 2 parameters from a typical MOSIS' 3mm process. The delay is obtained from the addition of the input vectors (1 1111111) and (00000000) having first bit carry input equal to a step function at time zero. The dynamic power dissipation was obtained by SPICE as proposed by Kang [Kang 86]. By using the model mentioned in the previous section, the total delay of an n–bit CRA is:

$$T_{CRA} = \sum_{i=1}^{n} \Delta t1_i \quad \leq n \; \Delta t1$$

$$\textit{where } \Delta t1_i \textit{ is the actual dealy of stage } i$$

$$\text{and } \Delta t1 = \max(\Delta t1_i) = (1 + 7 \;\; 2)\tau \cdot = 15\tau \qquad (6.4)$$

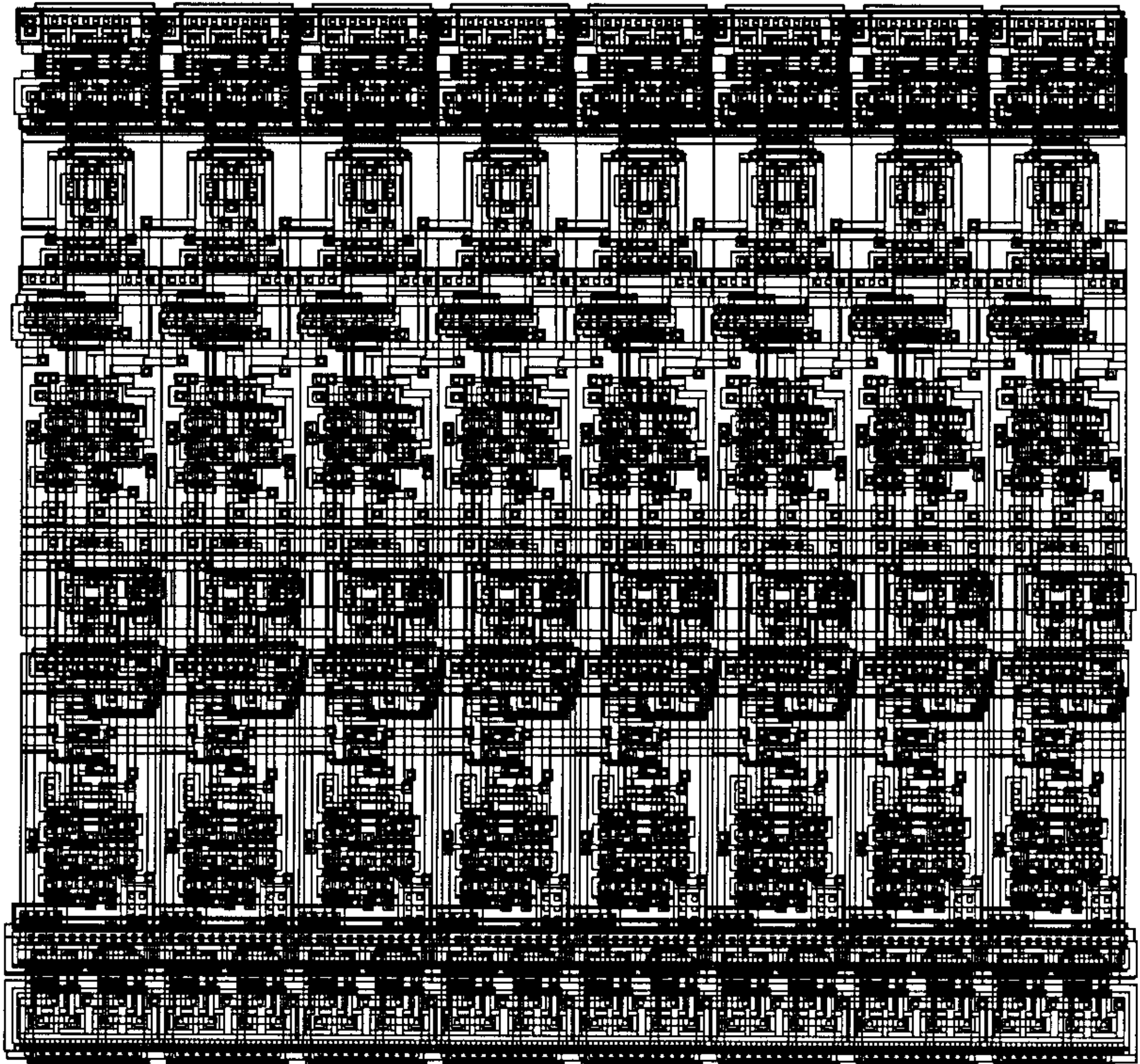


Figure 6.2 Layout of an 8-bit ECDL CRA

6.3.2 Carry–Completion–Sensing Adder (CCA)

Speeding up the carry chain propagation time in an adder has been a central problem in the design of high–speed arithmetic units. In the carry–ripple adder discussed above the carry propagation time is linearly proportional to the number of bits of the operands. Although the design is simple, the speed is unacceptable for addition of long operands. However, the addition time, on the average, is proportional to the length of the longest carry propagation chain if the adder is asynchronous. In 1946, Burks, Goldstine and von Neumann [Bur 46] showed in a classic analysis that if two binary numbers to be added are evenly distributed in its range, an upper bound of the expected longest carry chain is $log_2(n)$ for an n–bit adder. Reitwiesner [Rei 60] derived an algorithm to determine the longest zero or non–zero carry chain length. Briley [Bri 73] further tightened the bound suggested in [Bur 46]. All three works assume the operands are evenly distributed in the range. The implementation of this adder in ECDL is basically the same as the simple ripple adder except that the completion logic is modified so that it will exploit a smaller average delay time. This modification allows the *Done* signal to be generated whenever the two inputs are both zero or both one. An additional logic block is required to detect the presence of completion signals from all stages. This is realized with an n–input NAND gate. The inputs of this NAND are the completion signals of each bit without the inverters. Figure 6.2 illustrates the carry part of the ECDL carry–completion sensing adder. Notice that there are maximum of three P–channel devices in series instead of one as in a ripple adder carry cell. The sum cell for carry–completion sensing adder is the same as in the ripple adder. The ANDing unit used as the completion sensor can be implemented with conventional static CMOS or with an extra level of ECDL logic. If n becomes large so that an n–input NAND is unrealizable with a single gate, a tree of NAND gates can be used to realize the ANDing unit. As a result, the final "done" signal remains high until all the carries have been enabled. In effect the final "done" signal corresponds to the slowest individual "done" signal. The total number of transistors per bit in a carry–completion sensing adder is 41 of which 13 are p–channels, assuming the static CMOS NAND is used. Although on average the total delay is $(log_2 n)$, the worst case delay for an n–bit carry–completion sensing adder is still proportional to the total carry chain length. In a fixed–clock system this adder does not provide any advantage since we still need the clock period to accommodate the longest delay. However in a self–timed system, there is no need to wait for the longest delay and, on average, the system will run faster. Assuming that each stage has an actual delay of $\Delta t2_i$ and the completion circuit has a delay of T_{cc}, and let $\Delta t2 \triangleq \max(\Delta t2_i)$, the worst case delay of this carry–completion sensing adder is:

$$T_{CCA}(total) = \sum_{i=1}^{n} \Delta t2_i + T_{cc} < n\Delta t2 + T_{cc} \tag{6.5}$$

while the average delay is:

$$T_{CCA}(average) = \sum_{i=1}^{\log_2 n} \Delta t2_i + T_{cc} < (\log_2 n)\Delta t2 + T_{cc} \tag{6.6}$$

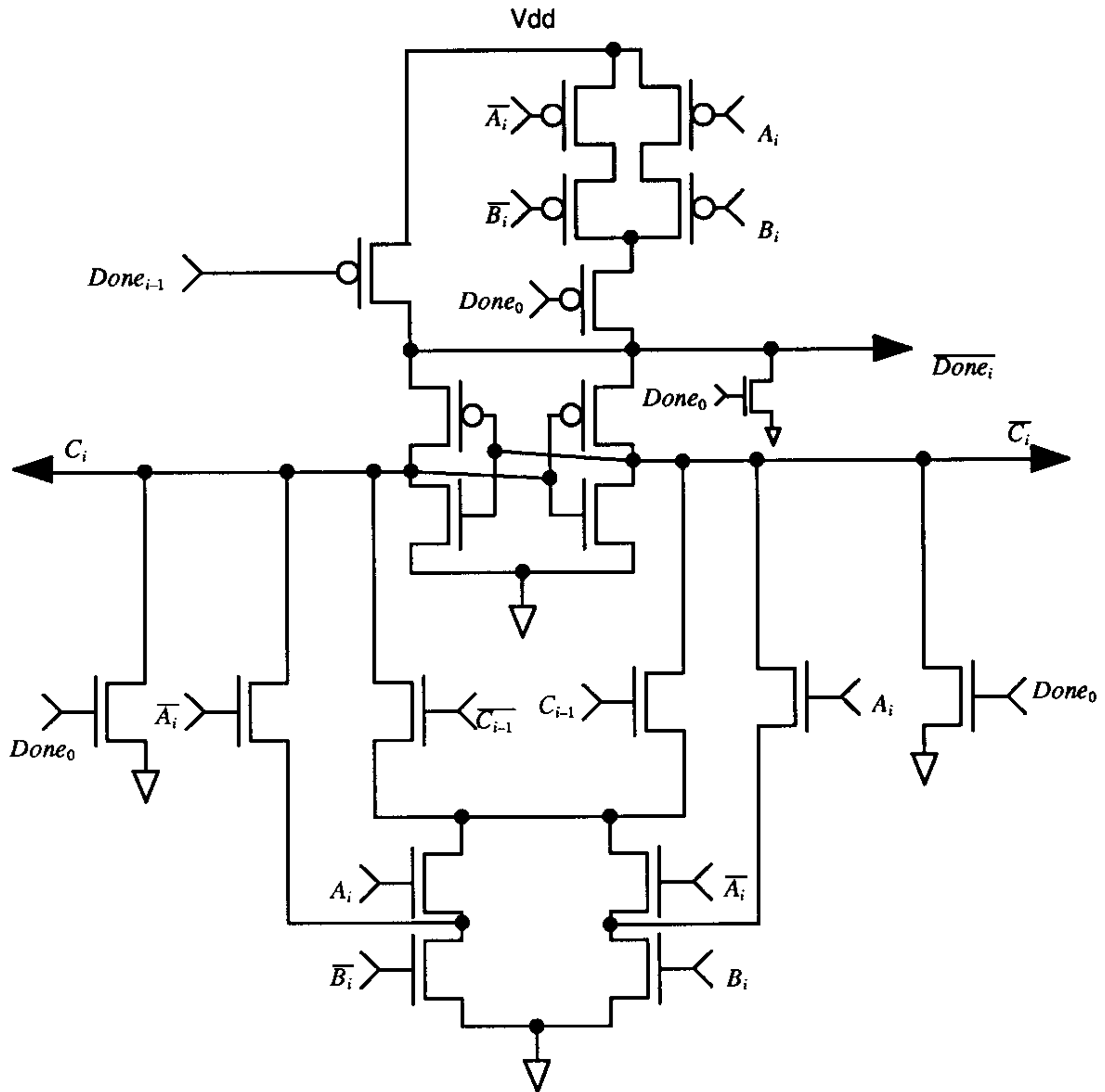


Figure 6.3 The ECDL Design of the Carry Cell for a Carry–Completion–Sensing Adder

If all transistors are of the same size, then the current in the carry–completion sensing adder used to charge and discharge is smaller than the current of a carry–ripple adder. With the same loading capacitance value we can expect a longer delay per stage for the carry–completion sensing adder than $Dt1$. From the delay model we expect the per–stage delay of the car-

ry–ripple adder to be:

$$\Delta t2 = 3 + (3 \cdot 2 \cdot 3 + 4 \cdot 2)\tau = 29\tau \simeq 2\Delta t1 \tag{6.7}$$

That is, in the worst case this adder is expected to require about twice the time of the carry–ripple adder. However, on the average this adder will be faster than carry–ripple adder when n is larger than 8. A layout of an 8–bit carry–completion sensing adder in MOSIS' SCMOS rule has area of 433 lambda by 485 lambda. It is shown in Figure 6.3. This is an 18% increase in the area compared with the ECDL carry–ripple adder. It consumes almost the same dynamic power as the ECDL CRA.

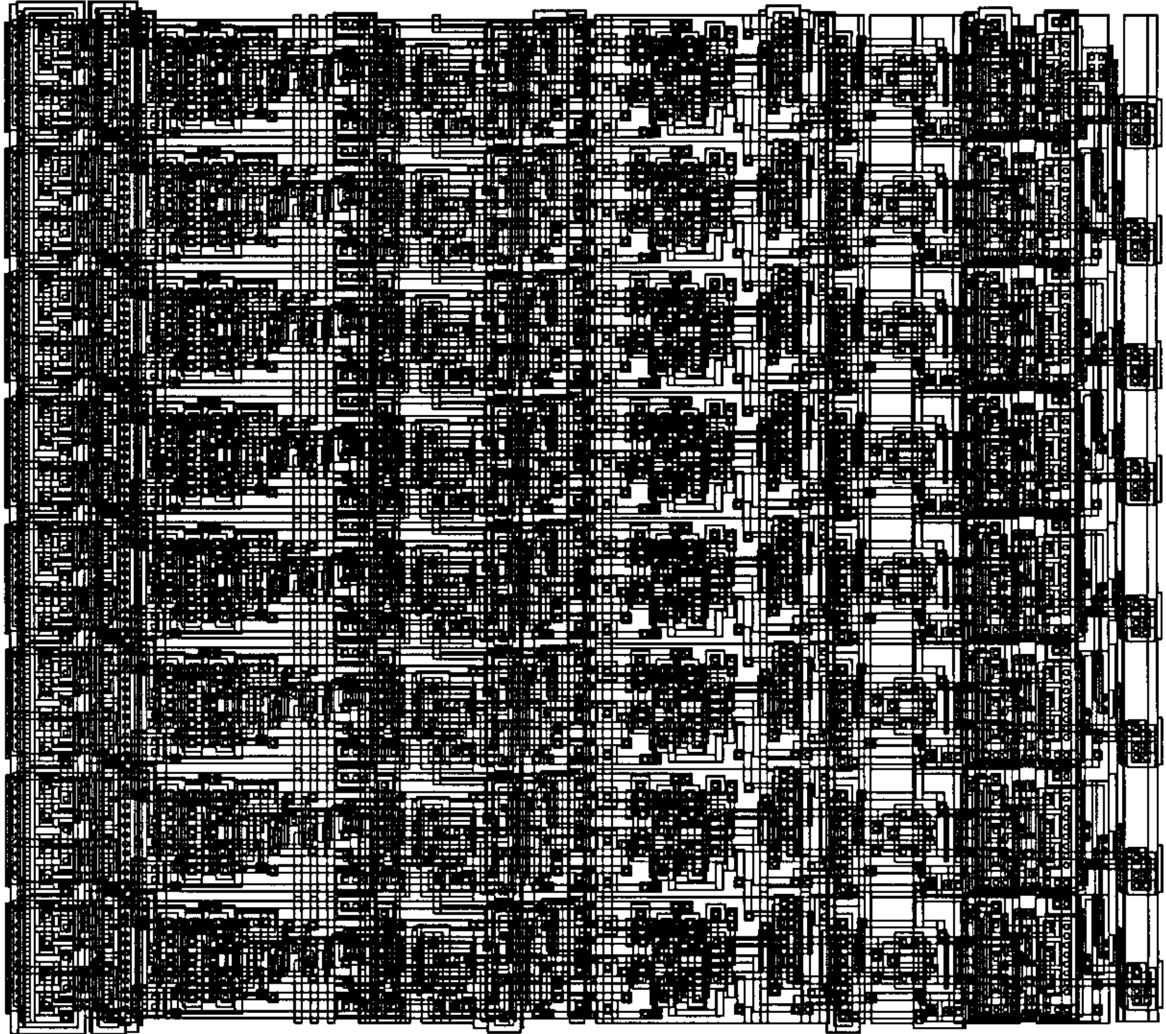


Figure 6.4 Layout of an 8-bit ECDL CCA

6.3.3 Carry–Skip Adder (CSA)

The previous adder only minimized the average delay. A simple but efficient way to speed up the carry propagation is by using the carry skip technique [Leh 61]. We divide an n–bit adder into n/k groups of k–bits. For each group, there is a carry bypass circuit besides the k–bit ripple carry path. The final carry for a group is obtained as follows:

$$P_i = A_i + B_i$$

$$\mathbf{P} = \prod_{i=1}^{k} P_i$$

$$C_k = \mathbf{P} \quad C_0 + \quad A_k B_k + P_k C_{k-1} \tag{6.8}$$

Lehman and Burla [Leh 61] derived the optimal group size which gives the least number of gate delays for $k = \sqrt{n}$. Majerski [Maj 67] proposed to include groups of different sizes to further reduce the worst–case gate delay by reducing the number of skips. These two approaches are optimized with the basic uniform delays assuming the ratio between the skip time and the ripple time is 1. Barnes and Oklobdzija [Bar 85] considered the case when cells are implemented with VLSI technology and relaxed the ratio to be integers in the range of two to seven inclusively. Guyot et al [Guy 87] further extended it to include ratios of rational numbers. Further improvements in the design method of carry–skip adders are discussed by Chan and Schlag [Cha 90]. The later three works all conclude that carry–skip adders are a good compromise between ripple–carry adders and carry look ahead adders with respect to area and delay. While the ripple adder is simple, easy to implement but rather slow, the carry–look ahead adder requires large silicon area and is much harder to design and implement. The carry–skip adder, however, gives good performance with a minimum area penalty. To implement a carry–skip adder in ECDL, we designed a special carry cell at the k–th bit of each group, assuming a fixed size group. Even with variable sized group the last bit's carry part can still be modified. This special carry cell only depends on the initial carry and all groups inputs, as illustrated in Figure 6.5. The first k–1 bits of the group consist of the carry–ripple adder discussed in the previous section. A block diagram showing how cells are assembled to form a k–bit group is given in Figure 6.6. The final completion signal is generated from the k–th bit of the SUM cell belonging to the last group of an n–bit adder. The total area of an 8–bit carry skip adder which contains two $k=4$ carry skip groups is measured at 467 lambda x 473 lambda. Layout is shown in Figure 6.7. This is only a 5% increase in comparison with an 8–bit carry–completion sensing adder. The total number of transistors in this 8–bit adder with 4–bit fixed sized groups is 308 of which 28 are P–channel. In general, the number of transistors for a k–bit group of an ECDL carry skip adder is:

$$39k - 2 \tag{6.9}$$

where 7k are P–channel devices. Assuming that a P–channel device is twice the size of an N–channel device, the equivalent transistor count for a k–bit group is:

$$46k - 2 \tag{6.10}$$

Since for every k bits, the carry cell needs $2k$ non–local inputs and for every non–local input a space must be reserved from the starting point of the signal, the total equivalent wiring number is:

$$\frac{n}{k}\sum_{i=1}^{k} 2i = n(k+1) \tag{6.11}$$

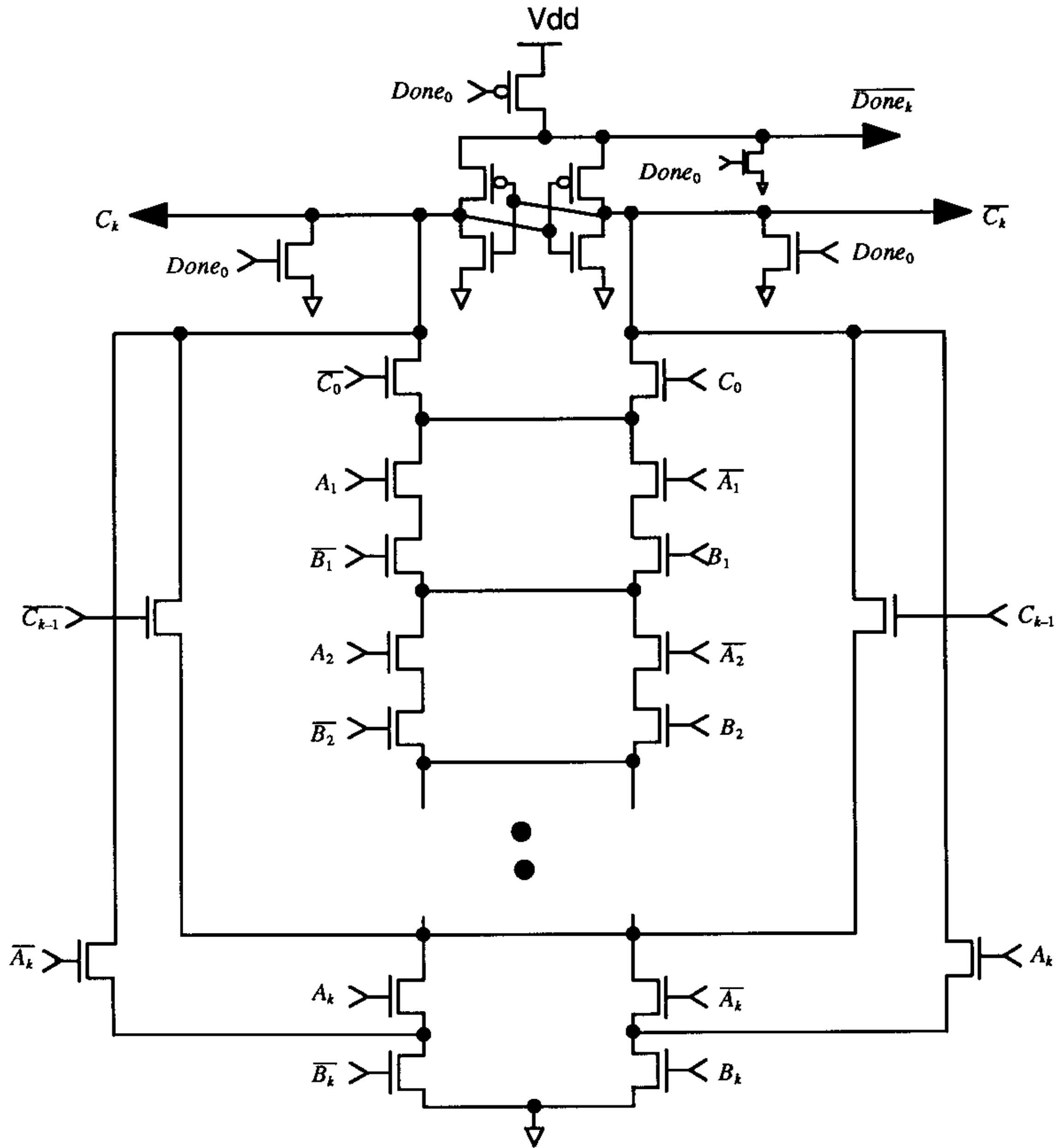


Figure 6.5 The Kth Bit Carry Cell of a ECDL Carry–Skip Adder

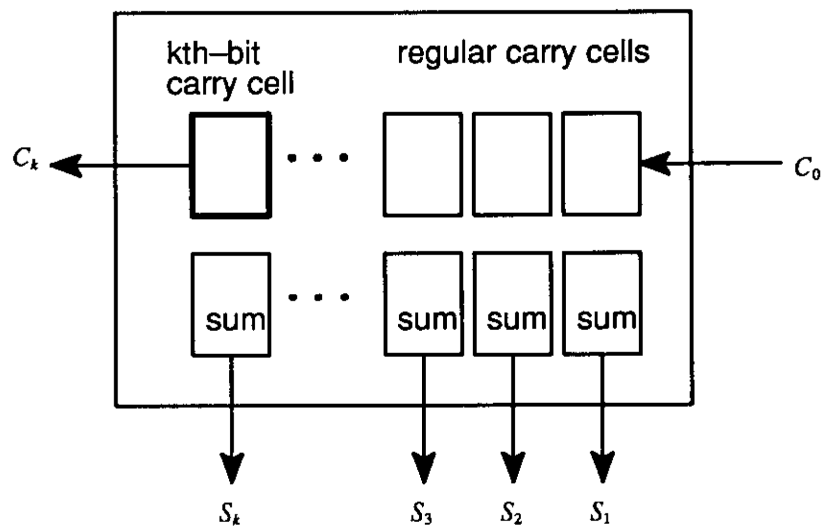


Figure 6.6 (a) The Block Diagram of a k–bit ECDL Carry–Skip Adder

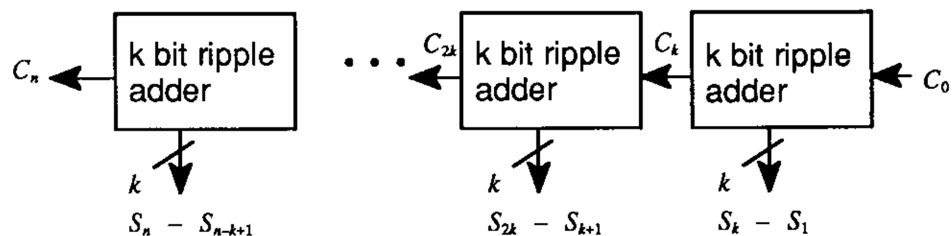


Figure 6.6 (b) The Block Diagram of an n–bit ECDL Carry–Skip Adder with Fixed k–bit size grouping

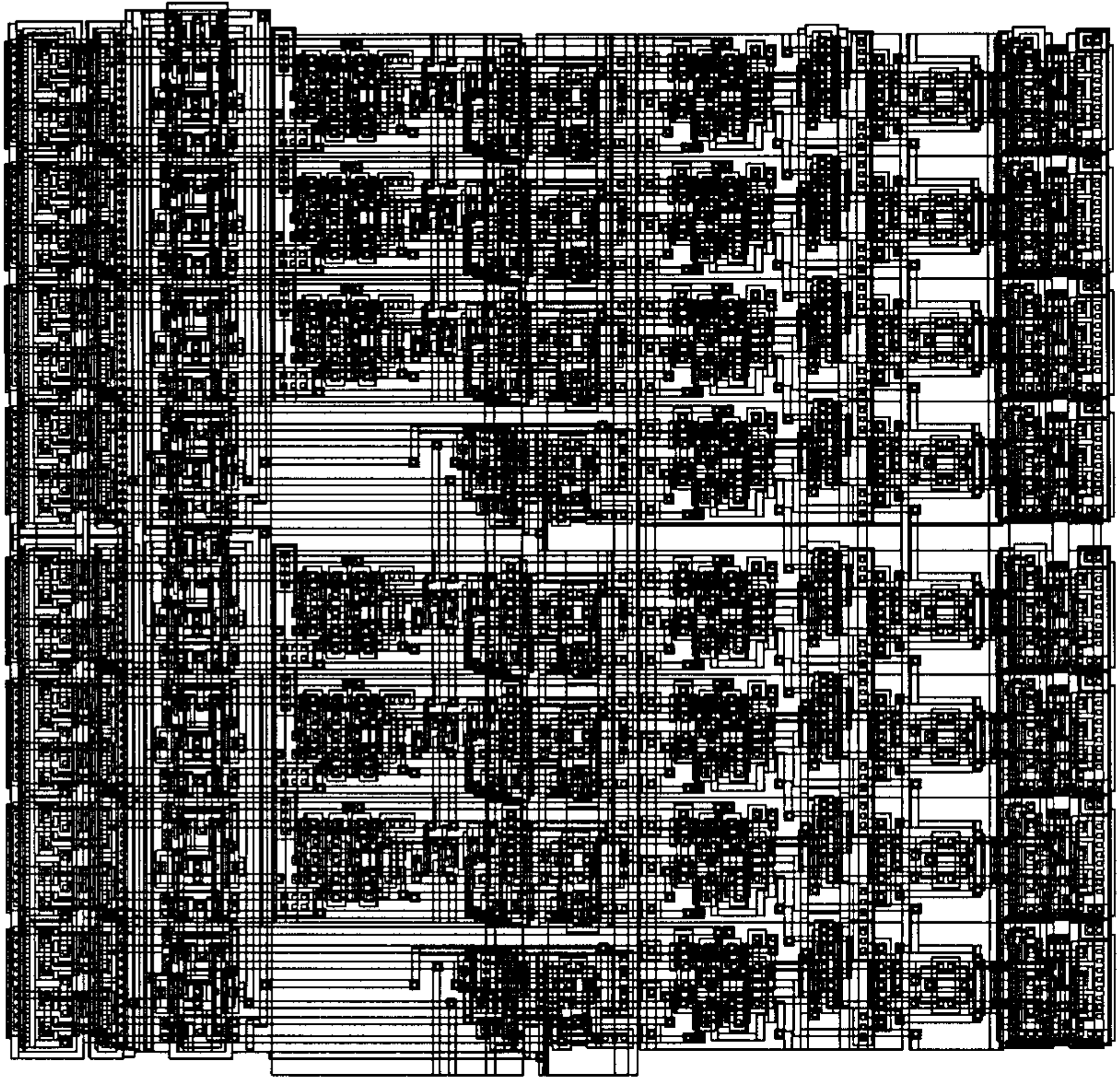


Figure 6.7 Layout of an 8-bit ECDL CSA

From (6.10) and (6.11) we obtain the model of the total area for an n–bit ECDL CSA with fixed k–bit group as:

$$Total\ Area = (47+k)n-2(n/k) \quad (6.12)$$

Let D$t3$ be the delay of the ECDL cell that generates the skip carry. The worst case delay for a fixed size group carry–skip adder is:

$$T_{CSA} = \sum_{i=1}^{k} \Delta t1_i \qquad \textit{(carry–ripple in group 1)}$$

$$+ \sum_{i=2}^{\frac{n}{k}-1} \Delta t3_i \qquad \textit{(carry bypass)}$$

$$+ \sum_{i=n-k+1}^{n} \Delta t1_i \qquad \textit{(carry–ripple in group n/k)} \quad (6.13)$$

The current used to charge and to discharge the output depends on the number of P–channel and N–channel devices in series. Therefore, there is no difference between the simple carry cell and the k–th bit carry cell of the k–bit group. Assume the majority of the load capacitance is contributed by diffusion area, we observe that the total number of diffusion area for the k–bit carry is always two more than that of the regular carry adder. In addition the N–channel tree logic of the carry cell which generates the skip carry is lar ger than the regular carry cell's N–channel tree network. Therefore, the number of N–channel devices in series for a k–th carry cell is more than the number of the N–channel devices in series for a simple carry cell. That is, the n–channel transistors in series is $(2k+1)$ instead of 3. This gives a way to estimate D$t3$ as:

$$\Delta t3 = (1 + 1 \cdot 3 \ \ (2k+1) + 4 \ \ 3)\tau \approx (\frac{k}{3} + 1)\Delta t1 \quad (6.14)$$

Substituting (6.14) into (6.13) we have:

$$T_{CSA} = (\frac{4k}{3} + \frac{n}{k} + \frac{n}{3} - 2)\Delta t1 \quad (6.15)$$

To obtain optimal delay for a fixed size ECDL carry–skip adder, we should choose:

$$k_{opt} = \frac{\sqrt{3n}}{2} \quad (6.16)$$

6.3.4 Carry–Lookahead Adder (CLA)

The limited speed of a carry–ripple adder is a consequence of the carry–out dependence on the carry–in signal. To reduce this dependency, a logic relation may be established to have all carries depend solely on the primary inputs. Given two *n*–bit binary numbers A and B:

$$A = (A_n A_{n-1} A_{n-2} \ \ldots\ldots \ A_2 A_1)_2$$

$$B = (B_n B_{n-1} B_{n-2} \ \ldots\ldots \ B_2 B_1)_2$$

Define two auxilliary local functions:

$$G_i = A_i \ B_i \qquad \textit{(generate)}$$
$$P_i = A_i + B_i \qquad \textit{(propagate)}$$

From G and P express all carries in an n–bit addition as:

$$\begin{aligned} C_1 &= G_1 + P_1 \ C_0 \\ C_2 &= G_2 + P_2 \ C_1 = G_2 + P_2(G_1 + P_1 \ C_0) \\ &\vdots \\ C_i &= G_i + P_i \ C_{i-1} \\ &\vdots \\ C_n &= G_n + P_n \ C_{n-1} = G_n + P_n \ G_{n-1} + \\ &\qquad P_n \ P_{n-1} \ P_{n-2} \ G_{n-2} + \ldots + P_n \ P_{n-1} \ldots P_2 \ P_1 \ C_0 \end{aligned} \tag{6.17}$$

This, in principle, can produces all carries with three logic gate levels assuming no restrictions on fan–in, fan–out and the number of gates. Clearly, as *n* increases, two problems arise. One is the number of logic gates. To implement the *n*–th carry C_n as given by (6.17) with only two–input AND and OR gates requires $n(n+1)/2$ of two–input AND gates and *n* two–input OR gates excluding gates used to generate *Pi* and *Gi*. The other problem is the fanout for *Pi* and *Gi*. An alternative is to build carry look ahead modules and then assemble them to form larger adders. In contrast, using ECDL to implement CLA do not have these two problems. The equation (6.17) is implemented with N–channel logic and there is no theoretical limitation of the number of transistors in series. Therefore, all carries can be generated in one stage delay. However, the offset voltage is not zero, and in fact it can be as high as 20 mV. When the number of transistors in series is too large, the differential voltage created is not large enough for the flip–flop (sense amplifier) to set. This causes the ECDL cell to give a wrong result. Figure 6.8 gives the *k*–th bit carry cell implemented in ECDL. Other carry cells are implemented similiarly. The typical delay to generate C_k is $\Delta t4$. Our model gives approximately this delay as:

$$\Delta t4 \cong (1 + (2k+1) \ (k+1) + 4 \ (k+1))\tau = (2k^2 + 7k + 6)\tau \tag{6.18}$$

Therefore, the total delay of an n–bit ECDL CLA implemented with k–bit modules is:

$$T_{CLA} = \left(\frac{n}{k}\right)\frac{2k^2 + 7k + 6}{15} \quad . Dtl \tag{6.19}$$

This adder takes more area than other adders discussed above. An 8–bit adder uses an area of 672 lambda by 479 lambda which is an 81% increase in comparison with the simple ECDL carry–ripple adder. Layout is shown in Figure 6.9. A fixed group 8–bit carry–look ahead adder using two 4–bit carry–look ahead adders is implemented. To compare with other ECDL adders discussed, we determine the total number of N–type transistors:

$$\sum_{i=1}^{k}(6i + 23) = 6(k^2 + k) + 20k = 6k^2 + 29k \tag{6.20}$$

The total area cost of an n–bit ECDL CLA contributed by wiring is approximated by:

$$\left(\frac{2n}{k}\right)\sum_{i+1}^{k} i^2 = \frac{2nk(k+1)(2k+1)}{6k} = \frac{n(2k^2 + 3k + 1)}{3} \tag{6.21}$$

