1996

Design and optimization of high performance binary adders for digital signal processing.

Jinghong. Wang
University of Windsor

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

ISBN 0-612-11035-4

Canada

Name TINGHUNG WANG

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Electronics and Electrical Engineering

SUBJECT TERM

0544

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Abstract

In this thesis, recently developed concepts and circuit design techniques associated with Enhanced Multiple-Output Domino Logic (EMODL) are explored for the design of high performance dynamic carry lookahead adders (CLAs). In order to improve the area and speed of the EMODL design techniques, we investigate the trade-off between the number of cascaded gate stages and the gate fan-in of each stage by varying these factors in four different architectural structures for a 32-bit CLA. From HSPICE simulation results, which show that the number of cascaded stages is a more critical factor than the gate fan-in, a three stage 32-bit dynamic CLA is designed and fabricated in a 1.2μ CMOS technology. Both HSPICE simulation and test measurements show a critical path of 2.7ns. In comparison with a recent design reported by Hwang and Fisher, which requires five stages, our new architecture demonstrates improved area-speed performance.

Acknowledgments

I would like to express my sincere gratitude first and foremost to my research advisor Dr. Graham A. Jullien for his support and guidance throughout my study at the University of Windsor. I am grateful to Dr. William C. Miller for his constant support and thoughtful comments for this work. I would also like to thank Dr. S. Bandyopadhyay for serving on my thesis committee. I was very fortunate to have the opportunity working closely with Professor Zhongde Wang, who has been a constant source of advice and encouragement. I would also like to thank him for spending his busiest nights proofreading my preliminary draft of this thesis. Dr. Richard J. Caron from Department of Mathematics and Statistics also took interest in my research. I appreciate his enthusiasm and patience in many discussions I had with him. I am also thankful to many of my friends and the fellow students in our VLSI Research Group for their help and friendship. Last, but not the least, my most sincere gratitude must go to my husband and our parents for their understanding and loving support so that I could reach my goal.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

In data processing systems such as microprocessors and digital signal processors (DSP), the Arithmetic/Logic Unit (ALU) and multiplier are the engines of the system. In the ALU and multiplier, which perform the operations of ADD, SUBTRACT, MULTIPLY and DIVIDE; the ADD operation is ubiquitous. In the multiplier, for example, partial products are obtained through several stages of full adders or carry save adders, and the final product is received from a final multi-bit adder stage. It is clear that the speed of the processors is directly affected by the speed of the adder.

It is well-known that processor speed slows down dramatically when communicating with external memory. Utilization of cache memory and register files is one of the solutions. The more memory and registers that can be placed on the chip, the faster the processor will operate. Since smaller components result in more chip components, the size of the adder is an equally important design issue.

The performance of a system is characterized by its speed, size and power dissipation. As portable computers become popular,

minimizing power dissipation of adders is important in saving battery life. Generally speaking, dynamic power dissipation of a dynamic logic circuit is caused by current flow during precharge and discharge. The magnitude of current flow per unit time depends primarily on the number of transitions, number of transistors that is involved in logic transitions and the size of those transistors (i.e. factors that affect charge movement during transitions). Although studies on reduction of power dissipation is an active area, we have restricted the objectives of this thesis to the design of high speed and hardware efficient carry lookahead adders. Since the number of transistors used by the adder is a vital element for estimating the silicon area and power dissipation, however, design philosophies that decrease the number of transistors will be advantageous to improving the overall circuit performance.

Starting from analyzing and comparing various adders at both the architectural level and circuit level, we determine that CLA architectures, associated with dynamic logic implementations, are very efficient. This choice is enhanced by the availability of several new concepts, including pseudo complement mapping techniques and EMODL circuit styles. A 32-bit CLA is designed accordingly and demonstrates that these new concepts lead to area efficient low critical path delay CLA designs.

We show that the 32-bit adder, laid out in 1.2μ CMOS technology, has a smaller area than a recent design by Hwang and Fisher [15] using a 0.9μ CMOS technology. Our HSPICE simulation results also show that the new design is 15% faster than Hwang and Fisher's design. A ten serial connected critical path adder is fabricated and I/O pad calibrators used to minimize the error introduced by I/O pad delays. The test result is consistent with the HSPICE simulation results obtained from the complete 32-bit adder.

When the adder is designed using EMODL circuit techniques, its critical path delay is affected by many factors. The most important ones are: the fan-in and fan-out of each gate, the dimension of each MOS transistor and the number of cascaded gates in the critical path. In this thesis, we investigate trade-offs of these factors with regard to area and speed by designing and analyzing four different schemes of a 32-bit adder. Optimization

approaches, including sizing transistors, and modifying the number of cascaded gate stages and the fan-in of each stage, are discussed. Examples of four adder schemes and their simulation results are provided. By comparing the performance of the four designs, we conclude that the 32-bit CLA, with only three worst case gate stages, is the best structure.

1.2 Thesis Organization

The thesis is outlined as follows. Chapter 2 reviews the current efforts in high speed adder design, at both the algorithm and architectural level. The CLA algorithm, which is considered the fastest, is then selected as the basis of our new adder design. Finally, a recent design concept, referred to as "pseudo complement" and associated with the CLA algorithm, is discussed.

In chapter 3, evolution of high speed CMOS circuit design techniques from static CMOS to domino logic is illustrated. The multiple output domino logic (MODL) technique is discussed specifically. The idea of MODL is then broadened to suit more general and flexible circuit designs. This results in a new circuit family which we refer to as enhanced multiple output domino logic (EMODL). This new circuit style proves advantageous to speeding up the CLA operation. Examples of EMODL gates are also given in this chapter.

Details of our newly designed 32-bit CLA are described in chapter 4. The reasons for using our proposed new design methodologies, such as pseudo complement sparse carry chain and EMODL sum unit, are explained, and a conventional 32-bit CLA is used as a comparison target design. Layout and HSPICE simulation results of the new 32-bit CLA are presented, and the comparison confirms that our CLA offers better performance than the best existing published design.

In chapter 5, further studies on optimizing the area-time characteristics of the 32-bit CLA are conducted. Elements that influence the CLA performance are examined carefully. Four 32-bit CLA designs, with different configurations, are constructed based on varying

these elements over a reasonable range. Critical paths of the four CLAs are simulated for experiment verification. Comparison results obtained from the simulations provide us with an empirical guide to optimal CLA design.

Chapter 6 summaries the main contributions of this thesis and suggests some directions for future research.

Schematics associated with layout examples of all the building blocks of our 32-bit adder are given in Appendix A. The schematics are also labeled with transistor sizes. A automated optimization procedure is proposed in Appendix B. A preliminary C program for gate delay minimization is given in Appendix C.

Chapter 2

Architecture Discussions

2.1 Introduction

Addition is one of the most important arithmetic operations performed by both general-purpose and signal-processing microprocessor systems. Implementation of high-speed adders is demanded and motivated by the progress in VLSI technology. High-speed adders can be designed using carry-lookahead algorithms, carry-skip mechanisms, carry-select structures or various combinations. In this chapter, we shall look into some typical implementations of word-wide adders as follows:

1) carry-ripple adder [37];
2) carry-lookahead adder [6];
3) carry-skip adder [9];
4) carry-select adder [1] and
5) hybrid adder [21];

The carry-ripple adder is a straightforward implementation of adders, but it is slow in operation. Its long carry propagation path extends from the least significant bit position to the most significant bit, and so the ripple-carry generation delay is linearly proportional to the size of the adder. The carry-lookahead adder improves the

performance of the carry-ripple adder by generating the slow signals, i.e. carries, earlier than the evaluation of the sum. The carry-skip adder improves the performance of the carry-ripple adder by making the carries available earlier under certain conditions, trading the available time against resources. In the carry-select adder, the sums with predefined carries are duplicated, at the expense of additional resources, to reduce the number of levels in the adder. The hybrid adder combines two or more types of the above adder designs to generate better trade-offs between speed and hardware requirements.

Both carry-lookahead and carry select adders have $O(\log n)$ time complexity, but carry-select adders require more actual hardware. This is because carry-select adders have to precompute the sum results for all possible carry bit values, i.e., 1 and 0, and use the carry from the previous bit position to choose the proper result; thus the speed advantage is a trade off driven by the replication of carry select sections. When the word length, n, of the adder is larger than a certain bit number, carry-lookahead adders are faster than carry-skip adders since carry-skip adders exhibit a total delay of $O(\sqrt{n})$. The carry-lookahead architecture is chosen in this work because it is considered to be among the fastest circuit topologies for performing addition, and its size is relatively small. In this chapter, after brief discussions of the other adder architectures, we will describe the carry lookahead algorithm in more detail, and then discuss the recently introduced concept of "pseudo complements" [35] which is beneficial to improving the CLA performance.

2.2 Typical Adder Architectures

2.2.1 Carry-Ripple Adder

An N-bit carry-ripple adder consists of N full adders linked together by the carry inputs and outputs. This is the smallest possible parallel adder. However, it is also the slowest. This is because the carries are connected in series, and the addition will not finish until a carry generated by the least significant full adder ripples through $N-2$ intermediate full adders to the most significant full adder. As an example, the block diagram of a 4-bit

carry-ripple adder is shown in Figure 2.1. This adder takes $(a_3, \ldots, a_0)$, $(b_3, \ldots, b_0)$ and carry-in c_0 as inputs, and produces the sum $(s_3, \ldots, s_0)$ and carry output c_4. The i-th stage has inputs a_i, b_i, and c_i (the carry-in) so that the basic addition algorithms for the i-th bit position are given by

$$s_i = a_i \oplus b_i \oplus c_i \tag{2.1}$$

$$c_{i+1} = a_i b_i + (a_i \oplus b_i)\, c_i = a_i b_i + (a_i + b_i)\, c_i \tag{2.2}$$

where c_{i+1} is the carry-out bit at the i-th stage. The cascaded carry scheme used in connecting the full-adder circuits introduces significant delay into the circuit. For example, the third bit sum output s_2 is not valid until c_2 is valid, which in turn needs the carry bit c_1 generated by the 0-th bit position.

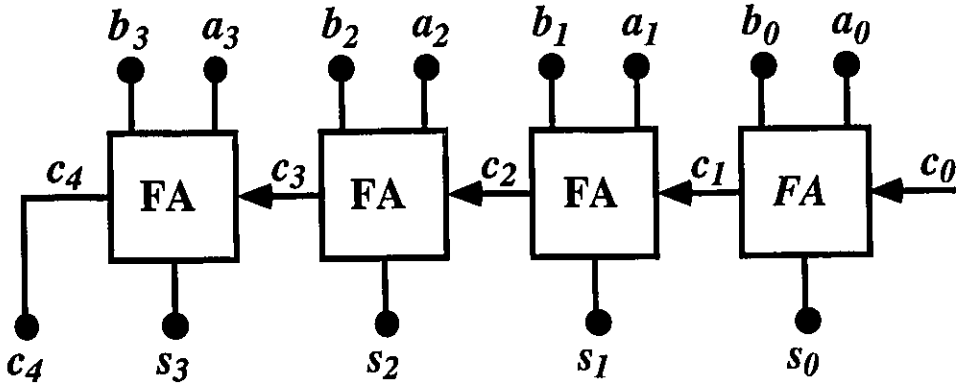

Figure 2.1 4-Bit Carry-Ripple Adder

2.2.1.1Full-Adder Designs

By expanding and reformulating eqn. (2.1) and eqn. (2.2), one can get the following results:

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i \tag{2.3}$$

$$\bar{c}_{i+1} = \bar{a}_i \bar{b}_i + \bar{a}_i \bar{c}_i + \bar{b}_i \bar{c}_i \tag{2.4}$$

$$s_i = a_i b_i c_i + a_i \bar{b}_i \bar{c}_i + \bar{a}_i \bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i \tag{2.5}$$

By substituting eqn. (2.4) into eqn. (2.5), sum s_{i+1} can be re-written as:

$$s_i = a_i b_i c_i + (a_i + b_i + c_i)\,\bar{c}_{i+1} \tag{2.6}$$

A full-adder schematic according to eqn. (2.3) and eqn. (2.6) is given in Figure 2.2. By simple optimization of the p-channel transistors, a full adder schematic with the same function yet different configuration is shown in Figure 2.3. The interesting feature of this circuit is that the pFET logic block identical topology as the nFET block, which tends to simplify and reduce the layout design than of Figure 2.3 compared to Figure 2.2. In addition, the full adder in Figure 2.3 is faster due to a lower height in the pFET chain. Both full adders consist of 28 transistors with buffered s_i outputs.

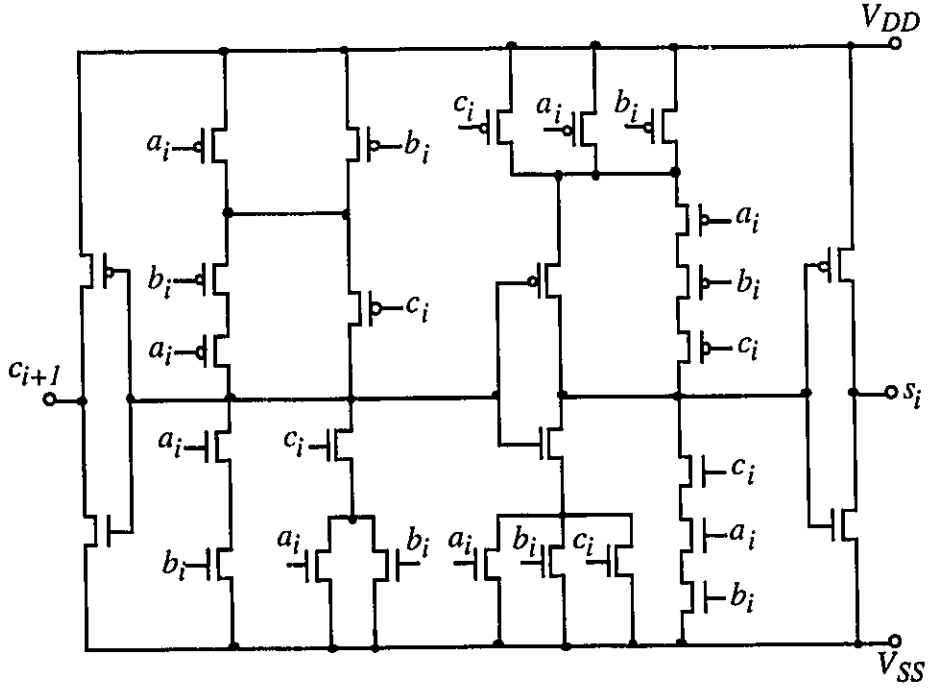

Figure 2.2 Full Adder Using Standard CMOS Gate

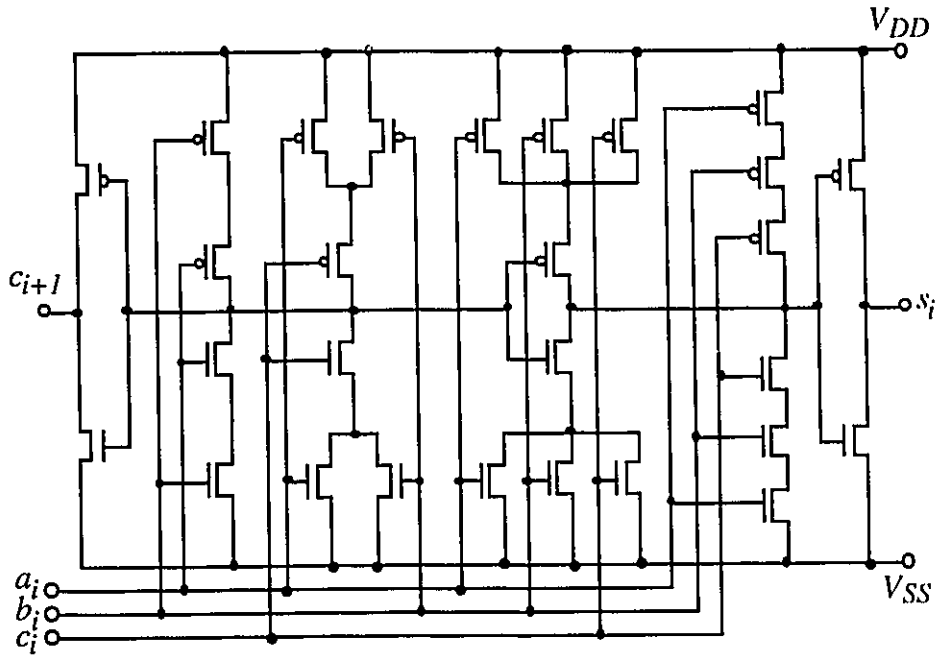

Figure 2.3 Reconfigured Full Adder

A rather different full adder configuration employes a transmission gate to accomplish the exclusive-or (*XOR*) function [23]. This adder configuration is illustrated in Figure 2.4. The adder has 24 transistors which is less than the ones shown in Figure 2.2 and Figure 2.3, and has the advantage of having equal s_i and c_{i+1} delay times.

Modification of the transmission gate full adder is given in Figure 2.5. This adder was discussed in [38] and implements the same function as the full adder shown in Figure 2.4 but with fewer transistors.

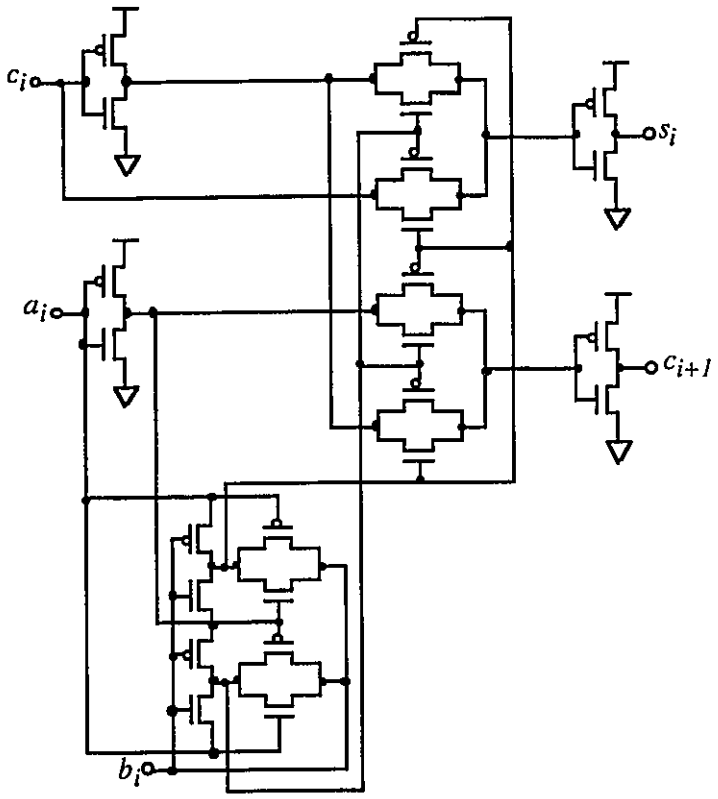

Figure 2.4 Full Adder Using Transmission Gate

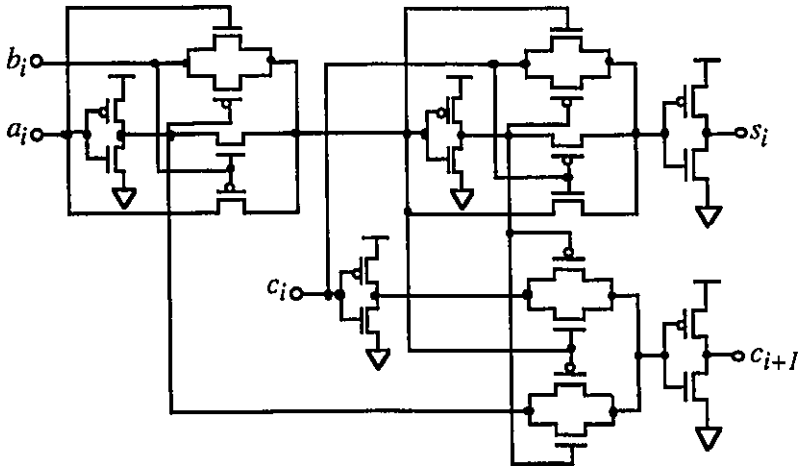

Figure 2.5 Modified Transmission Gate Full Adder

2.2.2 Carry-Lookahead Adder

Carry-lookahead adder (CLA) algorithms are designed to give "look-ahead" capabilities which provide the carry-in bits to each stage without waiting for each bit evaluation. To see the basic idea, we define a carry generate function

$$g_i = a_i b_i \tag{2.7}$$

and a carry propagate function

$$p_i = a_i \oplus b_i \tag{2.8}$$

so that the carry-out bit is given by

$$c_{i+1} = g_i + p_i c_i \tag{2.9}$$

applying this to the 4-bit adder gives the carry-out bits as

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 (g_0 + p_0 c_0)$$

$$c_3 = g_2 + p_2 (g_1 + p_1 (g_0 + p_0 c_0))$$

$$c_4 = g_3 + p_3 (g_2 + p_2 (g_1 + p_1 (g_0 + p_0 c_0)))$$

Adding a CLA unit to the parallel adder as shown in Figure 2.6 greatly increases the speed of the network. The circuit of Figure 2.7 is a 4-bit carry lookahead unit with 3 level c_i logic.

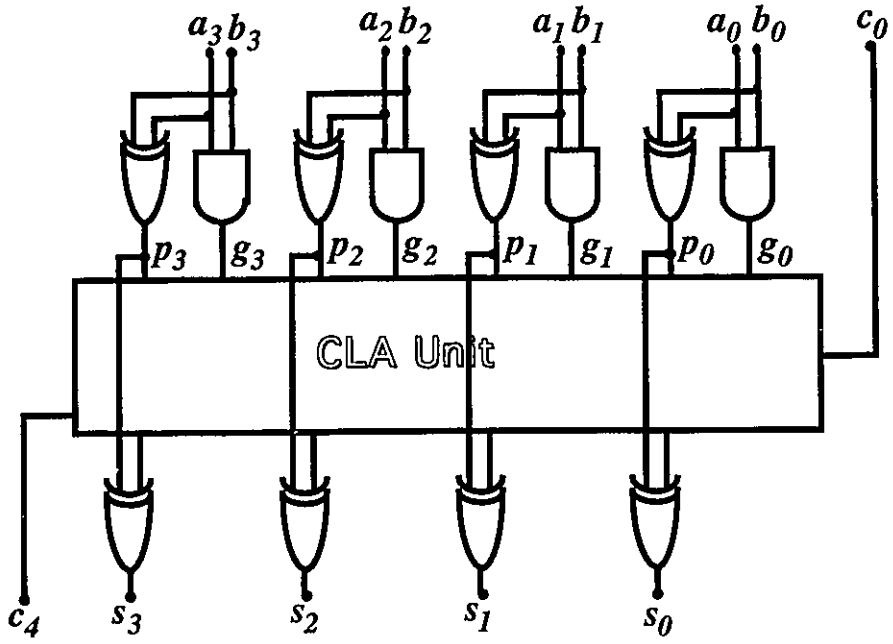

Figure 2.6 Block Diagram of 4-Bit Carry-Lookahead Adder

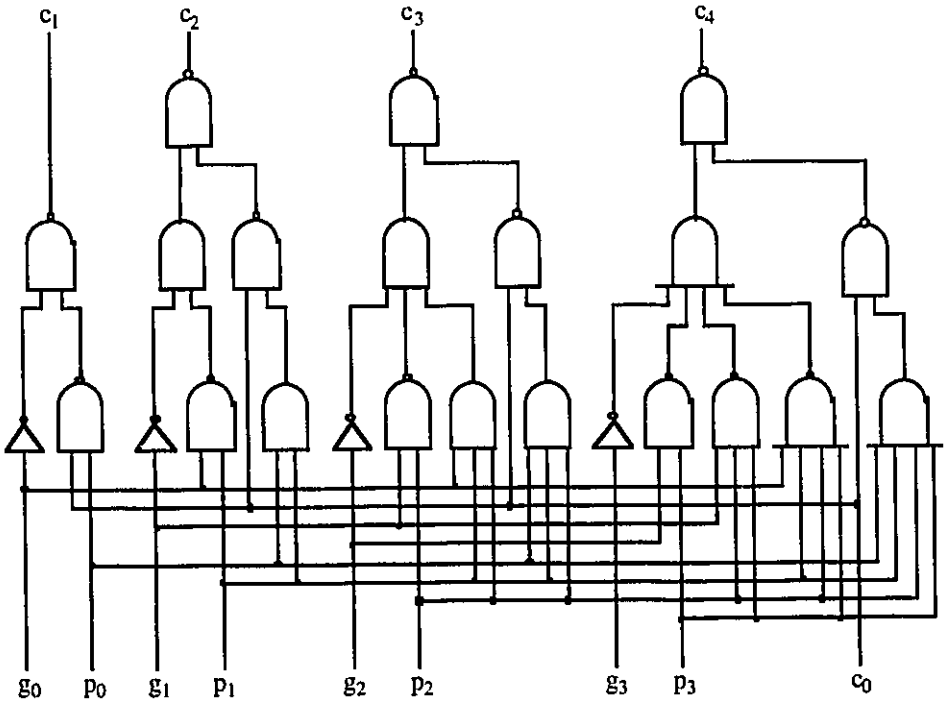

Figure 2.7 4-Bit Carry-Lookahead Unit

2.2.3 Carry-Skip Adder

The carry-skip adder uses the idea that if corresponding bits in the two words to be added are not equal, a carry-in signal of that bit position will be passed to the next bit position. Using the two N-bit binary operands, A= $(a_n, \ldots, a_0)$ and B= $(b_n, \ldots, b_0)$, the carry input and carry output at bit position i as c_i and c_{i+1} respectively, and the sum at the bit position i as s_i, the following relations for s_i and c_{i+1} are well-known:

$$s_i = a_i \oplus b_i \oplus c_i \tag{2.10}$$

$$c_{i+1} = a_i b_i + (a_i + b_i)\, c_i \tag{2.11}$$

Carry-skip adders are best understood by first looking at a carry-ripple adder. In this adder a carry signal may have to propagate all the way from the least significant position to the most significant position. An example would be the case where $\forall i$ except 0, a_i=0, a_0=1, and $\forall i$, b_i=1. Note that in every bit position through which the carry signal travels, a_i and b_i are different. This is a significant fact because, in general, a carry signal will propagate through a bit position i if and only if $a_i \neq b_i$. In another words, if $a_i = b_i$, then all carry signals to the left of position i are independent of all inputs in positions 0 through $i-1$. Thus, the chances that a carry signal travels a long way is small, because the probability of having a long string of bit positions all with $a_i \neq b_i$ is small. The carry-skip adder takes advantage of this observation. The bit positions are divided into blocks of contiguous positions, as shown in the example of Figure 2.8. The blocks are numbered 0, 1, 2 and 3 from right to left. When a block has all the positions i with $a_i \neq b_i$, such as block 2 in Figure 2.8, the carry-out of that block is the same as the carry-in to the block. Also the carry input to the bit positions in block 2 are either all 0's or all 1's, depending on whether the carry input to the least significant bit of the block is a 0 or a 1, respectively. Thus no carry signal has to propagate through this block in a position-by-position fashion.

A	10100	01011	10100	01011
B	01101	10100	01010	01100
	block 3	block 2	block 1	block 0

Figure 2.8 Input Data Example for Carry-Skip Mechanism

There are two general types of carry-skip adders: constant block width, as illustrated in Figure 2.8, and variable block width. For a variable block width carry-skip adder, multiple levels of carry-skip mechanism and variable block sizes are used in order to create a faster circuit. For example, Figure 2.9 illustrates a basic structure of a two-level constant block width carry-skip adder. This stage consists of three adder blocks and a second level carry-skip mechanism. The carry skip can be implemented with a multiplexor selected by the group propagate. Each adder block has five full adders grouped together and a first-level carry-skip mechanism. In variable block width carry-skip adder, the number of full adders in the adder blocks can be different, and the number of adder blocks in each stage can also vary in order to achieve fast carry-skip speed.

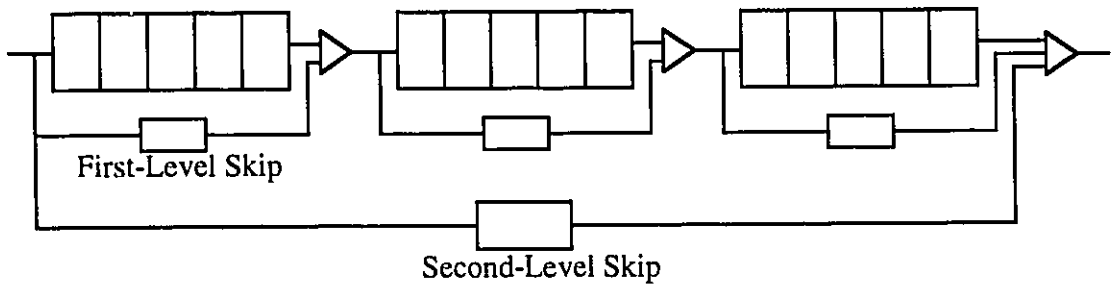

Figure 2.9 Two Level Carry-Skip Scheme

Recently carry-skip adders, using a Manchester adder with carry skip mechanism, has become popular. Figure 2.10 shows a 4-bit Manchester carry chain implemented in dynamic CMOS logic. This adder circuit operates as follows: the nodes are precharged

during phase $\overline{ck}$ through p-channel transistors, and the circuit is evaluated during phase ck. Depending on the values of a_i, b_i and c_i, a carry can either be generated or propagated. Theoretically, a Manchester carry chain adder is a carry-lookahead chain implemented by dynamic logic. This is because a Manchester adder also uses **generate** and **propagate** terms to produce a sum and carry as shown in Figure 2.10. Again, carry generate and carry propagate signals are defined as:

$$g_i = a_i b_i \qquad p_i = a_i \oplus b_i$$

So that sum s_i and carry c_{i+1} are determined by

$$s_i = p_i \oplus c_i$$

$$c_{i+1} = g_i + p_i c_i$$

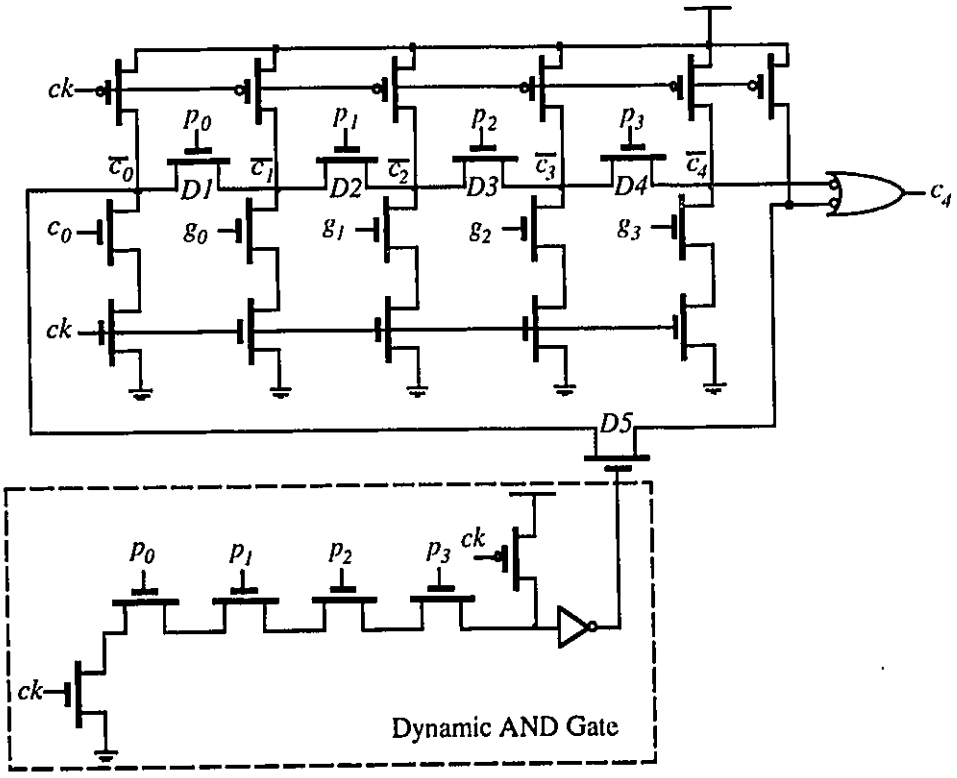

Figure 2.10 4-Bit Manchester Adder with Carry-Skip

The difference between the dynamic carry-lookahead adder and the carry-skip adder, using a Manchester carry chain, is the attachment of a bypass transistor $D5$ and an additional dynamic AND gate (carry-skip circuitry) which turns on $D5$ if all carry propagate signals are true. Note that the critical path of the Manchester adder is the series connected pass transistors $D1 - D4$ in the carry chain. Although the carry-skip circuitry has a similar construction to the carry chain $D1 - D4$, the node capacitance at intermediate nodes is approximately half that of the carry chain. Thus the carry-skip arrangement improves the worst case carry propagation time if all carry propagates are true; this, in turn, improves the overall speed of the adder.

2.2.4 Carry-Select Adder

Another approach to building fast adders that trades-off area for speed is to use a carry-select adder. The basic scheme is shown in Figure 2.11. Two carry-ripple adder structures are built, one with a zero carry-in and the other with a one carry-in. This is repeated for a certain sized adder, for instance 4-bits. The previous carry then selects the appropriate sum using a multiplexor or tri-state adder gates. The stage carries and the previous carry are gated to form the carry for the succeeding stage.