Therefore, the estimated area for an n–bit ECDL CLA with k–bit modules is:

$$\frac{n(2k^2 + 21k + 88)}{3} \tag{6.22}$$

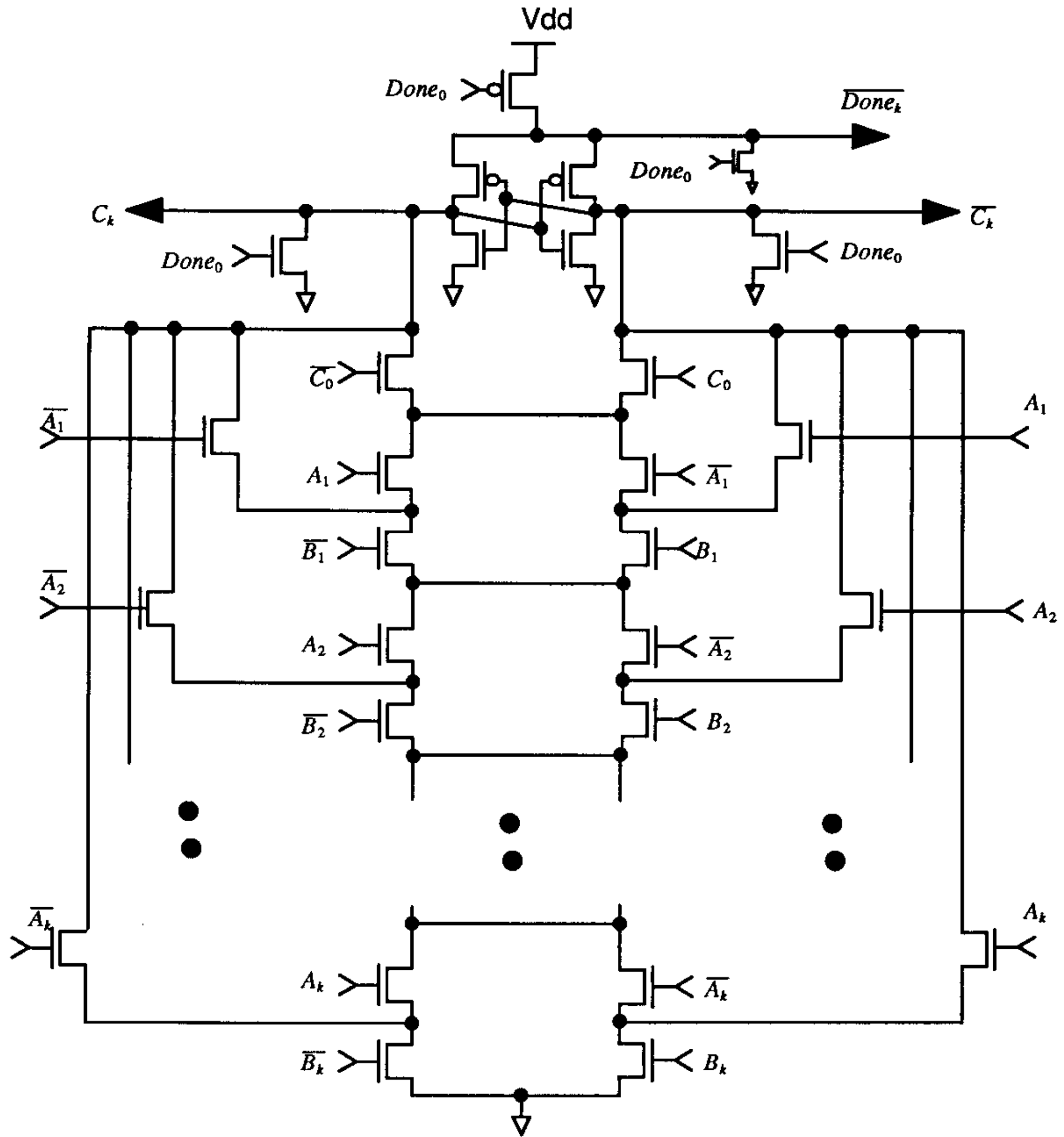


Figure 6.8 The *i*th bit Carry Cell of an ECDL Carry–Lookahead Adder

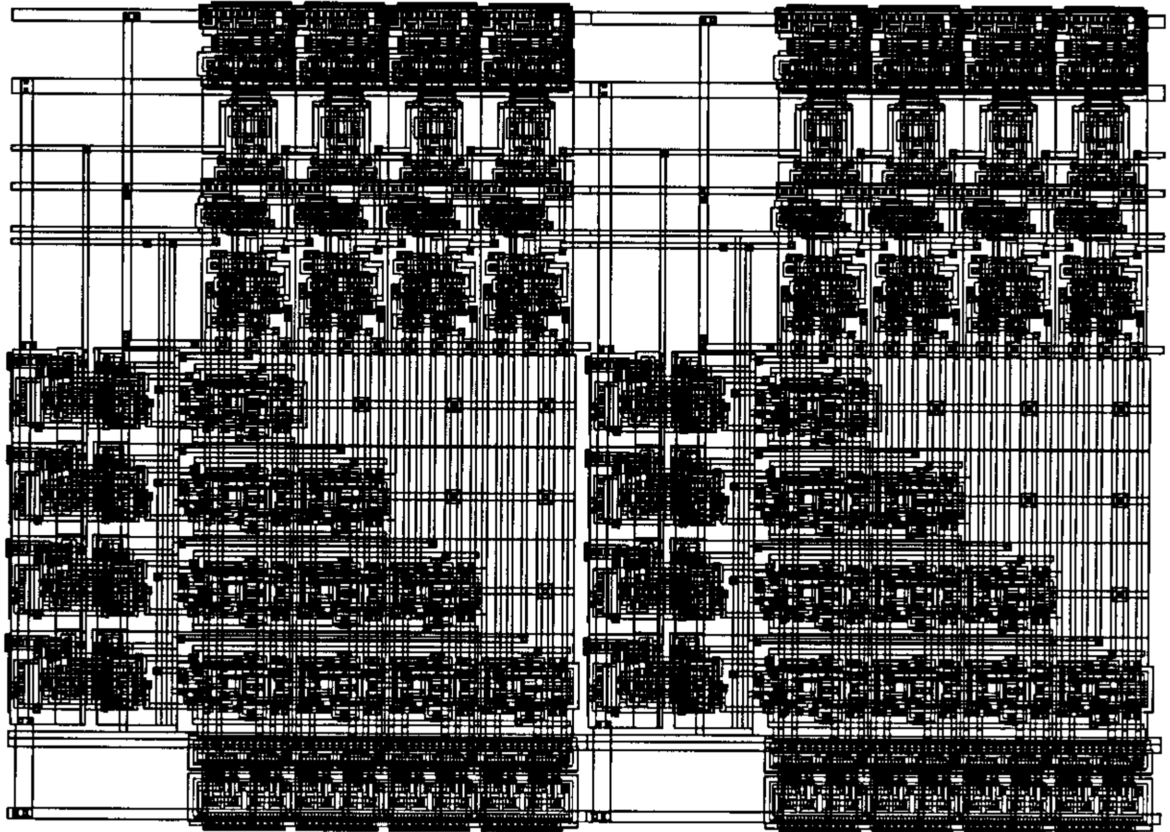


Figure 6.9 Layout of an 8-bit ECDL CLA

6.4. Measurement and Comparison

6.4.1 Delay Comparison

Four 8–bit adders were implemented and fabricated using MOSIS' 3 micron Tiny chip service. Results from measurements are shown in Figure 6.10. All measurements use the worst case input combinations of A = (11111111), B = (00000000) and $C_0 = 1$, except for the carry–skip adder which has inputs (11111111) and (00000001). SPICE simulation from extracted layouts is summarized in Figure 6.11. Simulated results include delays of different feature sizes. Since the design is done in MOSIS Scalable rules, the simulation results are obtained by setting the layout extractor to a proper value of lambda and by using a different set of SPICE MODEL parameters for each lambda value. In Figure 6.12, the delays predicted by the proposed model in the previous section are calculated and plotted against the measured values and SPICE simulation results for 3 m feature size. Except for the carry completion sensing adder, results of delays are predictable. The reason for a large difference between the measured result and simulation for CCA is that the size of the P–channel transistors are not minimized sized. They have been scaled to match the CRA adder design.

6.4.2 Area Comparison

All four adders are implemented with MOSIS' scalable CMOS rules. The total area measured does not include pad drivers. All units are in lambdas. Table 6.1 gives the tabulated results, while Figure 6.13 gives the normalized results in comparison with what the model predicts. As expected, the ECDL carry lookahead adder consumes much more space than other ECDL adders. However the model predicts the area fairly well.

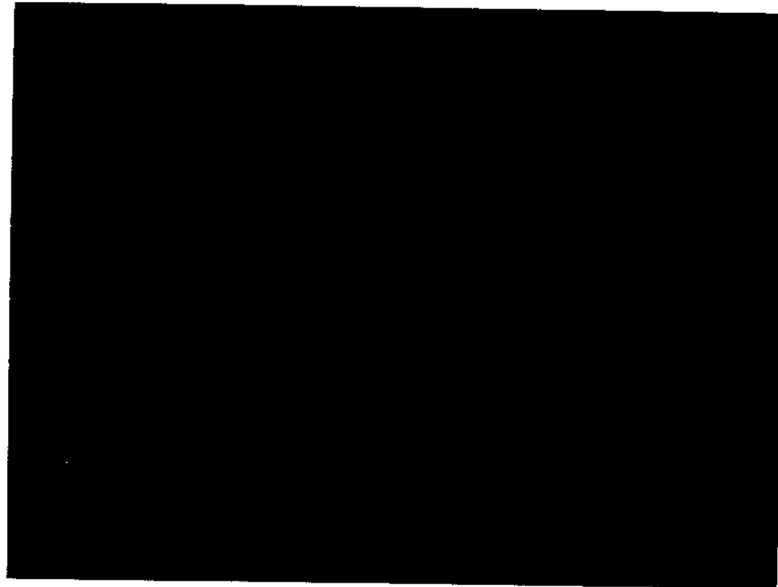


Figure 6.10 (a) Measured Worst Case Total Delay of an 8-bit ECDL CRA

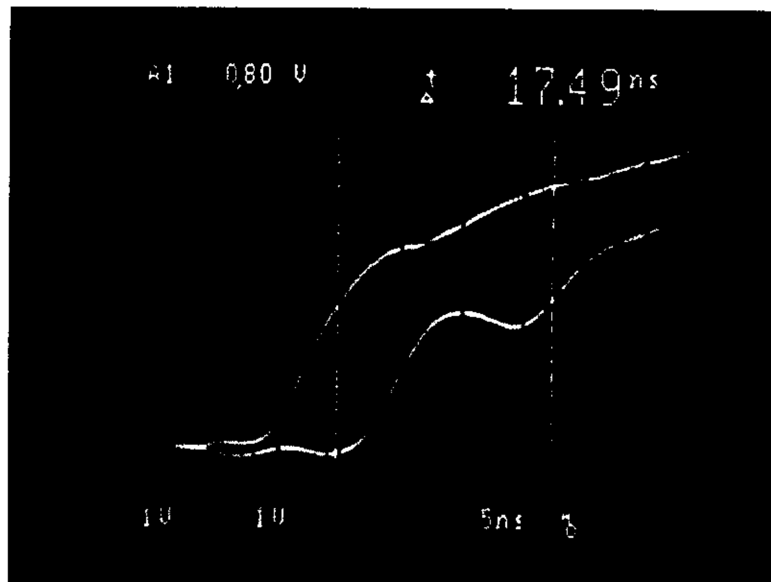


Figure 6.10 (b) Measured Worst Case Total Delay of an 8-bit ECDL CCA (measurement)

Figure 6.10 (c) Measured Worst Case Total Delay of an 8-bit ECDL CSA

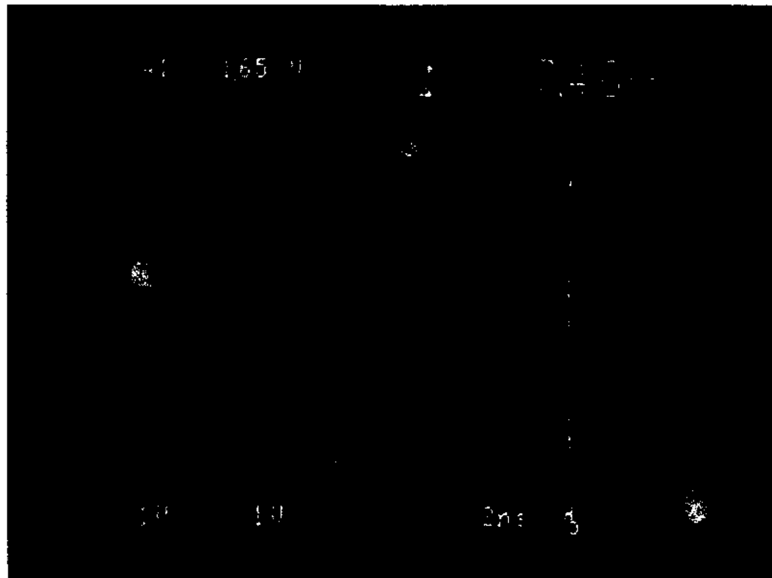


Figure 6.10 (d) Measured Worst Case Total Delay of an 8-bit ECDL CLA (measurement)

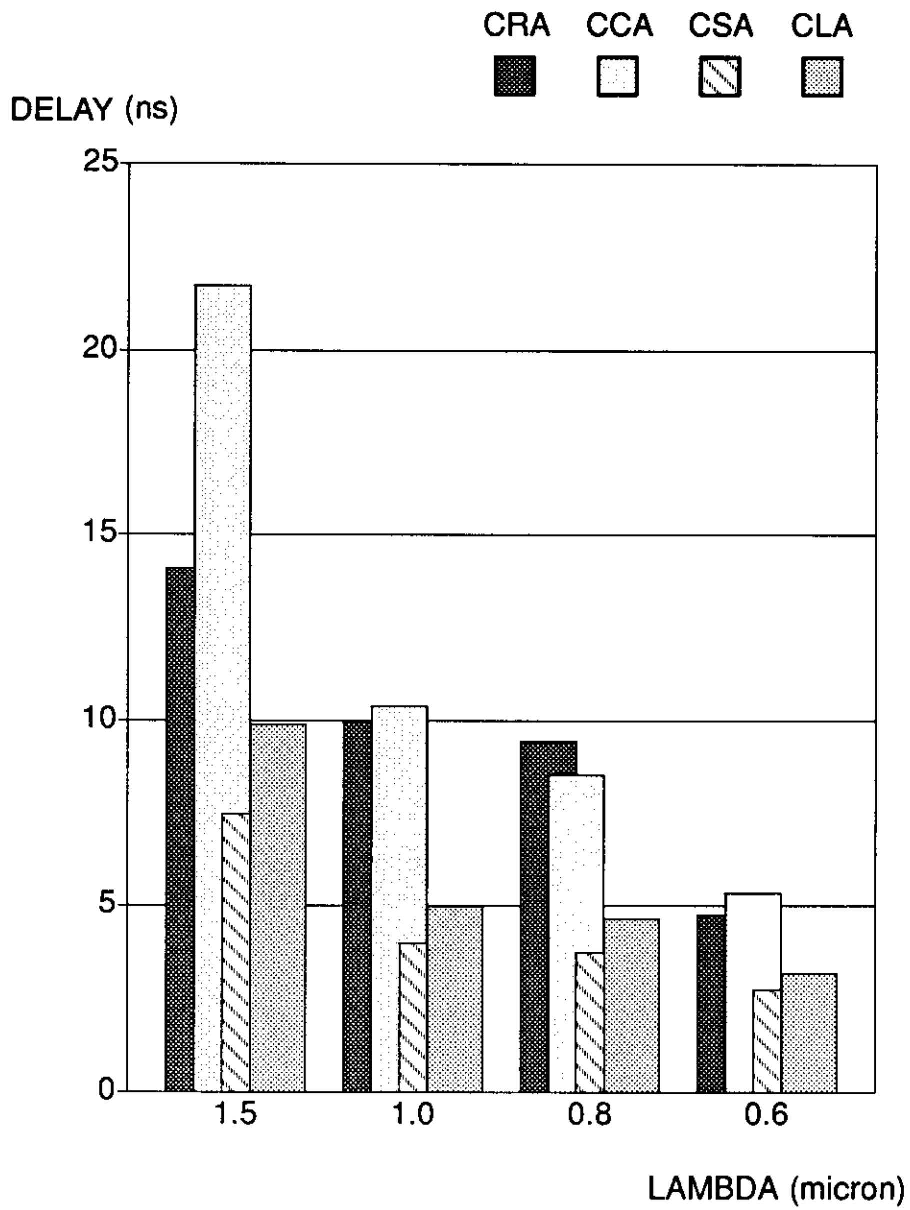


Figure 6.11 SPICE Simulation Results of all 8–bit ECDL Adders

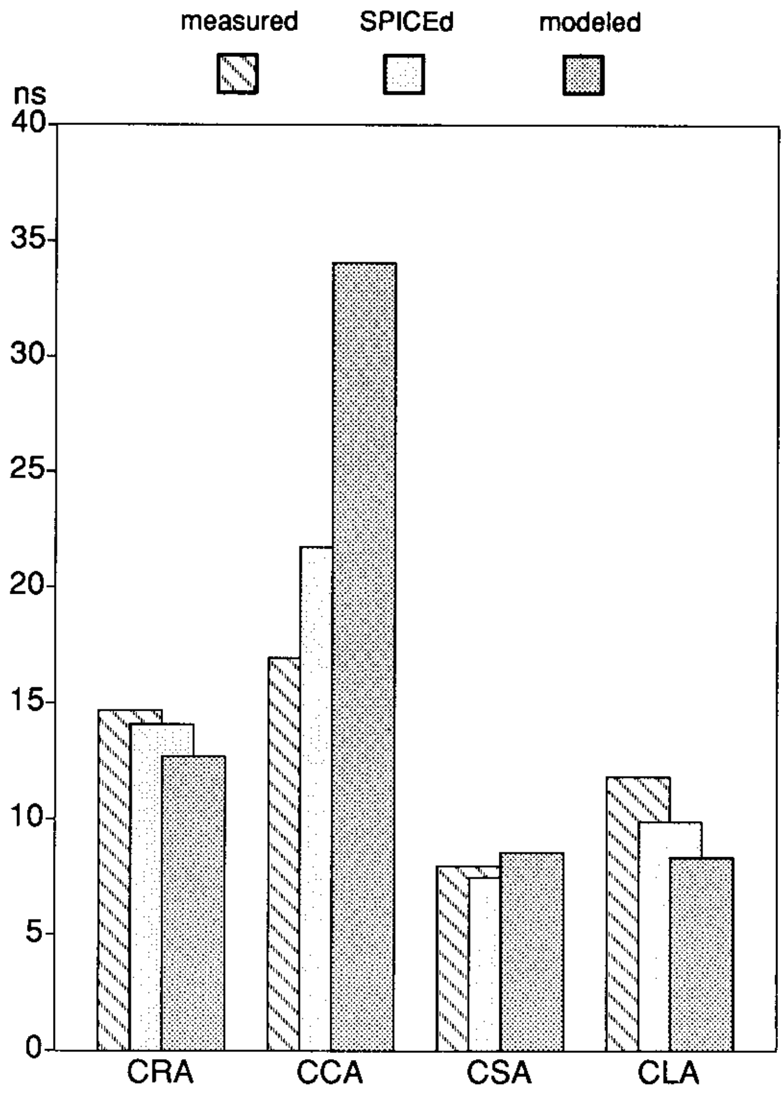


Figure 6.12 Delay Comparison of all 8–bit ECDL Adders for 3m Features Sizes

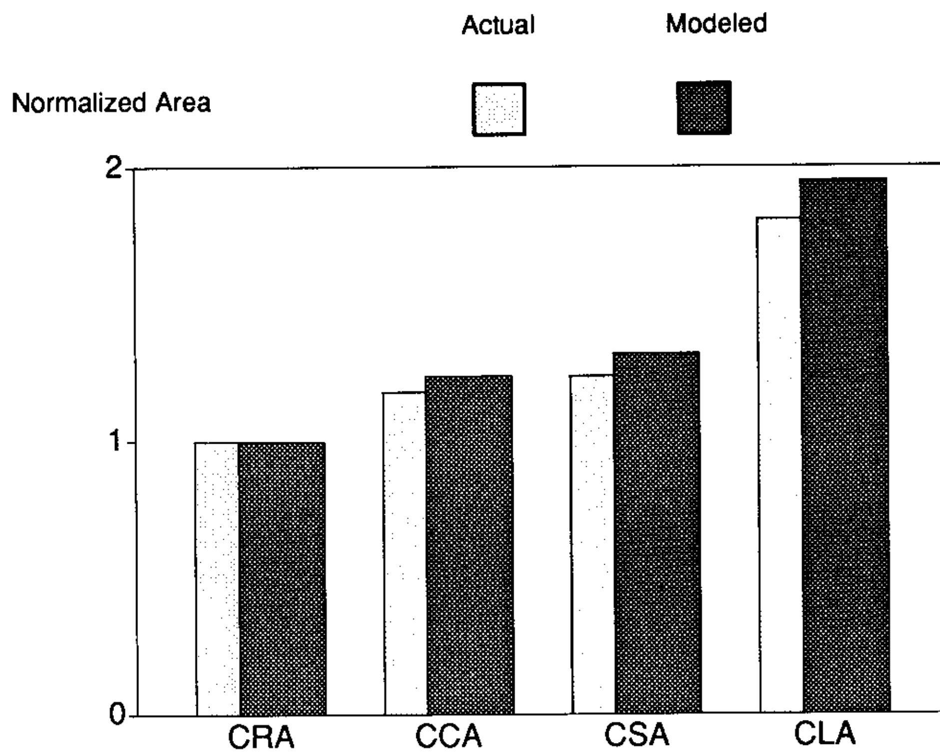


Figure 6.13 Area Comparison of 8–bit ECDL Adders

6.5. Conclusion

Four different adders were implemented using a CMOS differential logic –ECDL. The area and speed of each adder were measured and compared. One conclusion can be made is that carry skip adder seems to have the best speed area combined performance. A first order modeling method is used to estimate the area and speed of different implementation. This methodology may be used to predict other additional algorithms including variable–size carry skip adder and other arithmetic structures such as array multipliers and array dividers implemented in ECDL .