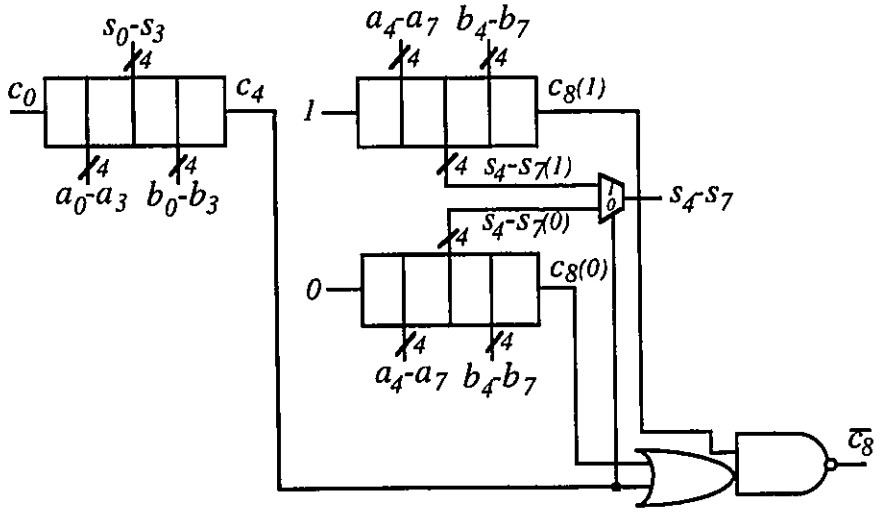

Figure 2.11 8-Bit Carry-Select Adder

2.2.5 Hybrid Adder

The above discussion reveals that different types of adder implementations have their own advantages and disadvantages in operation speed and hardware expense. Hybrid adders tend to combine two or more types of the above adders for maximizing advantages while minimizing the disadvantages. For example, the 32-bit adder used as the 16x16 multiplier accumulator in the ADSP-2105 DSP device (developed by Analog Devices Inc.) is a hybrid adder employing a carry-skip scheme combined with a carry-select adder. In [21], another hybrid adder is developed which utilizes carry-lookahead and carry-select structures for obtaining a balanced load distribution and regular layout, with the expected increase in hardware resources.

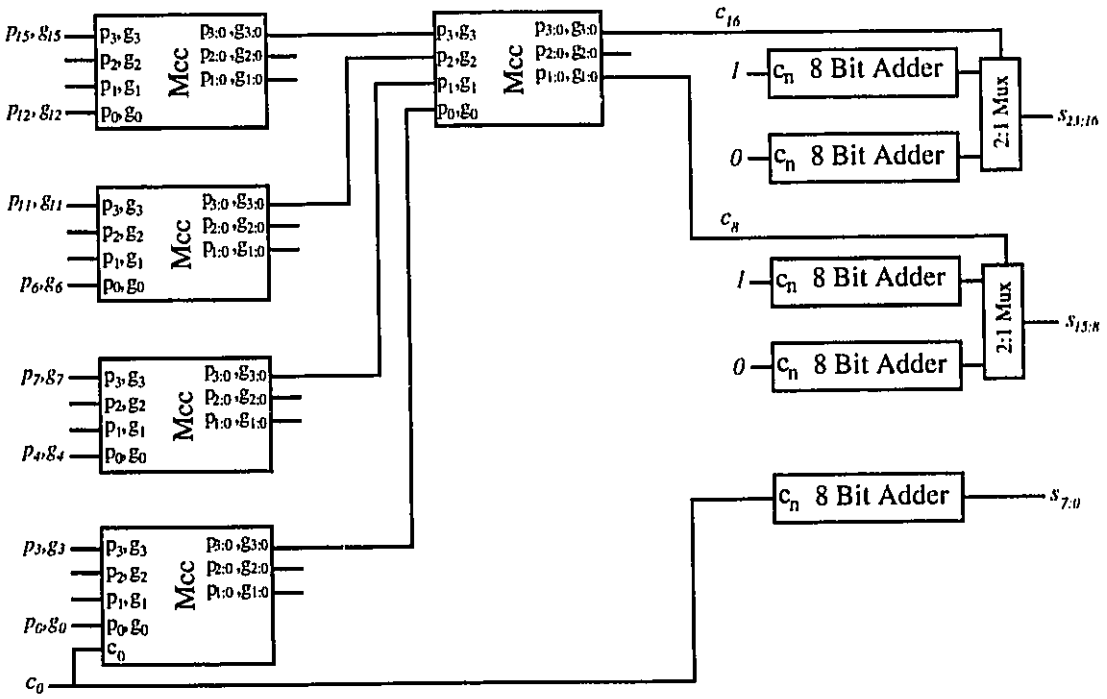

Figure 2.12 24-Bit Hybrid Adder

Figure 2.12 shows the block diagram of a 24-bit hybrid adder using combined carry-lookahead and carry-select structures. The carry-lookahead logic is realized by a Manchester carry chain, denoted as Mcc in the block diagram. Each Mcc block contains

four bits and produces carries on the boundaries of 0, 8 and 16. Therefore the uniform carry select section is composed of an 8 bit-wide carry-ripple adder. Eight bit boundaries prove to be convenient because the eight bit carry-ripple adder operates in slightly less time than the carry tree, thus the sum data are set up at the selection multiplexers just before the select signals arrive.

2.3 CLA and the Pseudo Complement Concept

2.3.1 Carry Lookahead Algorithm

Referring to the discussion of basic CLA algorithm in Section 2.2.2, the arithmetic description of the basic organization of an n-bit CLA can be stated as follows:

Given an n-bit adder with two binary summands $(a_n, ..., a_0)$ and $(b_n, ..., b_0)$, and a carry-in c_0, we first define three auxiliary logic variables: carry generate g_i; carry propagate p_i; and half sum x_i, as follows:

$$g_i = a_i b_i \tag{2.12}$$

$$p_i = a_i + b_i \tag{2.13}$$

$$x_i = a_i \oplus b_i \tag{2.14}$$

a_i, b_i and c_i are the inputs at the i-th bit position, p_i denotes that a carry will propagate across bit position i, and g_i denotes that a carry is generated at bit position i. The final sum s_i and carry c_{i+1} can be expressed as:

$$s_i = x_i \oplus c_i \tag{2.15}$$

$$c_{i+1} = g_i + p_i c_i \tag{2.16}$$

An alternative definition of p_i, as shown in eqn. (2.17), also frequently appears in the literature. (We refer to it as *XOR*-p_i in later discussions, and refer to eqn. (2.13) as *OR*-p_i.)

$$p_i = a_i \oplus b_i \tag{2.17}$$

In such case the half sum variable x_i is no longer necessary to produce the final sum s_i and can be replaced by p_i. Therefore, the final sum s_i and carry c_i are computed as:

$$s_i = p_i \oplus c_i \tag{2.18}$$

$$c_{i+1} = g_i + p_i c_i \tag{2.19}$$

Preference of either definition of p_i depends on the adder algorithm chosen and the design criteria. A detailed comparison of the two will be given in Chapter 5.

Continuing with the definition, an operator "*o*", which is introduced by Ladner and Fischer [18], is defined as follows:

$$(g_i, p_i) o (g_j, p_j) = (g_i + p_i g_j, p_i p_j) \tag{2.20}$$

Operator "o" is associative [6], and can be proved by the following:

For any (g_{i+2}, p_{i+2}), (g_{i+1}, p_{i+1}) and (g_i, p_i), we have

$$\begin{aligned} & [(g_{i+2}, p_{i+2}) o (g_{i+1}, p_{i+1})] o (g_i, p_i) \\ &= [(g_{i+2} + p_{i+2} g_{i+1}, p_{i+2} p_{i+1})] o (g_i, p_i) \\ &= [(g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i), (p_{i+2} p_{i+1} p_i)] \end{aligned}$$

and

$$(g_{i+2}, p_{i+2}) o [(g_{i+1}, p_{i+1}) o (g_i, p_i)]$$

$$= (g_{i+2}, p_{i+2}) o [(g_{i+1} + p_{i+1}g_i, p_{i+1}p_i)]$$

$$= [(g_{i+2} + p_{i+2}g_{i+1} + p_{i+2}p_{i+1}g_i), (p_{i+2}p_{i+1}p_i)]$$

Therefore,

$$[(g_{i+2}, p_{i+2}) o (g_{i+1}, p_{i+1})] o (g_i, p_i)$$

$$= (g_{i+2}, p_{i+2}) o [(g_{i+1}, p_{i+1}) o (g_i, p_i)]$$

Note that operator "o" is not commutative. This is easily seen in eqn. (2.20); the left argument (g_i,p_i) on the left side of the equation is treated differently from the right argument (g_j, p_j). If $p_j = 0$, we define the operator "o" as

$$(g_i, p_i) o (g_j, 0) = (g_i, p_i) o g_j = g_i + p_i g_j \tag{2.21}$$

Given the fact that operator "o" is associative, we can choose a k and we define two more logic variables "group carry generate" $gg_{i,k}$ and "group carry propagate" $gp_{i,k}$. Then the pair $(gg_{i,k}, gp_{i,k})$ can be formulated as:

$$(gg_{i,k}, gp_{i,k}) = (g_{i+k}, p_{i+k}) o (g_{i+k-1}, p_{i+k-1}) o \ldots o (g_i, p_i) \tag{2.22}$$

where i is the starting bit position of the group and k indicates that the length of the group is $k+1$. $gp_{i,k}$ signifies that a carry will propagate from bit i to bit $i+k+1$. Similarly $gg_{i,k}$ denotes that a carry is generated in at least one of the bit positions from i to $i+k+1$ inclusive and propagated to bit position $i+k+1$.

From eqn. (2.19) to (2.22) we get

$$c_{i+1} = (g_i, p_i) o c_i \tag{2.23}$$

$$= (g_i, p_i)\, o\, (g_{i-1}, p_{i-1})\, o \ldots o\, (p_0, g_0)\, oc_0$$

and

$$c_{i+k+1} = (gg_{i,k}, gp_{i,k})\, oc_i \tag{2.24}$$

It is worth mentioning that both expressions of p_i can be applied to eqn. (2.20) to (2.24). However, different CLA structures will be generated from different choices of the p_i definition. In the case where *XOR* logic is applied to the carry propagate, as defined in eqn. (2.17), the generation of p_i for each pair of input summands is required. This stage, however, is not required when using *OR*-p_i logic. The disadvantage with the latter approach is the potential for false evaluation caused by sneak paths that can occur in MODL circuit implementations. This problem will be discussed in Section 3.3.2.

2.3.2 Pseudo Complement Concept

Using the *XOR* function in sum generation requires both true and complement signals for x_i and c_i. In a static CMOS logic implementation, the complements can be produced by adding an inverter to the original variable. In domino logic design, the complement result can not be produced in the same way. Since we are considering full domino logic design for the adder implementation, we have to create both the true and complement signals for x_i and c_i using separate domino logic circuits prior to the evaluation of the sum s_i. The circuits for the complements of $gg_{i,k}$ and $gp_{i,k}$ may need more transistors and greater fan-in, and thus more silicon area and critical path delay, than their true counterparts. This fact will be illustrated in detail in Section 4.3.2 of Chapter 4. In order to generate the complement carry chain in parallel with the true carry chain, while retaining the same circuit delay and silicon area, we define a pseudo complement generate $\hat{g}_i = \overline{a_i}\,\overline{b_i}$, and pseudo complement propagate $\hat{p}_i = \overline{a_i} + \overline{b_i}$, or $\hat{p}_i = \overline{a_i} \oplus \overline{b_i} = a_i \oplus b_i$ [35]. Note that $\hat{g}_i$ is not the real complement of g_i, since $\bar{g}_i = \overline{a_i b_i} = \bar{a}_i + \bar{b}_i$, and $\hat{p}_i$ is not the real complement of p_i for $\bar{p}_i = \overline{a_i + b_i} = \bar{a}_i \bar{b}_i$, or $\bar{p}_i = \overline{a_i \oplus b_i}$.

When the operator "o" applies to two pairs of pseudo complement generate and propagate, it is defined as:

$$(\hat{g}_i, \hat{p}_i)\, o\, (\hat{g}_j, \hat{p}_j) \;=\; (\hat{g}_i + \hat{p}_i\hat{g}_j, \hat{p}_i\hat{p}_j) \tag{2.25}$$

Then the pseudo-complement group generate $\hat{gg}_{i,k}$, and the pseudo-complement group propagate $\hat{gp}_{i,k}$ are generate in a fully parallel way:

$$(\hat{gg}_{i,k}, \hat{gp}_{i,k}) \;=\; (\hat{g}_{i+k}, \hat{p}_{i+k})\, o\, (\hat{g}_{i+k-1}, \hat{p}_{i+k-1})\, o \ldots o\, (\hat{g}_i, \hat{p}_i) \tag{2.26}$$

The complement carry chain can be derived from $\hat{g}_i$ and $\hat{p}_i$ in the same manner as the carry chain is derived from g_i and p_i.

$$\bar{c}_i \;=\; (\hat{g}_i, \hat{p}_i)\, o\bar{c}_{i-1} \tag{2.27}$$

$$=\; (\hat{g}_i, \hat{p}_i)\, o\, (\hat{g_{i-1}}, \hat{p_{i-1}})\, o \ldots o\, (\hat{p}_0, \hat{g}_0)\, o\overline{c_0}$$

and

$$\bar{c}_{i+k+1} \;=\; (\hat{gg}_{i,k}, \hat{gp}_{i,k})\, o\bar{c}_i \tag{2.28}$$

Note that substituting a_i and b_i with $\bar{a}_i$ and $\bar{b}_i$ to either *XOR*-p_i or *OR*-p_i also gives different types of pseudo p_i, i.e. *XOR*-$\hat{p}_i$ or *OR*-$\hat{p}_i$. Either one will function correctly for computing the complement carries. The proof is presented in the following section.

2.3.3 Proof of Correctness for the Pseudo Complement Concept

Type I: p_i Using *OR* Logic

Given: $\quad g_i = a_ib_i \quad p_i = a_i + b_i \quad c_{i+1} = (g_i, p_i)\, oc_i$

Define: $\quad \hat{g}_i = \bar{a}_i\bar{b}_i \quad \hat{p}_i = \bar{a}_i + \bar{b}_i$

Prove: $\bar{c}_{i+1} = (\hat{g}_i, \hat{p}_i) o \bar{c}_i$

Proof: $c_{i+1} = (g_i, p_i) o c_i$

$= g_i + p_i c_i$

$= a_i b_i + (a_i + b_i) c_i$

$\therefore \bar{c}_{i+1} = \overline{a_i b_i + (a_i + b_i) c_i}$

$= (\bar{a}_i + \bar{b}_i) \cdot (\bar{a}_i \bar{b}_i + \bar{c}_i)$

$= \bar{a}_i \bar{b}_i + (\bar{a}_i + \bar{b}_i) \bar{c}_i$

$= \hat{g}_i + \hat{p}_i \bar{c}_i$

$= (\hat{g}_i, \hat{p}_i) o \bar{c}_i$

Type II: p_i Using *XOR* Logic

Given: $g_i = a_i b_i \quad p_i = a_i \oplus b_i \quad c_{i+1} = (g_i, p_i) o c_i$

Define: $\hat{g}_i = \bar{a}_i \bar{b}_i \quad \hat{p}_i = \bar{a}_i \oplus \bar{b}_i$

Prove: $\bar{c}_{i+1} = (\hat{g}_i, \hat{p}_i) o \bar{c}_i$

Proof: Let us prove $(\hat{g}_i, \hat{p}_i) o \bar{c}_i$ is actually $\bar{c}_{i+1}$.

$(\hat{g}_i, \hat{p}_i) o \bar{c}_i = \hat{g}_i + \hat{p}_i \bar{c}_i$

$= \bar{a}_i \bar{b}_i + (\bar{a}_i \oplus \bar{b}_i) \bar{c}_i$

$= \bar{a}_i \bar{b}_i + (\bar{a}_i b_i + a_i \bar{b}_i) \bar{c}_i$

$= \bar{a}_i \bar{b}_i + \bar{a}_i \bar{c}_i + \bar{b}_i \bar{c}_i = \bar{a}_i \bar{b}_i + (\bar{a}_i + \bar{b}_i) \bar{c}_i$

Now look at $\bar{c}_{i+1}$:

Since: $c_{i+1} = (g_i, p_i) o c_i$

$= g_i + p_i c_i$

$$= a_i b_i + (a_i \oplus b_i) c_i$$

$$\therefore \bar{c}_{i+1} = \overline{a_i b_i + (a_i \oplus b_i) c_i}$$

$$= (\bar{a}_i + \bar{b}_i) \cdot (\bar{a}_i \bar{b}_i + \bar{c}_i)$$

$$= \bar{a}_i \bar{b}_i + (\bar{a}_i + \bar{b}_i) \bar{c}_i$$

$$\therefore \bar{c}_{i+1} = (\hat{g}_i, \hat{p}_i) o \bar{c}_i$$

2.4 Summary

Five types of adder structures have been reviewed. The implementation and comparison work of these five types, using 16-bit adders, was published by Callaway and Swartzlander [7]. These adders are carry-ripple adder, constant block width carry-skip adder, variable block width carry-skip adder, carry-lookahead adder, and carry-select adder. The performance comparisons of these adders are summarized in Table 2.1

Adder Type	Estimate Worst Case Delay (Gate Unit)	Simulated Worst Case Delay (ns)	Number of Gates	Adder Size $\left(mm^2\right)$
Carry-Ripple	36	54.27	144	0.2527
Constant Width Carry-Skip	23	28.38	156	0.4492
Variable Width Carry-Skip	17	21.84	170	0.5149
Carry-Looka-head	10	17.13	200	0.7454
Carry-Select	14	19.56	284	1.0532

Table 2.1. Area-Speed Characteristics of Five 16-Bit Adders

Table 2.1 shows that carry-lookahead is the fastest, but its size is relatively large. This can be improved by utilizing advanced circuit implementation techniques. We also learn,

from the table, that the relative adder sizes are consistent with the number of gates. In another words, comparing the gate counts of adder designs may be helpful in judging relative sizes before the actual layout implementation.

Knowing that the carry-lookahead algorithm offers the fastest adder architecture, and with the help of pseudo complement techniques in mapping complement carry chains, we are confident that the speed of the adder we develop will be superior to other adder designs.

We now invest our effort in reducing the cost of the adder by exploiting high performance circuit design methods.

Chapter 3

Circuit Implementation Techniques

3.1 Introduction

Circuit design is equally important as algorithm and architecture design in achieving high performance systems. Since CMOS technology is still in a dominant position in most IC products, we examine possible CMOS circuit alternatives to find a solution for high speed adder design.

Static CMOS is the most widely used CMOS circuit style because it gives full-rail output voltage swing and exhibits high noise immunity. Also it provides the basis for more advanced dynamic circuit designs. The shortcoming of static CMOS circuit is that it uses both nMOS and pMOS transistors in equal quantities, with typically larger sized pMOS devices. This leads to large charge movement during transitions, slowing down the circuitry and increasing the power dissipation.

An N-input static CMOS gate thus requires 2N MOSFETs, which is twice the number needed to perform the logical operation. An N input pseudo nMOS gate only requires N+1 transistors which may be useful to consider when layout problems start to overwhelm the need to maintain low power dissipation; this type of gate produces

static power dissipation in addition to dynamic dissipation during switching. Other drawbacks of pseudo nMOS circuits are that the logic swing $V_{OH} - V_{OL}$ is always less than V_{DD}, and ratioed transistor sizes are usually needed to achieve acceptable output voltages. This places constraints on the layout design.

Dynamic logic circuits perform logic operations using the properties of capacitive charge storage nodes. Clock signals are required to predefine charge states and to evaluated the logic function. The main advantage to using dynamic CMOS logic circuits is that they are generally faster and more compact than equivalent static gates. However, the operation is more sensitive to layout geometries, and the problems of charge leakage and charge sharing can become critical. Also glitches will occur when single phase dynamic CMOS gates are connected in series.

Domino logic [17] is designed to provide glitch-free cascades of nMOS logic blocks by adding a static inverter to a basic dynamic circuit. Any number of logic gates may be cascaded and operated by a single phase clock, provided that the sequence can evaluate within the evaluation clock phase. The limitation of domino logic is that only non-inverting structures are possible. Also, charge sharing can be a problem, which is common to all the clocked-CMOS circuits.

Zipper logic [20] is designed in a way that both nMOS and pMOS dynamic circuits are used. Four clocks are grouped in two pairs. Each pair is used for one type of dynamic circuit. The two clocks in each pair are in phase but have different amplitudes in order to diminish charge sharing problems. The two types of dynamic circuits (n-p) are serially connected one type after the other resembling a zipper structure. Problems that occur with this type of logic include poor speed response of the pMOS blocks and the extra effort involved in accomplishing four clock controls.

Differential cascode voltage switch logic (CVSL) [11] combines complementary logic arrays with a differential latching circuit. Usually a cross-coupled pMOS latch circuit is

used. The circuit has the advantage of providing complementary logic functions. However, the sizing of the latching circuit can be highly influential to the overall circuit speed. Also CVSL performance is not necessarily superior to a conventional static CMOS gate [31].

Multiple output domino logic (MODL) [15] provides more than one output, using a single circuit, where one output is a subfunction of the other. This approach increases the overall system speed by reducing the device count and gate interconnection complexity.

In the following sections we will take an in-depth look at the above mentioned circuit design techniques. Then we discuss a recently introduced CMOS gate family called enhanced multiple output logic (EMODL) [34] and [35]. EMODL extends the MODL style to a more general case, where a more complex logic gate can be built upon a common base which may not necessarily be a subfunction output. This technique provides a higher degree of device saving.

3.2 Typical CMOS Circuit Realization Approaches

3.2.1 Static CMOS Logic

Static CMOS logic gates are formed with completely symmetric nMOS and pMOS transistor arrays. The logic function of each gate is implemented twice. For example, a combinational gate that performs the AND/OR invert function having one 3-input AND logic OR with one 2-input AND, is shown in Figure 3.1. The five n-channel transistors have all the information to represent the required logic function and so do the five p-channel transistors. Using both arrays in a fully complementary CMOS circuit provides the advantage that, except for the very brief period when the output or the inputs are making logic transitions, there will be no current flowing through the circuit and no static power consumption.

The problem with this fully complementary approach is that for complex gates of the type shown in Figure 3.1, a substantial amount of area can be wasted, particularly considering the larger width p-MOS transistors required for symmetrical characteristics (the same function could be performed with six transistors in the pseudo-nMOS configuration shown in Figure 3.2 and discussed in the following section). As a result of the extra transistors, the capacitive load of the input gates of a fully complementary circuit are considerable higher than the loads of a pseudo nMOS circuit, since each gate connection goes to both a p-channel and an n-channel transistor. Taking into account the fact that p-channel transistors are generally two or three times the size of n-channel transistors, the total gate load of each connection will be three or four times higher than that of a pseudo nMOS circuit.

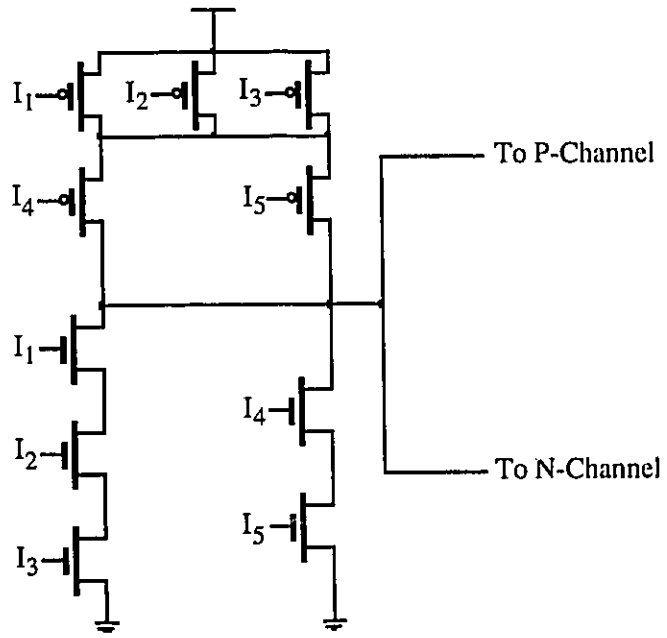

Figure 3.1 Static Combinational CMOS gate

3.2.2 Pseudo nMOS Logic

Pseudo nMOS logic uses only nMOS transistors to form the logic function, and a single pMOS as a load. The pseudo nMOS gate which represents the equivalent logic function as

the circuit of Figure 3.1 is shown in Figure 3.2. The pMOS load is grounded so that it is always in a conducting mode. As a result, the output voltage will never reach $0v$ when a logic low is supposed to be presented. This is typical of ratioed nMOS logic circuit where the gain ratio of the p-transistor load to n-driver transistors $\beta_{load}/\beta_{driver}$ sets V_{OL}.

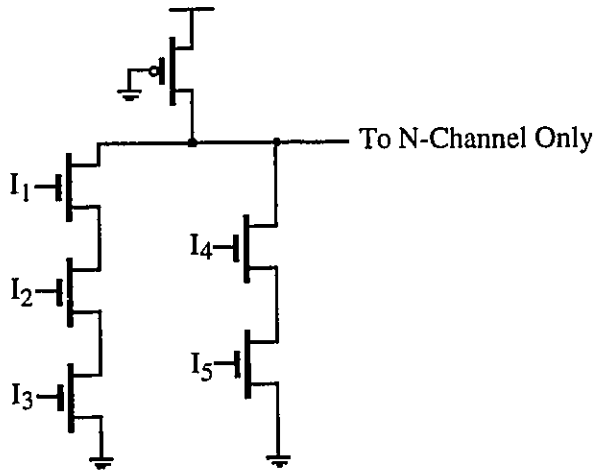

Figure 3.2 Pseudo nMOS Logic Gate

There are N+1 transistors in an N-input pseudo nMOS gate. Compared to a complementary gate, the capacitive load on each output is lower as a result of using only one transistor for each term of the input function. The density advantage pseudo nMOS gates have over fully complementary gates is obvious. There is, however, no real speed advantage over standard CMOS because pull-up current always flows caused by the grounded pMOS load, and this increases the pull-down time. Making the pull-up current very small does not solve this problem because then the pull-up time is large. As a result, the speed of CMOS and pseudo-NMOS are very close; CMOS has twice the capacitance but also twice the available current. The other problem with pseudo nMOS gates is the inevitable static power dissipation in the p-transistor load.

3.2.3 Dynamic CMOS Logic

The operation of a dynamic circuit is divided by the clock into two distinct phases: the **precharge** and **evaluate** intervals. For an N-input dynamic nMOS circuit, there will be N+2 transistors involved including N n-channel transistors for the logic function, one p-channel transistor for precharge and one n-channel transistor as a "ground switch" for enabling the evaluate interval. A dynamic CMOS gate with the same logic function as the static CMOS gate described in Section 3.2.1 is shown in Figure 3.3. CK is a single phase clock. $CK = 0$ defines the precharge interval when the p-channel transistor is conducting while the n-channel "ground switch" is turned off. The main purpose of the precharge is to allow the output to be charged to V_{DD} before the actual output value is determined. The input voltages are also accepted at this time. When the clock switches to $CK = 1$, the circuit enters the evaluate interval where the p-channel transistor is cutoff and the "ground switch" is active. A conditional discharge occurs depending on the logic levels of the inputs to the n-transistor chain.

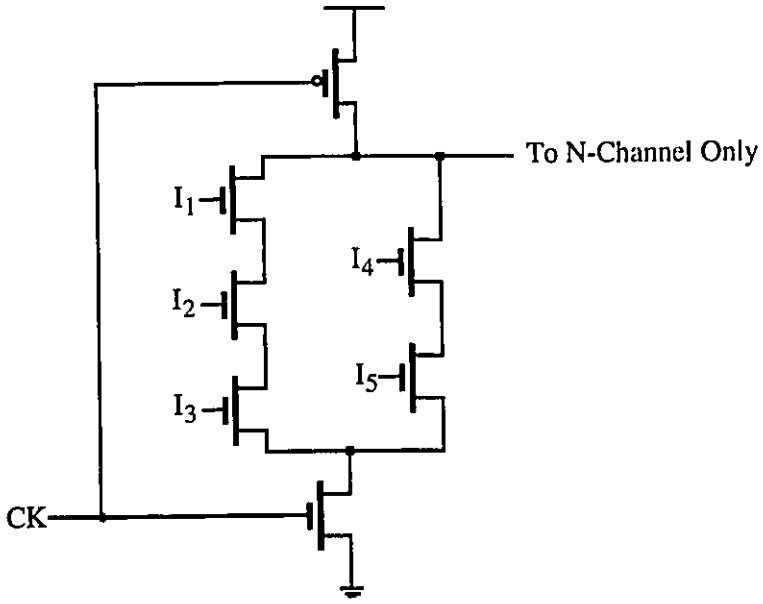

Figure 3.3 Dynamic nMOS Gate

The advantage of the dynamic circuit is that the load capacitance is the same as the pseudo nMOS gate but the full pull-down current is available. The pull-up time is improved by use of the "ground switch" and the pull-down time is also increased due to the same reason. The dynamic circuit is roughly twice as fast as either pseudo nMOS or full CMOS. In addition, there is no static current path so static power dissipation is much closer to CMOS than to pseudo nMOS. There is a dynamic power penalty compared to CMOS because each gate must be precharged high in every cycle, but this is often compensated for logic circuits with high transition rates by the reduced charge movement during these transitions.

There are inherent problems with dynamic circuits where several logic gates are connected in series. Firstly, input logic transitions can only be allowed during the precharge phase. If this condition is not met, charge redistribution effects can corrupt the output node voltage. Secondly, simple single phase dynamic CMOS gates can not be cascaded. When the gates are precharged, the output nodes are charged to V_{DD}. During the evaluate phase, the output of the first gate will conditionally discharge. However, some delay will be incurred due to the finite pull-down time. Thus the precharged node of the first stage can discharge the output node of the following gate before the first gate is correctly evaluated, which results in erroneous evaluation for the second gate.

This problem can be avoid by using the alternating polarity arrangement shown in Figure 3.4. Connecting nMOS outputs to the gates of pMOS transistors does not cause problems because the precharge voltage of V_{DD} forces a pMOS transistor to turn off. The same holds true for pMOS outputs connected to the gates of nMOS transistors. The main drawback of this type of circuit is that pMOS logic chains are slower than equivalent nMOS arrays. Unfortunately they can not be avoided.

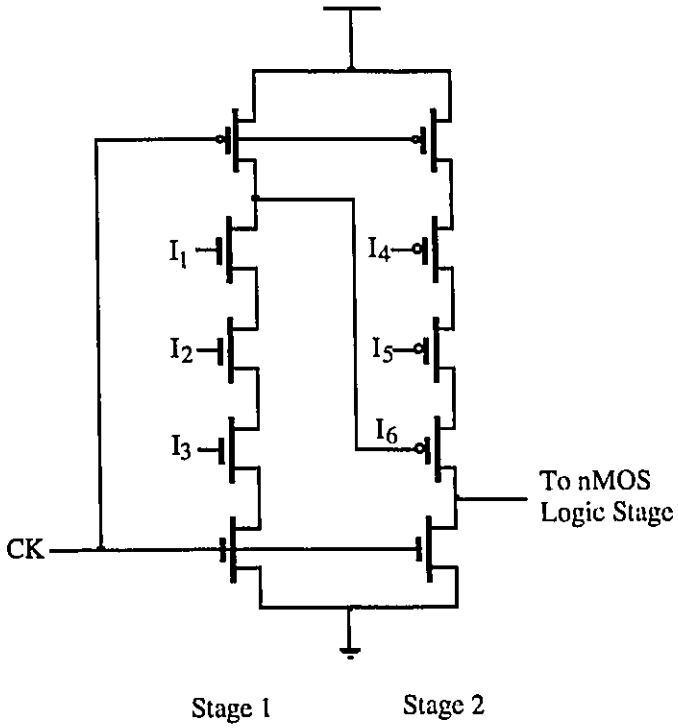

Figure 3.4 Correct Dynamic Cascades

3.2.4 CMOS Domino Logic

The CMOS Domino circuit shares the same characteristics as dynamic circuits. A corresponding domino circuit gate is shown in Figure 3.5. It consists of two parts. The first part looks the same as the dynamic pseudo nMOS gate in Figure 3.3 and is clocked in the same way, with a precharge phase followed by an evaluation phase. The second part is a static CMOS buffer. Only the output of the static buffer is fed to other gates of the circuit. All the above mentioned problems of dynamic circuits are solved by attaching this inverter buffer. This is because during precharge, all the domino gate outputs are set to low as the outputs of the dynamic chains are precharged high, therefore all the transistors that are driven by the domino gate outputs are turned off. In addition, during evaluation, a domino gate can make only a single transition, namely from low to high. Because of the nature of the dynamic gate which drives it, it is impossible for the buffer to go from high to low during evaluation. All nodes can make at most only one transition and then must

stay there until the next precharge. As a result there can be no timing glitches at any nodes in a row of domino gates.

Since there is no need to worry about timing glitches and since, during precharge, all domino outputs turn off the transistors they drive, all domino gates may be switched from precharge to evaluate with the same clock edge.

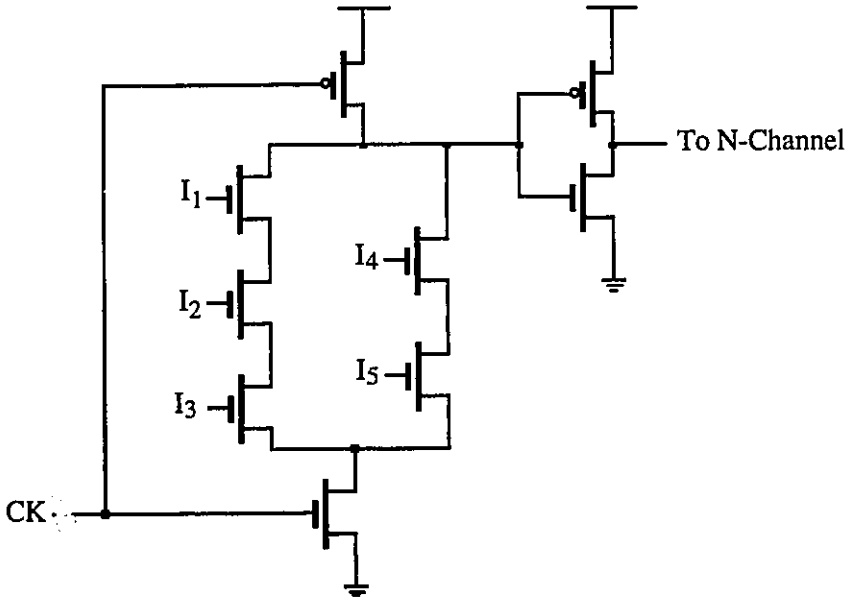

Figure 3.5 Domino Logic Gate

Many types of circuits when made with domino gates can be significantly faster than a corresponding circuit made with other techniques. The circuit has the low power of a dynamic circuit since there is never a dc path to ground. Also, the full pull-down current is available to drive the output nodes. At the same time the load capacitance is much smaller than for CMOS because most of the p-channel transistors have been eliminated from the load. Meanwhile, the use of a single clock edge to activate the circuit provides simple operation and full utilization of the speed of each gate. One limitation of this circuit technique is that all of the gates are noninverting. This initially appears serious since, for example, an *XOR* function is not possible. *XOR* logic is commonly used in many complex circuits including arithmetic logic unit (ALU). The problem can be solved by inverting the output of the last stage of domino circuit to drive the needed static XOR, since the domino

gate is fully compatible with standard CMOS gates. An example of this approach can be found in Section 4.2.2.

3.2.5 Zipper CMOS Logic

Zipper CMOS is another example of a dynamic logic family. The main feature of zipper CMOS is the use of four clocks to control an nMOS-pMOS logic cascade as shown in Figure 3.6.

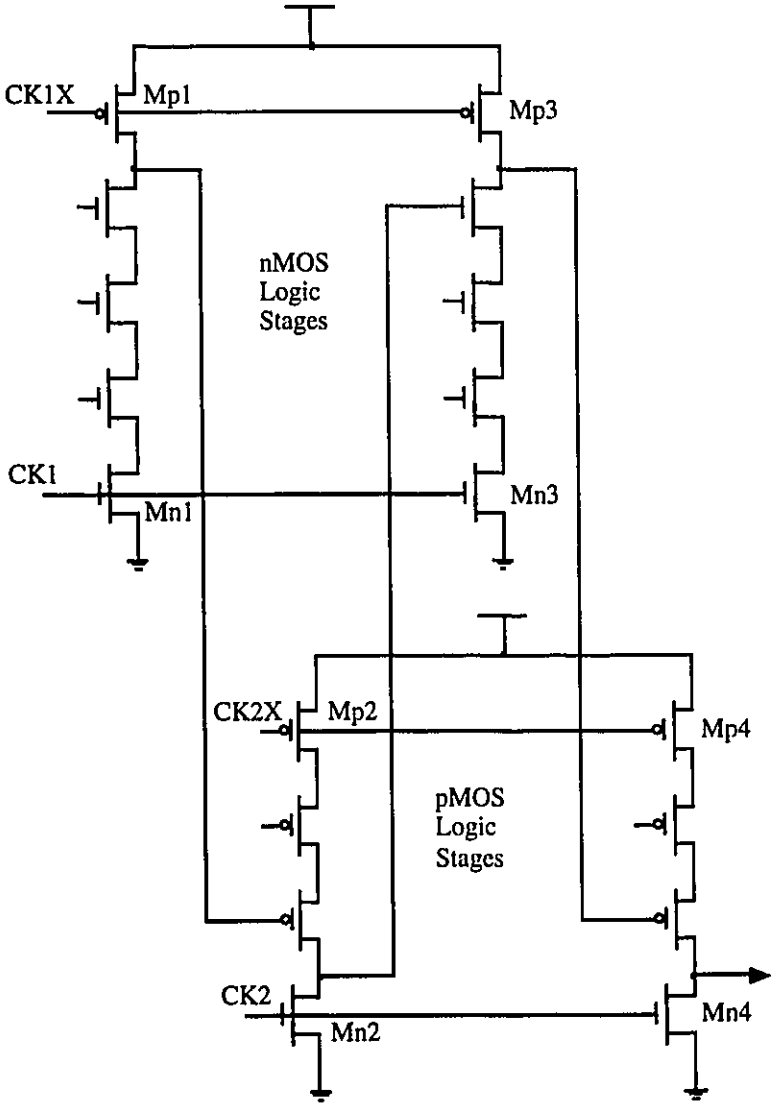

Figure 3.6 Zipper Logic Circuit

Clock signals CK1 and CK1X are in phase with each other, and are connected to the nMOS logic stages. Similarly, CK2 and CK2X are in phase, and define the timing intervals in the pMOS circuits. The clocks are defined so that

$$CK1 \cdot CK2 = 0 \tag{3.1}$$

Figure 3.7 illustrates the clocking signals. Consider first the waveforms for CK1 and CK1X. We see that CK1 has an amplitude which ranges from 0 to V_{DD}, but that CK1X is restricted to the range $[0, (V_{DD} - V_{Tp})]$. where V_{Tp} is the threshold voltage of the pMOS device. Since CK1X is applied to the p-channel precharge transistors Mp1 and Mp3, limiting the amplitude to a maximum of $(VDD - V_{Tp})$ keeps these devices on the edge of conduction during logic evaluation. This reduced amplitude is designed to overcome the problems of charge sharing and charge leakage in the nMOS logic stage.

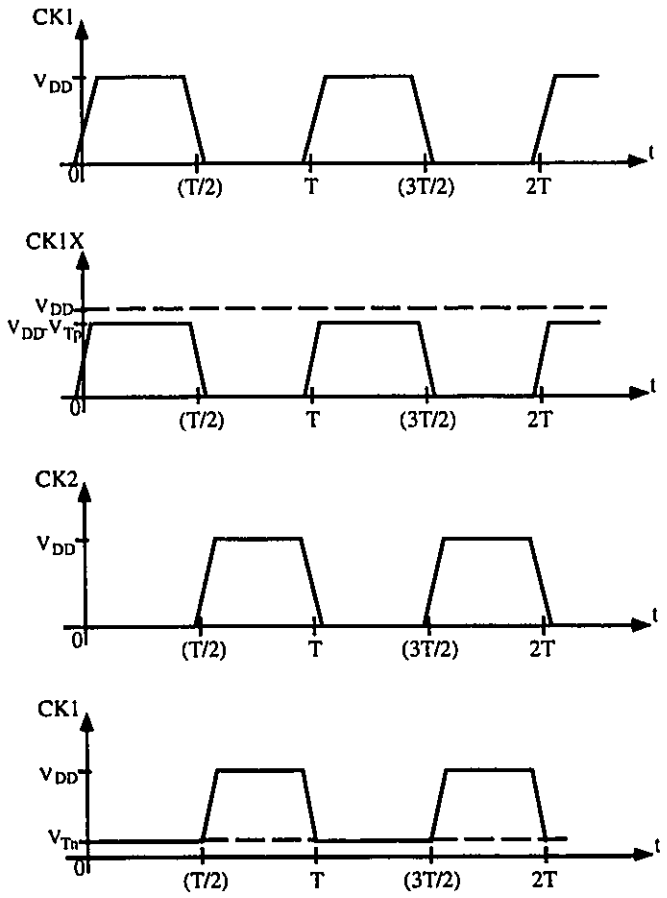

Figure 3.7 Clocks for Zipper Logic

The clock pair CK2 and CK2X are design to perform a similar function. CK2 undergoes a full-rail swing, but CK2 is restricted to the range $[V_{Tn}, V_{DD}]$. Noting that CK2X controls the n-channel precharge transistors Mn2 and Mn4, we see that the increased amplitude keeps these devices on the edge of conduction during a pMOS evaluation interval.

The name "zipper CMOS" arises from tracing the logic signal "up" and "down" as it propagates from left to right in the network; this is similar to a zipper closing. The design style itself is complicated by the need to generate and route four clocking signals.

3.2.6 Cascode Voltage Switch Logic (CVSL)

This is a differential style of CMOS logic requiring both true and complement signals to be routed to gates. Two complementary nMOS switch structures are constructed and then connected to a pair of cross-coupled p-channel transistors. A symbolized gate is illustrated in Figure 3.8. The logic circuit on the right side uses input variables $(\bar{a}, b, \bar{c})$ to form a logic block $\bar{f}$, while the left hand side logic has complementary inputs $(a, \bar{b}, c)$ and is denoted by f. A condition of $f = 1$ implies that conduction is established through the f logic block. Gate level outputs are denoted by $(q, \bar{q})$. This illustrates one implementation of **dual-rail** logic where both a function and its complement are generated at the same time. The dual-rail advantage is achieved at the expense of the extra routing, active area, and complexity associated with dealing with double rail logic. In addition, this form of circuit is slower than a conventional complementary gate because during the switching action, the p-channel pull-ups have to fight the n pull-down trees.

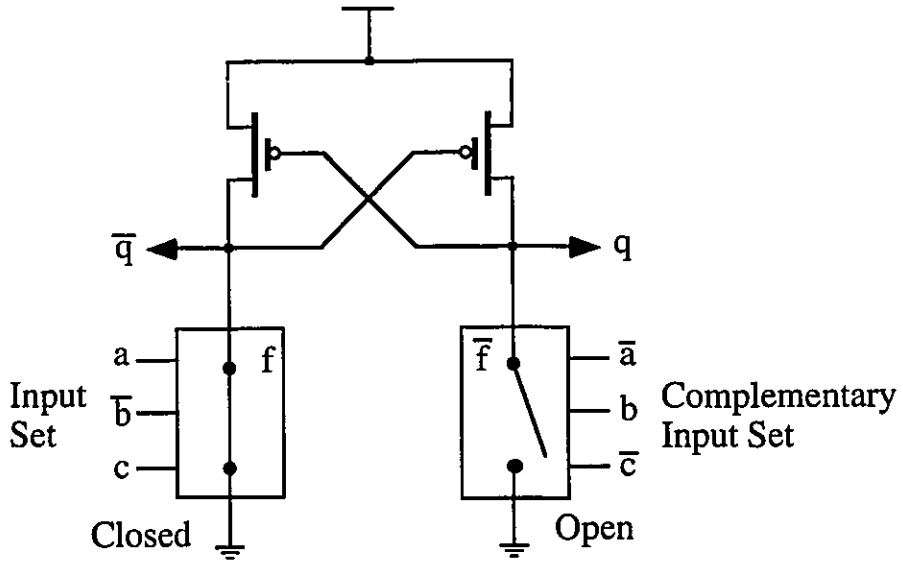

Figure 3.8 Cascode Voltage Switch Logic

3.3 Multiple Output Domino Logic (MODL)

3.3.1 Basic Structure

Multiple output domino logic (MODL) circuit was introduced in [15] to provide a faster and more compact system than standard domino logic (SDL). Unlike SDL, MODL produces more than one function outputs from a single logic gate.

As we know from the domino logic introduced in Section 3.2.4, there is only one output generated from a SDL gate. In many cases, especially in the case where recurrent logic, such as carry generation, is involved, one function is often a subfunction of the other. When the function and its subfunction are needed to be generated as separate output signals, replication of circuitry will be required for SDL circuits, as illustrated in Figure 3.9. As function $f1$ is a subfunction of function f two SDL gates are necessary to provide both outputs, as a result the nMOS logic tree $T1$ is reused in the f generation circuit. Note

that in the SDL gate for function f, an additional precharge pMOS is often necessary to prevent charge sharing.

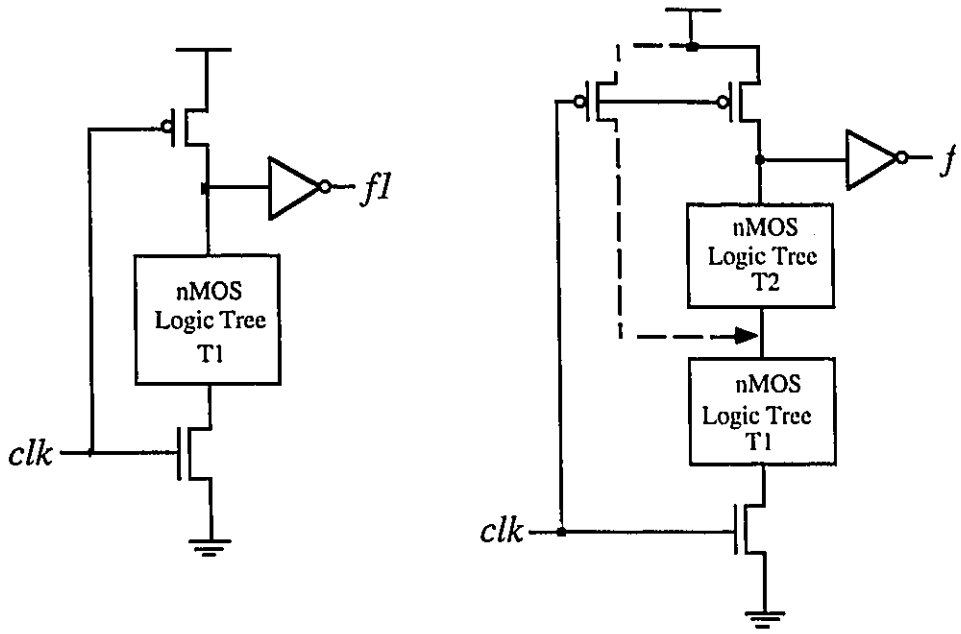

Figure 3.9 SDL Gates Generating *f1* and *f*

The main concept behind MODL is the elimination of replicating circuitry, by using the subfunction available in the logic tree as an output by adding a precharge device and a static inverter at the corresponding node, as shown in Figure 3.10. Comparing with Figure 3.9, we can see that MODL requires fewer extra precharge devices since nodes internal to the logic tree are being precharged for functional output purposes as well as for preventing charge sharing, whereas these additional devices usually exist in a SDL tree to prevent charge sharing. Thus MODL is considered less susceptible to charge sharing than SDL circuits and saves additional devices.

By looking once more at Figure 3.10, we notice that the nMOS logic tree of each MODL gate drives more than one static inverter so that the overall capacitive loading of a MODL gate is more than that found in an SDL gate. This calls for the enlargement of the pull-down devices in order to discharge the increased capacitance. In the actual layout,

however, the area savings due to reduction of gate count and interconnection complexity exceeds the area increase due to incrementally larger device sizes.

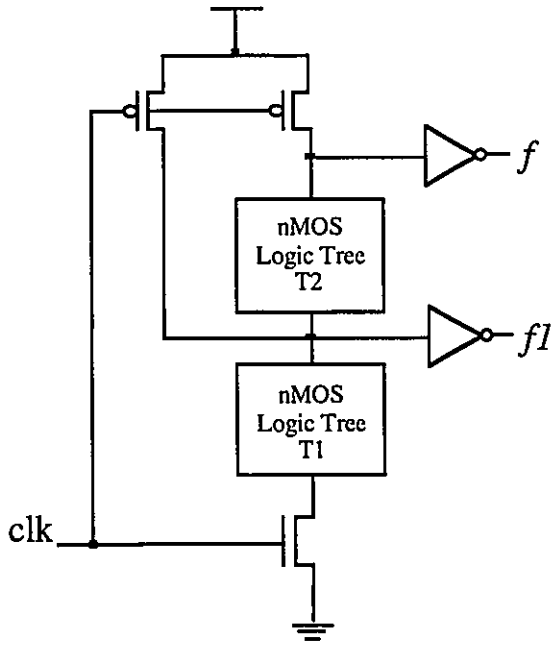

Figure 3.10 MODL Structure

Although the area advantage of MODL over SDL is apparent, the speed advantage at the gate level is not so obvious because each MODL gate implements more than one function. However, performance improvement can be achieved by capitalizing on this fact at architectural level. Therefore, the overall organization of a given block of logic is optimized in terms of the use of MODL techniques. By using a MODL circuit style, the overall device count is reduced and therefore the fan-out for a given output is reduced. Meanwhile, parasitic wiring capacitance is lessened as a consequence of a smaller overall layout. This results in an overall reduction of load capacitance for a given logic stage. As a final point, the reduction of capacitance also results in a reduction in power dissipation. Generally speaking, use of MODL can reduce silicon area, increase circuit performance, and decrease power, because of the reduction of device count, wire length, and consequently output loading.

3.3.2 Sneak Paths

One of the problems a MODL circuit may encounter is the occurrence of *sneak paths*, when an OR-AND form of MODL gate is constructed. A *sneak path* is defined as a path in the circuit which causes a false discharge from a lower output node through a higher device to ground. This is illustrated through an example in Figure 3.11. As mentioned in Section 2.3.1, carry propagate p_i can be expressed as either $a_i + b_i$ (OR-p_i) or $a_i \oplus b_i$ (XOR-p_i). If the *OR* logic is used for p_i, a sneak path will be formed through node *N*1, devices *D1*, node *N2*, device *D2* and "ground switch" *D3* to ground, under the condition $a_{i+2} = b_{i+2} = 1$ and $a_{i+1} = b_{i+1} = 0$. No matter what value c_i holds, since $a_{i+1} = b_{i+1} = 0$ results in $g_{i+1} = p_{i+1} = 0$, node *N1* is expected to hold at logic high, thus c_{i+1} is supposed to be logic "0" during evaluation phase. On the other hand, since $a_{i+2} = b_{i+2} = 1$ forces $g_{i+2} = p_{i+2} = 1$, node *N2* will be discharged to a logic low through *D2*. Unfortunately, during evaluation the p-channel devices are turned off which terminates the power supply to node *N1*, meanwhile devices *D1*, *D2* and *D3* are turned on by p_{i+2}, g_{i+2} and clk. Device *D1* behaves as a bidirectional pass transistor which form a path conducting through *D1*, *D2* and *D3* to falsely discharge node *N1* to ground. Thus c_{i+1} is evaluated mistakenly as logic "1" instead of "0"y. Sneak path problems appear to only occur in MODL type circuits.

In order to avoid this sneak path problem, the *XOR* form of p_i should replace the OR-p_i in the same circuit. In this way, under the same condition $a_{i+1} = b_{i+1} = 0$ and $a_{i+2} = b_{i+2} = 1$, p_{i+2} is no longer "1" for $p_{i+2} = a_{i+2} \oplus b_{i+2} = 0$, as a result, *D1* is cutoff and effectively blocks the sneak path from *N1* through *N2* to ground.

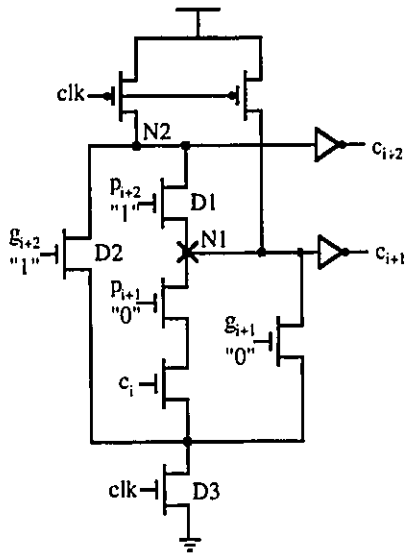

Figure 3.11 Sneak Path Example

3.3.3 Example of MODL gates

Since the overall savings in device count is due to a reduction of circuit replication, the actual advantage of MODL over SDL is directly dependent upon the degree of recurrence in the logic functions being realized. For example, a 2-bit carry propagate using *XOR* logic is defined as $dp_{i+1} = p_{i+1}p_i$ and the 1-bit propagate term is $p_i = a_i \oplus b_i$. Note that p_i is a subfunction of dp_{i+1}. A MODL gate that generates both terms are shown in Figure 3.12.

Another example is a carry generation circuit. Referring to eqn. (2.10) and reformulating it to eqn. (3.2), we find that carry generation logic is in a highly recurrent form. Implementing the logic described in eqn. (3.2) using MODL gives a 4-bit carry generator shown in Figure 3.13. This MODL gate uses 22 devices, while the equivalent four SDL carry-generating gates would require 46 devices including those pMOS for preventing charge sharing. The number of transistors used with MODL is less than half of that with SDL. There is no doubt that MODL is advantageous for implementing highly recurrent logic like carry generating functions.

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 (g_0 + p_0 c_0) = g_1 + p_1 c_1$$

$$c_3 = g_2 + p_2 (g_1 + p_1 (g_0 + p_0 c_0)) = g_2 + p_2 c_2$$

$$c_4 = g_3 + p_3 (g_2 + p_2 (g_1 + p_1 (g_0 + p_0 c_0))) = g_3 + p_3 c_3 \qquad (3.2)$$

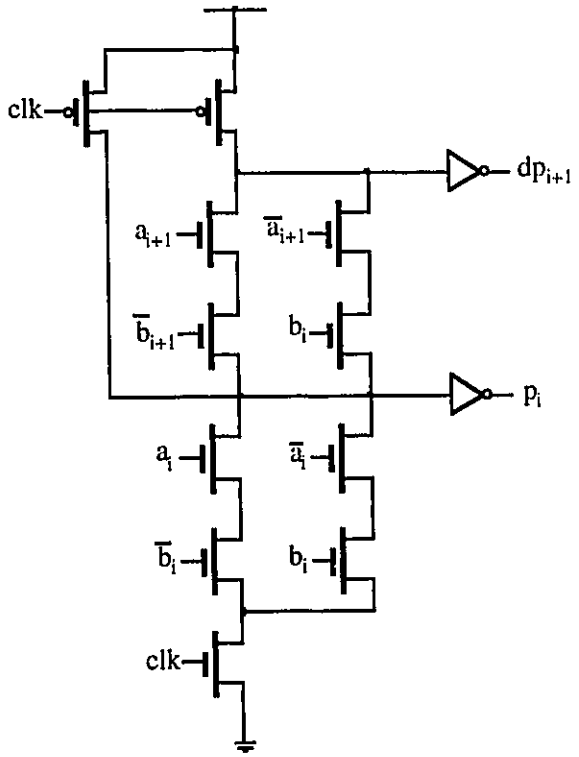

Figure 3.12 1- and 2-Bit Propagate Terms

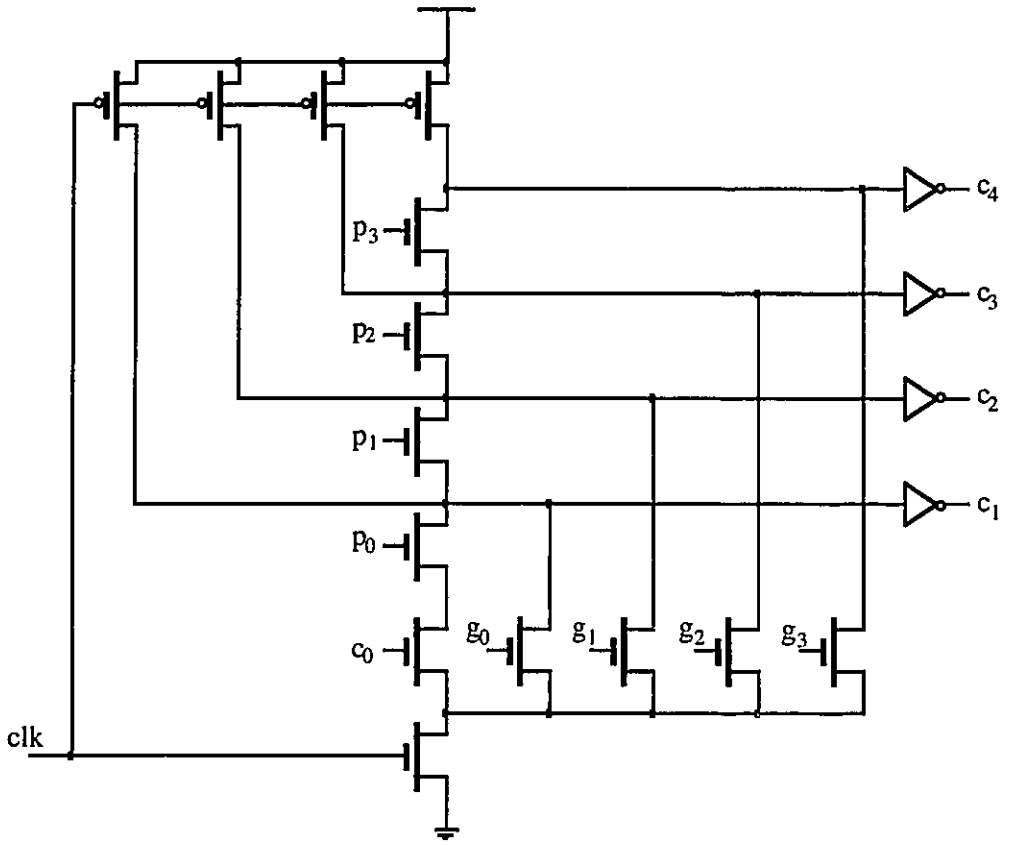

Figure 3.13 4-Bit MODL Carry Generator

3.3.4 MODL and Manchester carry chain

If we look back at the Manchester carry chain embedded in Figure 2.10 and make some corresponding arrangement to have a Manchester carry chain for generating consecutive carries, we get the circuit shown in Figure 3.14.

Comparing Figure 3.14 with Figure 3.13 we can see that a MODL carry generator is functionally and structurally equivalent to a dynamic Manchester carry chain. The difference between these two is perhaps in their evolution and layout style. That is, a MODL carry generator is treated as a single combinational logic gate while a Manchester carry chain has evolved from a blend of MOS pass transistor logic and combinational

logic. Regardless, MODL is a general CMOS logic style, but the Manchester carry chain is an MOS circuit specific to carry generation.

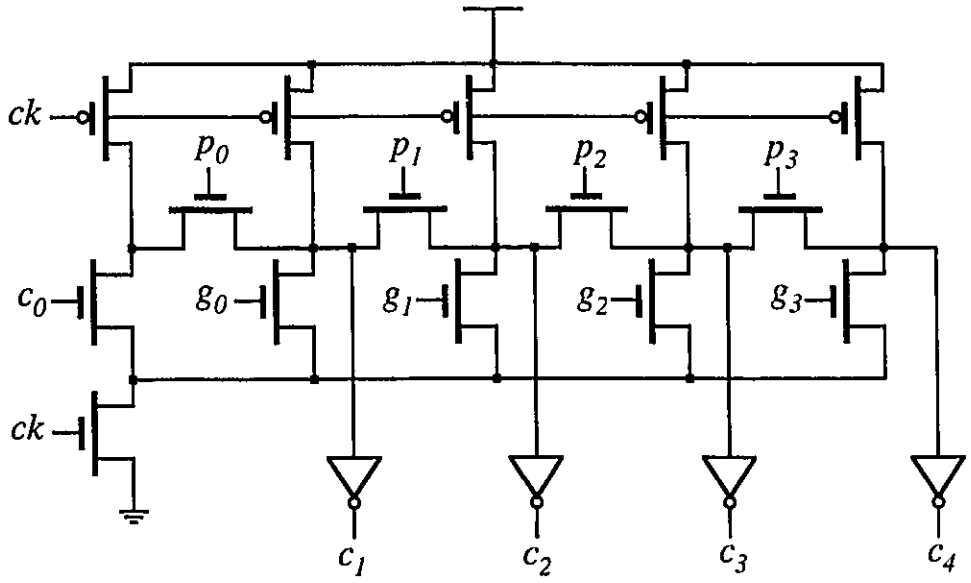

Figure 3.14 4-Bit Manchester Carry Chain with Carry Output for Each Bit Position

3.4 Enhanced Multiple Output Domino Logic (EMODL)

The MODL concept arises from the idea that one logic tree can generate its function and subfunction without replicating subfunction circuits. This can be extended to a more general multiple output domino logic structure which we have referred to as enhanced multiple output domino logic (EMODL). Unlike MODL circuits, one output of an EMODL gate is not necessary to be a subfunction of the other. Nor is a subfunction necessary to be an output of the EMODL gate. In another word, a subfunction of an nMOS logic tree, which may not lead to an output, forms a common base to two or more functional outputs. Similarly, these functional outputs may not be restricted to a function-subfunction relationship. As illustrated in Figure 3.15, the nMOS logic tree $T0$ is a

common base to both functions *f1* and *f2*. *f1* is not a subfunction of *f2*, and *f2* is not a subfunction of *f1*. Merging the two SDL gates to an EMODL gate, as shown in Figure 3.16, eliminates the requirement of replicating the nMOS logic tree *T0*, and also saves additional precharge devices required in the equivalent SDL circuits. The enhanced MODL concept provides the same area advantage over SDL circuit as MODL does, but it provides more flexibility and freedom for creating circuit designs than MODL.

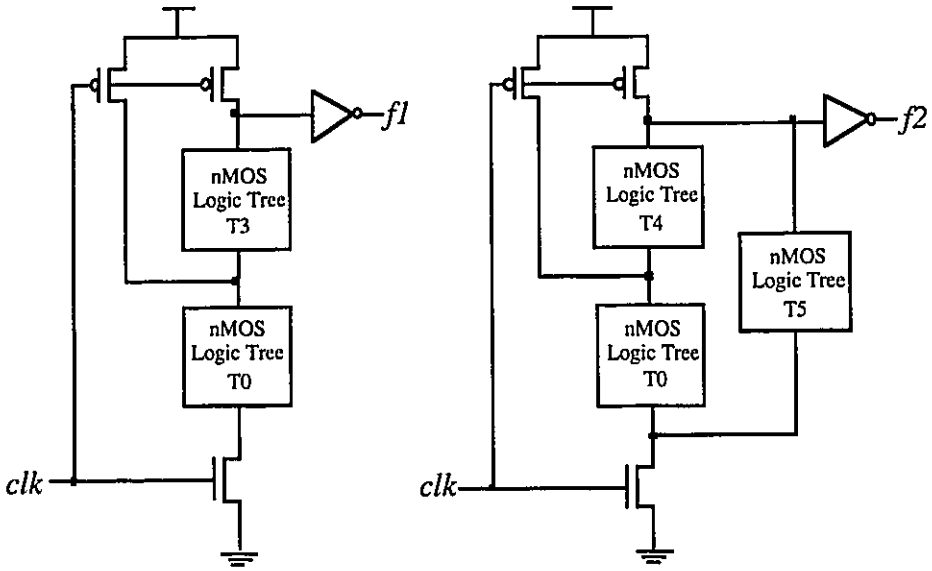

Figure 3.15 SDL Gates with Common Base *T0*

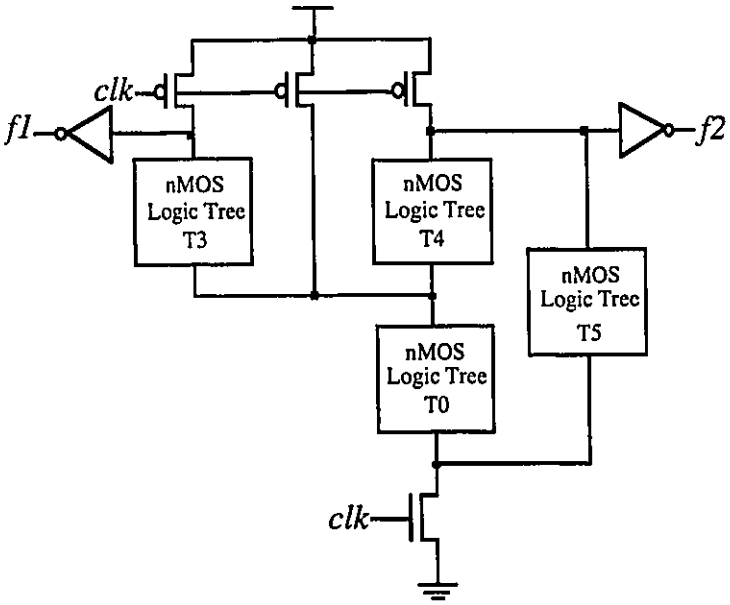

Figure 3.16 EMODL Structure

An immediate example of EMODL design is the generation of four consecutive sums as shown in Figure 3.17. With carry input c_k, sums s_{k+1}, s_{k+2}, s_{k+3} and s_{k+4} are generated from a single EMODL tree. Note that all the carry propagate p_k inputs used in Figure 3.17 are produced by *XOR* of variables a_k and b_k. The nMOS tree in the dashed box, which represents functions c_{k+1} and $\bar{c}_{k+1}$, is the common base to the generation of s_{k+2}, s_{k+3} and s_{k+4}. Note that neither c_{k+1} nor $\bar{c}_{k+1}$ is required as output signal. And the four consecutive sums have no function-subfunction relationship.