Chapter 7

Implementation of Array Multiplication and Division using ECDL

7.1 Introduction

Hardware multipliers and dividers have become standard in modern processors. For example the current IBM RISC System/6000 processor chip set consists of six VLSI chips [Misra90]. Among the six chips are the Fixed–Point Unit and the Floating–Point Unit. Both of these chips have dedicated multipliers as part of the chip. In particular, the floating-point unit consists of a 56–bit multiplier. In [Hok 90], it is stated that the goal of the new processor design is to include simple self–contained low latency hardware to reduce the number of cycles required for each instruction. In this chapter we investigate three different array multipliers and one array divider using ECDL. Their performance is presented and compared.

7.2 Array Multiplier

Array multipliers utilize the observation that partial products in the multiplication process can be obtained independently in parallel. Consider two unsigned binary integers:

$A = a_{m-1}a_{m-2}a_{m-3} \ldots\ldots a_1a_0$ *and* $B = b_{n-1}b_{n-2}a_{m-3} \ldots\ldots b_1b_0$ *with values*

$$A = \sum_{i=0}^{m-1} a_i 2^i$$

$$B = \sum_{j=0}^{n-1} b_j 2^j \qquad (7.1)$$

The product of these two binary integers is obtained by

$$P = A\ B = \sum_{i=0}^{m-1} a_i 2^i \cdot \sum_{j=0}^{n-1} b_j 2^j = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} (a_i 2^i \cdot b_j 2^j)$$

$$= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} (a_i \cdot b_i) 2^{i+j} = \sum_{k=0}^{m+n-1} p_k 2^k \qquad (7.2)$$

In a conventional gate level implementation, an *m x n* multiplier requires *m(n–1)* full adders and *mn* AND gates. The total multiply time of this unit is estimated as:

$$D = d_{AND} + [(n-1) + (n-1)] \cdot d_{FA} \qquad (7.3)$$

where d_{AND} *and* d_{FA} *are the delays of AND gate and full adder, respectively*

Since an AND gate uses 6 transistors and a full–adder uses 30 transistors when implemented with static CMOS, to implement a $m \times n$ array multiplier will require a total number of transistors:

$$(6mn + 30m(n-1)) = 36mn - 30m \qquad (7.4)$$

As mentioned in chapter 4, ECDL can be used to implement array networks (2 D iterative networks). In the following section three array multipliers are implemented in ECDL. Tradeoffs are discussed.

7.2.1 Unsigned Array Multiplier

In the nineteen sixties, various iterative array multiplication schemes have been proposed. In this section an unsigned array multiplier is implemented using ECDL. This multiplier was first introduced by Braun and is summarized in many texts [Hwa 79] [Wes 85]. An example of a 3x3 array multiplier is shown in Figure 7.1. There are two types of cells. The multiplier cell which is used to build the core and the full–adder cell which is used to generate the final product.

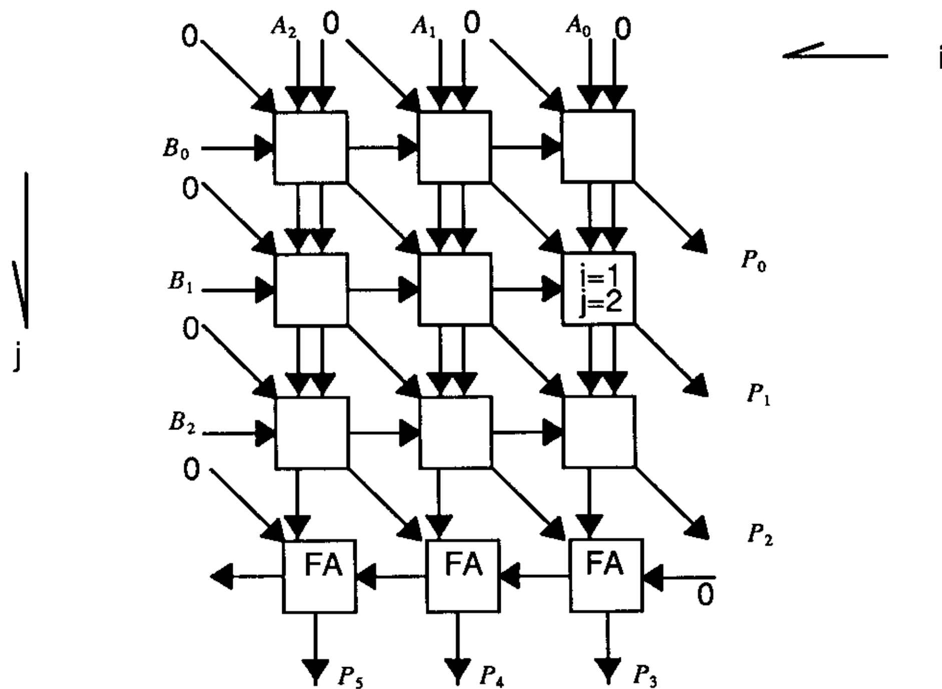


Figure 7.1 An example of a 3x3 array multiplier with ripple carry adder

Each of the multiplying cells has 4 inputs and 4 outputs. The logic diagram of the *i*th column and *j*th row multiply cell is depicted in Figure 7.2.

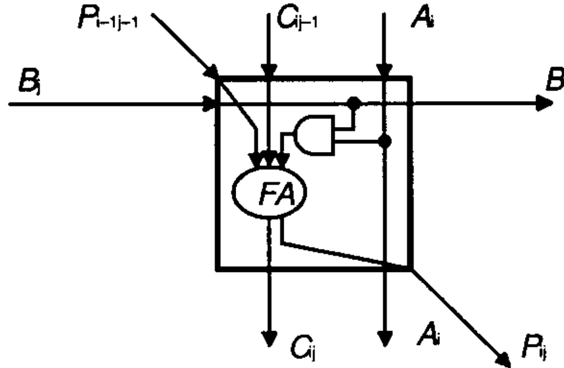


Figure 7.2 The logic of a multiplier cell (*i*th column and *j*th row)

Two of the ouputs are really inputs that are being passed through. The other two outputs can be expressed with the following logic equations:

$$
\begin{aligned}
P_{ij} &= P_{i-1j-i} \oplus C_{ij}-1 \oplus A_iB_i \\
C_{ij} &= P_{i-1j-1}(A_iB_i) \;+\; P_{i-1j-1}C_{ij-1} \;+\; (A_iB_i)C_{ij-1}
\end{aligned}
\qquad (7.5)
$$

Since ECDL is a double–railed logic family, both the function and its complement are needed. The following logic expressions for the two outputs and their complements are chosen to help to maximize the sharing of terms and, thus, to minimize the number of devices needed to implement the multiply cell.

$$
\begin{aligned}
P_{ij} &= (P_{i-1j-1}C_{ij-1} + \overline{P}_{i-1j-1}\overline{C}_{ij-1})(A_iB_i) \;+\; (\overline{P}_{i-1j-1}C_{ij-1} + P_{i-1j-1}\overline{C}_{ij-1})(\overline{A_iB_i}) \\
\overline{P}_{ij} &= (P_{i-1j-1}C_{ij-1} + \overline{P}_{i-1j-1}\overline{C}_{ij-1})(\overline{A_iB_i}) \;+\; (\overline{P}_{i-1j-1}C_{ij-1} + P_{i-1j-1}\overline{C}_{ij-1})(A_iB_i) \\
C_{ij} &= P_{i-1j-1}C_{ij-1} + (\overline{P}_{i-1j-1}C_{ij-1} + P_{i-1j-1}\overline{C}_{ij-1})(A_iB_i) \\
\overline{C}_{ij} &= \overline{P}_{i-1j-1}\overline{C}_{ij-1} + (\overline{P}_{i-1j-1}C_{ij-1} + P_{i-1j-1}\overline{C}_{ij-1})(\overline{A_iB_i})
\end{aligned}
\qquad (7.6)
$$

Method proposed by Chu [Chu 86] is used to build the N–core network which will implement the logic equation (7.6). The final ECDL implementation of two outputs of the multiply cell is shown in Figure 7.3 (a) and (b) respectively. However from Figure 7.1 we observe that some of the inputs are zeros constants. Thus the multiplier cell can be simplified. The out-

puts' logic equations, with complement, for cells of row one and cells of column one, the two outputs and their complements are:

$$P_{ij} = A_j B_i$$

$$\overline{P}_{ij} = \overline{A_j B_i} = \overline{A}_j + \overline{B}_i$$

$$C_{ij} = 0$$

$$\overline{C}_{ij} = 1 \qquad (7.7)$$

Optimized logic expression for cells at the second row:

$$P_{ij} = \overline{P}_{i-1j-1}(A_j B_i) + P_{i-1j-1}(\overline{A_j B_i})$$

$$\overline{P}_{ij} = \overline{P}_{i-1j-1}(\overline{A_j B_i}) + P_{i-1j-1}(A_j B_i)$$

$$C_{ij} = P_{i-1j-1}(A_j B_i)$$

$$\overline{C}_{ij} = \overline{P}_{i-1j-1} + P_{i-1j-1}(\overline{A_j B_i}) \qquad (7.8)$$

At the bottom of the array are ripple adder cells. The implementation of these full–adder cell is the same as the carry–ripple–adder cell implementation presented in the previous chapter. The block implementation of a 4x4 unsigned iterative array multiplier with controlling signals is shown in Figure 7.4, where cell M1 implements (7.7), cell M2 implements (7.8), and cell M3 implements (7.6)

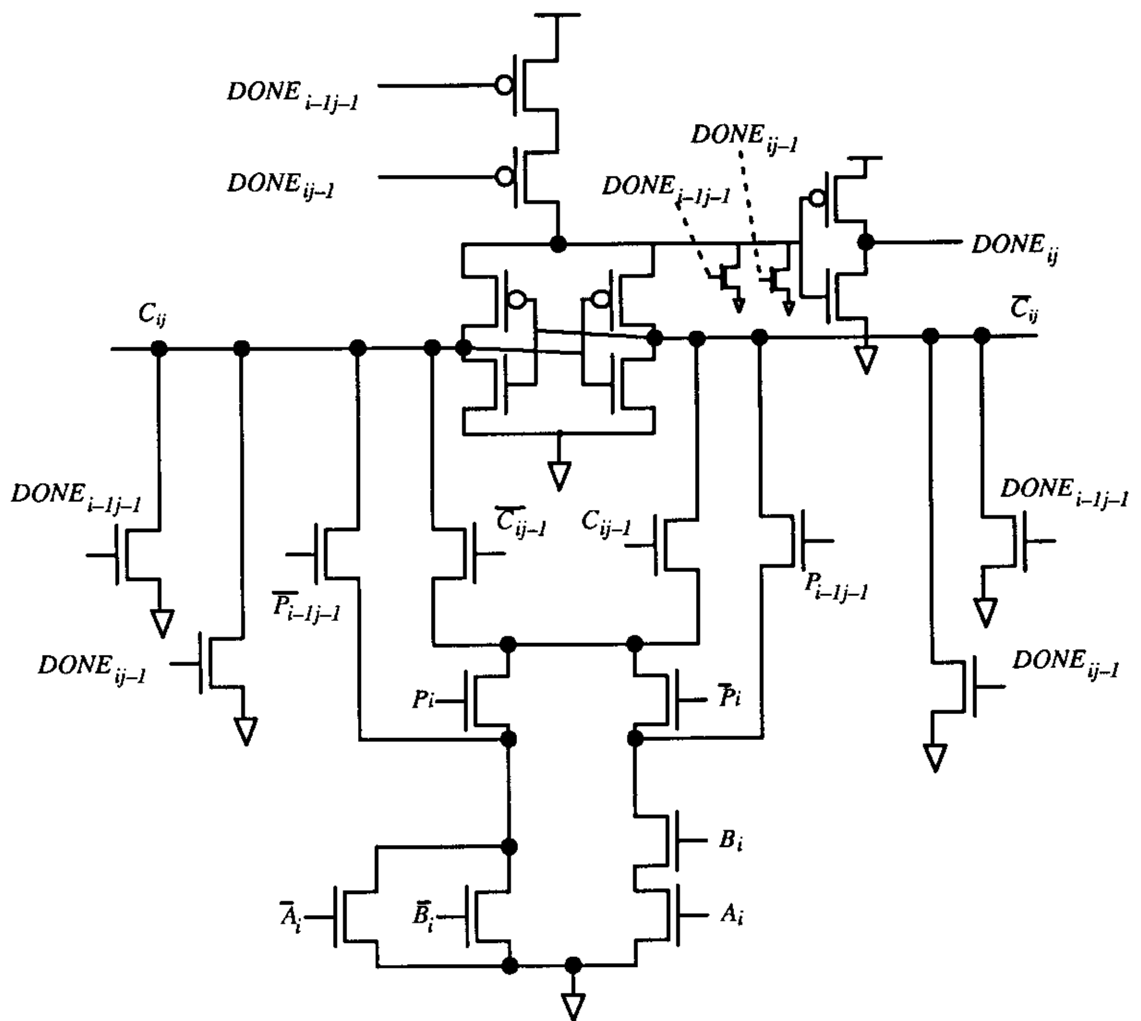


Figure 7.3 (b) ECDL Implementation of C_{ij}

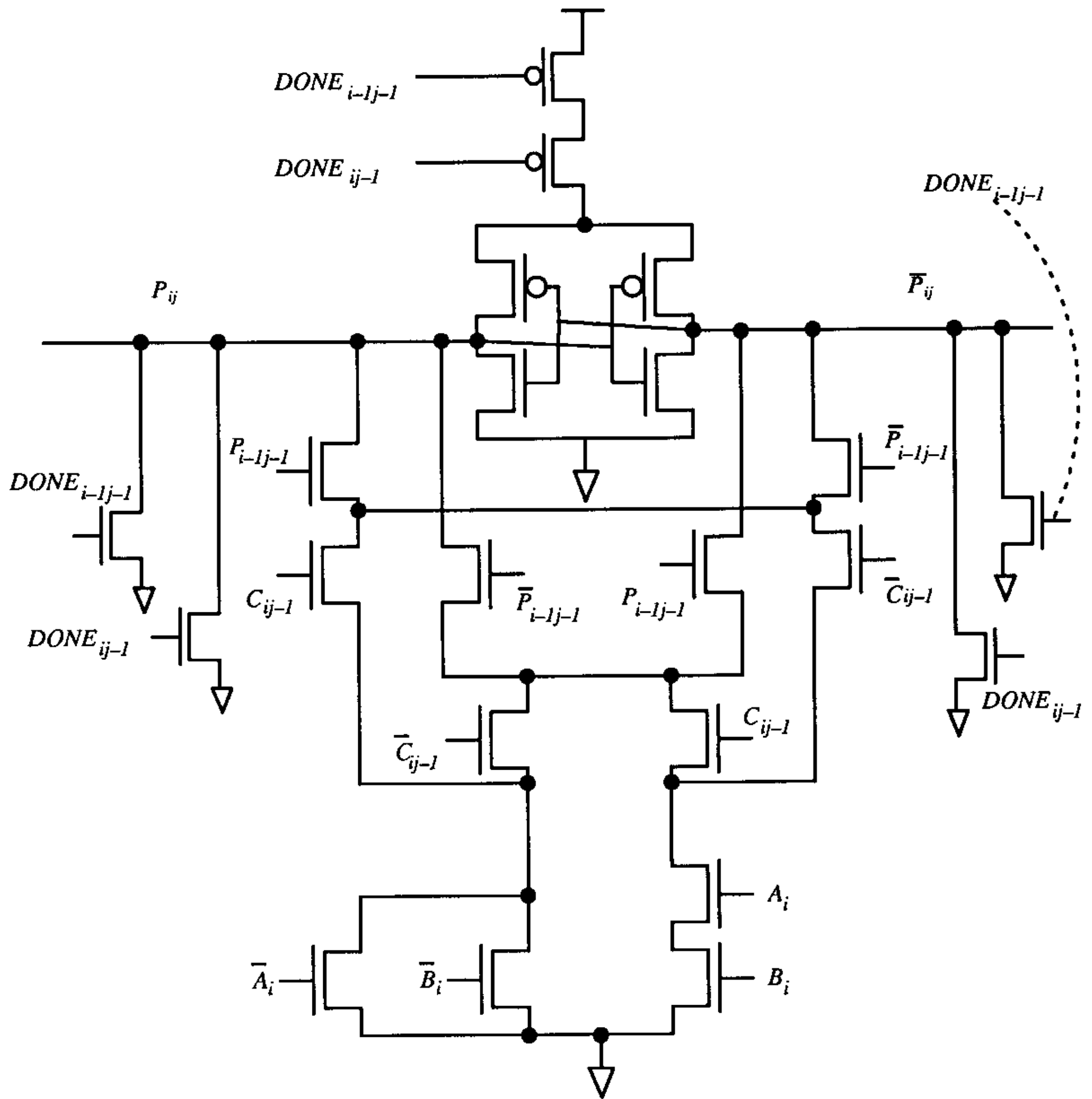


Figure 7.3 (b) ECDL Implementation of P_{ij}

There are a total of 50 devices used to implement the multiplier cell M3. Out of 50, 9 of them are P–channel devices. The two boundary cells, M1 and M2, require 14 and 32 devices respectively. The carry–ripple–adder cells uses 35 devices. The total number of devices for a *m x n* multiplier is:

$$50(m-1)(n-2)+32(m-1)+32(m-1)+14(n+m-1) = 50mn-22m-36n-76 \quad (7.9)$$

However when the number of P–channel devices is scaled by 2 to reflect the size difference between an N–channel and a P–channel transistor, static CMOS implementation requires *53mn+* devices and the ECDL implementation uses *57mn+* devices.