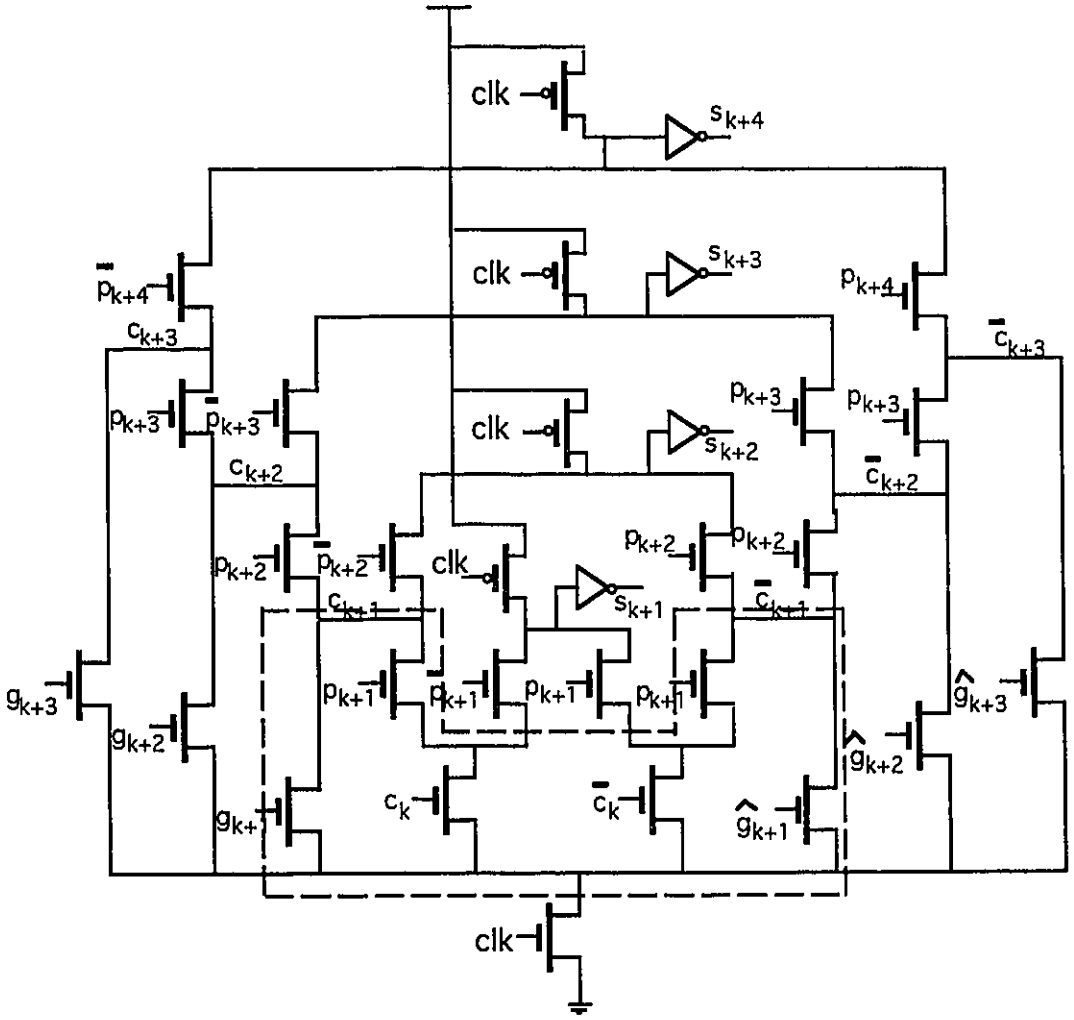

Figure 3.17 4-Bit EMODL Sum Generator

Another example of the advantage of the flexible design style provided by EMODL is the carry generation circuit shown in Figure 3.18. Five sparse carries are generated from group generates and propagates and the carry-in. It is easy to see that this EMODL carry chain is a modification of the MODL circuit discussed in Section 3.3.3. Without increasing the height of the gate, carry c_q is generated from the same nMOS tree (indicated by dashed box) that generates the carry c_n. In this case, the common base is related to an output node. It is important that the input variables $p_{m,n}$ in the circuit are generated originally from the *XOR* logic of summands a_i and b_i. This guarantees that the EMODL carry generator is sneak path free.

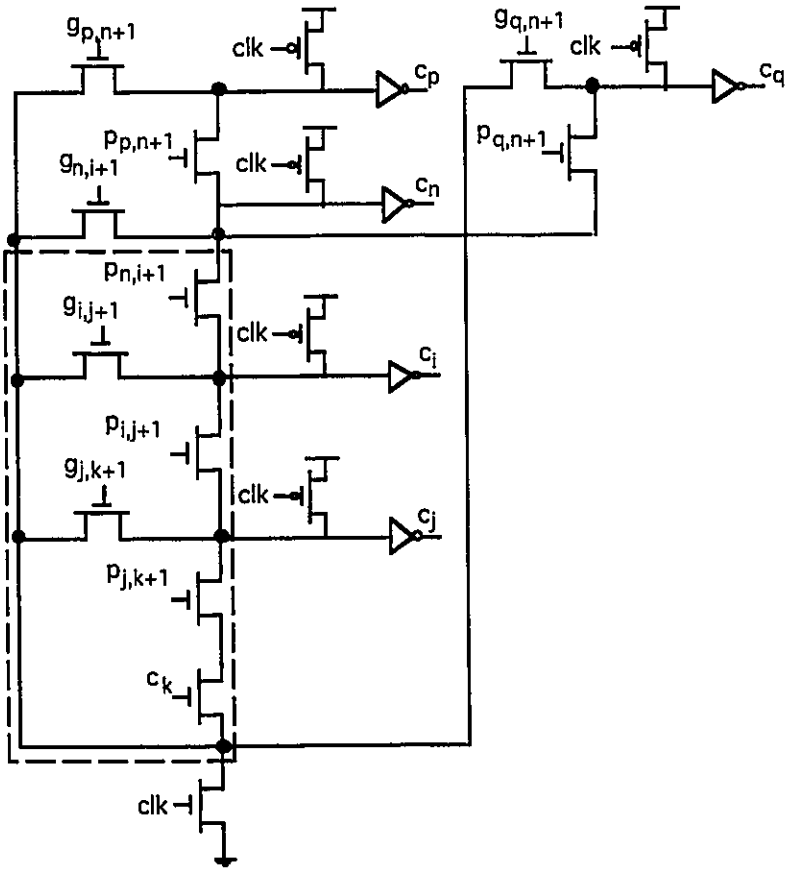

Figure 3.18 A Carry Chain Built by Serial and Parallel Circuit.

3.5 Summary

The evolution of high speed low area circuit design techniques from static CMOS to EMODL logic has been traced in this chapter. Dynamic circuits provide higher speed and lower cost than equivalent static circuits in implementing complex gates with high fan-in and fan-out. Among the various types of dynamic circuits, domino logic is probably the most suitable in system designs with cascaded gates. Multiple output domino logic allows single logic gates to produce multiple functions. It is particularly recommended for recurrent logic such as carry generating circuits where device count reduction is desired. Moreover, this kind of circuit is more stable than standard domino and other dynamic CMOS circuits. The MODL idea is then extended using the enhanced MODL technique which results in greater transistor savings and more design flexibility.

Chapter4

A New 32-Bit CLA Design

4.1 Introduction

In this chapter a novel high performance dynamic carry-lookahead adder is introduced. We compare this new adder with an ultrafast and compact CLA which was recently reported by Hwang and Fisher [15]. The novel features of this new 32-bit adder are the use of the pseudo complement concept, sparse carry chains and EMODL circuit style. The pseudo complement technique guarantees a dual-rail carry chain possessing identical circuit delays, so that use of a dynamic sum stage becomes possible. As a result, the speed advantage of a full dynamic CLA is realized. The dynamic sum stage is composed of an EMODL gate, which is capable of producing consecutive sums from a single carry input. This allows the carry chain to generate only those carries which are required by the EMODL sum stage. This so-called sparse carry chain technique eliminates the usual stages for generating the carries for each bit position which are required in all other published adders, including the comparison design of Hwang and Fisher [15]. The circuit and layout architecture of the new 32-bit CLA are detailed in this chapter. HSPICE simulation and chip test results demonstrate that the performance of this design is superior to other published designs.

4.2 Existing Approaches

4.2.1 Typical CLA Architecture

The conventional CLA architecture consists of three major parts, as shown in Figure 4.1.

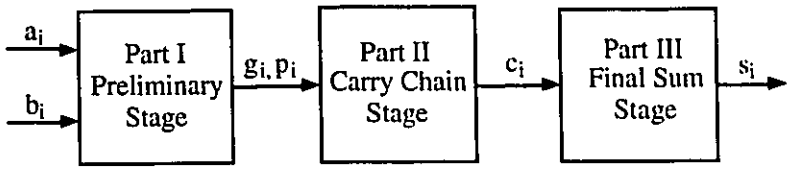

Figure 4.1 Conventional CLA Structure

The first part is the preliminary stage where carry generates and propagates for all bit positions are produced. The second stage is the carry chain which produces the carry and its complement for each bit position. This part normally contains several gate stages and is the major contribution to delays in the adder critical path. The third part is the final sum stage. It is usually comprised of *XOR* cells to produce sums for all bit positions. The inputs of this stage are the two input operands and the carries of each bit position generated by the carry chain. For a typical 32-bit static CLA, with moderate fan-in and fan-out, a worst case delay of seven gated stages [15] is normally required.

4.2.2 Hwang and Fisher's 32-Bit Adder

Using the MODL technique, Hwang and Fisher reported a 32-bit CLA with five gate delays in the worst case. Figure 4.2 shows the block diagram of this adder. A non-complementary circuit structure was exploited for generating carries for each bit position. The complements of carries for all the bit positions are obtained by using a group of static inverters. Finally static *XOR* gates are used to produce the final sums.

Unlike the first stage of the conventional CLA, where carry generate and propagate terms of every bit position are produced from each individual gate by the definition $g_i = a_i b_i$

and $p_i = a_i \oplus b_i$, Hwang and Fisher's CLA produces 1- and 2-bit generate and propagate terms g_i, dg_{i+1} and p_i, dp_{i+1} from each MODL gate. This saves one gate delay. Note that $dp_{i+1} = p_{i+1}p_i$. Since the static sum unit requires p_i for each bit position, additional circuits have to be used for generating p_{i+1} separately.

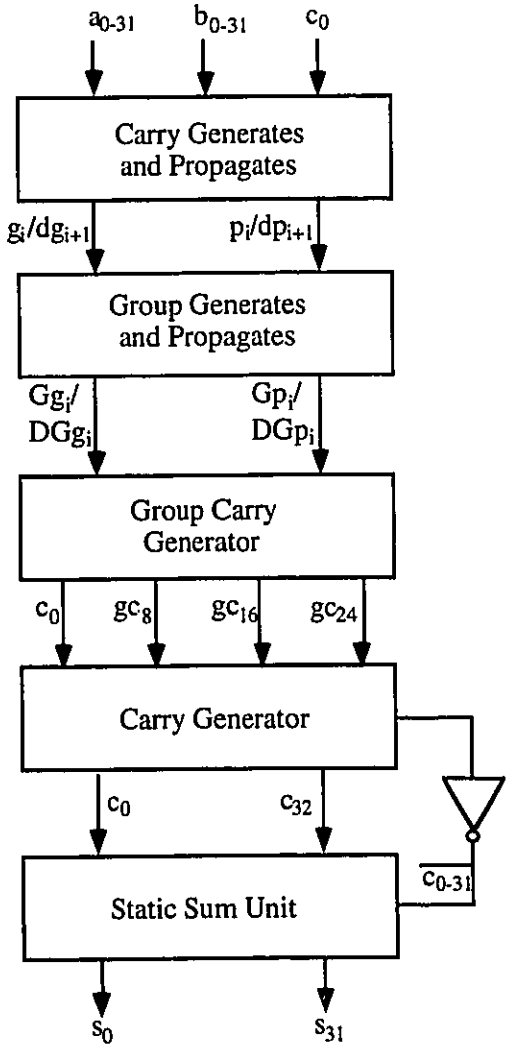

Figure 4.2 Architecture of Hwang and Fisher's 32-Bit CLA

This adder is hybrid in circuit style since it uses dynamic circuits to generate carries and static circuits to produce sums. Obviously, the inherent speed and area disadvantages of

the static circuits limit the performance of this adder. Unfortunately, the use of static *XOR* gates can not be avoided when the non-complementary carry chain approach is used, since a dynamic *XOR* cannot be built in a single phase clocked dynamic circuit (except at the first stage), as demonstrated in Figure 4.3.

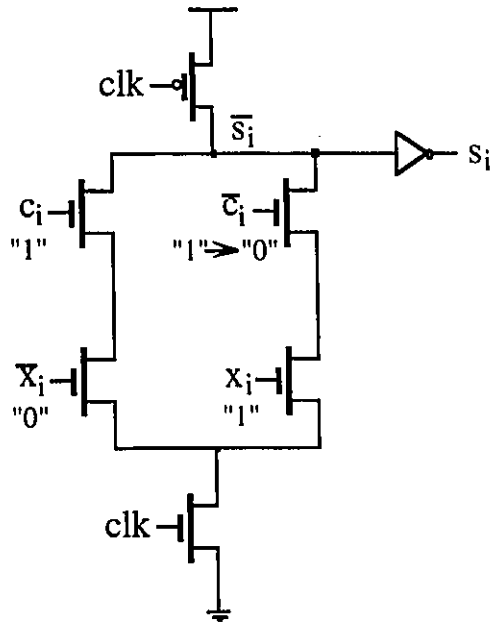

Figure 4.3 A Dynamic *XOR* Gate

Assuming $x_i = a_i \oplus b_i = 1$ and $\bar{x}_i = \overline{a_i \oplus b_i} = 0$ are generated at the same time during evaluation. When clock clk is in evaluation phase, $c_i = 1$ reaches the input of *XOR* gate after the delay required by the carry chain. Suppose, after another inverter delay, $\bar{c}_i$ changes to "0", so that output s_i also should be "0"; unfortunately, this does not happen. The problem is that $\bar{c}_i$ is always driven to "1" by the static inverter during precharge, since c_i is precharged to "0". Only when c_i switches from "0" to "1", can $\bar{c}_i$ change to "0" after an inverter delay. In the situation shown in Figure 4.3, as soon as clk changes to "1", the predetermined $\bar{c}_i$ is high, when x_i is also high; a false discharge of node $\bar{s}_i$ then

occurs before $\bar{c}_i$ has the time to turn to "0". Once this false discharge has taken place, s_i can not be recovered. Thus the nature of the non-complementary structure limits the applications of high-speed circuit techniques. A pure dynamic CLA, which is faster than the hybrid one, can only be realized by a dual-rail carry chain.

4.3 New 32-Bit CLA Design Criteria

4.3.1 Architecture Description

A block diagram of our new 32-bit CLA is given in Figure 4.4. The three parts are actually three gate stages in the real circuit implementation. From the first stage, c_4, group generates gg_k and propagates gp_k are produced. Then c_8, c_{12}, c_{16}, and further group generates Gg_j and propagates Gp_j are generated in the second stage. Finally, all the sums are produced by carries c_4, c_8, c_{12}, c_{16} as well as Gg_j and Gp_j by the third stage. In parallel with this, the pseudo complement carry chain, which utilizes exactly the same domino circuits as the corresponding carry chain, supplies the complement sparse carries and pseudo complements of further group generates and propagates to the EMODL sum stage. There are only three gate delays in the worst case.

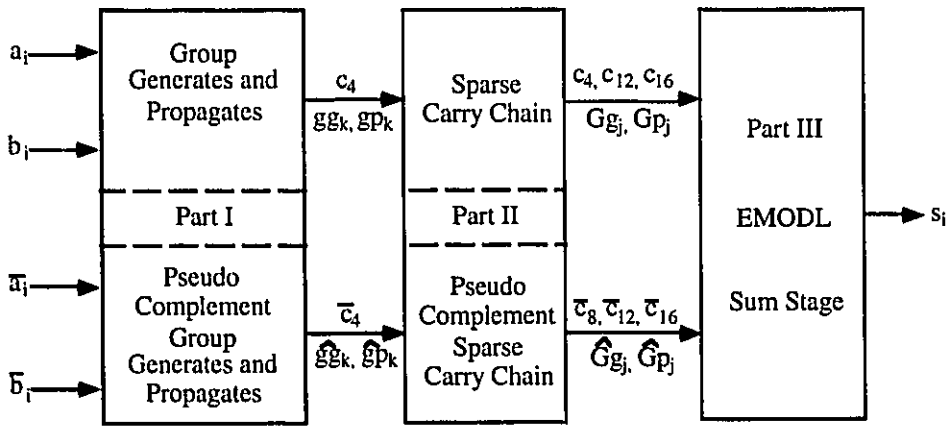

Figure 4.4 The New 32-Bit CLA Structure

4.3.2 Why Pseudo Complement

The "pseudo complement" concept was first introduced by Wang in [35]. The term "pseudo" was chosen to differ from the "real" complement of a logic term. As we know, complementary carries are necessary for performing the *XOR* operation used in generating the sum of a binary adder. The function of a *pseudo complement* is to produce the complement carries from pseudo complement generate and propagate $(\hat{g}_i, \hat{p}_i)$ instead of the complement generate and propagate $(\bar{g}_i, \bar{p}_i)$. We will show, later, that the use of *pseudo complement* expressions to generate the complement carries is superior to the use of complement terms since it results in a circuit with higher speed and smaller silicon area.

To illustrate the idea further, let us compare the MODL implementation of group carry generates $(gg_{k,1}, gg_{k,2}, gg_{k,3})$ and propagates $(gp_{k,1}, gp_{k,2}, gp_{k,3})$, their complements and *pseudo complements*. Note that k is the starting bit position of the group. The Boolean expressions for $(gg_{k,1}, gg_{k,2}, gg_{k,3})$ and $(gp_{k,1}, gp_{k,2}, gp_{k,3})$ are:

$$gg_{k,1} = g_{k+1} + p_{k+1}g_k \qquad gp_{k,1} = p_{k+1}p_k$$

$$gg_{k,2} = g_{k+2} + p_{k+2}gg_{k,1} \qquad gp_{k,2} = p_{k+2}gp_{k,1}$$

$$gg_{k,3} = g_{k+3} + p_{k+3}gg_{k,2} \qquad gp_{k,3} = p_{k+3}gp_{k,2}$$

The MODL circuit implementations are shown in Figure 4.5 and Figure 4.6.

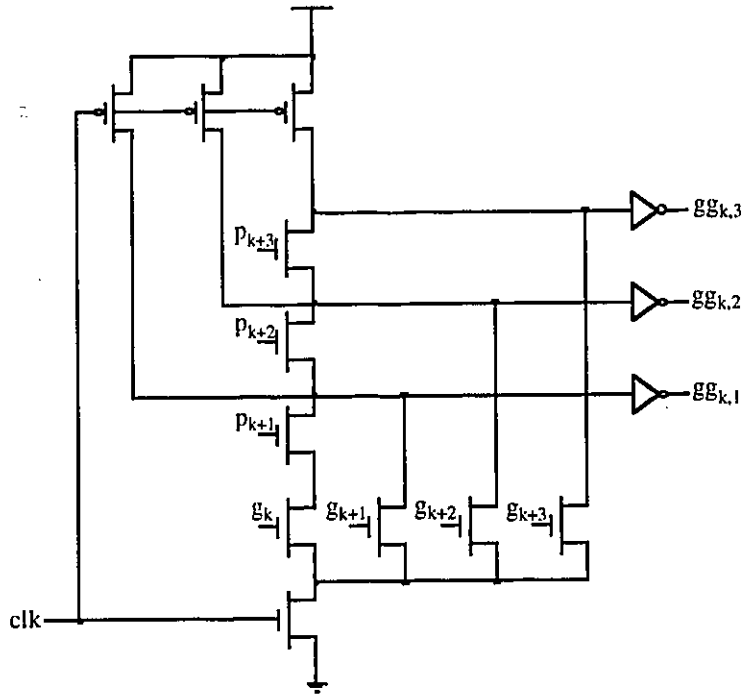

Figure 4.5 Group Generates for $(gg_{k,1}, gg_{k,2}, gg_{k,3})$

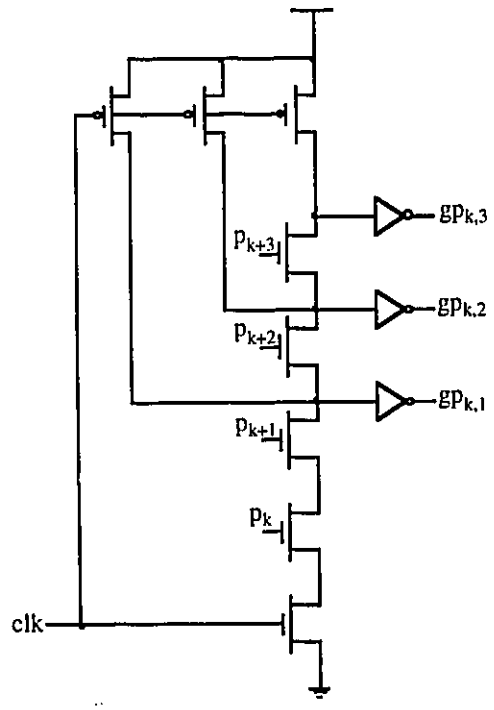

Figure 4.6 Group Propagates for $(gp_{k,1}, gp_{k,2}, gp_{k,3})$

The complements of $(gg_{k,1}, gg_{k,2}, gg_{k,3})$ and $(gp_{k,1}, gp_{k,2}, gp_{k,3})$ are given as follows:

$$\overline{gg}_{k,1} = \bar{g}_{k+1}(\bar{p}_{k+1} + \bar{g}_k) \qquad \overline{gp}_{k,1} = \bar{p}_{k+1} + \bar{p}_k$$

$$\overline{gg}_{k,2} = \bar{g}_{k+2}(\bar{p}_{k+2} + \overline{gg}_{k,1}) \qquad \overline{gp}_{k,2} = \bar{p}_{k,2} + \overline{gp}_{k,1}$$

$$\overline{gg}_{k,3} = \bar{g}_{k+3}(\bar{p}_{k+3} + \overline{gg}_{k,2}) \qquad \overline{gp}_{k,3} = \bar{p}_{k+3} + \overline{gp}_{k,2}$$

Obviously, $\overline{gg}_{k,1}$, $\overline{gg}_{k,2}$ and $\overline{gg}_{k,3}$ can not be integrated in a MODL gate directly. Nor can $\overline{gp}_{k,1}$, $\overline{gp}_{k,2}$ and $\overline{gp}_{k,3}$. Either they have to be implemented in three separate SDL gates, or their logic formulation should be modified to the following equations in order to realize the MODL implementations.

$$\overline{gg}_{k,1} = \bar{g}_{k+1}(\bar{p}_{k+1} + p_{k+1}\bar{g}_k) \qquad \overline{gp}_{k,1} = \bar{p}_{k+1} + p_{k+1}\bar{p}_k$$

$$\overline{gg}_{k,2} = \bar{g}_{k+2}(\bar{p}_{k+2} + p_{k+2}\overline{gg}_{k,1}) \qquad \overline{gp}_{k,2} = \bar{p}_{k+2} + p_{k+2}\overline{gp}_{k,1}$$

$$\overline{gg}_{k,3} = \bar{g}_{k+3}(\bar{p}_{k+3} + p_{k+3}\overline{gg}_{k,2}) \qquad \overline{gp}_{k,3} = \bar{p}_{k+3} + p_{k+3}\overline{gp}_{k,2}$$

The MODL implementations are shown in Figure 4.7 and Figure 4.8. It is easy to see that not only the transistor counts are more than the circuits shown in the previous two figures, but also the fan-in of the gates for generating complement group generates are higher. Thus the resultant speed of the circuits is slower.

The *pseudo complement* approach solves the above problem in a perfect way. It comes from the idea that the complement carries that are needed by the *XOR* sum stage are not required to be produced from the intermediate complement generate and propagate terms. If we replace the real complements of these intermediate terms, such as $\bar{g}_i = \bar{a}_i + \bar{b}_i$ and $\bar{p}_i = \overline{a_i \oplus b_i}$, based on $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$, with pseudo complement terms $\hat{g}_i = \bar{a}_i \bar{b}_i$ and $\hat{p}_i = \bar{a}_i \oplus \bar{b}_i = a_i \oplus b_i$, then the pseudo complements of

$(gg_{k,1}, gg_{k,2}, gg_{k,3})$ and $(gp_{k,1}, gp_{k,2}, gp_{k,3})$ can be expressed in the same logic forms as their true terms respectively.

$$\hat{gg}_{k,1} = \hat{g}_{k+1} + \hat{p}_{k+1}\hat{g}_k \qquad \hat{gp}_{k,1} = \hat{p}_{k+1}\hat{p}_k$$

$$\hat{gg}_{k,2} = \hat{g}_{k+2} + \hat{p}_{k+2}\hat{gg}_{k,1} \qquad \hat{gp}_{k,2} = \hat{p}_{k+2}\hat{gp}_{k,1}$$

$$\hat{gg}_{k,3} = \hat{g}_{k+3} + \hat{p}_{k+3}\hat{gg}_{k,2} \qquad \hat{gp}_{k,3} = \hat{p}_{k+3}\hat{gp}_{k,2}$$

As described by Eqn. (2.25)-(2.28), eventually the complement carries can be obtained from these pseudo complement terms in the same way as the carries are derived from the original generate and propagate terms.

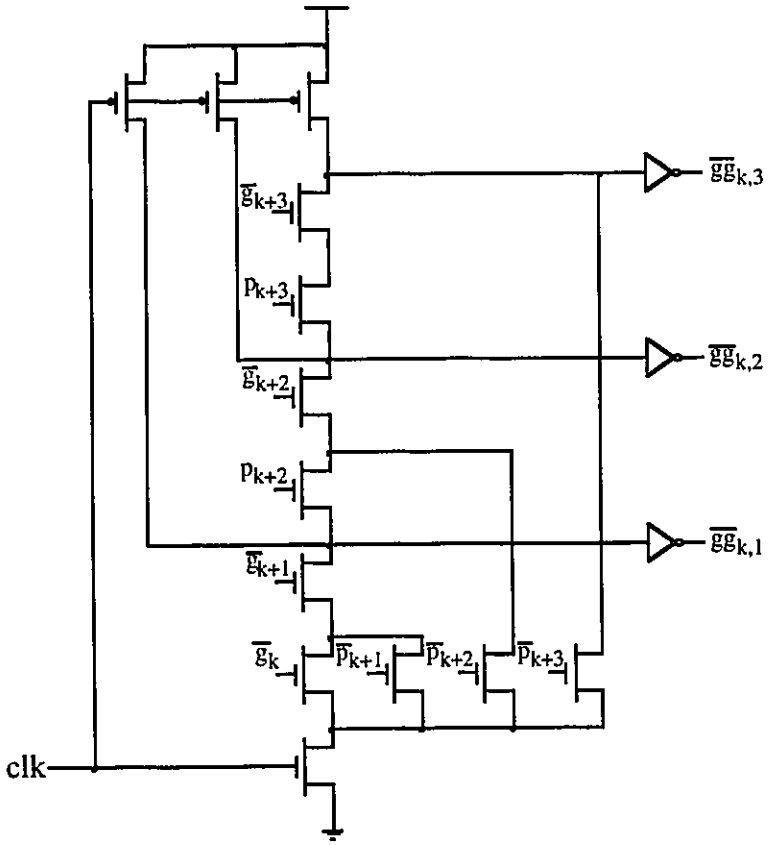

Figure 4.7 Complements of Group Generates ($gg_{k,1}$, $gg_{k,2}$, $gg_{k,3}$)

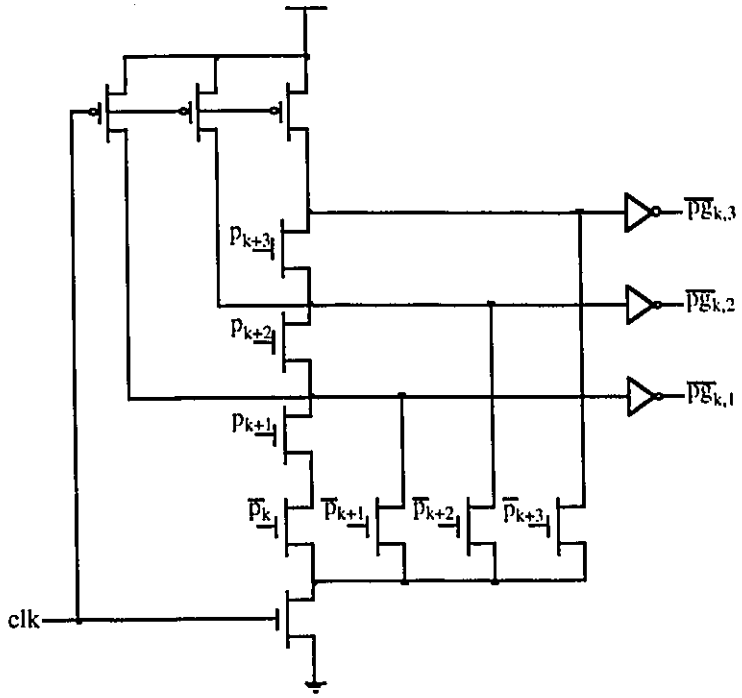

Figure 4.8 Complements of Group Propagates ($gp_{k,1}$, $gp_{k,2}$, $gp_{k,3}$)

Figure 4.9 and Figure 4.10 are the MODL gates which produce the pseudo complements of $(gg_{k,1}, gg_{k,2}, gg_{k,3})$ and $(gp_{k,1}, gp_{k,2}, gp_{k,3})$. The circuit structures are exactly the same as those of Figure 4.5 and Figure 4.6 respectively. In another words, by substituting the inputs of the circuits in Figure 4.5 and Figure 4.6 with pseudo complement terms, we get pseudo complement outputs. This approach can be applied continuously in the complement carry chain until the complement carries are obtained.

The advantages of pseudo complement approach can be summarized to three points: (1) less device count and fan-in than the usual complement approach, hence smaller hardware and higher circuit speed; (2) complementary carry chain with the same delay guarantees that EMODL sum unit function correctly, therefore a full dynamic design is realized; and (3) layout design load is reduced since the complement carry chain layout is simply the duplication of the carry chain.

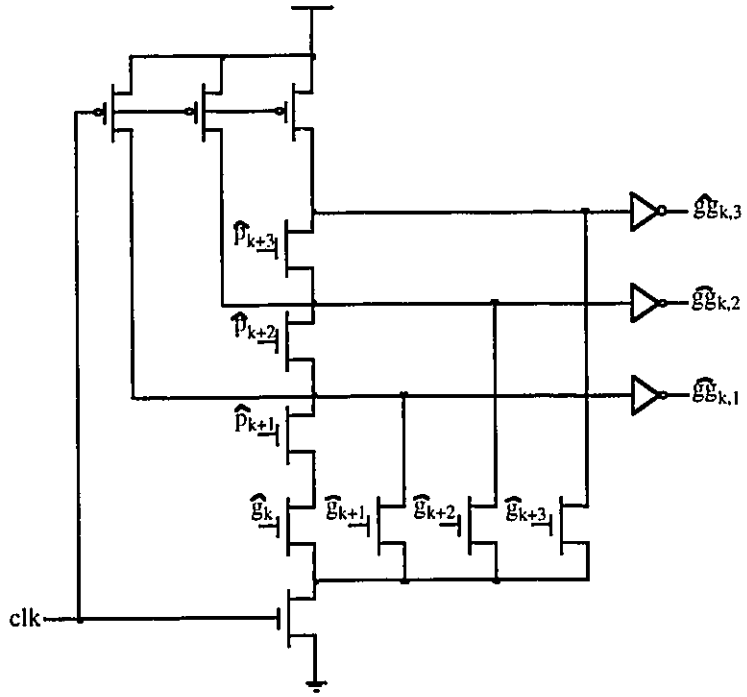

Figure 4.9 Pseudo Complements of Group Generates ($gg_{k,1}$, $gg_{k,2}$, $gg_{k,3}$)

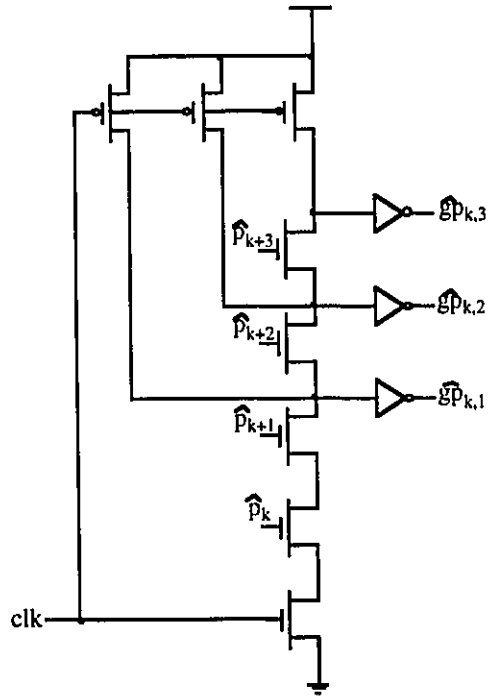

Figure 4.10 Pseudo Complements of Group Propagates ($gp_{k,1}$, $gp_{k,2}$, $gp_{k,3}$)

4.4 Implementation of A 32-Bit CLA

4.4.1 Architecture Floor Plan

Our new 32-bit adder is constructed by nine different types of blocks, denoted as $B_1 - B_8$ and XOR. Figure 4.11 is the architectural floor plan of this 32-bit CLA which consists of three stages in total. The pseudo complement carry chain is omitted from the diagram for simplicity. Block XOR accepts adder inputs a_i and b_i, or in the complement chain $\bar{a}_i$ and $\bar{b}_i$, and produces the XOR results for use in the final sum stage.

The first stage is composed of three different types of blocks $B_1 - B_3$. Two B_1 blocks generate c_4 and $\bar{c}_4$ separately. Eight (4×2) B_2 blocks prepare group generates and propagates ($gg_{m,4}$ and $gp_{m,4}$) and their pseudo complements ($\hat{gg}_{m,4}$ and $\hat{gp}_{m,4}$) in 4-bit length. Six (3×2) B_3 blocks prepare group generates and propagates ($gg_{n,3}$ and $gp_{n,3}$) and their pseudo complements ($\hat{gg}_{n,3}$ and $\hat{gp}_{n,3}$) in 3-bit length. In the second stage, c_8, c_{12} and c_{16} are produced from B_4 in the sparse carry chain, which are sufficient for the third stage to generate the final sums for each bit position. In addition, further group carry generates and propagates $Gg_{k,3}$ and $Gp_{k,3}$ which group lengths of 3 are produced from B_5 for the final stage. This final stage in Figure 4.11 consists of three types of EMODL gates $B_6 - B_8$. Five B_6 cells take complementary sparse carries and the XOR of input operands as inputs and generate the sums from bit position 1 up to bit position 20, with each cell having four successive sum outputs. Another three B_7 cells generate sums from bit 21 to 29. Each cell gives three sum outputs based on c_{16}, $\bar{c}_{16}$, $Gg_{k,3}$, $Gp_{k,3}$ and their pseudo complements. The only difference between block B_7 and B_8 is the additional circuit in B_8 that produces carry c_{32} for the most significant bit. Schematic examples of blocks $B_1 - B_8$ are included in Appendix B.

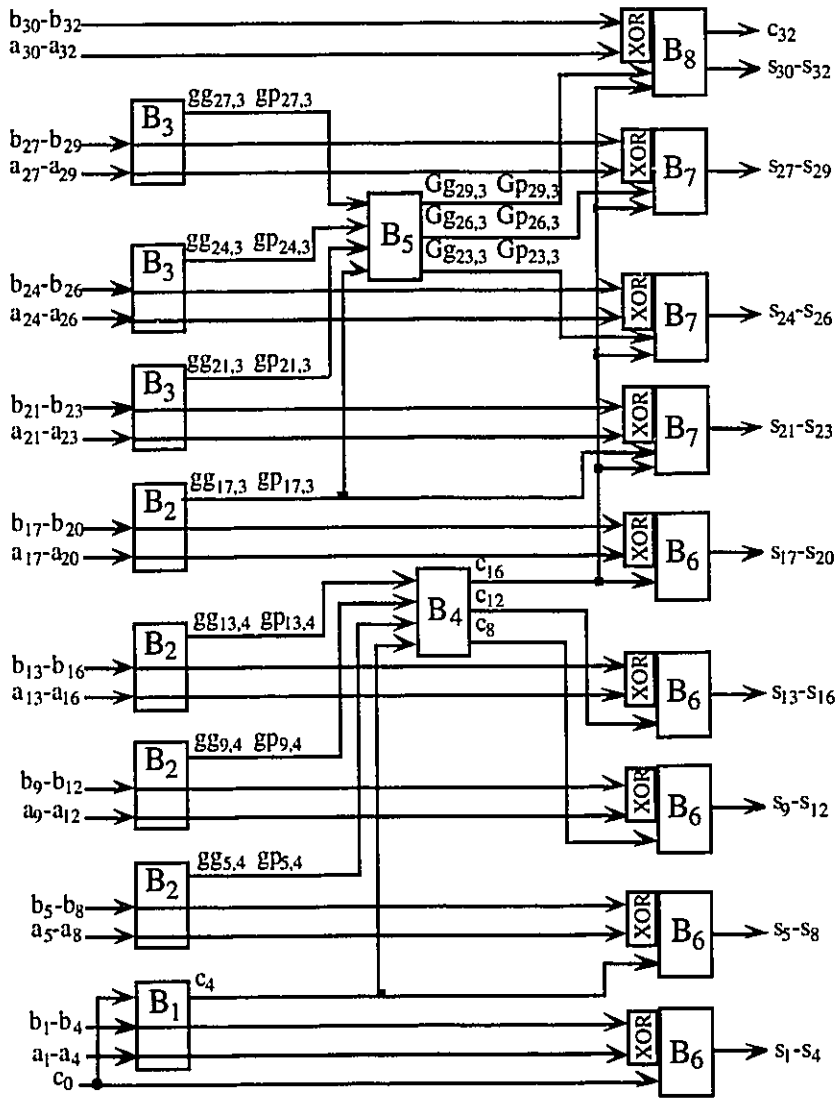

Figure 4.11 Architecture of New 32-Bit Adder

4.4.2 Layout Floor Plan

The 32-bit adder has been full-custom laid out, using 1.2μ CMOS double metal double-well technology. The layout and the physical distribution of the cell blocks for the 32-bit CLA core are shown in Figure 4.12. From the cell distribution diagram shown on the right hand side of the figure, we can see that there are two different columns which are partly interleaved to build the first stage and the last stage respectively. The second stage is inserted between the two columns and takes no extra space, since the area of the two types

of blocks B_4 and B_5 are relatively small. The circuits outside the CLA core are inverters for driving the inputs of the adder so as to provide a realistic input waveform for comparing with HSPICE simulation results.

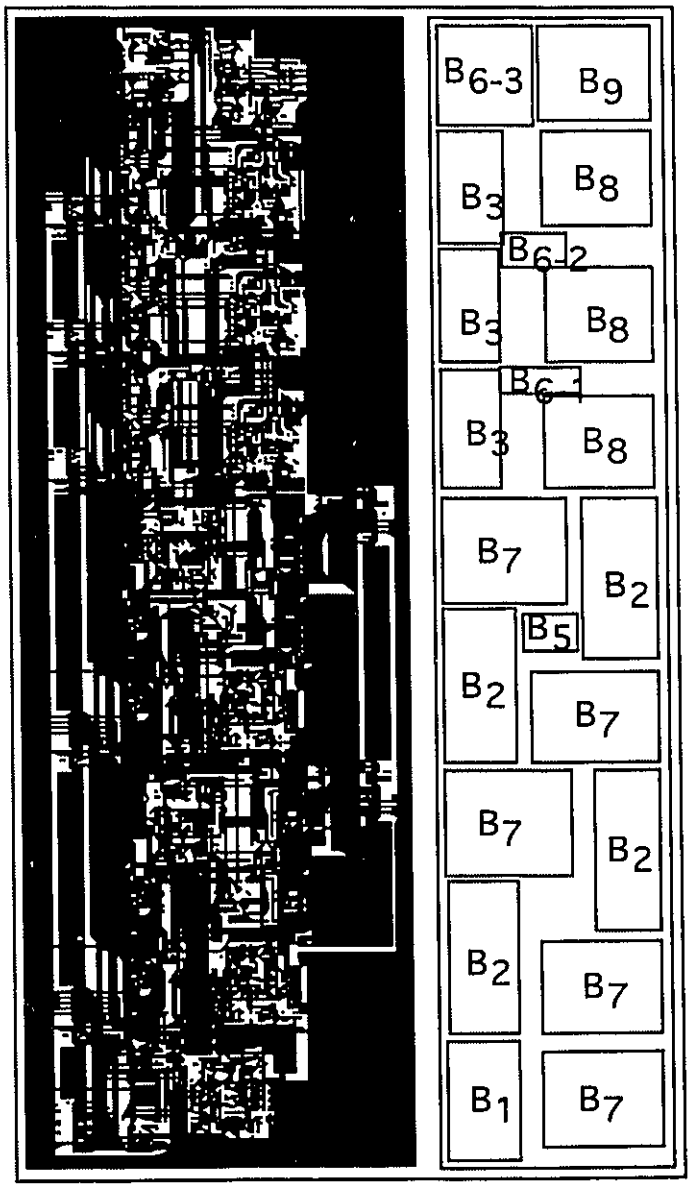

Figure 4.12 New 32-Bit CLA Layout Floor Plan

In the layout, metal two is used for long horizontal connections between the cells. Metal one runs vertically among cells and is used for local interconnections. The LSB of the adder is at the bottom, The carry-in is zero. The total width of the adder area is the sum of the widths of the first and last stage cells. The height of the adder is kept low since the cells are placed carefully as illustrated in Figure 4.12. The total area of the core is $337 \times 2169 \mu m^2$. Figure 4.13 shows a scanned micrograph of the chip.

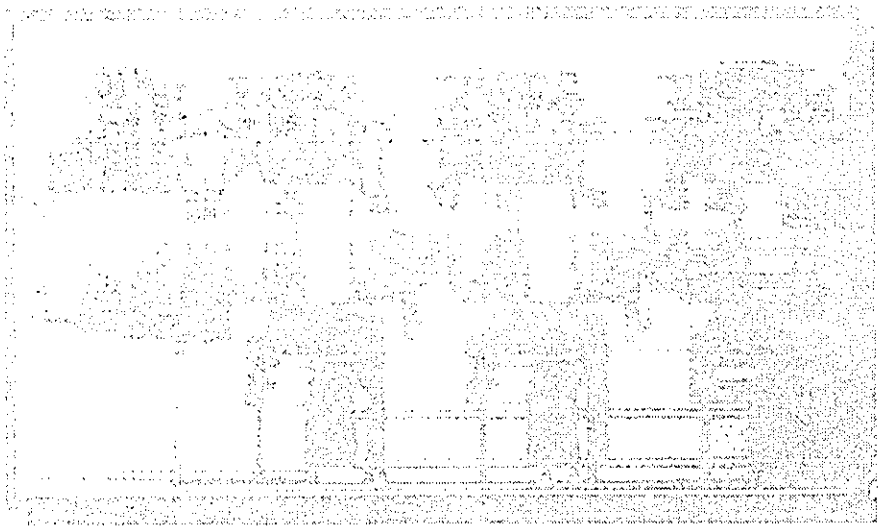

Figure 4.13 Scanned Picture of the Chip

4.4.3 Cell Examples

The schematics and layouts of all the cells designed are listed in Appendix A. We will only give an example design of cell B_7 in this section. B_7 is an EMODL gate with three consecutive sum outputs. Even though this cell has the same maximum fan-in as the schematic shown in Figure 3.17, which is the cell type B_6 used in our adder, it can only generate three sums since the carries have to be produced by both carry-in and group generates and propagates.

We should notice that the height of the nFET tree strongly affects the efficiency of the EMODL circuit. The higher the tree is, the more sums can be generated from a single EMODL gate. thus the fewer the carries that have to be produced throughout the carry chain; this means that the number of stages in the critical path is reduced. However, the investigation in reference [8] points out that if a uniform size is applied to all the n-channel transistors, the discharge time of a serial connected n-channel transistor tree is quadratically related to the height of the tree. In this case, we should avoid using gates with very high fan-in.

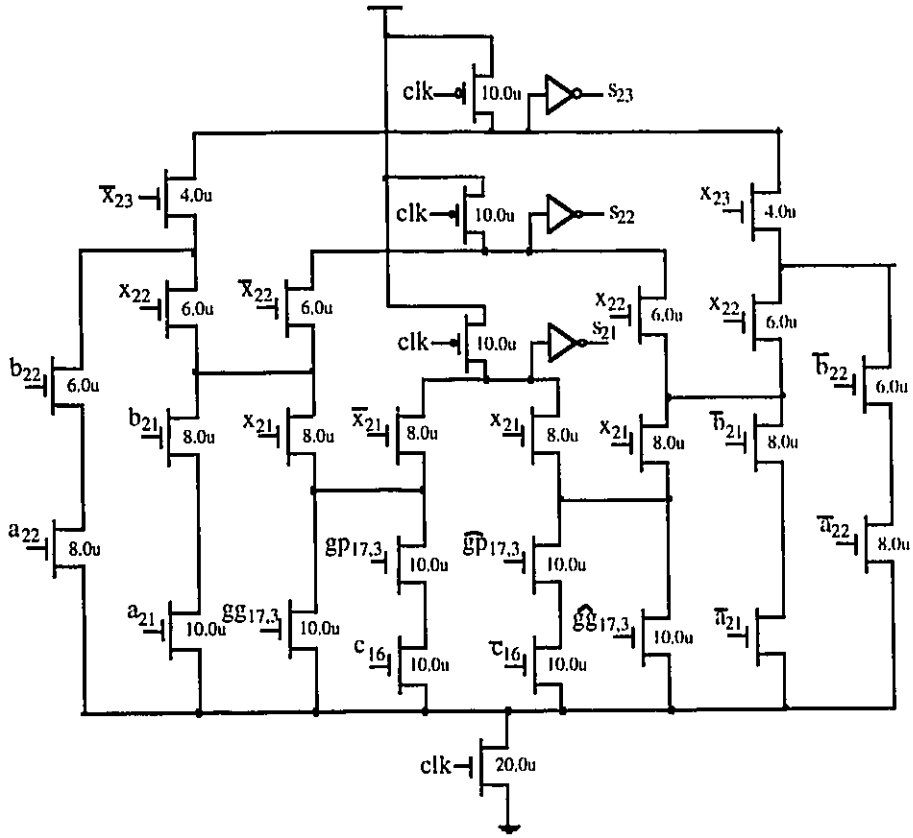

Figure 4.14 Schematic of Cell B_7 with Transistor Sizing

On the other hand, transistor sizing techniques may mitigate the quadratic relationship between the discharge time and tree height. As we know from the *RC* delay model [12], the delay of a dynamic gate depends in a complex way on the sizes of the series path

transistors, since the size of each device controls both its effective resistance and its capacitance. Generally, the delay versus size function is monotonic (i.e. without a minimum), and so we limit the maximum size of the transistor based on area constraints. If we assume that each transistor uses a minimum length given by the process design rules, we can minimize the delay by varying the width of each transistor. Using the analytical optimization approach, introduced in [3], we can minimize the gate delay by tapering the transistor chain so that the transistor closest to ground has the largest width, with transistor size decreasing monotonically from ground to output. In this way, the discharge time is reduced by 30%, and the utilization of gates with relatively high fan-in and complexity becomes practical. As shown in Figure 4.14, the maximum fan-in of B_7 is 5, where the width of each transistor is labeled beside it. The length of each transistor is 1.2μ. A layout according to this design is given in Figure 4.15. The 32-bit CLA using this complex gate gives improved overall performance compared to existing adder designs.

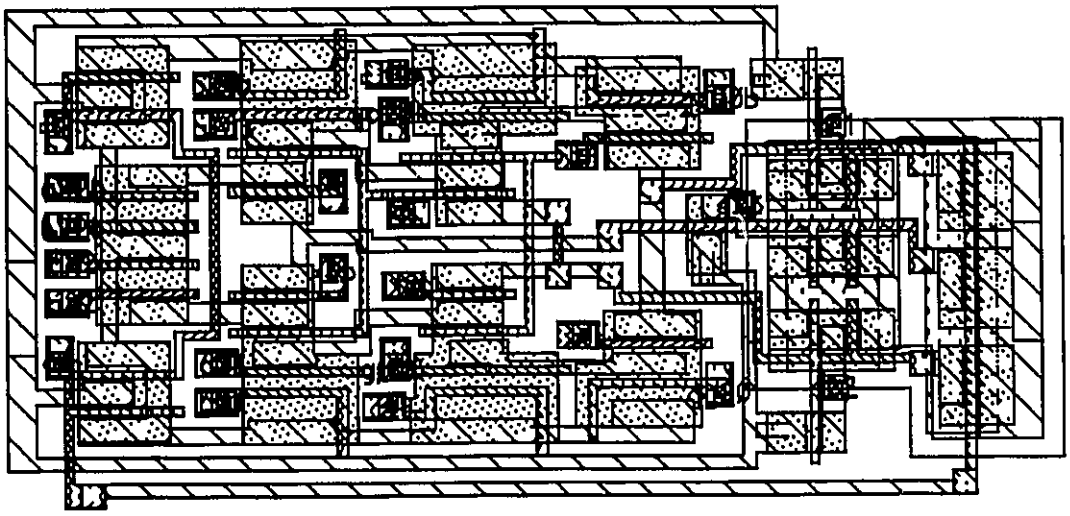

Figure 4.15 Layout of Cell B_7

4.4.4 Simulation and Testing Results

HSPICE simulations on mask extractions of the new 32-bit CLA have been performed. The results show a worst case delay of 2.7ns (Figure 4.16). For comparison, the delay of the 32-bit adder given by Hwang and Fisher, which was fabricated in a 0.9u CMOS

technology, is 3.1ns. Applying reasonable scaling criteria we would expect our design to be faster and smaller than their design.

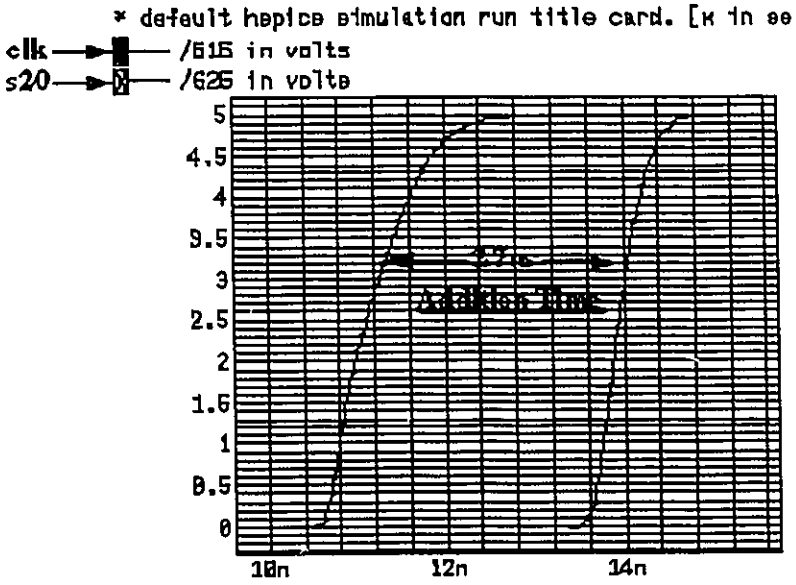

Figure 4.16 HSPICE Simulation from Mask Extracted Circuit

First order MOS scaling theory based on a constant field model is introduced in [37]. Although this is not an accurate model for sub-micron device scaling, it is still a powerful guideline to identify the improvements that can be expected as processes are scaled. Basically, if we apply the scaling factor α to all the dimensions including those vertical to the surface, the concentration densities in the process, as well as the device voltage as shown in Table 4.1, we should get the resultant effects as listed in Table 4.1. Table 4.1 clearly shows that delay, area and power are all decreased in a smaller dimension technology. However, when supply voltage is not scaled, the comparison of delays and power dissipations between two technologies is not as straightforward as the table implies. Clearly though, as parasitic capacitance is decreased in the smaller dimension technology, the overall speed of the circuit will be faster. The voltage scaling does not affect the layout, and so the area scaling is still applicable.

PARAMETERS	SCALING FACTOR
Length: L	$1/\alpha$
Width: W	$1/\alpha$
Gate Oxide thickness: t_{ox}	$1/\alpha$
Junction depth: X_j	$1/\alpha$
Substrate doping: N_a or N_d	α
Supply voltage: V_{DD}	$1/\alpha$

Table 4.1 First-Order Scaling on MOS Device Parameters

PARAMETERS	SCALING FACTOR
Electric field across gate oxide: E	1
Depletion layer thickness: d	$1/\alpha$
Parasitic capacitance: WL/t_{ox}	$1/\alpha$
Gate delay: VC/I	$1/\alpha$
Dynamic power dissipation: P_d	$1/\alpha^2$
Gate area: WL	$1/\alpha^2$

Table 4.2 Influence of First-Order Scaling on MOS Device Characteristics

Table 4.3 compares the main characteristics of our design with Hwang and Fisher's. The equivalent area of our design, scaled from 1.2μ technology to 0.9μ is calculated according to the first-order scaling theory. The scaling can not be applied to the worst case delay, but the speed advantage of 0.9μ technology over 1.2μ is clear. This means that our design in 1.2μ technology with a 2.7ns worst case delay is clearly faster than Hwang and Fisher's CLA in 0.9μ technology with a reported delay of 3.1ns. The performance of the new CLA is unquestionable better than the conventional design.

N.A.---Not Applicable

Design	Technology	No. of Stages	Delay (ns)	Area (μm^2)	No. of Devices
Hwang and Fisher's	0.9μ	5	3.1	341x1976	1325
Our New Design	1.2μ	3	2.7	337x2169	1269
Scaled Estimate	0.9μ	3	N.A.	278x1627	1269

Table 4.3 Comparison Between Two Designs

Obtaining an accurate test result from the 32-bit adder chip is difficult because the expected addition time is less than 3.0ns while the delay introduced by the I/O drivers and the measurement equipment is around 20ns. In order to reduce this delay effect, we have fabricated a chip with ten specially designed adders connected in series. These adders contain only the critical path in the actual 32-bit adder with equivalent loads for the missing circuitry. Figure 4.17 shows the layout of the ten-critical-path-adder along with a calibration circuit built by connecting an input pad directly to an output pad (including drive and protection circuitry). In testing, two parameters have to be measured. One is the delay of the ten-critical-path-adder through the I/O pads, the other is the delay between the calibration input and output pads. In this way, the delay of one critical-path-adder can be calculated as $\frac{TotalDelay}{Calibration\ \text{I/O Pad Delay}} \times \frac{1}{10}$. The test results in Figure 4.18 show a measured delay time of 27ns for ten critical-path-adders after calibration. Therefore, each critical-path adder is equivalent to 2.7ns. This verifies our HSPICE simulation result, of Figure 4.16, which was performed on the entire 32-bit adder.

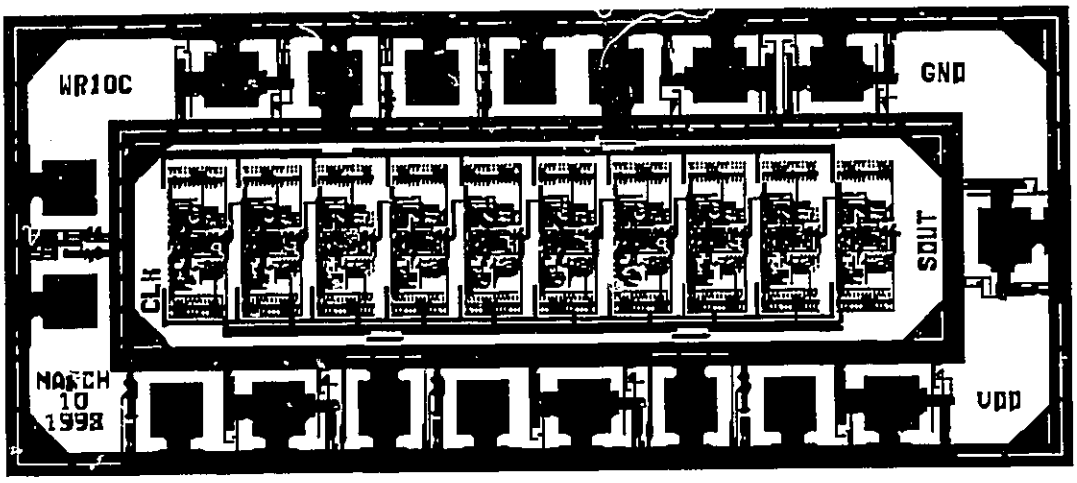

Figure 4.17 Chip Layout with Ten Critical Path Adders

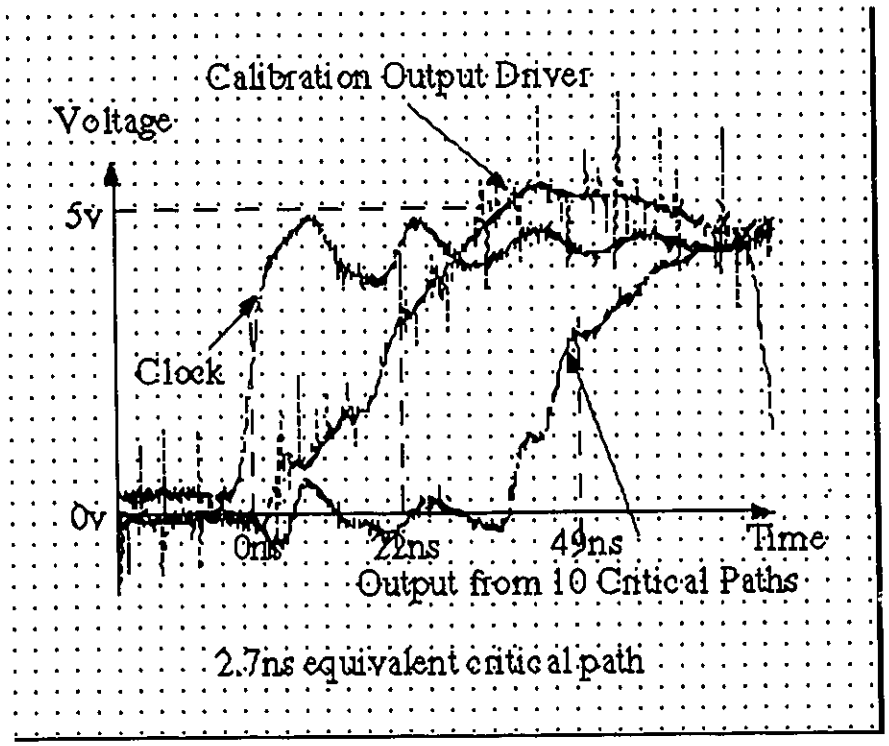

Figure 4.18 Testing Result of the 10-Critical-Path-Adder

4.5 Summary

A new CLA architecture has been designed and implemented in 1.2μ CMOS technology. The new feature of this design is the utilization of a recently introduced *pseudo complement* mapping technique, an enhanced multiple Domino logic (EMODL) circuit style and a sparse carry chain. The worst case delay path of this 32-bit adder is reduced to three gate stages instead of five reported in a recent paper by Hwang and Fisher.

The 32-bit adder is fabricated in 1.2μ CMOS technology with a smaller area than the Hwang and Fisher design (using a 0.9μ CMOS technology). HSPICE simulation results show that the new design is 13% faster than Hwang and Fisher's. A ten serial connected critical path test circuit is fabricated with calibration circuitry to remove the delay caused by the I/O pads and drivers. The measurement results from this test chip verify the HSPICE simulation results obtained from the complete 32-bit adder.

Chapter5

Area-Time Optimization

5.1 Introduction

The *Area-Speed* optimization of adders has been extensively studied. Lee and Oklobdzija [19] examined the delay characteristics of carry-lookahead adders with respect to a delay model that accounts for fan-in and fan-out dependencies, and a heuristic method for finding optimal CLA structures wes presented. Wei and Thompson [36] proposed a systematic method of implementing and optimizing a static CMOS CLA with regard to area and time. In [10] Chan, et. al. investigated the worst case carry propagation delays in carry-skip adders and block carry-lookahead adders, and found that delays not only depend on the intrinsic gate delays, but also on how the full adders are grouped structurally into blocks as well as the number of levels. All these studies indicate that area-speed optimization of the adders has to take into account many factors such as fan-in and fan-out contributions to individual gate delay and the effect of varying gate sizes and number of cascaded gated stages.

In this chapter we will analyze the area-delay characteristics of the dynamic CLA designed with our new techniques. In particular, we

will focus on analyzing the critical path delays of several CLA configurations and use the simulation results as guidelines for future CLA design.

5.2 Timing Analysis

The critical path delay is the longest time a signal takes to propagate from the input of the system to the output. This delay depends on the individual gate delay and the number of gates the signal has to travel through in the critical path. The gate delay is a function of the gate fan-in and fan-out, the gate complexity and the size of the devices. The number of cascaded gates in the critical path is mainly affected by gate fan-in and complexity [32] and [33]. In this section we will examine the critical path delay from three major factors.

5.2.1 Selection of $OR\text{-}p_i$ or $XOR\text{-}p_i$

As we know, there are two ways of defining the carry propagate p_i as the intermediate term used in CLA algorithms. The definitions are given in eqn. (2.13) and (2.17). In eqn. (2.13), an inclusive-OR ($OR\text{-}p_i$) operation is applied to the two summands, whereas eqn. (2.17) uses the exclusive-OR ($XOR\text{-}p_i$) operation. Using different p_i definitions will result in a CLA with different architectures and different performances. Especially with the introduction of the MODL circuit technique, which increases the speed and reduces the area, the CLA structure will substantially change, based on the different choices of p_i, based on removal of the sneak path problem.

The sneak path problem has been discussed in some detail in Section 3.3.2. An example has shown how a sneak path occurs when using $OR\text{-}p_i$ and how it can be prevented by replacing $OR\text{-}p_i$ with $XOR\text{-}p_i$. Another solution to this problem, avoiding the use of $XOR\text{-}p_i$, is by decomposing the MODL circuit into several SDL gates. Neither solution is perfect. The drawback of the $OR\text{-}p_i$ with SDL approach is the increase in device count, fan-out load and interconnection complexity between the gates. The advantage is that one

stage of gates which is required for generating the XOR-p_i of each input pair, is eliminated from the critical path. On the contrary, the use of the XOR operation sacrifices one stage to generate p_i for each input pair, but a sneak path free MODL circuit can be obtained.

5.2.2 Fan-In and Number of Stages

Both fan-in size and the number of cascaded gate stages determine the circuit speed. For an adder with a given input word length, the number of stages in the critical path decreases when the gate fan-in increases. Although fewer number of stages may be advantageous to improving critical path delay, it also requires a CLA structure with higher fan-in and fan-out for each stage and more complex interconnection. This tends to increase the circuit delay.

MODL techniques provide circuits with high speed and low device count. In addition, we acquire further hardware savings by introducing EMODL, which generates several sums from a single gate by sharing common factors. With the introduction of EMODL, highly complex gates can be realized such that the number of stages in the critical path can be reduced, which may be advantageous for decreasing the critical path delay. On the other hand, complex EMODL gates may slow down the evaluation speed due to high fan-in, large fan-out load to the previous stage, and non-negligible parasitic capacitance present due to complex interconnections between the transistors in the gate.

According to the RC delay model [12], the dynamic discharge delay is quadratically proportional to the fan-in size, assuming that the transistors in the nFET chain have the same width (square-law delay model) [8]. From this point of view, we should avoid increasing the fan-in of each gate. However, If we limit the gate fan-in to a small size, the number of stages in the critical path will increase. In spite of this, proper transistor sizing may reduce the discharge time when the fan-in is relatively high. This leads to another design topic of transistor sizing. We will describe it qualitatively in the next section.

5.2.3 Transistor Sizing

Here we concentrate on clocked dynamic circuits. There are two phases in the operation of the dynamic circuit: precharge phase and evaluation phase. In the precharge phase, the clock is low, and the output is precharged to V_{DD} through a pFET. The input signals are allowed to stabilized during this phase. In the evaluate phase, the clock switches to logic "1". The worst-case delay appears when all the inputs of a series-connected nFET chain from output node to ground are high, resulting in discharging of the output node to ground. In a typical dynamic circuit, the worst-case evaluation time is longer than the precharge time; therefore, the limiting factor in the clocking speed is the worst-case discharge time of the nFET chain.

Transistor sizing influence on the discharge time can be analyzed qualitatively by observing the domino CMOS five-input NAND gate given by Figure 5.1. FET0 activated the evaluation phase when clock "clk" is high. FET6 is a pFET used to precharge the internal node "N" to V_{DD} when the clock is low. A buffer inverter is connected to the output node to couple the consecutive stages of the Domino logic chain.

When the size of FET5 in Figure 5.1 is reduced, two effects are present: one tends to increase delay; the other tends to decrease delay. First, the delay tends to increase as the resistance of FET5 increases. This is because the charge on the output node takes more time to discharge. Second, the delay tends to decrease as the parasitic capacitance decreases, since the charge stored in the capacitance decreases. When the length of the nFET chain is very long, the second effect is more significant than the first one. This is due to the fact that the charge on the output node and the parasitic capacitance of FET5 has to drain through the summed resistances of FET0-FET5. Decreasing the capacitance of FET5 plays a more significant role in decreasing the discharge time than the increase in resistance of only one FET. We can conclude that in a long nFET chain the delay time can be decreased by decreasing the size of FET5.

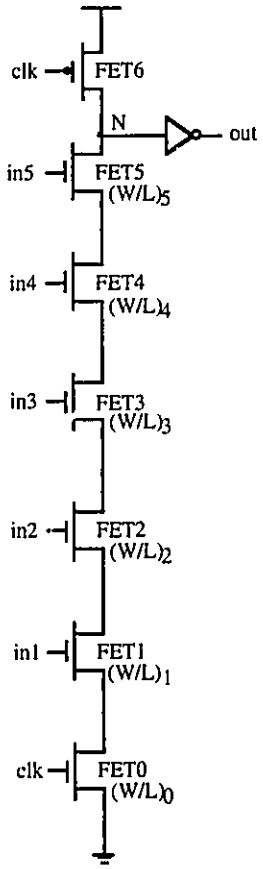

Figure 5.1 nFET Chain with Fan-in of 6

We get a deeper understanding of the problem by studying the generic RC model, in which each nFET is replaced by one link of the RC chain representing the channel resistance and the parasitic capacitance. Such an nFET chain model is shown in Figure 5.2.