A 6x6 unsigned iterative array multiplier was laid out and fabricated through MOSIS using a 3–micron process. The total delay from initial enable signal to the last product bit is measured at 49 ns. This measurement compares reasonably well with the SPICE simulation result. Table 7.1 summarizes all the simulated results.

lambda / data	1.5 micron	1.0 micron	0.8 micron	0.6 micron
total delay of a 6x6 array multiplier	44.7 ns	22.0 ns	18.7 ns	12.3 ns

Table 7.1 Delays summary of an ECDL 6x6 array multiplier

By using the delay model proposed in the previous chapter, we simply substitute proper values for the variables used in equation (6.1). Thus, the average delay for a cell M3 used in an unsigned iterative array multiplier is:

$$T(M3)_{ij} = (x+xyz+4xy)\tau = (2+2\ 4\ 4+4\ 2\ 4)\cdot\tau = 66\tau \quad (7.10)$$

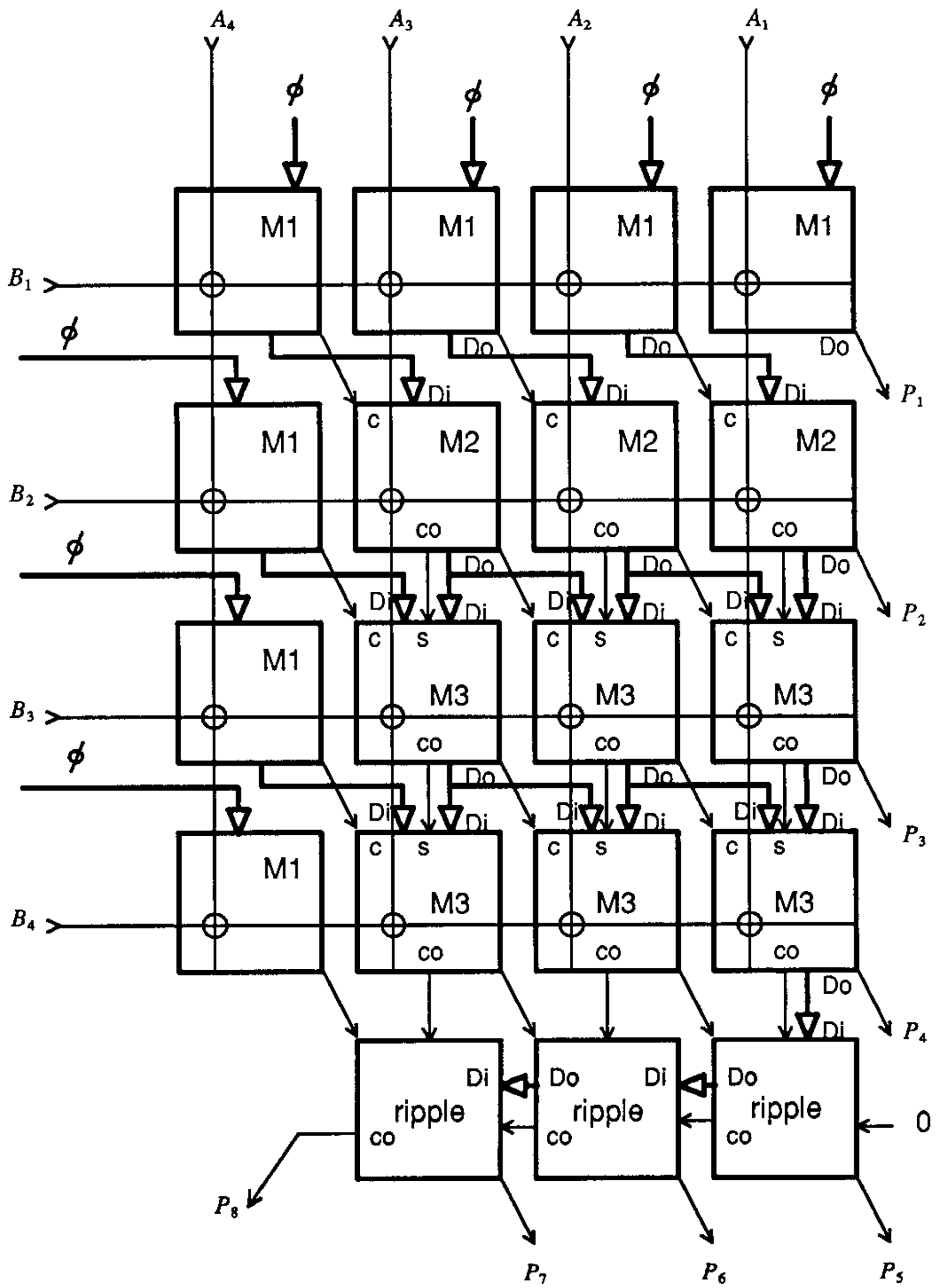


Figure 7.4 A 4x4 ECDL Unsigned Array Multiplier

7.2.2 Radix–4 Iterative Array Multiplier

As mentioned in chapter 5, an obvious way to speed up the iterative/array networks is to combine many cells into a single cell. Therefore the delay is only a fraction of the original network if the resulted cell has the same delay as the original cell. In the case of the simple unsigned array multiplier, we can improve the speed by using a similar interconnection structure between cells of a higher radix numbering representation. For example the same multiplication as that of the binary 4x4 array multiplier of Figure 7.4 can be realized by a 2x2 array multiplier of radix 4. Each of the multiplier cells can be built in ECDL. If each of the radix 4 ECDL cell is the same as the original one, there would be a speedup of 2 in total multiplication delay time. However, each the resulting radix 4 multiplier cells has 8 inputs and 8 outputs. This causes the complexity of the cell to increase greatly. In fact, the total number of devices used to implement an ECDL radix–4 multiplier cell is 294. This is more than 5 times the device count in comparison with the cell of a simple binary array multiplier. However, the total number cells needed is reduced from *nxm* to one–fourth of *nxm*. Therefore the total increase in devices counts is less than expected. If a straight forward implementation of this radix 4 multiplier cell using logic gates is used, the expected complexity should have been 16 times greater rather than five times greater per cell. The explanation for the reduction in complexity is that there are many terms which are shared in using ECDL or any other CVS–like differential logic technique to implement a complex logic function. This is because both the output and its complement are implemented with N–channel devices. In static CMOS implementation, the function is implemented in N–channel devices while the dual of the function is implemented with the P–channel devices. The total number of an *mxn* radix–4 multiplier implemented in ECDL uses around *74mn* devices.

The procedure used to build ECDL cells ais summarized in the following steps [Chu 88]:

(1) Generate truth table for each of the outputs.
(2) Establish 1–list, 0–list, 01–list and 10–list
(3) Run these 4 lists through a minimization program to obtain the reduced truth table.
(4) Construct the N–channel networks for each outputs according to the what is shown in Figure 7.5

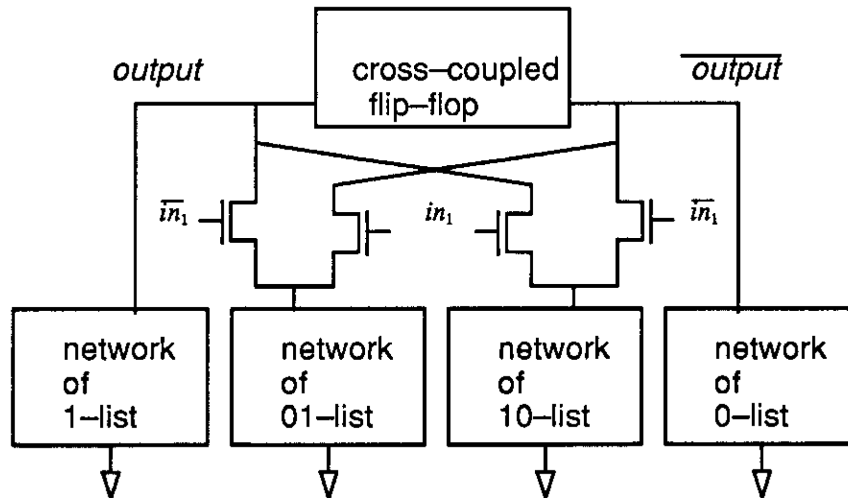


Figure 7.5 The general network used in the procedure to obtain final tree structure

An ECDL radix–4 multiplier cell is built using the above procedure. It is SPICE simulated. However the delay of this cell is much slower than twice the delay of the original cell. This is caused by the parasitic and the interconnection wiring delay. By giving the proper values for (6.1) we obtained:

$$T(Rad-4\ \ Cell) = 836\tau \qquad (7.11)$$

We did not investigate further because it is a bad implementation choice. It would be better off if we use recoding to begin with. If we use recoding the number of cells used will be reduced by one–half.

7.2.3 Array Multiplier using 5–Counters

There are alternative designs which will reduce the total delay in half without increasing the complexity of the cell by much. One particular method was proposed by Nakamura [Nak 86]. This method calls for the implementation of array multipliers using 5–counters and it looks cost effective. The 5–counter cell has 5 inputs and produces 3 outputs. The multiply cell using the 5–counter has 7 inputs and 3 outputs. Of the 7 inputs, 4 are used to generate "partial products". These four inputs are ANDed in pairs to produce two partial products. These two partial products and the other three inputs form the inputs for the 5–inputs count block. The outputs of the counts blocks are the outputs of the 5–counter multiplier cell, and it

is the binary representation of the number of ones in the 5 inputs to the 5–inputs count block. Figure 7.6(a) illustrates the logic operation of a 5:3 counter used in the 5–counter multiplier cell.

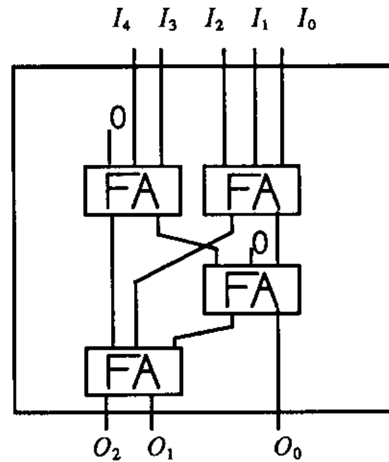


Figure 7.6 (a) 5–counter cell

Figure 7.6(b) shows the 5–counter multiplier cell. Again the same procedure as discussed in the previous section is used to obtain an ECDL implementation of 5–counter multiplier cell. The total number of devices used in the ECDL implementation of a single cell is 124. Out of the 124 devices 16 of them are P–channels.In comparison using static CMOS to implement the same cell requires 136 devices, which half of them are P–channel devices. The total number of an *nxm* multiplier uses $62nm + 39m + 39n - 2$ devices.

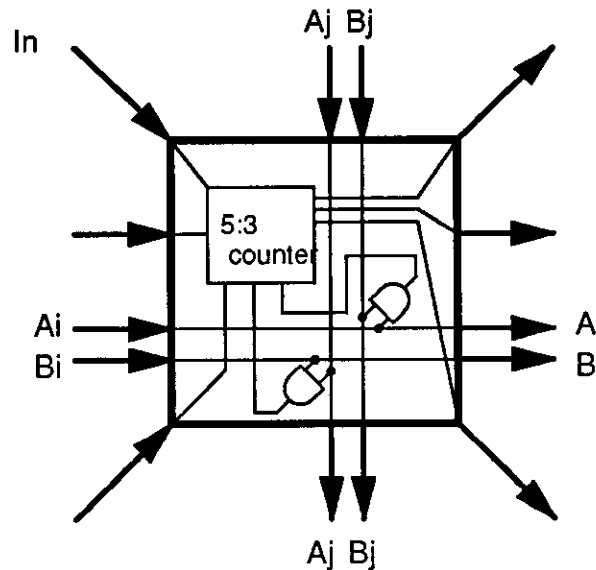


Figure 7.6(b) A Multiply cell using 5–counter cell

Two other cells are also needed to build the 5–counter array multiplier. They are the 2–input XOR and the 2–input OR cells. In Figure 7.7, a 5x5 array multiplier is illustrated which shows the interconnection of these three cells. Again the boundary cells, cells of the first and second columns, can be simplified since part of the inputs are always zero. However, the total number of devices saved is not significant in large array multipliers.

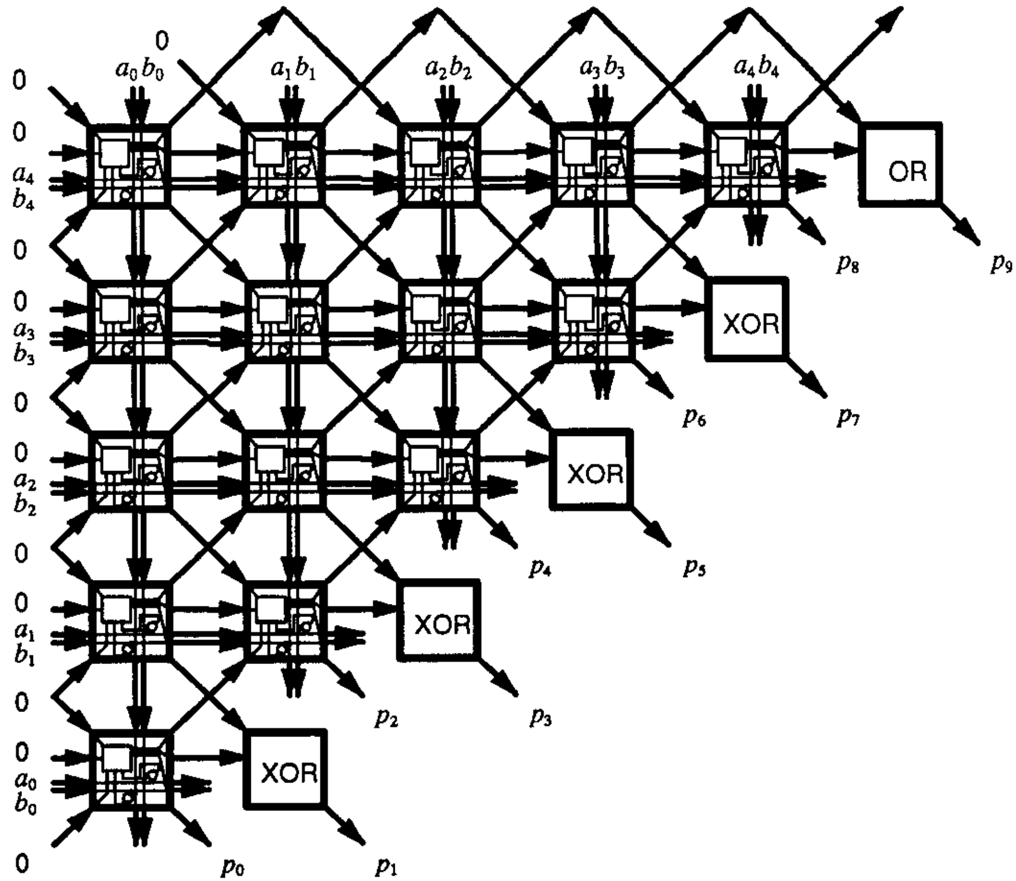


Figure 7.7 5x5 Array Multiplier with 5–counter Multiplier cell

Again, the modeled delay from (6.1) predicts that the average cell delay is:

$$T(5-Ctr\ \ Cell) = 111\tau \qquad (7.12)$$

A 6x6 ECDL 5–counter multiplier is built using MOSIS' 3 micron technology. The worst case delay is measured at 36 ns. Table 7.2 tabulates the SPICE simulation result for different feature sizes. We observe that the total simulated delay of this 6x6 multiplier implemented with 1.2 micron technology is 9.1 ns. This is an improvement of more than 3 ns from the simple binary array multiplier. With this value, a 32x32 5–counter array multiplier will have a delay of 45.5 ns.

lambda / data	1.5 micron	1.0 micron	0.8 micron	0.6 micron
total delay of a 6x6 array multiplier	30.5 ns	17.0 ns	14.5 ns	9.1 ns

Table 7.2 Simulated Delay Summary for Different Feature Sizes

7.2.4 Comparison

Two iterative array multipliers were fabricated using MOSIS' 3 micron process. The Radix–4 iterative array multiplier was not implemented, since it was too complicated to have any speed and area gain. Results of the measurements are listed in Table 7.3.

Methods	total delay
unsign array multiplier	49.0 ns
radix–4 array multiplier	N/A
5–counter array multiplier	36.2 ns

Table 7.3 Measurements for 3mm Implementation

Table 7.4 summarizes the simulated delay of a 6x6 array multiplier using the three different implementation methods discussed in this chapter.

Methods	delays simulated using 1.2 micron implementation
simple array multiplier	12.3 ns
radix–4 array multiplier	17.8 ns
5–counter array multiplier	9.1 ns

Table 7.4 Comparison of simulated delays of three 6x6 ECDL array multipliers

Table 7.5 compares the modeled delay.

Methods	model delay
simple array multiplier	66τ
radix–4 array multiplier	836τ
5–counter array multiplier	111τ

Table 7.4 Comparison of modeled delays of three ECDL array multiplier

Table 7.6 tabulates the comparison of total devices used to implement a *nxn* multiplier using three different methods.

Methods	total devices count
simple array multiplier	$50nm - 30m - 25n + 19$
radix–4 array multiplier	$74mn + 54m + 50n$
5–counter array multiplier	$62nm + 39m + 39n - 2$

Table 7.6 Comparison of devices used for three methods

The delay of an unsigned *mxn* array multiplier implemented with conventional method is:

$$T_1 = (m + n - 1)\Delta M_1 \qquad (7.13)$$

Where ΔM_1 is the delay of a single unsigned array multiplier cell. The delay of a radix–4 multiplier implemented with conventional methods is:

$$T_2 = (\frac{m}{2} + \frac{n}{2} - 1)\Delta M_2 \qquad (7.14)$$

Where ΔM_2 is the delay of a single radix–4 array multiplier cell. And the delay of a 5–counter array multiplier is:

$$T_3 = m\Delta M_3 + \Delta X \qquad (7.15)$$

Where ΔM_3 is the delay of a single 5–counter array multiplier cell and ΔX is the delay of a XOR cell. The delays of these three array multipliers implemented with self–timed circuits are:

$$T_1(ST) = \sum_{i=0}^{i=m-1} \Delta M_{1_i} + \sum_{i=0}^{i=n-2} \Delta M_{1_i} \qquad (7.16)$$

$$T_2(ST) = \sum_{i=0}^{i=\frac{m-1}{2}} \Delta M_{2_i} + \sum_{i=0}^{i=\frac{n-1}{2}} \Delta M_{2_i} \qquad (7.17)$$

$$T_3(ST) = \sum_{i=0}^{i=m-1} \Delta M_{3_i} + \Delta X_i \qquad (7.18)$$

Chapter 8

Clocking Schemes with Differential Logic

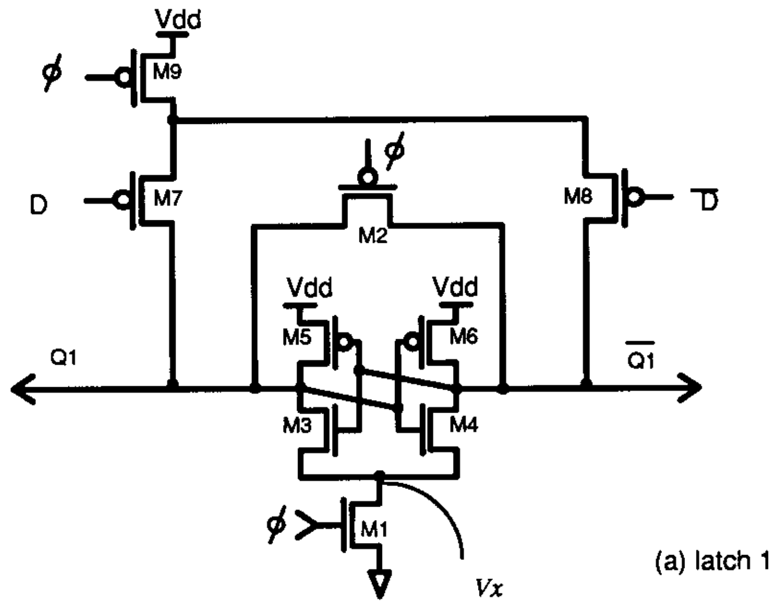
8.1 Introduction

In this thesis, so far, we have been concentrating on the design and implementation of self–timed modules using ECDL. However, ECDL is not restricted only to the implementation of asynchronous systems. It can also be used to implement digital systems with synchronous timing disciplines. In this chapter we propose a safe single phase timing discipline implemented using modified ECDL flip–flops. This timing discipline is called double–edge triggered single clock timing discipline. Traditionally the edge–triggered single phase timing scheme uses single–edge–triggered flip–flops (SET–FFs) to store its states. SET–FFs change states at the time when the clock signal goes from 0 to 1 or at the time when the clock goes from 1 to 0. The former is called the *positive–edge–triggered flip–flops* (PET–FFs) or *rising–edge–triggered flip–flops* (RET–FFs) and the latter is called *negative–edge–triggered flip–flops* (NET–FFs) or *trailing–edge triggered flip–flops* (TET–FFs). The advantage of edge triggering is that the data inputs (D) may change anytime, outside of the interval defined by the setup and hold time, without affecting the output of the flip–flop. This makes system design simpler. It is also less sensitive to noise. However, these flip–flops respond only once per clock pulse cycle. Energy and time are wasted. Unger proposed in [Unger 74], a class of flip–flops (FFs) which will respond to both the positive and the negative edges of the clock pulse. Using these *double–edge–triggered flip–flops* (DET–FFs) to implement a single phase system has two major advantages. First, power dissipation is reduced. With the conventional single–edge–triggered flip–flops, one of the two clock transitions accomplishes nothing. However, this transition may cause changes in the outputs of some logic elements internal to the FFs. In addition, extra energy is wasted to charge or discharge the capacitive load of the global clock line in a system using SET–FFs. This is particularly true in CMOS where static power dissipation is small and the dynamic power dissipation is the main contributor of energy dissipation. Second, the speed of the system is accelerated. With both edges able to cause state transition, some redundant logic can be eliminated. Moreover, the clock period will be shortened because there is no need to wait for the clock signal to toggle up and down.

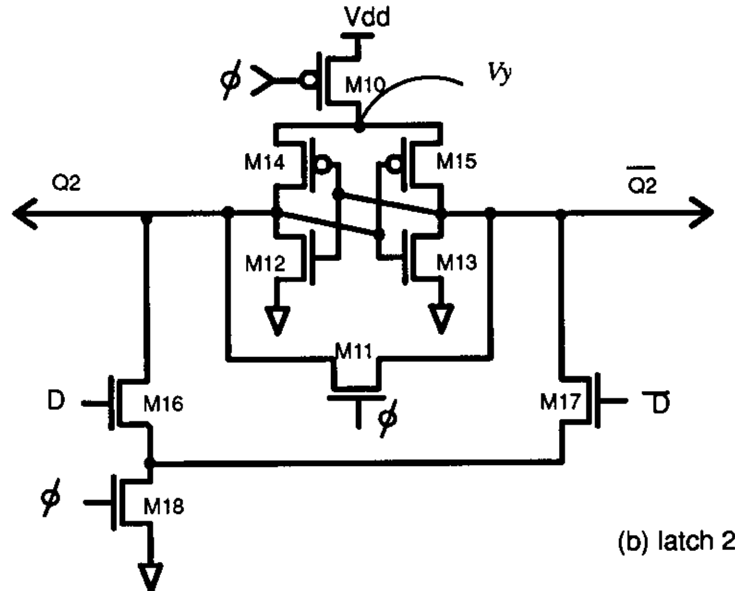
The main disadvantage of DET–FFs has been the substantial increase in the number of components required to build such FFs. In most cases, more than double the logic counts is expected. This paper proposes a novel design in CMOS which will implement static DET–FFs with relatively little increase in components. It is based on the single–phase CMOS register proposed by Lu in [Lu 88]. An implementation of a D–type DET–FF uses only 26 MOS devices in comparison with a typical static CMOS D–type flip–flop which requires 16 MOS devices. Another disadvantage of DET–FFs is in the extra delays caused by the extra gates needed to implement it by parallel decomposition. This CMOS implementation presented introduces little delays. It satisfies the speed requirement of the modern digital system. This D–FF is clocked at 50 MHz. Simulation performed with parameters obtained from a MOSIS (MOS Implementation System) [Tomovich 89] supported 2 micron CMOS/bulk process endorses the proposed implementation.

8.2 Circuit Design of a D–type DET–FF

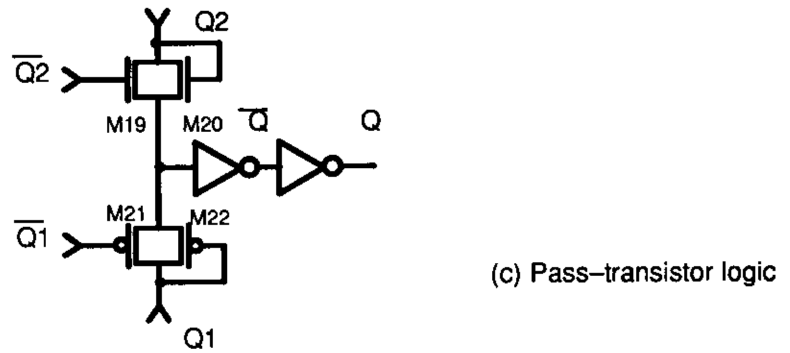
A D–type DET–FF consists of two cross–coupled latches with input gating devices and some simple pass transistor logic. A circuit diagram is illustrated in Figure 8.1. Its operation principle is similar to the one used by Mead and Wawrzynek [Waw 85]. The two cross–coupled latches are enabled/disabled by the clock signal as in ECDL. When the clock is low, *latch 1* is disabled and *latch 2* is enabled. With clock high, *latch 1* is enabled and *latch 2* disabled. A disabled *latch 1* has both its output and the complement set to high (*Vdd*). A disabled *latch 2* has both its output and the complement set to low (GND). During the rising edge of the clock signal, *latch 1* is being enabled. Depending on the D input value, either transistor M7 or M8 is conducting just before M9 switches off. Either output Q1 or its complement will remain charged to high (*Vdd*) while the other is discharged to low (GND). The set value will stay unchanged through out the half of the clock period while it is high. Similarly, on the trailing edge of the clock signal *latch 2* is being enabled. According to the value of D input, either Q2 or its complement will remain low (GND) while the other will be set to high. Output value remains stable for the duration of low clock signal. Thus, this DET–FF is a static flip–flop. It consumes no static power. Table 1 gives the logic required to obtain the final output value. We observed that when the clock is low, both Q1 and its complement are high. The final value is the value of Q2. When clock is high, both Q2 and its complement are low. The final takes the value of Q1. Pass–transistor logic, shown in Figure 8.1(c), is used to implement the logic function.



(a) latch 1



(b) latch 2



(c) Pass–transistor logic

Figure 8.1. Circuit Digram of a D–type DET–FF

Since the two latches operate during the "rising" and "trailing" edge of the clock, the rise and fall time of the clock signal needs to be long enough for the proper operation. Please refer to Figure 8.1 for the following explanation. When the clock is low, M1 is in the cut–off region. M9 is on, M2 is on and either M7 or M8 is on. They are all in the linear region. Q, Qbar and *Vx* are all at 5V (*Vdd*). M3 and M4 are also in cut–off region. There is no static current from *Vdd* to ground. When the clock rises, the gate voltage of M1 rises also. When the clock signal voltage is greater than the threshold voltage (*Vth*), M1 goes into saturation. *Vx* is discharged by the drain current of the M1. When *Vx* drops more than a threshold voltage, M3 and M4 also turn on (in saturation) to discharge the output and its complement. M9 is still on. Depending on the value of D input, either M7 or M8 is on, charging one of the output node. This causes a different in voltage between Q1 and its complement. When the clock signal rises to be more than *Vdd–Vth(p)*, M9 is off, and there is no more charging current. The value of D input will not affect the output anymore. The voltage difference caused during the time when the clock signal goes from *Vth(n)* to *Vdd–Vth(p)* must be large enough to overcome any offset voltage due to device mismatch. Similarly, the fall time of the clock needs to be long enough to create a difference in voltage for Q2 and its complement.

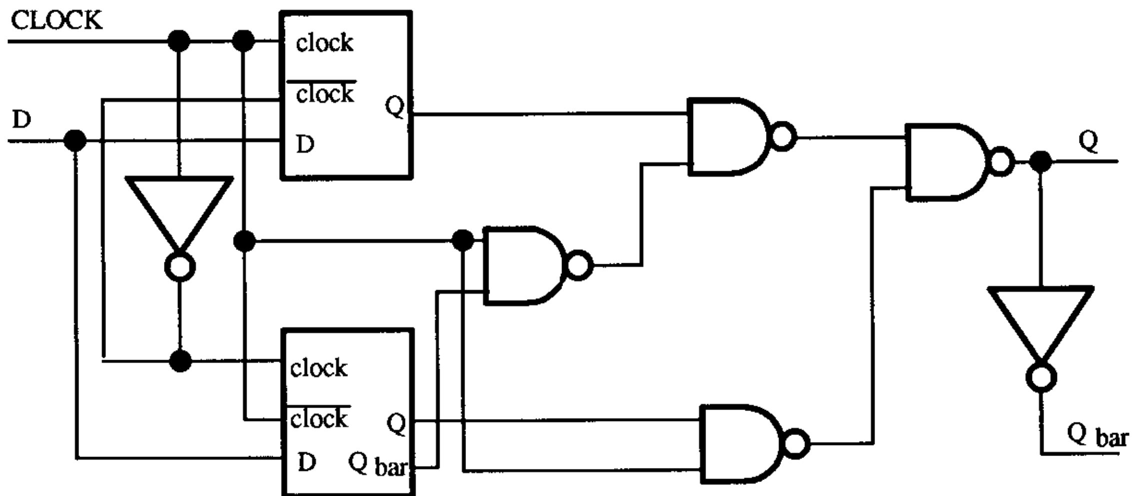


Figure 8.2. DET–D–FF gate level implementation in [Ung 81]

There are total of 26 transistors used in implementing this CMOS D–type DET–FF. The realization method proposed by Unger in [Ung 81] uses 36 transistors, since we can realize a D–latch with 8 devices, an NAND gate with 4 devices and an inverter with 2 devices. Figure 8.2 shows the gate level implementation of DET–FF proposed by [Ung 81]. Table 2 summarizes the comparison. Extract circuit from layout is simulated with SPICE. A clock signal of 50 Mhz is inputed. The D–DET FF shows a delay of less than 3ns from CLOCK to Q and Qbar. We also observe that the D–DET–FF does change its state with the transition of both rising and trailing edges of the clock signal. Simulation is shown in Figure 8.3.

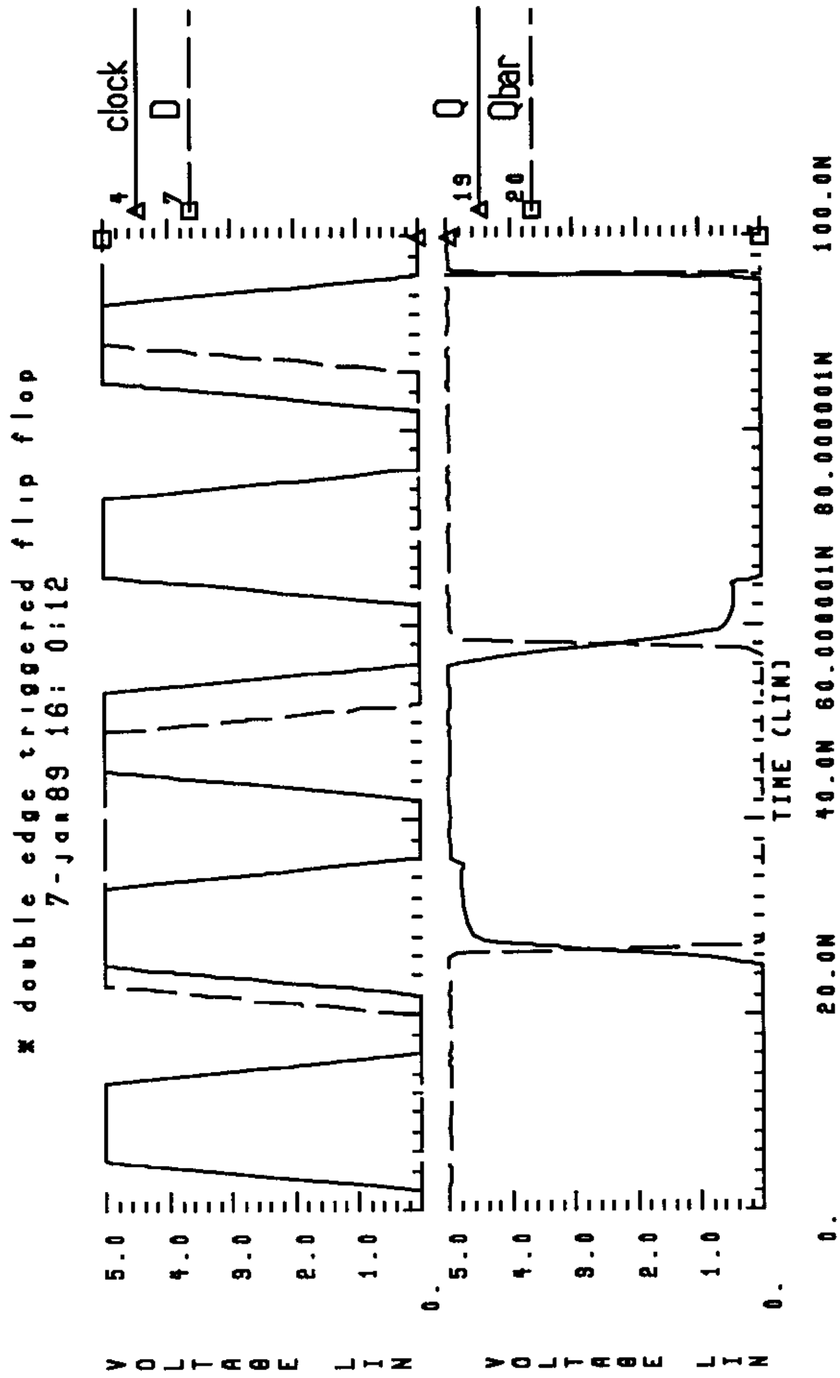


Figure 8.3. Simulation of a DET–FF

CLOCK	INPUT				OUTPUTS	
	Q_1	$\overline{Q_1}$	Q_2	$\overline{Q_2}$	Q	$\overline{Q}$
Low	High	High	–	–	Q_2	$\overline{Q_2}$
High	–	–	Low	Low	Q_1	$\overline{Q_1}$

Table 1. Truth Table to obtain the final output

	number of transistors	Eq. C of clock signal	Eq. C of D input
a typical SET–FF	16	5Cg	Cd
DET–FF in Figure 8.2	36	10Cg	2Cd
DET–FF in Figure 8.1	26	6Cg	2Cg

Table 2. Comparison of SET–FF and DET–FFs

Table 8.3 shows the State table of the implemented DET–FF.