The estimate discharge time can be approximated by Elmore's delay formula [12]:

$$t_D \cong \sum_{i=0}^{N} \left(\sum_{j=0}^{i} R_j \right) C_i \qquad (5.1)$$

As we know, R_j is inversely proportional to $(W/L)_j$; the parasitic capacitance C_i increases with channel width, and is approximately proportional to $(W/L)_i$

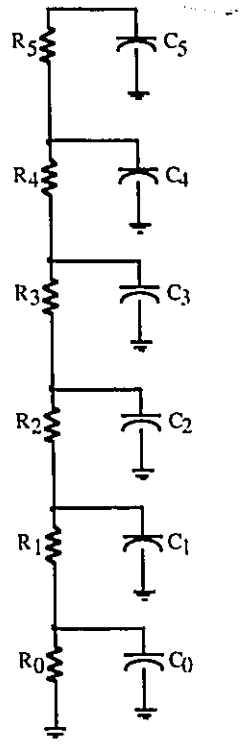

Figure 5.2 RC Model for 6 Fan-in nFET Chain

The discharge delay of the nFET chain, shown in eqn. (5.1), can be rewritten in the form:

$$t_D = R_0(C_0 + C_1 + C_2 + C_3 + C_4 + C_5)$$
$$+ R_1(C_1 + C_2 + C_3 + C_4 + C_5) + R_2(C_2 + C_3 + C_4 + C_5)$$
$$+ R_3(C_3 + C_4 + C_5) + R_4(C_4 + C_5) + R_5C_5$$

Suppose that we reduce the size of each FET by a factor $\alpha_i < 1$, such that

$$(W/L)_i' = \alpha_i (W/L)_i$$

The resistance and capacitance are changed accordingly to:

$$R_i' = \frac{R_i}{\alpha_i} \qquad C_i' = \alpha_i C_i$$

So that

$$\Delta t_D = t_D - t_D'$$

$$= R_0\left[\left(1-\frac{\alpha_1}{\alpha_0}\right)C_1 + \left(1-\frac{\alpha_2}{\alpha_0}\right)C_2 + \left(1-\frac{\alpha_3}{\alpha_0}\right)C_3 + \left(1-\frac{\alpha_4}{\alpha_0}\right)C_4 + \left(1-\frac{\alpha_5}{\alpha_0}\right)C_5\right]$$

$$+ R_1\left[\left(1-\frac{\alpha_2}{\alpha_1}\right)C_2 + \left(1-\frac{\alpha_3}{\alpha_1}\right)C_3 + \left(1-\frac{\alpha_4}{\alpha_1}\right)C_4 + \left(1-\frac{\alpha_5}{\alpha_1}\right)C_5\right]$$

$$+ R_2\left[\left(1-\frac{\alpha_3}{\alpha_2}\right)C_3 + \left(1-\frac{\alpha_4}{\alpha_2}\right)C_4 + \left(1-\frac{\alpha_5}{\alpha_2}\right)C_5\right]$$

$$+ R_3\left[\left(1-\frac{\alpha_4}{\alpha_3}\right)C_4 + \left(1-\frac{\alpha_5}{\alpha_3}\right)C_5\right] + R_4\left(1-\frac{\alpha_5}{\alpha_4}\right)C_5$$

Δt_D will be positive if the following relationship is established:

$$\alpha_5 < \alpha_4 < \alpha_3 < \alpha_2 < \alpha_1 < \alpha_0 < 1$$

This means that the best performance results when we successively reduce the MOSFET size ratios starting from the top transistor (closest to internal node "N") down towards ground. A 10-30% reduction of t_D is generally achieved by using this sizing approach compared to a design using the same area but identical transistor sizes.

The speed of an EMODL gate can be improved considerably by using the discussed transistor sizing techniques. The disadvantages introduced by the square-law model are alleviated so that we can trade gate height for fewer stages in the critical path.

5.3 Experiments with Four CLA Schemes

Based on the analysis of the previous section, we design four different circuit configurations of a 32-bit CLA, and compare their area-time characteristics. The structures of the four CLA adders are illustrated in Figure 5.3, 5.5, 5.7 and 5.9. Each square in the figure represents a domino logic gate. The number in the square indicates the fan-in of that gate. The bold line highlights the critical path in each architecture. The fan-out load is also identified by the bold line. The four architectures are configured differently with an intent to cover a reasonable range of the possible variations of the factors discussed in the previous sections. The first and third schemes employ $p_i = a_i + b_i$ as the carry propagate term. The second and forth schemes use $p_i = a_i \oplus b_i$. Scheme I has a fan-in pattern: 5, 4, 5, with three gate stages in the critical path. The second scheme has a fan-in pattern: 2, 4, 4, 5, with four stages. Scheme III has also four stages but the fan-in pattern is 4, 3, 3, 4. Scheme IV consists five stages with fan-in pattern: 2, 3, 3, 3, 4.

Note that with the utilization of EMODL circuitry, the critical path of the adder may not be the signal propagation path from the least significant bit data input to the most significant bit sum output, as one would expect. It may, in fact, vary according to the configuration of each particular design, depending on the number of stages, the complexity of the nFET tree in the gate, and the gate fan-in and fan-out. For example, the critical path for our first two schemes is from carry-in, c_0, to sum output at bit 20 s_{20}, while the critical path for the latter two schemes is from c_0 to s_{30}.

The first scheme, which is also the architecture presented in the previous chapter, has three cascaded stages only. The fan-out of the first stage in the critical path is 4, which is quite high since MODL gates are replaced by SDL gates in the second stage resulting from the use of *OR*-p_i. The highest fan-in is 5. And the maximum fan-out is 5. Transistor sizes are designed carefully based on the idea discussed in Section 5.2.3. Layout of this scheme is shown in Figure 5.4.

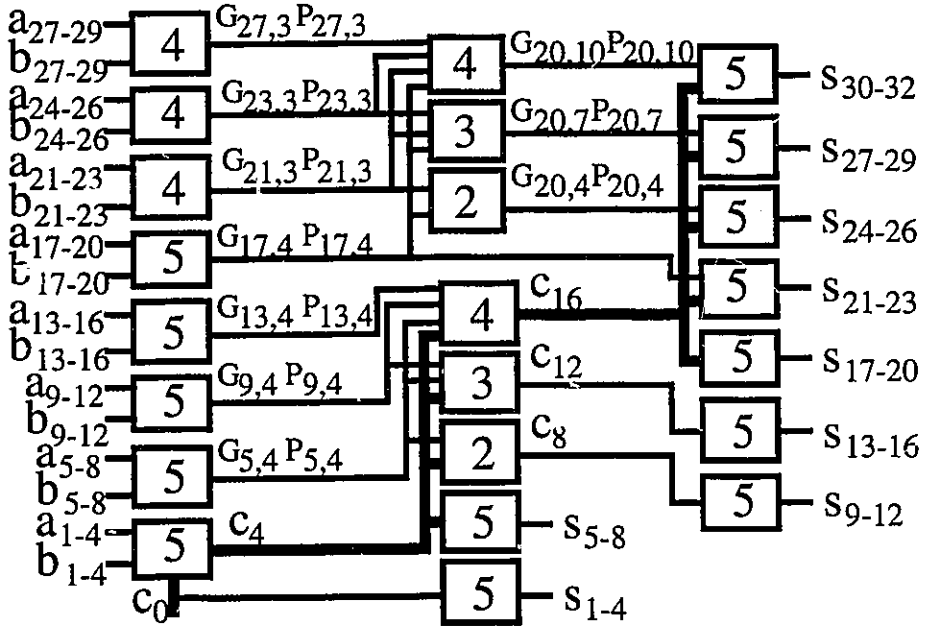

Figure 5.3 Configuration of Scheme I

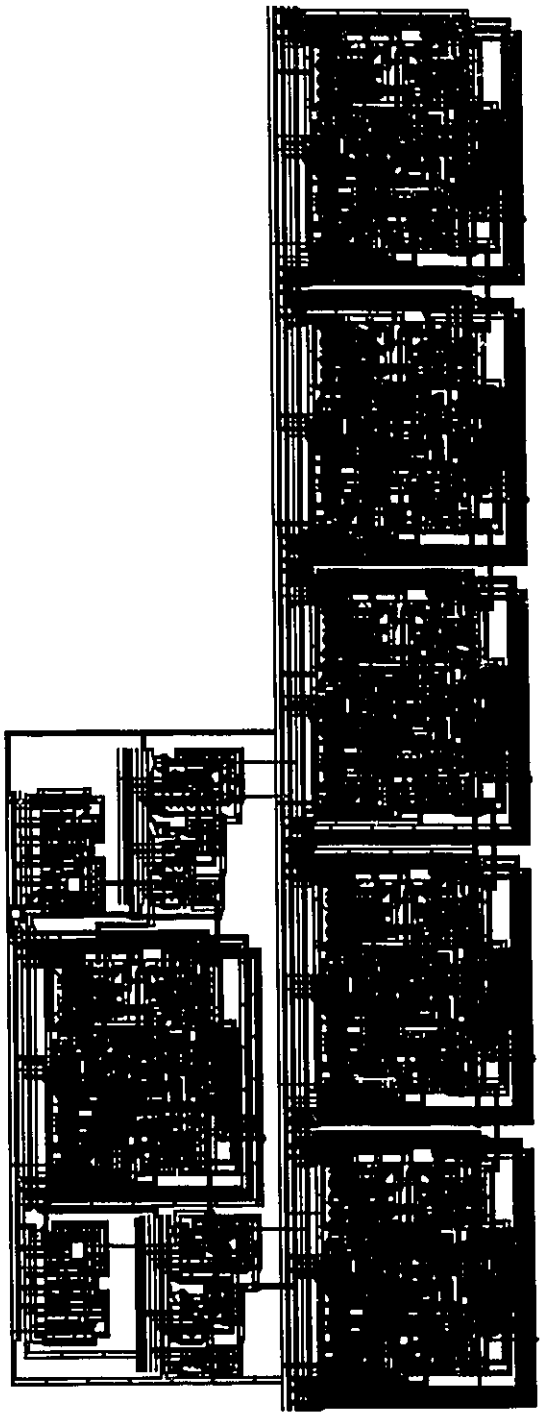

Figure 5.4 Layout of Critical Path for Scheme I

Alternatively, in scheme II, retaining a similar configuration to scheme I, we add a preliminary stage to provide the carry propagate $p_i = a_i \oplus b_i$ for each bit position. As a result, the number of stages is increased to four and MODL gates are able to be used to reduce the fan-out of the first stage to 2. The maximum gate fan-in and fan-out of this schemes are still five. The layout of the critical path is shown in Figure 5.6.

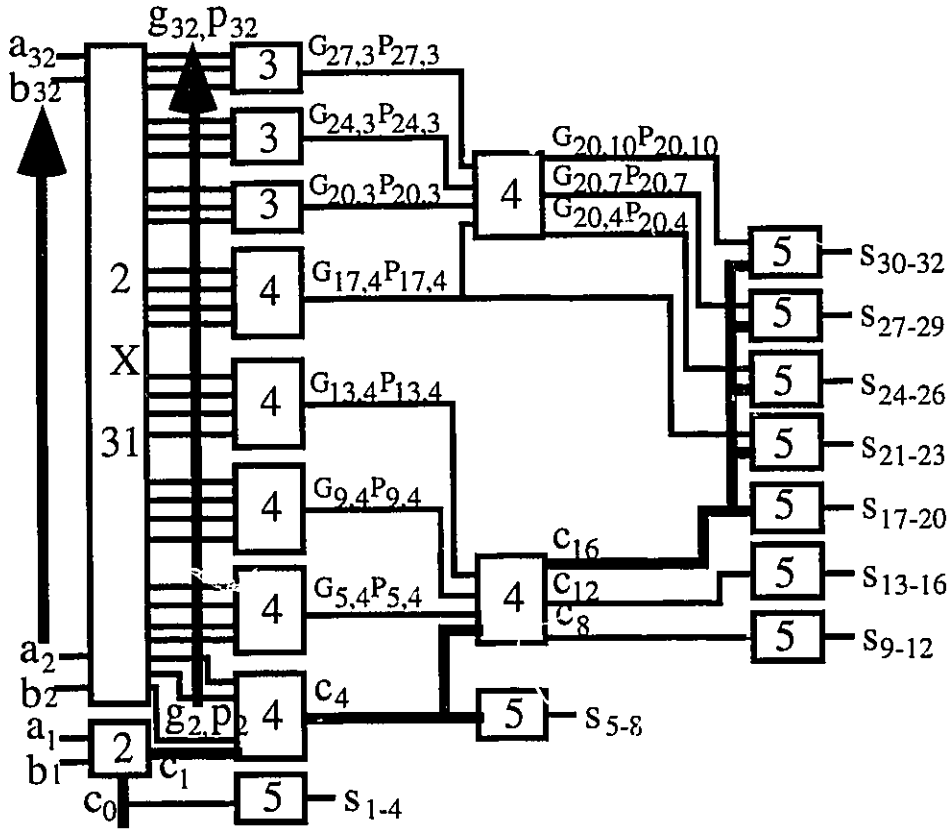

Figure 5.5 Configuration of Scheme II

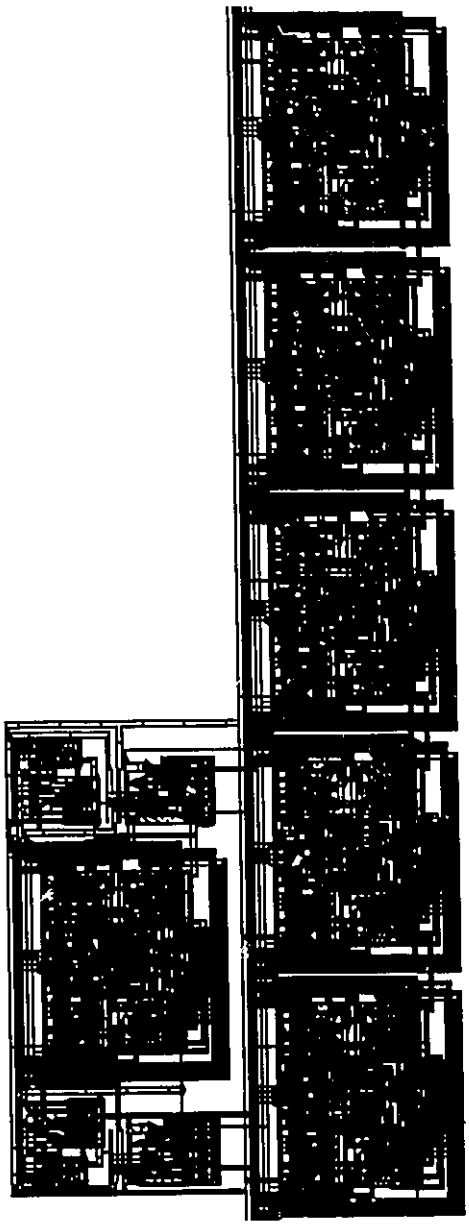

Figure 5.6 Layout of Critical Path for Scheme II

In scheme III, we limit the maximum fan-in to be four in order to shorten the intrinsic gate delay. However, we have to increase the number of stages from three, in scheme I, to four to accomplish the 32-bit addition. Note that the fan-out of the second stage increases significantly due to the use of SDL gates. The corresponding layout is shown in Figure 5.8.

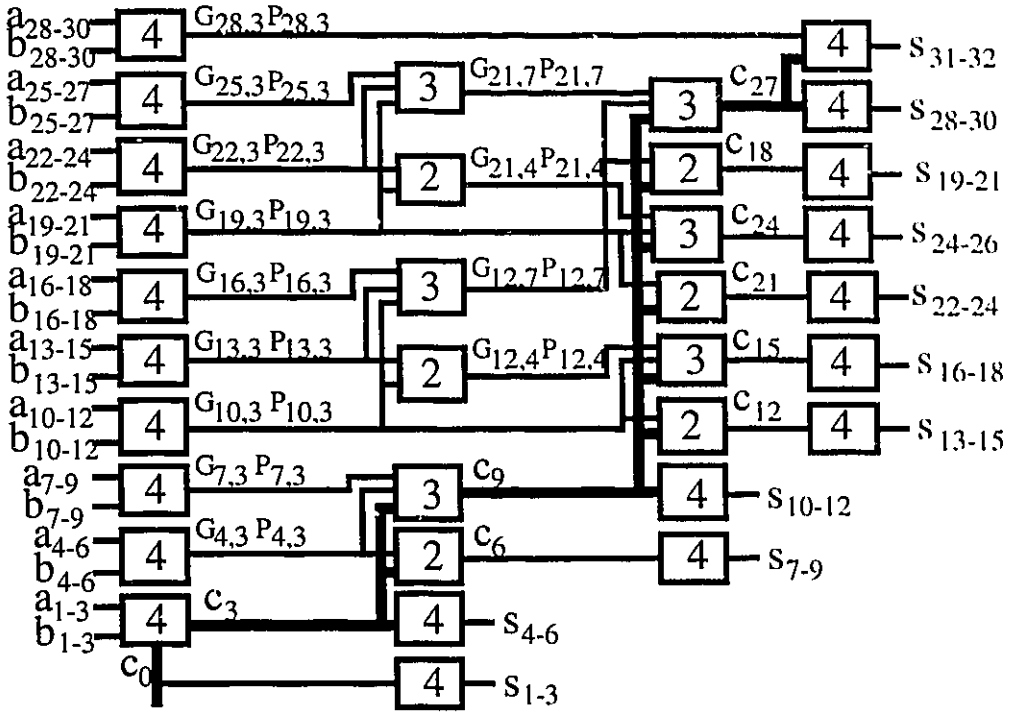

Figure 5.7 Configuration of Scheme III

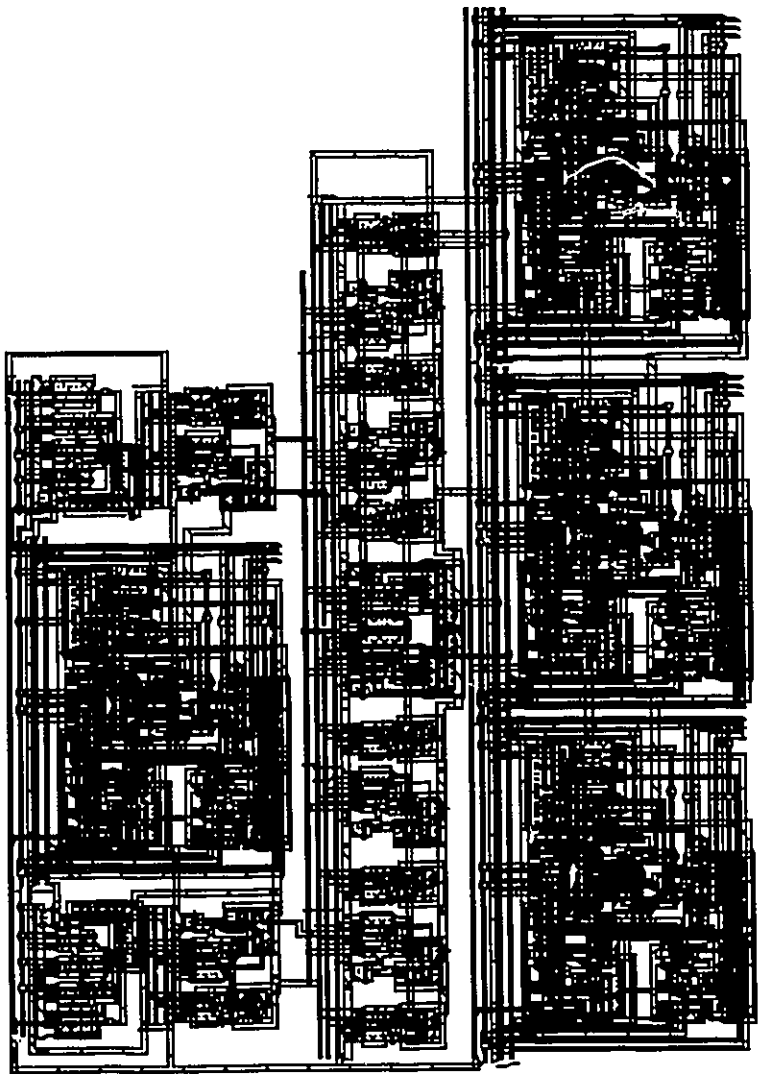

Figure 5.8 Layout of Critical Path for Scheme III

Similar to scheme II, scheme IV adds one stage of XOR-p_i generation circuit to scheme III and switches to MODL circuits while retaining the main configuration of scheme III. Figure 5.10 is the critical path layout of scheme IV.

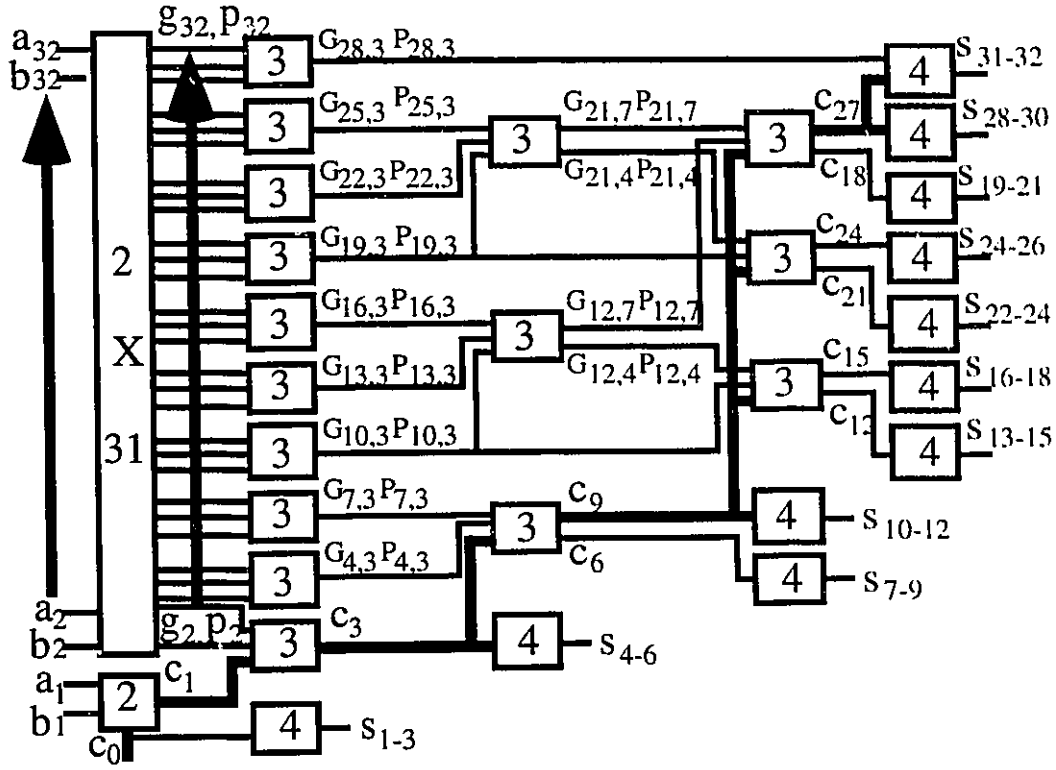

Figure 5.9 Configuration of Scheme IV

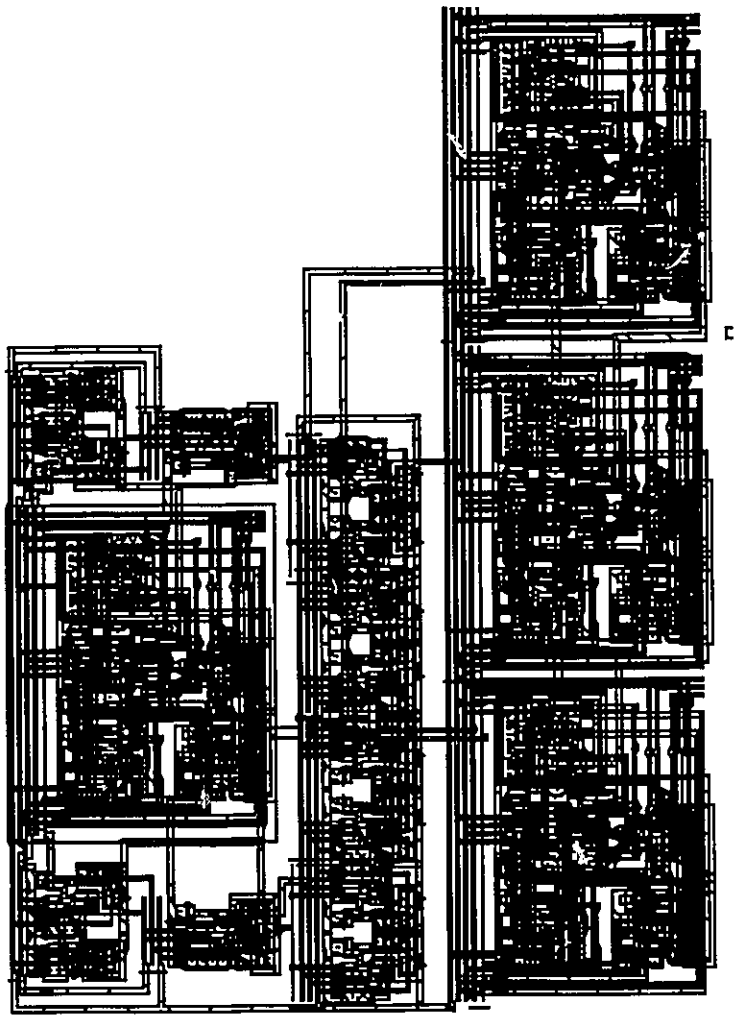

Figure 5.10 Layout of Critical Path for Scheme IV

5.4 Comparison and Summary

The results from the four different architectures are summarized in Table 5.1. The HSPICE simulation results from the mask extracted circuits, and the transistor counts, are also listed. Our experiments indicate that counting active devices provides a reasonably accurate comparison of layout area.

From Table 5.1 we see that scheme I and III, which use *OR* as the carry propagate operator, provide better area-time performance than their counterparts which require one more

stage to compute the *XOR* for every bit position. We also note that the *Speed-Area* result of scheme I, which uses the least number of stages and device count, is the best among all the four schemes, although the fan-in of gates in every stage are quite high. This result is counter-intuitive to the square-law delay model, which is more suitably applied to non-sized transistor trees. Comparing scheme II and III, which have the same fan-out load and same number of stages, scheme III is much faster. This is probably because a design with lower average fan-in results better performance. The delay difference between scheme II and IV is small. Although scheme IV has one stage more than scheme IV, its fan-out and average fan-in are smaller. This may cause its delay slightly smaller than that of scheme II.

Scheme	I	II	III	IV
Type of p_i	$a_i + b_i$	$a_i \oplus b_i$	$a_i + b_i$	$a_i \oplus b_i$
Fan-In Pattern	5, 4, 5	2, 4, 4, 5	4, 3, 3, 4	2, 3, 3, 3, 4
Outputs in Critical Path	c_4, c_{16}, c_{32}, s_{20}	c_1, c_4, c_{16}, c_{32}, s_{20}	c_3, c_9, c_{27}, s_{30}	c_1, c_3, c_9, c_{27}, s_{30}
Max. Fan-out	5	7	7	5
No. of Stages	3	4	4	5
HSPICE Delay	2.1ns	2.63ns	2.36ns	2.55ns
Device Count	1397	1691	1537	1826

Table 5.1. Performance Comparisons

Generally speaking, CLA critical path delay is affected by many factors in a very complex way, especially when transistor sizing technique is involved. Major concerns in determining the CLA architecture are the number of cascaded stages, the fan-in profile and distribution of fan-out load with respect to the related fan-in. The distinguish area-speed advantage of scheme I implies that the primary consideration is the number of stages in the critical path.

Chapter6

Conclusions and Future Research

6.1 Conclusions

High performance adder design techniques have been reviewed. In order to achieve an area-speed optimal adder we need to combine advanced architecture design methodologies with high performance circuit realization techniques. The 32-bit high performance adder introduced in this thesis employs the carry-lookahead algorithm and full dynamic logic circuit techniques. This is based on the investigation of various types of adder architectures and circuit implementation styles. We have found that the carry-lookahead architecture is the fastest and dynamic logic delivers a circuit with less silicon area and higher speed than static logic.

New design methodologies have been developed so that a superfast and compact 32-bit CLA design has been achieved. The recently introduced *pseudo complement* concept has been used to realize complementary carry chains with equal delays. As a result, a dynamic sum unit has been successfully designed. The new EMODL circuit style has been explored for building the sum unit in the adder, and resulting sparse carry chain circuits have been exploited. Only those carries that are necessary to the EMODL gate

are generated in the sparse carry chain and the usual circuits for generating every carry bit are eliminated.

An example 32-bit CLA has been designed based on these techniques and implemented in a 1.2μ CMOS technology. HSPICE simulations of the entire CLA and test results from a dedicated critical-path-adder chip have demonstrated that the performance of our CLA design is better than that of the advanced 32-bit CLA designs appearing in the recent literature.

Additional work has been conducted to provide guidelines for designing optimal CLA architectures, with respect to area and speed, using the above methods. Four CLA schemes, configured with changing fan-in, fan-out and number of stages, are used in a design experiment. Simulation results show that the number of stages is the primary variable in building high performance CLA adders.

6.2 Future Research

Topics for future studies should concentrate on the development of automated optimization approaches to search for area-time efficient CLA architectures. Resulting tools can probably be generalized to any digital system which uses multi-stage domino logic. An interesting addition to such a tool would be the estimation and optimization of power dissipation for CLA designs.

Although our work has concentrated solely on CLA adders, it is quite possible that the design concepts explored in this research work, can be extended to other arithmetic and logic units employed in modern computer systems on silicon.

REFERENCES

[1] Bedrig, O.J.: "Carry-Select Adders", *IRE Transactions on Electronic Computers*, pp. 340-346, vol. EC-11, 1962.

[2] Best, M.J. and Ritter, K.: "Linear Programming---Active Set Analysis and Computer Programs", *Prentice-Hall*, 1985.

[3] Bizzan, S.S.: "High Performance VLSI Circuit Techniques", *Master thesis*, Department of Electrical Engineering, University of Windsor, 1991.

[4] Bizzan, S.S., Jullien, G.A. and Miller, W.C.: "Analytical Approach to Sizing nFET Chains", *IEE Electronics Letters*, pp. 1334-1335, vol. 28, No. 14, 1992.

[5] Brayton, R.K., Hachtel, G.D. and Sangiovanni-Vincentelli, A.L.: "A Survey of Optimization Techniques for Integrated-Circuit Design", *Proceedings of The IEEE*, pp. 1334-1364, vol. 69, No. 10, October 1981.

[6] Brent, R.P. and Kung, H.T.: "A Regular Layout for Parallel Adders", IEEE Transactions on Computers, pp. 280-284, vol. c-31, No. 3, March 1982.

[7] Callaway, T.K. and Swartzlander, Jr., E.E.: "Estimating the Power Consumption of CMOS Adders", Proceedings of 11th Symposium on Computer Arithmetic, pp. 210-216, June 1993.

[8] Chan, P. K. and Schlag, M.D.F.: "Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip", *IEEE Transactions on Computers*, vol. 39, No. 8, pp. 983-992, August 1990.

[9] Chan, P. K. and Schlag, M.D.F.: "A Note on Designing Two-Level Carry-Skip Adders", *Journal of VLSI Signal Processing*, vol. 3, pp. 275-281, 1991.

[10] Chan, P.K., Schlag, M.D.F., Thomborson, C.D. and Oklobdzija, V.G.: "Delay Optimization of Carry-Skip Adders and Block Carry-Lookahead Adders Using Multidimensional Dynamic Programming", *IEEE Transactions on Computers*, pp. 920-930, vol. 41, No. 8, August 1992.

[11] Chu, K.M. and Pulfrey, D.I.: "Design Procedures for Differential Cascode Voltage Switch Circuits", IEEE Journal of Solid-State Circuits, pp. 1082-1087, vol. Sc-21, 1986.

[12] Elmore, W.C.: "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers", *Journal of Applied Physics*, pp. 55-63, vol. 19, January 1948.

[13] Fishburn, J.P. and Dunlop, A.E.: "TILOS: A Posynomial Programming Approach to Transistor Sizing", *Proceedings of International Conference of Computer Aided Design*, pp 326-328, November 1985.

[14] Glover, F. and Laguna, M.: "Tabu Search", *Modern Heuristic Techniques for Combinatorial Problems*, Colin R. Reeves (Ed.), pp. 70-150, Blackwell Scientific Publications, Oxford, 1993.

[15] Hwang, I.S. and Fisher, A.L.: "Ultrafast Compact 32-bit CMOS adders in Multiple-Output Domino Logic", *IEEE Journal of Solid-State Circuits*, pp. 358-367, vol. 24, No. 2, April 1989.

[16] Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P.: "Optimization by Simulated Annealing", *Science*, pp 671-680, vol. 220, No. 4598, May 1983.

[17] Krambeck, R.H., Lee, C.M. and Law, H.S.: "High-Speed Compact Circuits with CMOS", IEEE Journal of Solid-State Circuits, pp. 614-619, vol. Sc-17, No. 3, June 1982.

[18] Ladner, R.E. and Fischer, M.J.: "Parallel Prefix Computation", *Journal of ACM*, pp. 831-838, vol. 27, No. 4, October 1980.

[19] Lee, B.D. and Oklobdzija, V.G.: "Improved CLA Scheme with Optimized Delay", *Journal of VLSI Signal Processing*, pp. 265-274, vol.3, 1991.

[20] Lee, C.M. and Szeto, E.W.: "Zipper CMOS", *IEEE Circuits and Devices*, pp. 10-16, May 1986.

[21] Lynch, T. and Swartzlander, E.E.: "A Spanning Tree Carry Lookahead Adder", *IEEE Transactions on Computers*, pp. 931-939, vol. 41, No. 8, August 1992.

[22] Matson, M.D. and Glasser, L.A.: "Macromodeling and Optimization of Digital MOS VLSI Circuits", *IEEE Transactions on Computer-Aided Design*, pp. 659-678, vol. CAD-5, No. 4, October 1986.

[23] Mead, C. and Conway, L.: "Introduction to VLSI Systems", *Addison-Wesley*, 1980.

[24] Nye, W.T., Sangiovanni-Vincentelli, A.L., Spoto, J.P. and Tits, A.L.: "DELIGHT.SPICE: An Optimization-Based System for the Design of Integrated Circuits", *Proceedings of 1983 IEEE Custom Integrated Circuits Conference*, pp. 233-238, May 1983.

[25] Polak, E.: "Computational Methods in Optimization", *Academic*, New York, 1971.

[26] Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.: "Numerical Recipes in C---The Art of Scientific Computing", Second Edition, *Cambridge Press*, 1992.

[27] Shoji, M.: "CMOS Digital Circuit Technology", *Prentice Hall*, New Jersey, 1988.

[28] Shoji, M.: "FET Scaling in Domino CMOS Gates", *IEEE Journal of Solid-State Circuits*, pp. 1067-1071, vol. Sc-20, No. 5, October 1985.

[29] Shyu, J., Sangiovanni-Vincentelli, A., Fishburn, J.P. and Dunlop, A.E.: "Optimization-Based Transistor Sizing", IEEE Journal of Solid-State Circuits, pp. 400-409, vol. 23, No. 2, April 1988.

[30] Shyu, J.: "Performance Optimization of Integrated Circuits", *Ph.D. Dissertation*, College of Engineering, University of California, Berkeley, November 1988.

[31] Song, P.J. and De Micheli, G.: "Circuit and Architecture Trade-Offs for High-Speed Multiplication", *IEEE Journal of Solid-State Circuits*, pp. 1184-1198, vol. 26, No 9, September 1991.

[32] Wang, J., Wang, Z., Jullien, G. A., Bizzan, S. Luo, W., and Miller, W. C.: "Circuit driven delay optimization of EMODL carry lookahead adder", *Proceedings of 28th Asilomar Conference on Signals, Systems and Computers, Conference Record*, pp. 550-554, Pacific Grove, Cal., USA, Nov. 1-3, 1994.

[33] Wang, J., Wang, Z., Jullien, G. A. and Miller, W. C.: "Area-time analysis of carry lookahead adders using enhanced multiple output domino logic", *ISCAS'94*, May 30-June 2, 1994, London, UK.

[34] Wang, Z., Jullien, G.A., Miller, W.C., Wang, J. and Bizzan, S.S.: "Fast Adders Using Enhanced Multiple-Output Domino Logic", Submitted to *IEEE Transactions on Solid-State Circuits*, June, 1995.

[35] Wang, Z., Jullien, G.A., Miller, M.C. and Wang, J.: "New Concept for the Design of Carry Lookahead Adders", *Proceedings of 1993 ISCAS*, pp. 1837-1840, May 3-6, 1993, Chicago, USA.

[36] Wei, B.W.Y. and Thompson, C.D.: "Area-Time Optimal Adder Design", *IEEE Transactions on Computers*, pp. 666-675, vol. 39, No. 5, May 1990.

[37] Weste, N. and Eshranghian, K.: "Principles of CMOS VLSI Design-A System Perspective", *Addison-Wesley Publishing Company*, 1988.

[38] Zhuang, N. and Yu, H.: "A New Design of the CMOS Full Adder", *IEEE Journal of Solid-State Circuits*, pp. 840-844, vol. 27, No. 5, May 1992.

Appendix A

Examples of Full-Custom Designed Cells in the 32-Bit CLA

A.1 Introduction

Schematics and layouts of cell blocks B_1-B_8 and *XOR* are provided in this appendix. All designs are domino logic circuits. The *pseudo complements* of cells B_1-B_5 are not included because the schematics and layouts are exactly the same as the original cells except for a switching of the input polarities. All the schematics are labeled with transistor sizes, where the unit micron (μ) is omitted. The size of the inverter buffer used in the domino logic gate is labeled with only channel widths. The number above the inverter is the width of the p-channel transistor, and the number below it is the n-channel width. The channel length for all the transistors is 1.2μ.

The cell blocks are divided into three sections. As illustrated in Figure 4.11, there are three stages in the 32-bit adder. Therefore each section represents one stage. Stage one contains blocks B_1-B_3 and *XOR* cell. Stage two comprises blocks B_4 and B_5. Blocks B_6-B_9 are used in stage three. Blocks B_2-B_5 include more than one cell types. For instance, block B_2 includes dells $gk4$ and $pk4$. In such cases, all the cells that belong to one block are listed under the category of this block.

Labeling of inputs and outputs are taken from the schematics in Cadence. They are not the same as the logic terms appearing in the equations in this thesis.