CLOCK	INPUT D	OUTPUTS Q	$\overline{Q}$
↑	High	High	Low
↑	Low	Low	High
↓	High	High	Low
↓	Low	Low	High
Low	X	Q	$\overline{Q}$
High	X	Q	$\overline{Q}$

Table 3. State Table of a D–type DET–FF

8.3 Conclusion

A special CMOS logic family – ECDL is used to implement a D–type double–edge–triggered flip–flop (DET–FF). This D–type DET–FF offers speed and consumes no static power. A small price is paid in the number of devices used to build a DET–FF. The same method of powering up and down a cross–coupled latch with appropriate logic can be used to build other types of DET–FFs (eg. JK–type DET–FFs...). Other logic can also be added to form flip–flops with set/reset functions.

Chapter 9

Conclusion

9.1 Summary

Timing constraint and clock synchronization in system design remains a difficult problem. This problem will become more and more apparent with the continuing progress of fabrication technology. Fully self–timed design methodology is a promising approach to implementing complex systems where a global clocking scheme is the performance bottleneck. Before a fully, self–timed system can be built, self–timed subsystems need to be available.

This thesis presents a new differential CMOS logic family called Enable/Disable CMOS Differential Logic (ECDL). Similar to DCVSL, ECDL is a dual–rail logic family and it is suitable to build self–timed modules. Further modification by adding simple parts to the original ECDL together with other associated control circuitry such as C–element, allowed the accommodation of local generation of completion and acknowledge signals. The proposed ECDL is used to implement several computer subsystems as examples.

Several self–timed addition algorithms were implemented using ECDL. A first–order modeling technique is used to predict the performance and cost of these different adders. Results from measurement, SPICE simulation as well as results predicted by the model are tabulated and compared. First, we found that carry–skip adders are a good choice for implementation in self–timed styled using ECDL. It is comparable, in speed, with carry look–ahead adders but uses much less area. Second, we found that using ECDL to implement self–timed interactive networks helps relief the long delay resulted from implementing logic using iterative arrays.

Several array multipliers and array dividers are also implemented in ECDL. Again comparisons are made between different algorithms. We found that 5–counter array multiplier is the best choice in terms of cost and performance product.

We have also studied the use of ECDL circuits in synchronous systems. We have found that the proposed circuits are efficient in implementing double–edge triggered flip–flops.

9.2 Future work

There is a need to bring self–timed system design environments to the same level of support as the current synchronous design. On the physical level, we need more module generators. These module generators should be fully automated. The generator should also give system level performance feedback as the module is generated. On the specification level, there should be a high level language used to specify a self–timed system. In many ways functional programming language fits the design style of a self–timed timing strategy. On the testing and verification area, a none traditional approach must be taken. Of course the scan path technique as used by clocked system is still applicable. However one must distinguish between a real fault and a resulting datum taking infinite time to compute.

Another interesting question which has not been addressed in this thesis is that what is the real system performance benefit comparing the synchronous and the self–timed implementation. What needs to be done is to build a complete digital system using both synchronous method and the self–timed method and run some real benchmark programs to evaluate the performance of these two different approaches.

There is also extensive work at Caltech by A. Martin [Mar 57] [Mar 86] [Mar 90] that is not mentioned in this thesis. Their objective is to automate the design of complex self–timed systems. Other works by [Mol 85] [Mol 88], and works by [Chu 86] also address issues on high–level synthesis of self–timed systems. Both the theoretical bases and practical method of automating the design of complex asynchronous systems should be further explored.

References

[Afg 90] M. Afghahi, and C. Svensson, "A Unified Single–Phase Clocking Scheme for VLSI Systems", IEEE JSSC, Vol. 25, No. 1, Feb. 1990, pp. 225–233.

[Anc 82] F. Anceau, "A Synchronous Approach for Clocking VLSI Systems", IEEE JSSC, Vol. SC–17, No. 1, February 1982

[Anc 82] F. Anceau and R. A. Reis, "Complex Integrated Circuit Design Strategy", IEEE Journal od Solid–State Circuits, Vol. SC–17, No. 3, June 1982, pp. 459–464.

[Bri 73] B. Briley, "Some New Results on Average Worst Case Carry", IEEE Trans. on Computers, Vol. C–22, No. 5, May 1973

[Bru 89] E. Brunvand and R. Sproull, "Translating Concurrent Communicating Programs into Delay–Insensitive Circuits", CMU Technical Report: CMU–CS–89–126, April 30, 1989

[Bru 87] Erik Brunvand, "Parts–R–Us A chip aparts ...", CMU Technical Report: CMU–CS–87–119, May 20, 1987

[Cha 89] P. K. Chan and M. Schlag, "Analysis and Design of CMOS Manchester Adders with Variable Carry–Skip", proceedings of 9th IEEE Symp. on Arithmetic, 1989

[Cha 79] T. J. Chaney and F. U. Rosenberger, "Characterization and Scaling of MOS Flip Flop Performance in Synchronizer Applications", Proceedings of the Caltech Conference on VLSI, January 1979

[Chu 86] K. Chu, D. L. Pulfrey, "Design Procedures for Differential Cascode Voltage Switch Circuits", IEEE JSSC, Vol. SC–21, No. 6, December 1986

[Chu 87] K. Chu, D. L. Pulfrey, "A Comparison of CMOS Ciruit Techniques: Differential Cascode Voltage Switch Logic Versus Conventional Logic", IEEE JSSC, Vol. SC–22, No. 4, August 1987

[Chu 86] T–A. Chu, "On the Models for Designing VLSI Asynchronous Digital Systems", Integration, the VLSI Journal, Vol. 4, 1986 pp. 99–113

[Dal 86] William J. Dally, "A VLSI Architecture for Concurrent Data Structures", PhD thesis, Caltech, Techincal Report 5209:TR86, 1986

[Del 87] E. Delhaye, C. Rocher, Jean–Claude Baelde, Jean–Michel Gibereau, and M. Rocchi, "A 2.5–ns, 40–mW, 4x4 GaAs Multiplier in Two's Complement Mode", IEEE JSSC. Vol. SC–22, No. 3, June 1987

[Elzi 82] Y. M. El–ziq and S.Y.H. Su, "Fault Diagnosis of MOSIS Combinational Networks", *IEEE Trans. on Computers,* Vol. C–31, no. 2, February 1982, pp. 129–139.

[Elzi 83] Y. M. El–ziq, "Classifying, Testing, and Eliminating VLSI MOS Failures", *VLSI Design, September 1983,* pp.30–35.

[Erc 85] M. D. Ercegovac, and T. Lang, *Digital Systems and Hardware/Firmware Algorithm.* New York: Wiley, 1985 pp. 231–243.

[Erc 87] M. D. Ercegovac, T. Lang, J. G. Nash, and L. P. Chow, "An Area–Time Efficient Binary Divider", the Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors, Rye Brook, New York, Octboer 5–8, 1987

[Frie 84] V. Friedman and S. Liu Dynamic Logic CMOS Circuits. IEEE JSSC Vol. SC–19, No. 2, April 1984 pp. 263–266

[Gon 83] N. F. Goncalves, and H. J. De Man, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures", IEEE JSSC. Vol. SC–18, No. 3, June 1983

[Gro 86] T. A. Grotjohn, B. Hoefflinger, "Sample–Set Differential Logic (SSDL) for Complex High–Speed VLSI", IEEE JSSC, Vol. SC–21, No. 2, April 1986

[Guy 87] A. Guyot, B. Hochet and Jean–Michel Muller, "A Way to Build Efficient Carry–Skip Adders", IEEE Trans. on Computers, Vol. C–36, No. 10, October 1987

[Hat 86] Mehdi Hatamian, and Glenn L. Cash, "A 70–Mhz 8–bitx8–bit Parallel Pipelined Multiplier in 2.5–uM CMOS", IEEE JSSC. Vol. SC–21, No. 4, August 1986

[Hel 84] L. G. Heller, W. R. Griffin, J. W. Davis and N. G. Thoma Cascade Voltage Switch Logic: A Differential CMOS Logic Family IEEE Proceedings of ISSCC 1984, pp. 16–17

[Hof 85] M. Hofmann, "Automated Synthesis of Multi–Level Combinational Logic in CMOS Technology", PhD dissertation, Dept. of EECS., University of California, Berkeley, (Memorandum No. UCB/ERL M85/53), July 1985.

[Hok 90] E. Hokenek, R. K. Montoye and P. W. Cook, "Second Generation RISC Floating Point with Multiply–Add–Fused,", IBM Research Report RC 15649, April 90.

[Hol 82] L. A. Hollaar, "Direct Implementation of Asynchronous Control", IEEE Trans. on Computers, Vol. C–31, No. 12, December 1982

[Hwa 79] K. Hwang, *Computer Arithmetic – Principles, Architecture, and Design, John Wiley & Sons,* New York, 1979.

[Jha 90] N. K. Jha, "Testing of Differential Cascode Voltage Switch One–Count Generators", IEEE JSSC, Vol. 25, No. 1, February 1990

[Kang 86] S. M Kang, Accurate Simulation of Power Dissipation in VLSI Circuits, IEEE JSSC Vol. SC–21, No.5 , Oct. 1986, pp.889–891

[Kei 51] W. Keister, A. Ritchie, and S. Washburn, *The Design of Switching Circuits.* New York: Van Nostrand, 1951

[Kel 74] R. M. Keller, "Towards a Theory of Universal Speed–Independent Modules", IEEE Trans. on Computers, Vol. C–23, No. 1, Janurary 1974

[Ker 89] J. Kernhof, M. A. Beunder B. Hoefflinger, and Werner Hass, "High–Speed CMOS Adder and Multiplier Modules for Digital Signal Processing in a Semicustom Environment", IEEE JSSC, Vol. 24, No. 3, June 1989

[Kra 82] R. H. Krambeck, C.M. Thomas and H.S. Law "High–Speed compact circuits with CMOS" IEEE JSSC Vol. SC–17, No. June 1982, pp. 614–619

[Kun 82] H. T. Kung, "Why Systolic Architectures?", IEEE Computer, January 1982, pp. 37–46

[Lop 80] A. D. Lopez and H. F. Law, "A Dense Gate Matrix Layout Method for MOS VLSI", *IEEE Journal of Solid–State Circuits,* Vol. SC–15, No. 4, August 1980, pp. 736–740.

[Lu 88] S.–L. Lu, "A Safe Single–Phase Clocking Scheme for CMOS Circuits," IEEE Journal of Solid–State Circuits, Vol. 23, No. 1, February 1988, pp. 280–283.

[Lu 88a] S.–L. Lu, "Implementation of Iterative Networks with CMOS Differential Logic," IEEE Journal of Solid–State Circuits, Vol. 23, No. 4, August 1988, pp. 1013–1017.

[Lu 90] S.–L. Lu and M. Ercegovac, "A Novel CMOS Implementation of Double–Edge–Triggered Flip–Flops," IEEE Journal of Solid–State Circuits, Vol. 25, No. 4, August 1990, pp. 1008–1010.

[Lu 91] S.–L. Lu and M. Ercegovac, "Evaluation of Two Summand Adders in Differential CMOS", to be appear in IEEE JSSC, August 1991.

[Mar 85] A. Martin, "The Design of a Self–Timed Circuit for Distributed Mutual Exclusion", Proceedings of 1985 Chapel Hill Conference on VLSI, pp. 245–260.

[Mar 86] A. Martin, "Compiling Communicating Processes into Delay–Insensitive VLSI Circuits", Distributed Computing (1986), 1:226–234, Springer–Verlag, 1986.

[Mar 90] A. Martin, S. Burns, T.K. Lee, D. Borkovic and P. Hazewindus, "The Design of an Asynchronous Microprocessor", proceedings of the 10th Caltech Advanced VLSI Conference, 1990, pp. 351–373.

[Mas 84] Masaru Uya, Katsuyuki Kaneko, and Juro Yasui, "A CMOS Floating Point Multiplier", IEEE Journal of Solid–State Circuits, Vol. SC–19, No. 5, October 1984

[Mea 80] C. Mead and L. Conway. "Introduction to VLSI Systems", Chapter 7, written by C. Seitz, Addison–Wesley Publishing Company, 1980

[Mea 85] C. Mead and J. Wawrzynek, "A new discipline for CMOS design: An architecture for sound synthesis," in *Proceedings 1985 Chapel Hill Conference VLSI*, 1985, pp. 87–104.

[Men 89] T. H.–Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Automatic Synthesis of Asynchronous Circuits from High–Level Specifications", IEEE Trans. on Compuer–Aided Design, Vol. 8, No. 11, November 1989

[Misr 90] Mamata Misra, Editor, *IBM RISC System/6000 Technology* 1990

[Mol 85] C. Molnar, T.–P. Fang and F. Rosenberger, "Synthesis of Delay–Insensitive Modules", Proceedings of 1985 Chapel Hill Conference on VLSI, pp. 67–86.

[Mol 88] F. Rosenberger, C. Molnar, T. Chaney and T.P. Fang, "Q–Modules: Internally Clocked Dealy–Insensitive Modules", IEEE Trans. on Computers, Vol. 37, No. 9, Sept 1988. pp. 1005–1018.

[Mul 59] D. E. Muller, and W. S. Bartky, "A Theory of Asynchronous Circuits", Proceedings of an International Symposium on the Theory of Switching, the Annals of

the Computation Laboratory of Harvard University 29, Part 1 (Harvard Univrsity Press, Cambridge, MA, 1959)

[Nak 86] S. Nakamura, "Algorithms for Iterative Array Multiplication", IEEE Trans. on Computers, Vol. C–35, No. 8, August 1986

[Nel 83] Nelson F. Goncalves and Hugo J De Man NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures. IEEE JSSC Vol. SC–18, NO. 3, June 1983 pp. 261–266

[Nic 71] Jean–Daniel Nicoud, "Iterative Arrays for Radix Conversion", IEEE Trans. on Computers, Vol. C–20, No. 12, December 1971

[Okl 82] V. G. Oklobdzija and M. D. Ercegovac, "Testability Enhencement of VLSI Using Circuit Structures", the Proceedings of the IEEE International Conference on Circuits and Computers ICCC 1982, September–October 1982

[Okl 85] V. G. Oklobdzija, Robert K. Montoye, "Design–Performance Trade–Offs in CMOS Domino Logic", Proceedings of the IEEE 1985 Custom Integrated Circuits Conference

[Oust 86] J. Ousterhout et al., "1986 VLSI tools: Still more works by the original artists," University of California, Berkeley, Report, 1986.

[Par 81] R. Parthasarathy and S. M. Reddy, "A Testable Design of Iterative Logic Arrays", IEEE Trans. on Computers, Vol. C–30, No. 11, November 1981

[Pas 87] J. H. Pasternak, A. S. Shubat, and C. A. T. Salama, "CMOS Differential Pass–Transistor Logic Design", IEEE JSSC, Vol. SC–22, No. 2, April 1987

[Pete 81] James L. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice Hall, Inc. 1981

[Pfe 85] L. C. M. G. Pfennings et al. Differential Split–Level CMOS Logic for Subnanosecond Speeds, IEEE JSSC Vol. SC–20, No. 5, Oct. 1985. pp. 1050–1055

[Pre 84] J. A. Pretorius et al. Optimization of Domino CMOS Logic and Its Applications to Standard Cells Prooceedings of IEEE Custom Intergrated Circuits Conference 1984, pp 150–153

[Pre 86] J. A. Pretorius, A. S. Shubat, and C. A. T Salama, "Latched Domino CMOS Logic, IEEE JSSC, Vol. SC–21 No. 4, August 1986

[Rhy 86] T. Rhyne, and N. R. Strader, II, "A Signed Bit–Sequential Multiplier", IEEE Trans. on Computers, Vol. C–35, No. 10, October 1986

[Sang 85] A. L. Sangiovanni–Vincentelli, "An Overview of Synthesis Systems", *Proceedings of 1985 IEEE Custom Integrated Circuits Conference*, pp. 221–225.

[Sans 81] W. Sansen, W. Heyns and H. Beke, "Layout Automation Based on Placement and Routing Algorithms", in Computer Design Aids for VLSI Circuits, P. Antognetti, D. O. Pederson and H. DeMan (editor), Sijthoff anf Noordhoff, 1981, pp.470.

[Sei 79] Charles L. Seitz, "Self–Timed VLSI Systems", Proceedings of the Caltech Conference on VLSI, January 1979. pp. 345–355

[Sei 80] C. L. Seitz, "System Timing. In Introduction to VLSI Systems, C. A. Mead and L. A. Conway, Eds., Addison–Wesley, 1980

[Sha 89] R. Sharma, A. D. Lopez, J. A. Michejda, S. J. Hillenius, J. M. Andrews, and A. J. Studwell, "A 6.75–ns 16x16–bit Multiplier in Single–Level–Metal CMOS Technology", IEEE JSSC, Vol. 24, No. 4, August 1989

[Sho 88] M. Shoji, *CMOS Digital Circuit Technology,* Prentice–Hall Internation, Inc. 1988

[Skl 60] J. Sklansky, "An Evaluation of Several Two–Summand Binary Adders", IRE Trans. on Electronic Computers, June, 1960

[Skl 60] J. Sklansky, "Conditional–Sum Addition Logic", IRE Trans. on Electronic Computers, June, 1960

[Suth 89] I. E. Sutherland, "Mircopopelines", Communications of the ACM, Vol. 32, No. 6. June 1989

[Tom 88] C. Tomovich, "MOSIS–A Gateway to Silicon," IEEE Circuits and Devices Magazine, vol. 4, no. 2. pp.22–23, March 1988.

[Tur 89] Silvio Turrini, "Optimal Group Distribution in Carry–Skip Adders", proceedings of 9th IEEE Symp. on Arithmetic, 1989

[Ung 77] S. H. Unger, "Tree Realization of Iterative Circuits", IEEE Trans. on Computers, Vol. C–26, No. 4, April 1977

[Ung 77a] S. H. Unger, "The Generation of Completion Signals in Iterative Combination Circuits", IEEE Trans. on Computers, Vol. C–26, No. 1, Januray 1977

[Ung 81] S. H. Unger, "Double–Edge–Triggered Flip–Flops," IEEE Transactions on Computers, Vol. C–30, No. 6, June 1981, pp. 447–451.

[Ung 86] S. H. Unger and Chung–Jen Tan, "Clocking Schemes for High–Speed Digital Systems", IEEE Transactions on Computers, Vol. C–35, No. 10, Oct. 1986, pp. 880–895

[Wei 85] Belle W. Y. Wei, C. Thompsn and Yih–Farn Chen, "Time–Optimal Design of a CMOS Adder", UCB Research Report, August, 1985

[Wes 85] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A System Perspective*, Addison–Wesley Inc., 1985

[Wil 87] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang, "A Se;f–Timed Chip for Division", Proceedings of the 1987 Standford Conference, edited by Paul Losleben

[Win 85] O. Wing, "Automated Gate Matrix layout", Proc. 1985 International Symposium on Circuits and Systems, June 1985

[Yuan 87] J.–R. Yuan, I. Karlsson, and C. Svensson, "A True Single–Phase–Clock Dynamic CMOS Circuit Technique", IEEE JSSC, Vol. SC–22, No. 5, Oct. 1987

APPENDIX A

SPICE parameters

The corner parameters used in Chpater 2

```
*4 corners plus typical parameters from the following runs
*NMOS: fast:m5be  typical:m52s        slow:m5bg
*PMOS: fast:m57q  typical:m56g        slow:m5bg
*           TYPICAL PMOS
.MODEL PMOS PMOS LEVEL=2 LD=0.280000U TOX=510.000E-10
+NSUB=4.012077E+14 VTO=-0.754274 KP=1.342340E-05 GAMMA=0.705848
+PHI=0.600000 UO=100.000 UEXP=0.139572 UCRIT=10000.0
+DELTA=2.35927 VMAX=100000. XJ=0.400000U LAMBDA=4.720485E-02
+NFS=1.404551E+12          NEFF=1.001000E-02          NSS=0.000000E+00
TPG=-1.00000
+RSH=55 CGSO=4E-10 CGDO=4E-10 CJ=3.6E-4 MJ=0.5 CJSW=6.0E-10
+MJSW=0.33
*           FAST PMOS
.MODEL CMOSPF PMOS LEVEL=2 LD=0.280000U TOX=500.000E-10
+NSUB=7.708465E+14 VTO=-0.784985 KP=1.545932E-05 GAMMA=0.476981
+PHI=0.600000 UO=100.000 UEXP=0.176806 UCRIT=54557.2
+DELTA=1.46299 VMAX=100000. XJ=0.400000U LAMBDA=4.120570E-02
+NFS=4.868232E+11 NEFF=1.001000E-02 NSS=0.000000E+00 TPG=-1.00000
+RSH=55 CGSO=4E-10 CGDO=4E-10 CJ=3.6E-4 MJ=0.5 CJSW=6.0E-10
+MJSW=0.33
*           TYPICAL NMOS
.MODEL NMOS NMOS LEVEL=2 LD=0.100000U TOX=500.000E-10
+NSUB=1.000000E+16 VTO=0.884599 KP=4.163698E-05 GAMMA=1.49569
+PHI=0.600000 UO=200.000 UEXP=1.001000E-03 UCRIT=999000.
+DELTA=1.05750 VMAX=48267.7 XJ=0.100000U LAMBDA=7.923688E-03
+NFS=1.239917E+12 NEFF=1.001000E-02 NSS=0.000000E+00 TPG=1.00000
+RSH=25 CGSO=5.2E-10 CGDO=5.2E-10 CJ=3.2E-4 MJ=0.5 CJSW=9E-10
+MJSW=0.33
*           FAST NMOS
.MODEL CMOSNF NMOS LEVEL=2 LD=0.280000U TOX=520.000E-10
+NSUB=4.575777E+15 VTO=0.587229 KP=3.848050E-05 GAMMA=0.922197
+PHI=0.600000 UO=200.000 UEXP=1.001000E-03 UCRIT=999000.
+DELTA=1.59123 VMAX=100000. XJ=0.400000U LAMBDA=2.208002E-02
+NFS=5.033532E+11 NEFF=1.001000E-02 NSS=0.000000E+00 TPG=1.00000
+RSH=20 CGSO=5.20E-10 CGDO=5.2E-10 CJ=4.5E-4 MJ=0.5 CJSW=6.0E-10
+MJSW=0.33
*           SLOW NMOS
.MODEL CMOSNS NMOS LEVEL=2 LD=0.280000U TOX=532.000E-10
+NSUB=1.000000E+16 VTO=0.884728 KP=2.921911E-05 GAMMA=1.60185
+PHI=0.600000 UO=335.033 UEXP=1.001000E-03 UCRIT=999000.
+DELTA=0.940357 VMAX=35918.0 XJ=0.400000U LAMBDA=9.695906E-03
+NFS=8.705501E+11 NEFF=1.001000E-02 NSS=0.000000E+00 TPG=1.00000
+RSH=20 CGSO=5.2E-10 CGDO=5.2E-10 CJ=4.5E-4 MJ=0.5 CJSW=6.0E-10
```

```
+MJSW=0.33
*          SLOW PMOS
.MODEL CMOSPS PMOS LEVEL=2 LD=0.280000U TOX=532.000E-10
+NSUB=3.093068E+15 VTO=-0.717639 KP=1.000000E-05 GAMMA=0.643492
+PHI=0.600000 UO=100.000 UEXP=0.144944 UCRIT=23051.6
+DELTA=2.13959 VMAX=100000. XJ=0.400000U LAMBDA=4.389369E-02
+NFS=1.199305E+12          NEFF=1.001000E-02          NSS=0.000000E+00
TPG=-1.00000
+RSH=55 CGSO=4E-10 CGDO=4E-10 CJ=3.6E-4 MJ=0.5 CJSW=6.0E-10
+MJSW=0.33
```

3 um CMOS

```
.MODEL CMOSN NMOS LEVEL=2 LD=0.375000U TOX=425.000000E-10
+ NSUB=2.053000E+16 VTO=0.8718 KP=4.924000E-05 GAMMA=1.016
+ PHI=0.6 UO=606 UEXP=0.21634 UCRIT=112115
+     DELTA=1.000000E-06      VMAX=61949.5      XJ=0.500000U      LAMB-
DA=4.105125E-02
+ NFS=3.825579E+12 NEFF=1 NSS=1.000000E+12 TPG=1.000000
+ RSH=28.700000 CGDO=4.570337E-10 CGSO=4.570337E-10
+ CGBO=1.067298E-09
+ CJ=3.550000E-04 MJ=0.522000 CJSW=4.680000E-10 MJSW=0.339000
+ PB=0.800000
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 0.38 um
.MODEL CMOSP PMOS LEVEL=2 LD=0.327659U TOX=425.000000E-10
+ NSUB=5.636309E+15 VTO=-0.927564 KP=1.641000E-05 GAMMA=0.5324
+ PHI=0.6 UO=202.006 UEXP=0.237246 UCRIT=85028.5
+ DELTA=0.667921 VMAX=100000 XJ=0.500000U LAMBDA=3.549174E-02
+ NFS=5.538811E+11 NEFF=1.001 NSS=1.000000E+12 TPG=-1.000000
+ RSH=110.200000 CGDO=3.993366E-10 CGSO=3.993366E-10
+ CGBO=1.155782E-09
+ CJ=2.460000E-04 MJ=0.525000 CJSW=1.450000E-10 MJSW=0.021400
+ PB=0.850000
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 0.75 um
```

2 um CMOS

```
.MODEL CMOSN NMOS LEVEL=2 LD=0.238187U TOX=397.000000E-10
+ NSUB=5.726600E+15 VTO=0.760982 KP=5.590000E-05 GAMMA=0.501
+ PHI=0.6 UO=643 UEXP=0.172352 UCRIT=35918.3
+ DELTA=2.1874 VMAX=58960.2 XJ=0.150000U LAMBDA=3.684652E-02
+ NFS=1.796157E+12 NEFF=1 NSS=1.000000E+12 TPG=1.000000
+ RSH=31.000000 CGDO=3.107660E-10 CGSO=3.107660E-10
+ CGBO=4.920272E-10
+ CJ=1.060000E-04 MJ=0.716000 CJSW=5.340000E-10 MJSW=0.306000
+ PB=0.800000
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is -0.44 um
```

```
.MODEL CMOSP PMOS LEVEL=2 LD=0.250000U TOX=397.000000E-10
+ NSUB=5.770200E+15 VTO=-0.738663 KP=2.010000E-05 GAMMA=0.503
+ PHI=0.6 UO=230.919 UEXP=0.210543 UCRIT=22858.3
+ DELTA=2.198951E-06 VMAX=44177.9 XJ=0.050000U LAMBDA=5.218042E-02
+ NFS=2.106298E+11 NEFF=1.001 NSS=1.000000E+12 TPG=-1.000000
+ RSH=103.800000 CGDO=3.261786E-10 CGSO=3.261786E-10
+ CGBO=5.449557E-10
+ CJ=2.430000E-04 MJ=0.550000 CJSW=3.080000E-10 MJSW=0.342000
+ PB=0.800000
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is -0.20 um
```

1.6 um CMOS

```
*NM1 PM1 DU1 DU2 ML1 ML2
*
*PROCESS=hp
*RUN=m96h
*WAFER=15
*Gate-oxide thickness= 263.0 angstroms
*Geometries (W-drawn/L-drawn, units are um/um) of transistors measured were:
* 2.4/1.6, 4.8/1.6, 14.4/1.6, 14.4/4.0, 14.4/20.0
*Bias range to perform the extraction (Vdd)=5 volts
*DATE=04-aug-89
*
* Gate Oxide Thickness is 263 Angstroms
*
*
*NMOS PARAMETERS
*
.MODEL CMOSN NMOS LEVEL=13 VFB0=
+ -1.0374E+000,2.44562E-001,-7.9334E-002
+ 7.89709E-001,0.00000E+000,0.00000E+000
+ 1.42826E+000,-3.1383E-001,-5.9054E-002
+ 2.77692E-001,-2.3110E-002,-1.3491E-001
+ -7.7018E-003,2.37797E-002,-4.3878E-003
+ 5.15084E+002,5.67671E-001,2.10515E-001
+ 5.15700E-002,6.42654E-002,-5.7098E-002
+ 1.68199E-001,3.13036E-001,-1.0813E-002
+ 4.60281E+000,-5.3563E+000,2.90606E+001
+ -1.1803E-003,-6.2225E-003,-2.1317E-003
+ 1.84140E-003,-1.0919E-003,-8.0964E-003
+ 5.30010E-004,-1.6370E-003,1.11525E-002
+ -6.0318E-003,5.83179E-003,3.76801E-002
+ 6.38482E+002,2.68820E+002,-1.5919E+002
+ -1.2840E+001,2.03560E+001,6.97103E+001
+ 7.91526E+000,5.03075E+001,-3.1434E+001
+ 9.70793E-003,4.93089E-002,-2.7049E-002
+ 2.63000E-002,2.70000E+001,5.00000E+000
+ 5.59007E-010,5.59007E-010,6.86347E-010
+ 1.00000E+000,0.00000E+000,0.00000E+000
```

```
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
*
*N+ diffusion::
*
+ 60,   1.450000e-04,   5.730000e-10,   1.000000e-08,   0.8
+ 0.8,   0.546,   0.256,   0,   0
*
*PMOS PARAMETERS
*
.MODEL CMOSP PMOS LEVEL=13 VFB0=
+ -5.2077E-001,1.36479E-001,2.83455E-001
+ 7.03444E-001,0.00000E+000,0.00000E+000
+ 6.89859E-001,-1.8349E-001,-1.2756E-001
+ 1.41097E-002,-8.3915E-003,-5.9847E-002
+ -1.1190E-002,2.81221E-002,1.04709E-002
+ 1.87923E+002,3.87829E-001,4.94247E-001
+ 1.35309E-001,4.67897E-002,-7.9921E-002
+ -1.0380E-002,1.53733E-001,8.35597E-004
+ 8.84765E+000,-1.8765E+000,2.74088E+000
+ -2.6515E-003,2.07079E-003,-4.3892E-003
+ 8.34989E-004,-1.1908E-003,-5.5170E-003
+ 6.54914E-003,-7.1446E-004,1.38254E-003
+ -1.1653E-004,1.68407E-003,1.15214E-002
+ 1.97480E+002,7.08652E+001,-2.2999E+001
+ 8.00611E+000,1.71651E+000,7.21944E+000
+ -4.5868E-001,5.29499E+000,6.32307E-001
+ -8.7254E-003,-7.7511E-004,1.24242E-003
+ 2.63000E-002,2.70000E+001,5.00000E+000
+ 3.81910E-010,3.81910E-010,7.70293E-010
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
*
*P+ diffusion::
*
+ 125,   4.380000e-04,   1.780000e-10,   1.000000e-08,   0.85
+ 0.85,   0.476,   0.263,   0,   0
*
*METAL LAYER — 1
*
.MODEL PC_ML1 R
+ 5.200000e-02,   2.600000e-05,   0,   0,   0
+ 0,   0,   0,   0,   0
*
*METAL LAYER — 2
*
.MODEL PC_ML2 R
+ 2.600000e-02,   1.300000e-05,   0,   0,   0
+ 0,   0,   0,   0,   0
```

1.2 um CMOS

```
*NM1 PM1 DU1 DU2 ML1 ML2
*
*PROCESS=hp
*RUN=m94x
*WAFER=1
*Gate–oxide thickness= 205.0 angstroms
*Geometries (W–drawn/L–drawn, units are um/um) of transistors measured were:
* 1.8/1.2, 3.6/1.2, 10.8/1.2, 10.8/3.0, 10.8/15.0
*Bias range to perform the extraction (Vdd)=5 volts
*DATE=06–21–89
*
* Gate Oxide Thickness is 205 Angstroms
*
*
*NMOS PARAMETERS
*
.MODEL CMOSN NMOS LEVEL=13 VFB0=
+ –9.81476768398887e–01,–6.07944932205915e–02, 3.72813940266165e–01
+  7.96215280532583e–01, 0.00000000000000e+00, 0.00000000000000e+00
+  1.12375519410156e+00, 7.63500205883194e–02,–5.76778430194233e–01
+  1.49367909252274e–01, 5.25957513987947e–02,–1.59207417939029e–01
+ –1.52710039074865e–03,–1.21939637710616e–03, 1.84644642304276e–02
+  4.76963392775868e+02,4.63275E–001,4.50390E–001
+  5.68172900529153e–02, 1.46694298240633e–01,–8.95601010685062e–02
+ –1.06513517085362e–02, 3.84781220289973e–01,–1.52342371740312e–01
+ –3.51054066298352e+00,–1.06120246756420e+01, 7.73539726134143e+01
+ –3.08732377847749e–04,–7.84579680593054e–03, 9.42454212302451e–03
+  3.91559033349604e–04,–1.41711912708027e–03,–1.54912985677184e–03
+ –6.86817110688544e–03,–4.78978743097258e–03, 5.42078830292932e–02
+ –9.22650414845936e–03,–1.91300759014271e–03, 4.88911878824480e–03
+  5.58766791868520e+02, 3.58437125293079e+02,–3.61218524693922e+02
+ –1.93662258370486e+01,–8.31561646953385e+00, 1.16300507258313e+02
+  1.22805652202321e+01, 5.90487019134714e+01,–6.95995708607861e+01
+  1.27759656087596e–02, 5.43068898147907e–02,–5.35165912948455e–02
+ 2.05000E–002, 2.70000000000000e+01, 5.00000000000000e+00

+ 5.85276E–010,5.85276E–010,6.37954E–010
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
*
*N+ diffusion::
*
+ 95.43,    3.320000e–04,    4.000000e–10,    1.000000e–08,    0.8
+ 0.8,    0.9132,    0.1016,    0,    0
*
*PMOS PARAMETERS
*
.MODEL CMOSP PMOS LEVEL=13 VFB0=
```

```
+ -9.58010006654403e-02,-7.43065463031021e-02, 9.88280862179311e-02
+  7.11565457028002e-01, 0.00000000000000e+00, 0.00000000000000e+00
+  3.75187146149665e-01, 8.71026738287991e-02,-6.42707885872474e-02
+ -4.58744071599566e-02, 4.86416761317969e-02,-2.38561565904175e-02
+  4.35939582747579e-03, 7.46080733679095e-03,-1.96782259069246e-03
+  1.76434941025661e+02,1.62276E-001,7.11957E-001
+  1.42613259940565e-01, 8.33058547381846e-02,-7.91303395279619e-02
+  1.32150786102510e-01,-5.38982951472343e-02, 1.26058729863510e-02
+  8.60567635665549e+00,-4.96427689265064e+00, 8.68094496091161e+00
+ 2.93859809367867e-05,-2.28263811012316e-03,-1.95983622038314e-04
+ -2.09992856210145e-03, 1.01983507522258e-03,-1.19943847129812e-03
+  9.47268450042707e-03,-1.11008998869429e-02, 1.84585316820463e-02
+  1.80641321431356e-02,-2.40595235835579e-02, 1.03244273320566e-02
+  1.75627509264429e+02, 1.11021361553790e+02,-3.60327926371022e+01
+  6.53938354630934e+00, 9.47147721225049e-01, 1.34229902278225e+01
+  4.99012417708664e-01, 8.16096668479774e+00,-3.31716999871892e+00
+ -1.96961539954092e-02, 2.68823801224208e-02,-4.89793456656283e-03
+ 2.05000E-002, 2.70000000000000e+01, 5.00000000000000e+00

+ 2.05011E-010,2.05011E-010,7.21151E-010
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 1.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
+ 0.00000E+000,0.00000E+000,0.00000E+000
*
*P+ diffusion::
*
+ 128.2,    4.840000e-04,    1.270000e-10,    1.000000e-08,    0.85
+ 0.85,    0.5044,    0.1766,    0,    0
*
*METAL LAYER — 1
*
.MODEL PC_ML1 R
+ 5.480000e-02,    2.600000e-05,    0,    0,    0
+ 0,    0,    0,    0,    0
*
*METAL LAYER — 2
*
.MODEL PC_ML2 R
+ 3.600000e-02,    1.300000e-05,    0,    0,    0
+ 0,    0,    0,    0,    0
```