A.2 Stage One

A.2.1 Block B_1

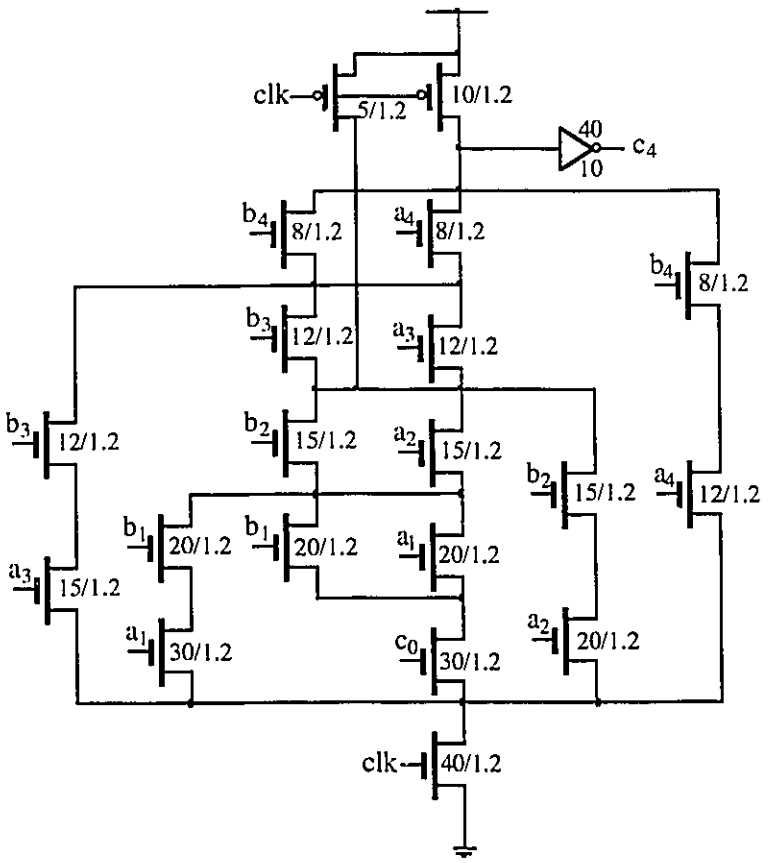

Figure A.1 Schematic of Cell c4

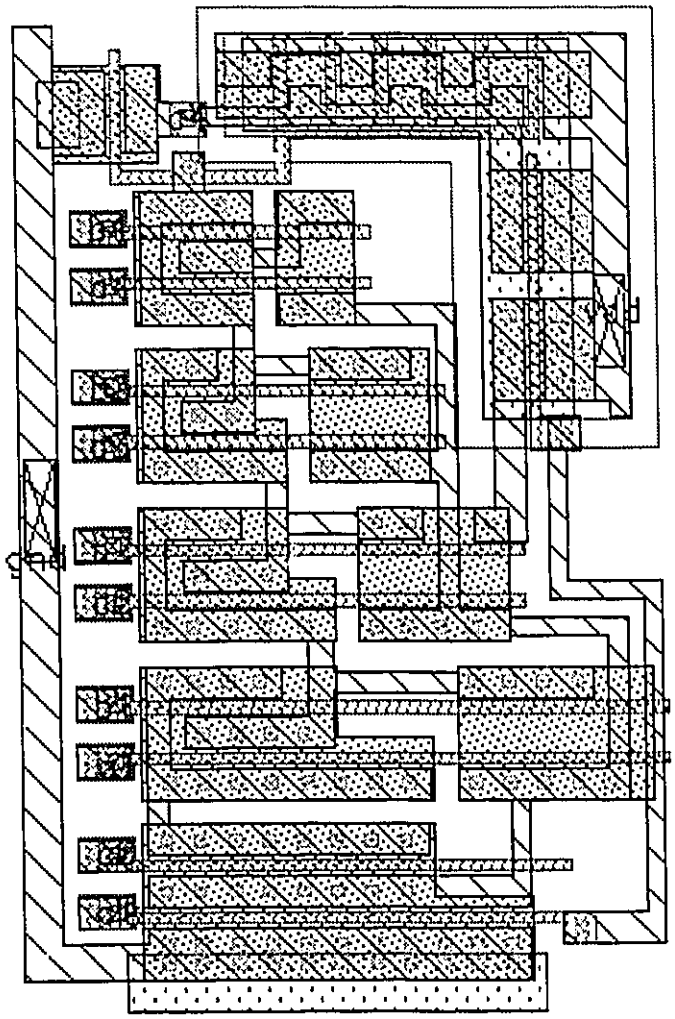

Figure A.2 Layout of Cell c4

A.6.3 Block B_2

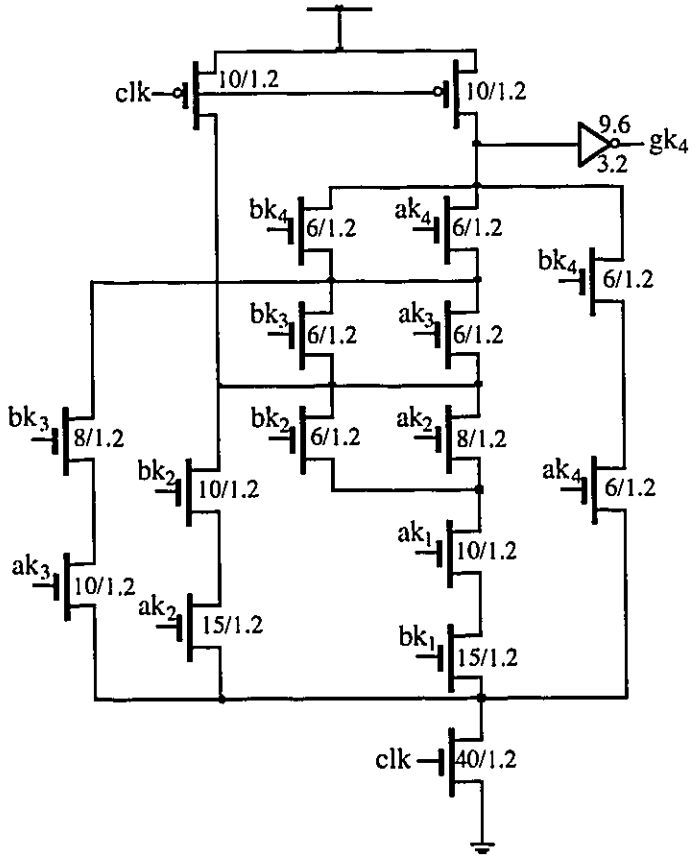

Figure A.3 Schematic of Cell gk4

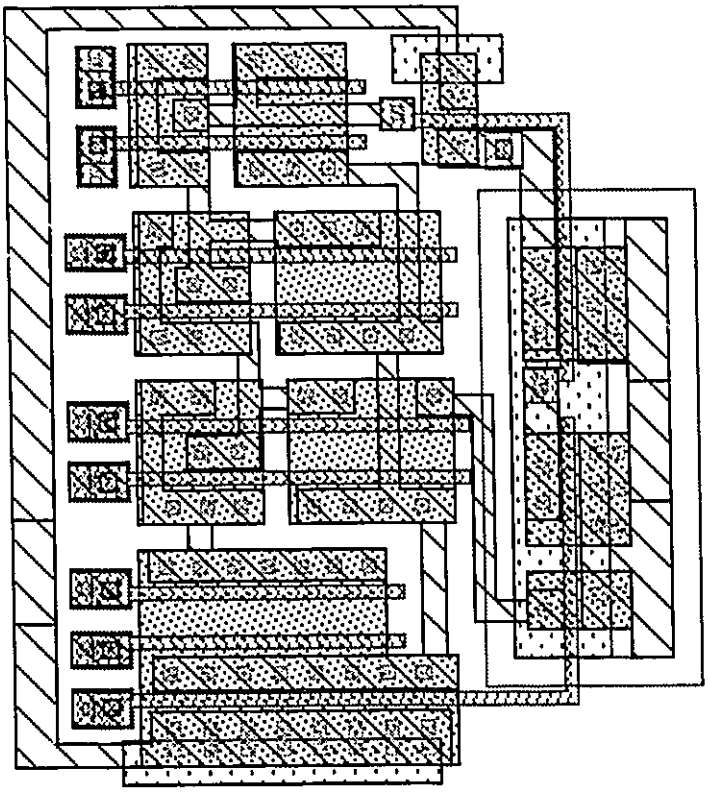

Figure A.4 Layout of Cell gk4

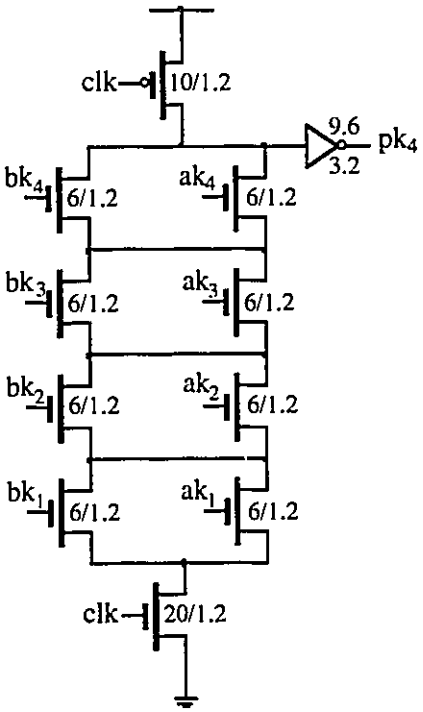

Figure A.5 Schematic of Cell pk4

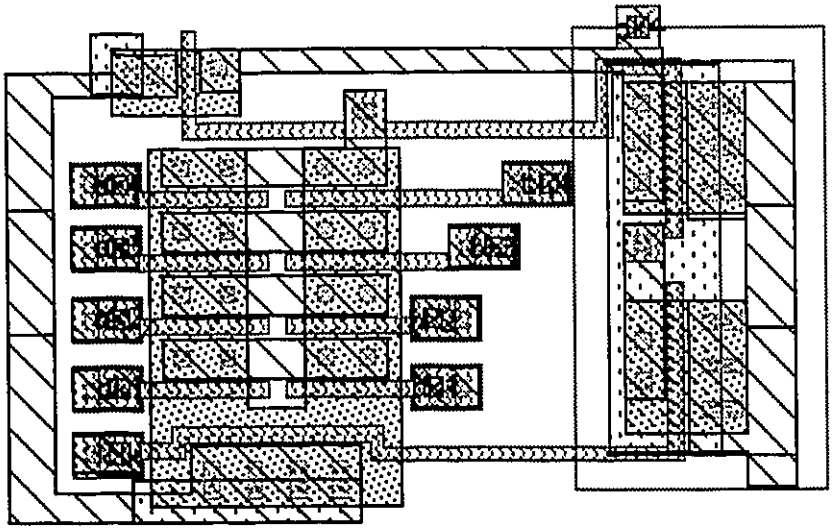

Figure A.6 Layout of Cell pk4

A.6.4 Block B_3

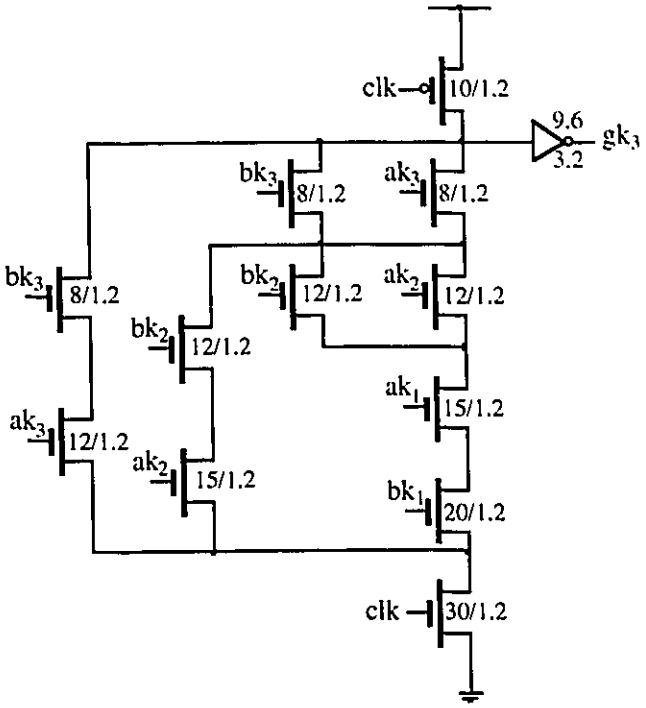

Figure A.7 Schematic of Cell gk3

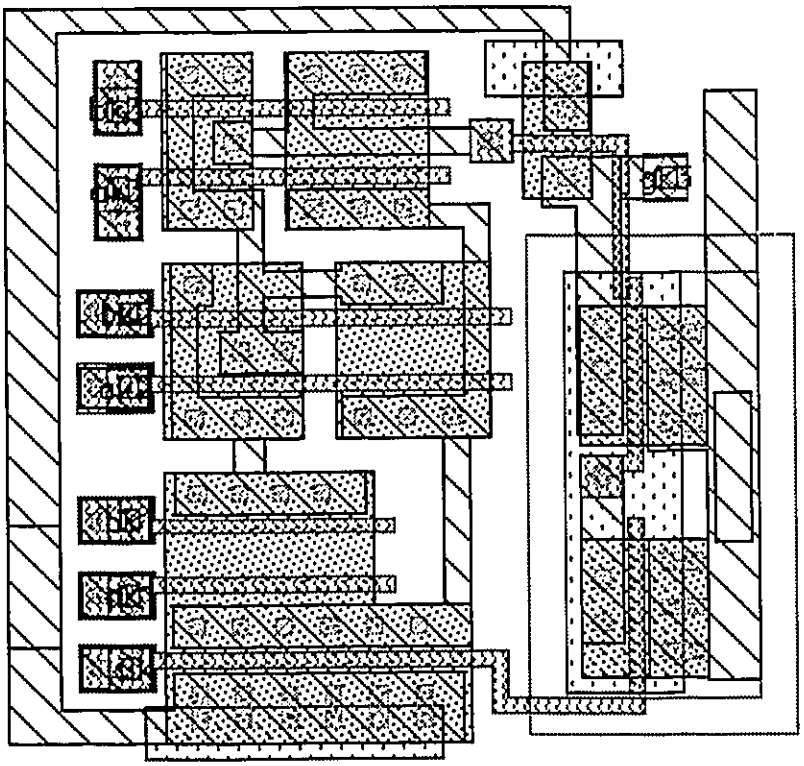

Figure A.8 Layout of Cell gk3

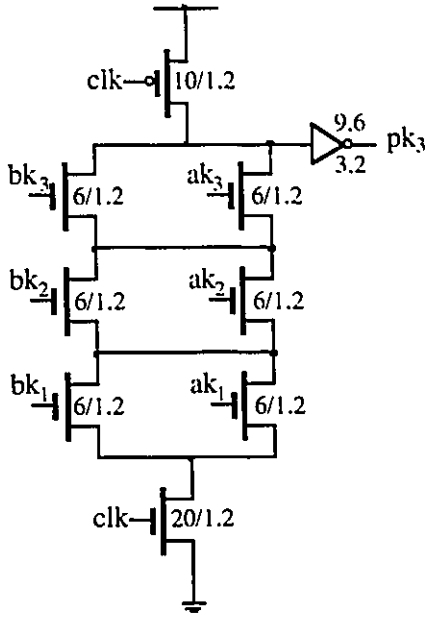

Figure A.9 Schematic of Cell pk3

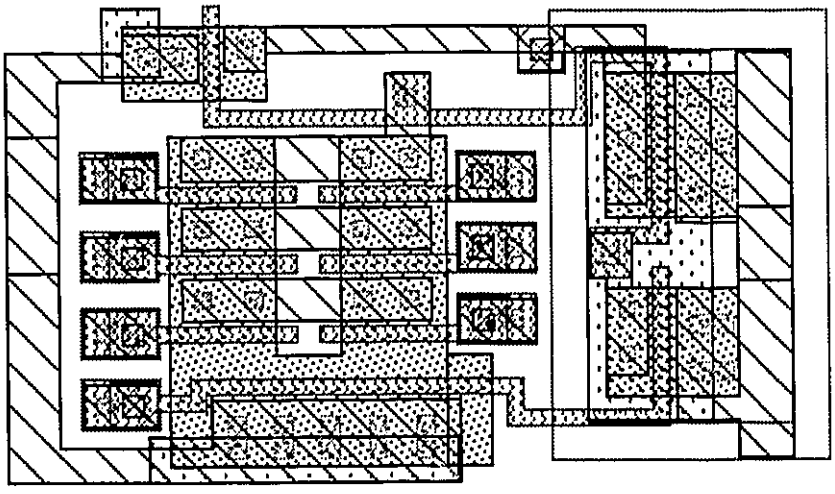

Figure A.10 Layout of Cell pk3

A.6.5 Cell *XOR*

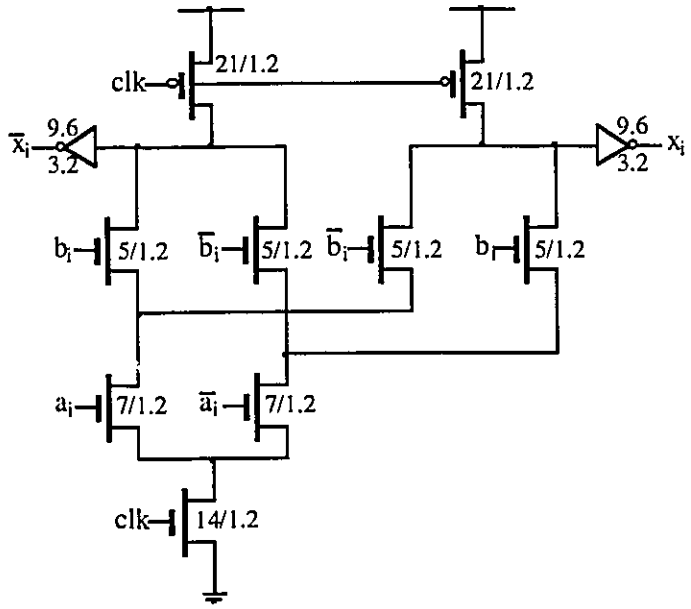

Figure A.11 Schematic of Cell xi

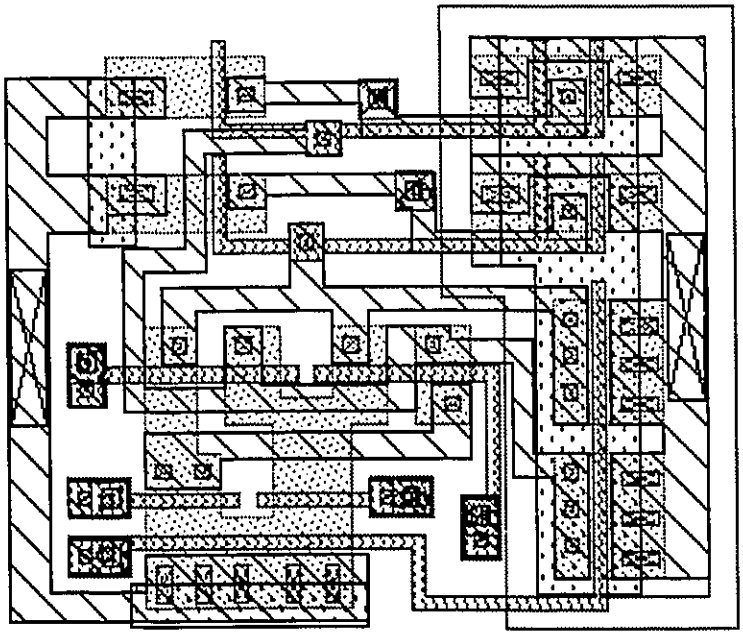

Figure A.12 Layout of Cell xi

A.1 Stage Two

A.3.1 Block B_4

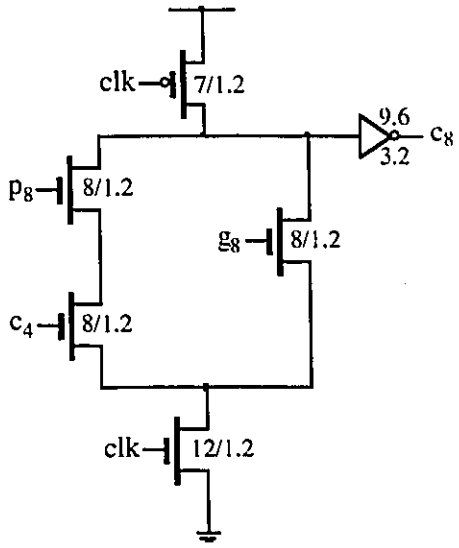

Figure A.13 Schematic if Cell c8

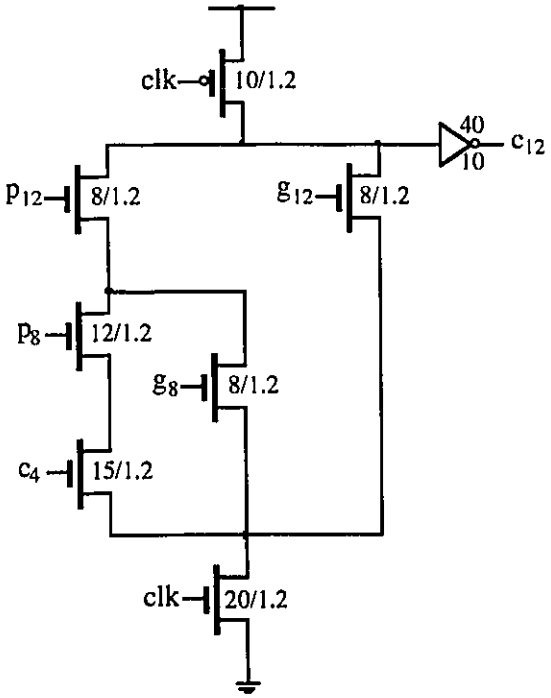

Figure A.14 Schematic of Cell c12

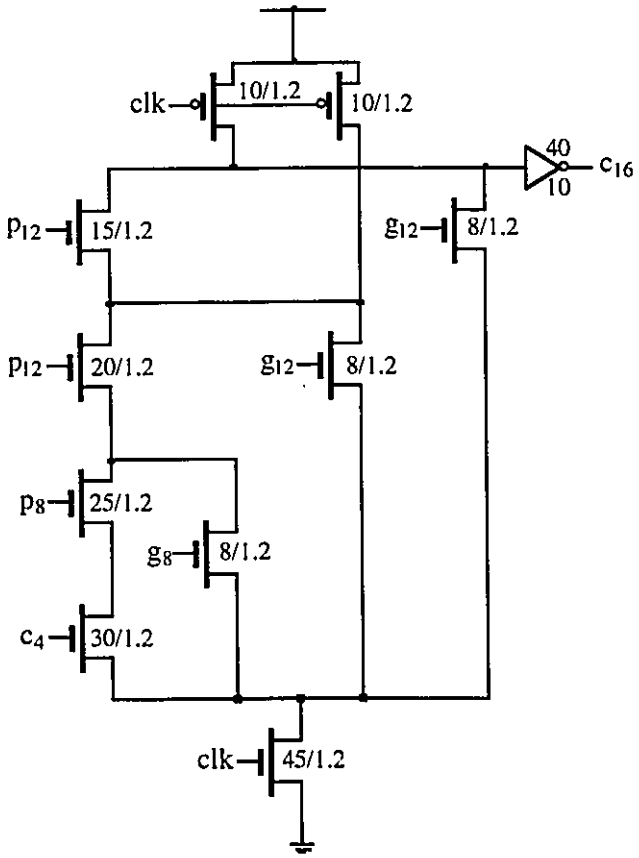

Figure A.15 Schematic of Cell c16

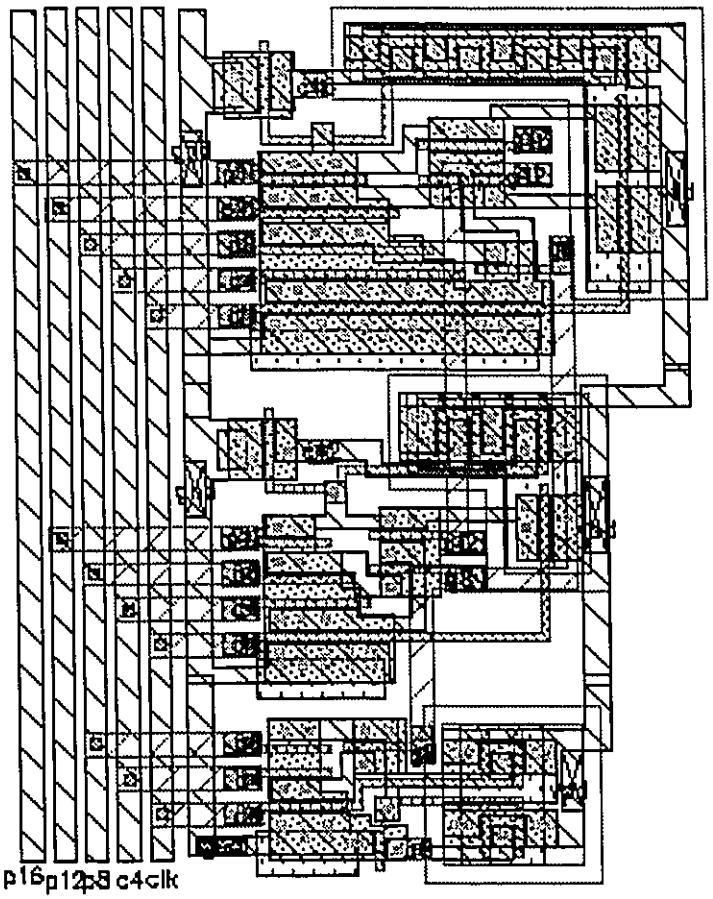

Figure A.16 Layout of Cells c8, c12 and c16 in One

A.3.2 Block B_5

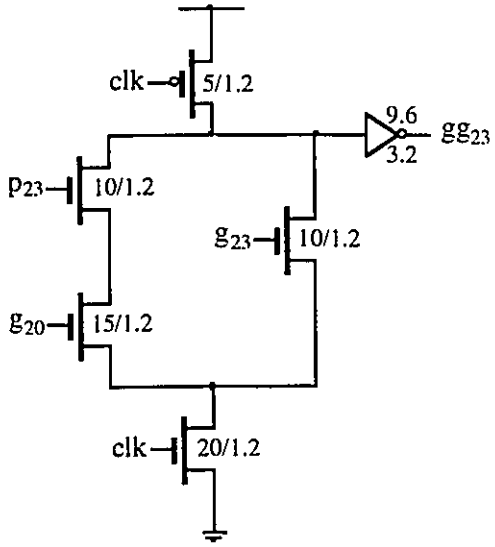

Figure A.17 Schematic of Cell gg23

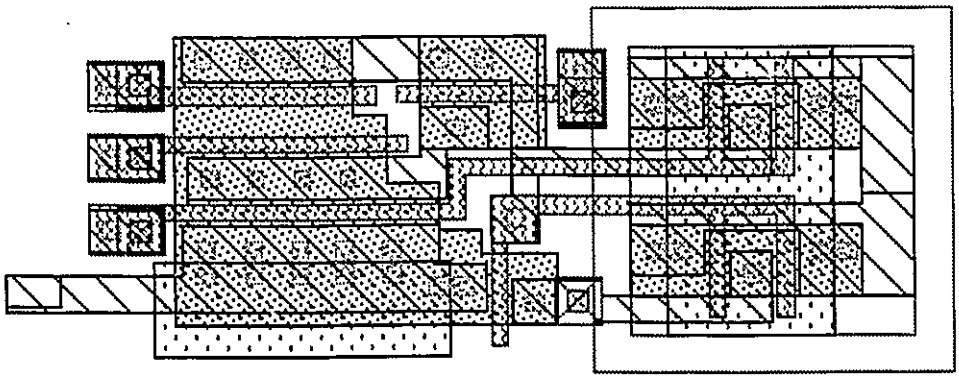

Figure A.18 Layout of Cell gg23

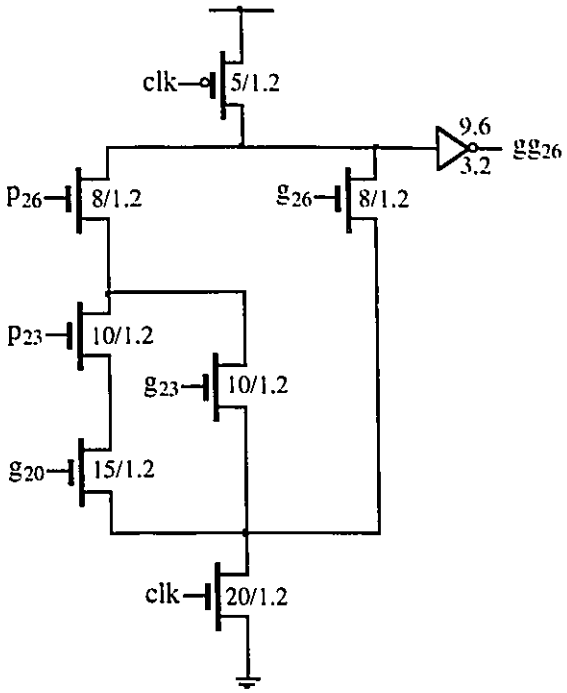

Figure A.19 Schematic of Cell gg26

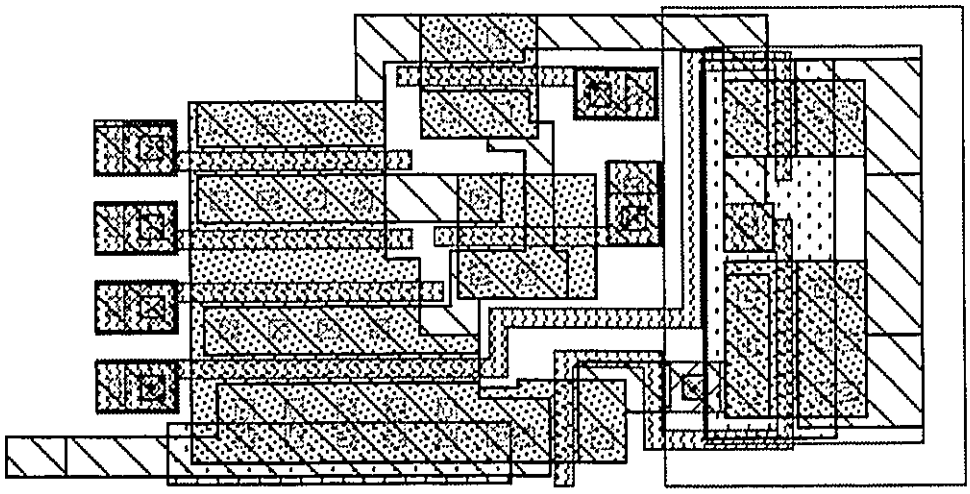

Figure A.20 Layout of Cell gg26

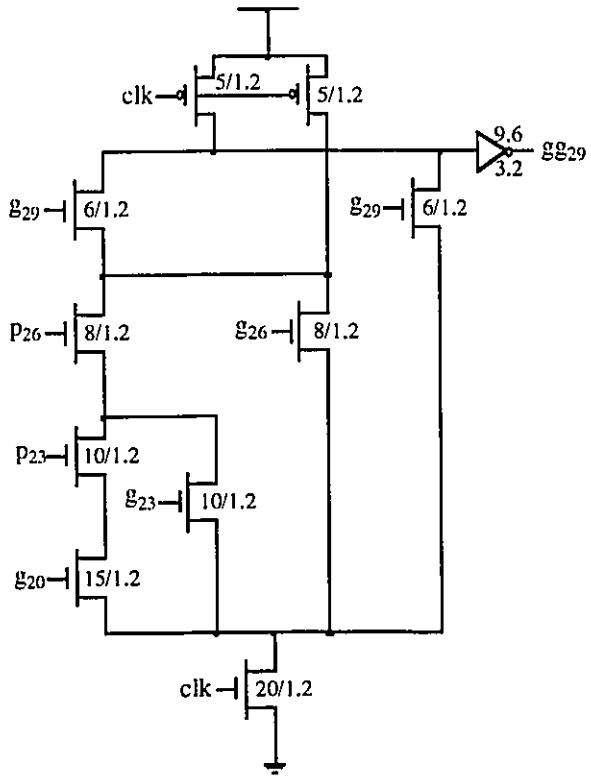

Figure A.21 Schematic of Cell gg29

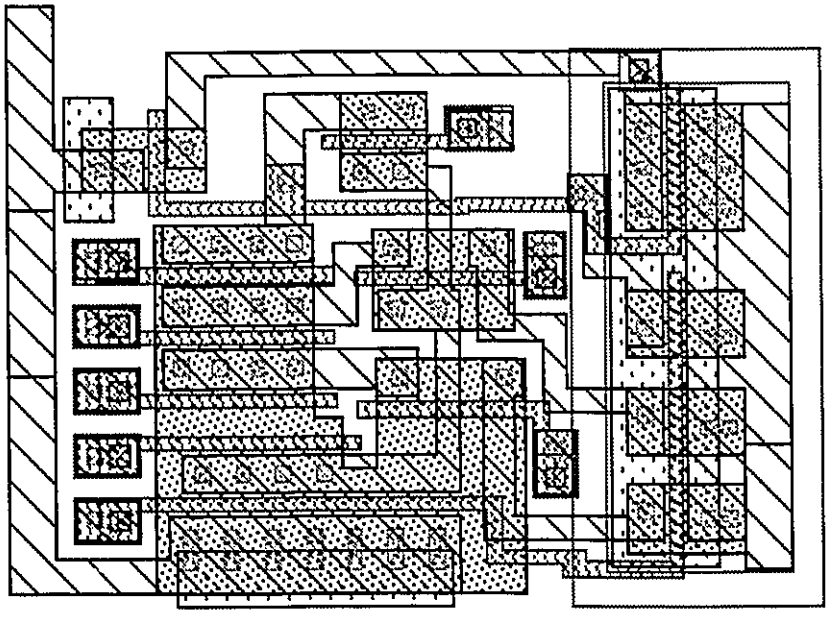

Figure A.22 Layout of Cell gg29

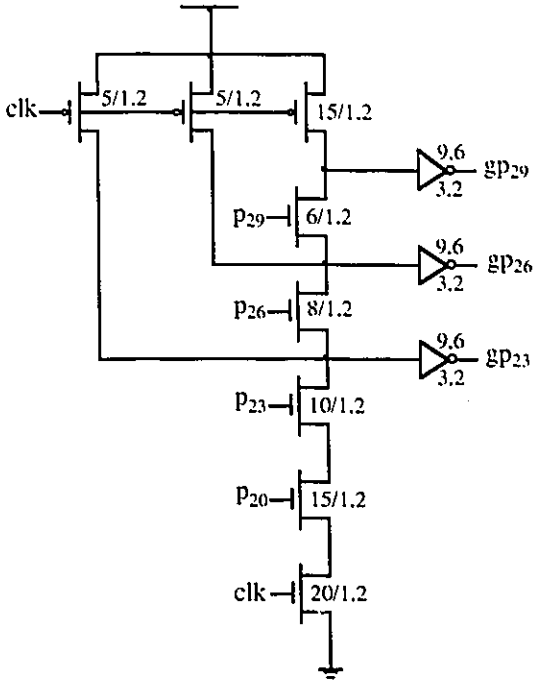

Figure A.23 Schematic of Cell gp23k

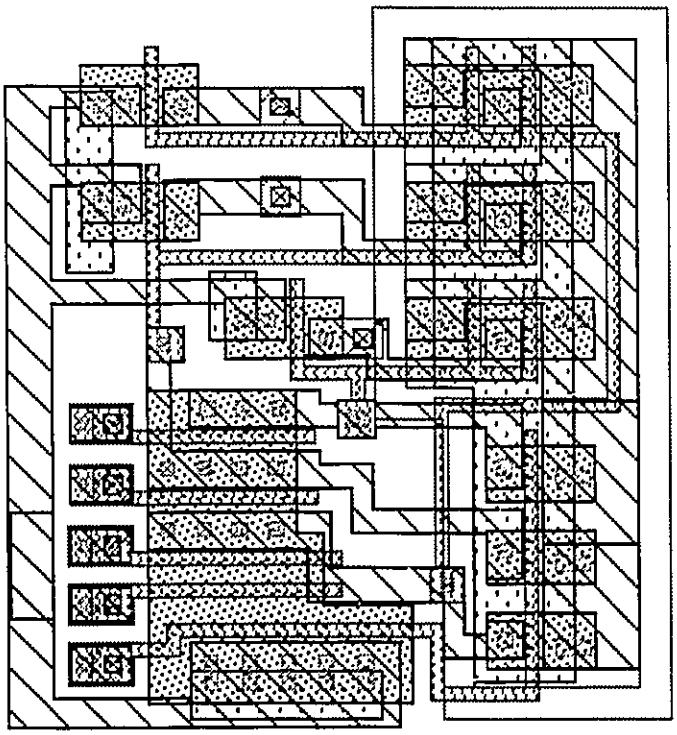

Figure A.24 Layout of Cell gp23k

A.2 Stage Three

A.4.1 Block B_6

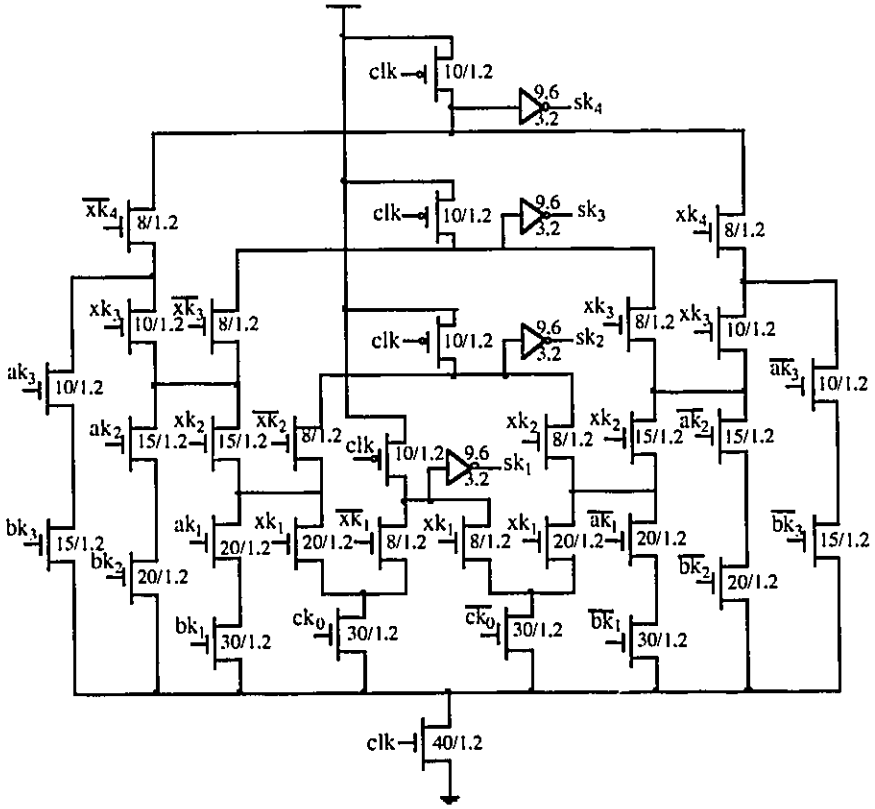

Figure A.25 Schematic of Cell sk4

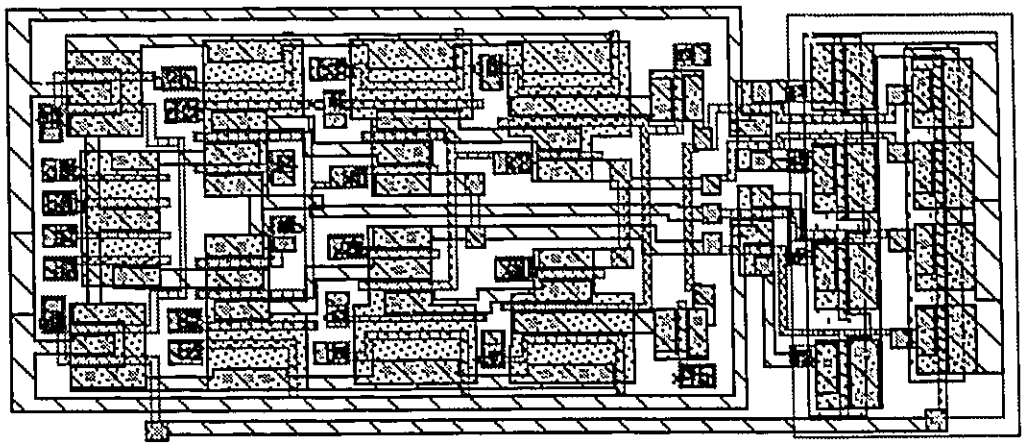

Figure A.26 Layout of Cell sk4

A.4.2 Block B_7

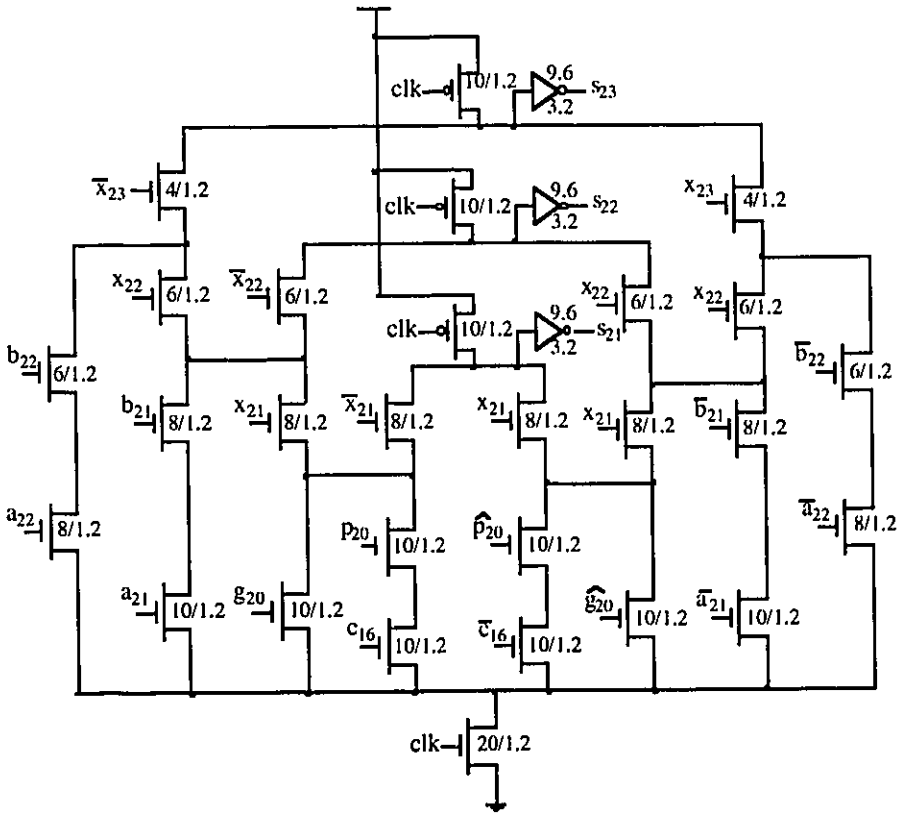

Figure A.27 Schematic of Cell sk3

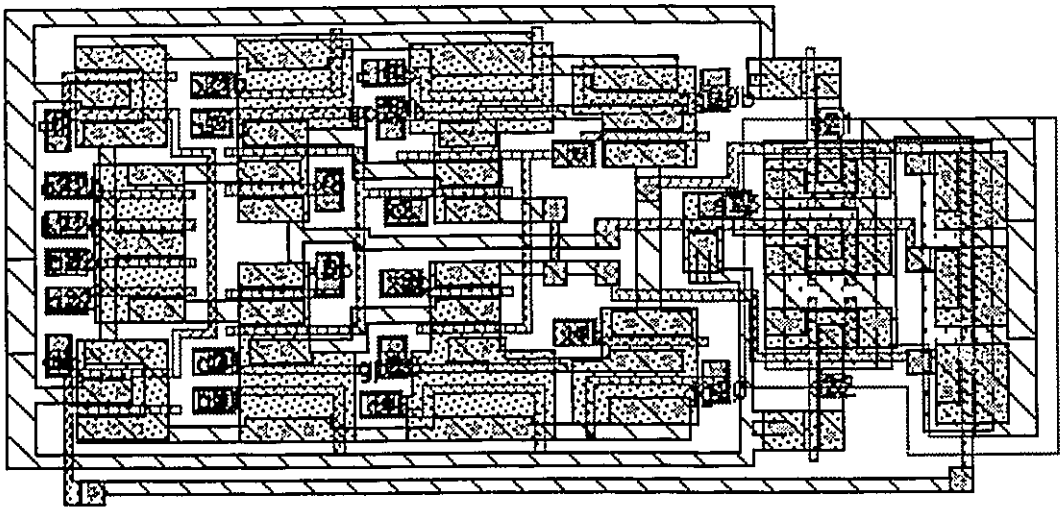

Figure A.28 Layout of Cell sk3

A.4.3 Block B_8

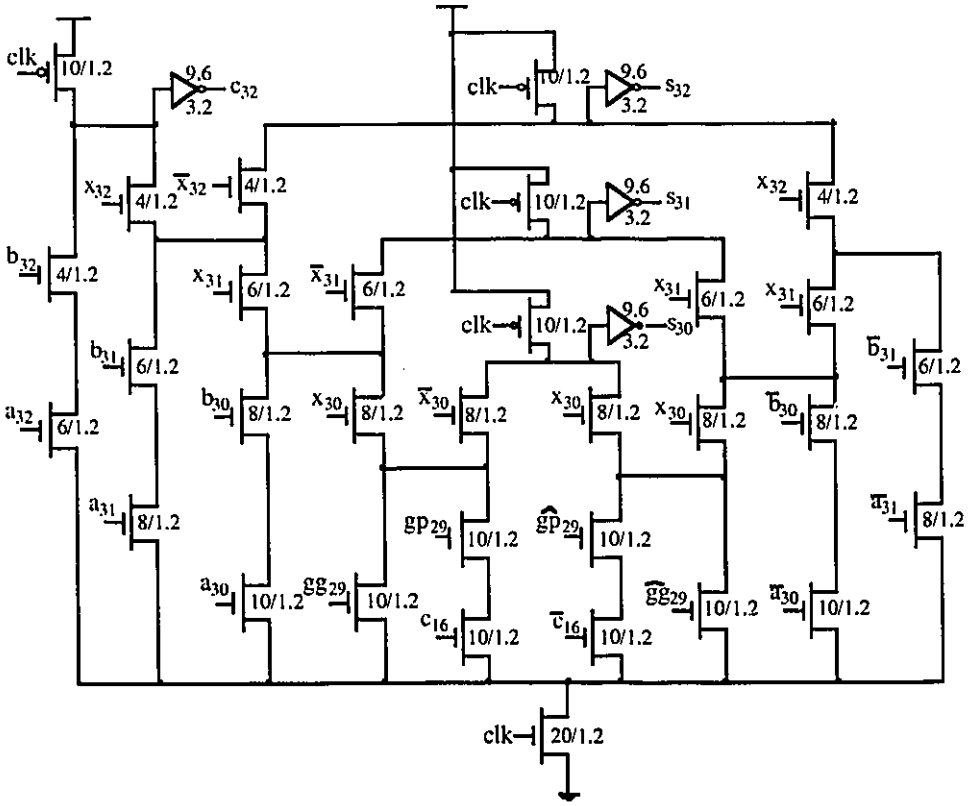

Figure A.29 Schematic of Cell sk32

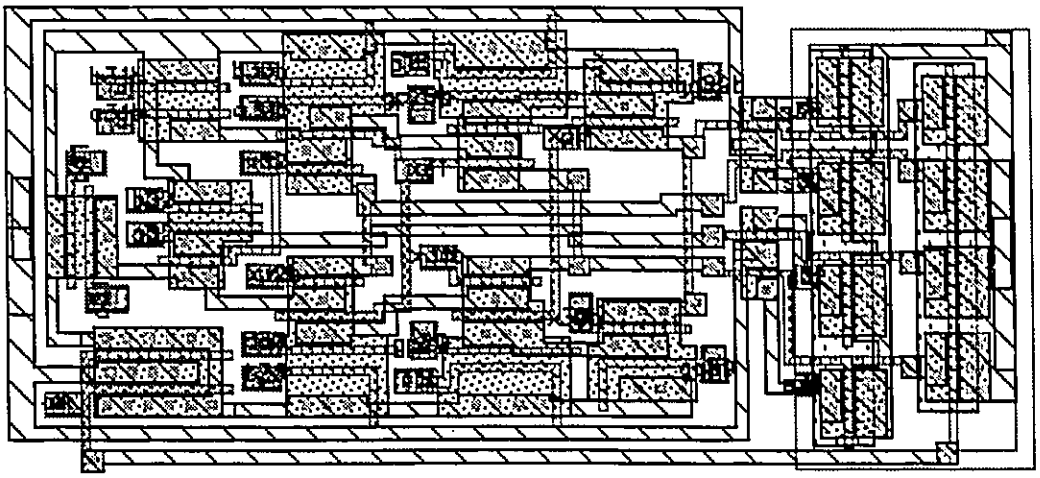

Figure A.30 Layout of Cell sk32

Appendix B

A PROPOSED METHODOLOGY FOR AUTOMATED DESIGN WITH TRANSISTOR SIZING

As we know, dynamic logic circuits evaluate within one clock cycle, and the speed is determined by circuit delay. We can estimate the system delay as the sum of the delay for each gate involved in the critical path.

$$D_T = \sum_{i=1}^{s} D_i$$

where D_T is the total system delay, D_i is the gate delay at stage i and s is the number of cascaded gate stages.

Here we propose an optimization procedure which comprises the following four steps:

Step 1. Set up circuit configuration, such as number of stages and fan-in pattern;

Step 2. Minimize each gate delay D_i by sizing transistors;

Step 3. Comparing resultant D_T with that of the previous structures;

Step 4. Continue with step 1 until D_T is no longer decreasing.

An appropriate algorithm has to be determined for step 1. A simple exhaustive search may be feasible enough for giving configurations that vary in a reasonable range. Other integer

programming technique such as simulated annealing [16] and tabu search [14] may also need to be investigated.

Step 2 is the core of this program. It is actually a transistor sizing problem whose object is to minimize the gate delay under the area constraint. For a dynamic circuit, it is equivalent to sizing the transistors in the series-connected nFET chain in order to improve the discharge time. A distributed RC delay model as described in Section 5.2.3, eqn. (5.1) is widely used. In [27] and [28], based on an RC delay model, Shoji proposed a heuristic scaling technique which can reduce the discharge delay of a Domino CMOS gate by up to 30 percent. The profile of the transistor sizes in the nFET chain can be featured as a ladder so that the FET closet to the ground is the largest, with FET size decreasing monotonically from ground to the output node.

Several authors ([5], [22] and [24]) use mathematical optimization techniques to solve the transistor sizing problem. In this approach, the problem is formulated as a constrained nonlinear mathematical program and a timing analyzer or a circuit simulator such as SPICE is used to evaluate the delay constraints. This approach usually converge to the global minimum and provides more accurate results. The drawback is the extensive computational time required for solving the problem in an unnecessarily large dimensional space.

A combined heuristic and mathematical programming approach to transistor sizing was presented by [29] and [30]. First a fast heuristic algorithm TILOS [13] is used to obtain an initial sizing of the circuit and then a nonlinear optimization method---feasible direction algorithm was adopted to the transistor sizing problem, which can get the solution in spaces of reduced dimensionality. This combined approach is a good compromise between the speed of the heuristic algorithm and the power of mathematical programming.

Sami Bizzan, from our VLSI research group, developed an analytical approach to sizing nFET chain with single variable based on certain assumptions [3] and [4]. The analytical technique uses an empirically determined condition on the time constants of the classical

RC delay mode. Simulation results demonstrate that the analytical technique is comparable to classical numerical sizing methods, yet consumes less computational time.

Our proposed gate delay minimization procedure is to combine the analytical approach and a nonlinear programming algorithm into two phases:

Phase I: Analytical approach which can obtain feasible initial sizing with low computational cost;

Phase II: Quasi-Newton nonlinear programming algorithm [25] and [26] associated with the active set method [2]. The optimal result can be achieved mathematically with quadratic convergence.

The reason we use a nonlinear programming algorithm is that in the RC delay model, discharge delay is a nonlinear function of transistor widths as the parasitic capacitance is proportional to transistor width and the channel resistance is inversely proportional to the width.

Quasi-Newton nonlinear programming with the active set method can be summarized as follows:

Suppose there are m series-connected transistors in the nFET chain, w_i is the width of the transistor i in the nFET chain. w is a set of w_i which includes all the transistors in the nFET chain. The gate delay minimization problem can be expressed as:

$$\min D(w)$$

$$\text{with constraint } a_i^T w \leq w_{imax} \qquad i = 1, \ldots, m$$

Although $a_i = 1$, we still keep it for explaining the active set method more explicitly. w_{imax} is the maximum width allowed for transistor i.

Note that the nonlinear function $D(w)$ can be related to a Taylor series approximation:

$$D(w) = D(w_j) + g_j^T (w - w_j) + \frac{1}{2}(w - w_j)^T H_j (w - w_j) + O\left(\|w - w_j\|^3\right)$$

where g_j is gradient $g_j = \nabla D(w_j)$, H_j is Hessian $H_j = \nabla^2 D(w_j)$.

We can start the computation with a given set of transistor widths $w_0 \leq w_{max}$, where $w_{max} = \{w_{imax} | i = 1, \ldots m\}$. This is obtained from the analytical transistor sizing stated in phase I. Now we define the active set as:

$$I(w_j) = \{i | a_i^T w_j = w_{imax}\}$$

We will manipulate the active set in phase II aiming at achieving a set of transistor widths of the m input nFET chain that leads to the least discharge delay. Starting with the first scheme, letting $w_{j+1} = w_j + \sigma_j s_j$, we will determine s_j with $g_j^T s_j < 0$ so that $D(w_j + \sigma_j s_j) < D(W_j)$. Then the $D(w)$ minimization problem is changed to:

$$\min g_j^T s + \frac{1}{2} s^T H_j s$$

$$\text{with } a_i^T s = 0, \quad i \in I(w_j)$$

Set $\sigma_j = min\{1, \hat{\sigma}_j\}$, $\hat{\sigma}_j$ is called the maximum feasible step size where

$$\hat{\sigma}_j = \frac{b_l - a_l^T w_j}{a_l^T s_j} = \min_{i = 1, \ldots, m} \left\{ \frac{b_j - a_j^T w_j}{a_j^T s_j} \middle| a_j^T s_j > 0 \right\}$$

Having σ_j and the above solved s_j, we now check if $D(w_j + \sigma_j s_j) < D(W_j)$. If it is true, stop working in this scheme. If it is not satisfied, let $\sigma_j \leftarrow \sigma_j / 2$ and repeat solving s_j until $D(w_j + \sigma_j s_j) < D(W_j)$. If it happens where $\sigma_j = \hat{\sigma}_j$, add l to active set $I(w_j)$

Dropping constraints from the active set, we need to compute the Lagrange multiplier μ_j. This second scheme sets the goal to find a set w_j such that

$$-g_j = \sum_{i \in I(w_j)} \mu_i a_i, \quad \text{with } \mu_i \geq 0, \; i \in I(w_j)$$

We now solve the above equation for μ_j. If $\mu_i \geq 0$, we have the optimum solution. If $(\mu_j)_k < 0$, then we drop constraint k from the active set $I(w_j)$. Repeating both schemes until $D(w)$ is minimized, we store the received transistor sizes and continue the automated program with step 3.

Even though the proposed transistor sizing procedure requires the gradient and Hessian to be computed, its quadratic convergence provides a significant reduction in the number of iterations, saving a great deal of computational effort. A preliminary C program for step 2 is attached in Appendix C. The interested reader can use it as a basis for future work

Appendix C

C Program for Transistor Sizing Using Nonlinear Programming with Active Set Method

The following is a preliminary C program for the gate delay minimization step discussed in Appendix B.

```
/*Matrix active[nxn]:active-set NXN dimensions
   Matrix gradient[nxn]:gradient of the objective function
delay(width)
   Matrix hessian[nxn]: second gradient of the objective
function delay(width)
   width[i]:          Width of the ith NFET
   num_fanin[i]       number of fan-in of the ith gate
   num_fanout[i]      number of fan-out of the ith gate
   delay(w[j], i):    objective function, delay of the ith
gate
  num_input:          number of input
  num_stages:         number of stages
*/

#include <stdio.h>
#include <math.h>

#define MINW  5.0e-6      /* Minimum transistor width */
#define MAXW  50.0e-6     /* Maximum transistor width */
#define ITMAX 100         /* Maximum allowed number of
iteration */
#define EPS   3.0e-8      /* Machine precision */
#define CJA   2.9e-4/* Diffusion area capacitance(pf/um2) */
```

```
#define DIFF_HANG  2.8e-6  /* Length diffusion extends
beyond the active area(um) */
#define CJP   3.3e-10      /* Diffusion perimeter
capacitance(pf/um) */
#define RSH   25.0e-6      /* Sheet resistance */
#define CGTA  1.38e-3      /* Gate capacitance(pf/um2) */
#define MASKCHANNEL 1.2e-6 /* Mask channel length(um) */
#define INV_WIDTH 23.0e-6  /* Inverter channel width */
#define FACTOR   CJA * DIFF_HANG + 2. * CJP
#define CONSTANT 4. * CJP * DIFF_HANG
#define WP    10.0e-6      /* Width of the charge
transistor PFET */


float delayfunc(int *height, float *width, int *fan_out,
float *gradient);
void direction(int *height, int *active_index, float
*gradient, float hessian[][20], float *sj);
int checkuj(int *height, int *active_index, float *uj);
void solveuj(int *height, int *active_index, float
*minusgrad, float *uj);
int mulztg(int *height, int *active_index, float *gradient,
float *ztg);
float lnsrch(int *active_index, int *colum, int *fanout,
float *oldwidth, float *funval_old, float *gradient, float
*sj, float *newwidth);
void hessianfunc(int *height, float *oldwidth, float
*newwidth, float *oldgradient, float *newgradient, float
hessian[][20]);
void gradientfunc(int *height, float *width, int *fan_out,
float *gradient);
float snrm(int *n, float *sx);
void solve(int *row, int *col, float zhz[][20], float *mzg,
float *p);
```

```
float delayfunc(int *height, float *width, int *fan_out,
float *gradient) /* calculate gate delay with RC model */
{
  int i, j;
  float gate_width, resist[20], capacit[20], delay, interm,
sumc, sumr;

  delay = 0.;
  gate_width = (float)(*fan_out / 3.) * INV_WIDTH;

  for(i = 0; i <= *height; i++) {
    resist[i] = RSH / width[i];
    if(i == 0)
      capacit[i] = FACTOR * (width[i] + WP) + CONSTANT +
CGTA * MASKCHANNEL * gate_width;
    else
      capacit[i] = FACTOR * (width[i] + width[i - 1]) +
CONSTANT;
  }

  for(i = 0; i <= *height; i++) {
    interm = 0;
    for(j = 0; j <= i; j++) interm += capacit[j];
    delay += interm * resist[i];
  }
/*  delay *= 1.e10;  */

  for(i = 0; i < *height; i++) {
    sumc = 0;
    sumr = 0;
    for(j = 0; j<= i; j++) sumc += capacit[j];
    for(j = *height; j < i; j--) sumr += resist[j];
    gradient[i] = 2 * FACTOR * sumr - resist[i] * sumc /
width[i] + FACTOR * resist[i];
  }
```

```
  sumc = 0;
  for(i = 0; i <= *height; i++) sumc += capacit[i];
  gradient[*height] = FACTOR * resist[*height] -
resist[*height] * sumc / width[*height];

printf("delay = %e\n", delay);

return delay;
}


void direction(int *height, int *active_index, float
*gradient, float hessian[][20], float *sj) /* determine sj,
the descent direction */
{
  int i, j, i_zh, i_zhz, j_zhz, i_p, colum;
  float zhz[20][20], mzg[20], p[20];

  i_zh = i_zhz = j_zhz = i_p = 0;
  colum = *height;

  for(i = 0; i <= colum; i++) {
    if(active_index[i] == 0) {
      j_zhz = 0;
      for(j = 0; j <= colum; j++) {
        if(active_index[j] == 0)
          zhz[i_zhz][j_zhz++] = hessian[i][j];
      } /* end of for(j) */
      i_zhz++;
    } /* end of if(active_index) */
  } /* end of for(i) */

  for(i = 0; i <= colum; i++) {
    if(active_index[i] == 0)
      mzg[i_zh++] = -gradient[i];
  }
```

```
  solve(&i_zhz, &j_zhz, zhz, mzg, p);  /* solve (zThz)p =
mzg */

  for(i = 0; i <= colum; i++) {
    if(active_index[i] == 0) {
      sj[i] = p[i_p];
      i_p++;
    }
    else
      sj[i] = 0.;
  }
}


int checkuj(int *height, int *active_index, float *uj) /*
check if all of uj are greater than 0 */
{
  int i, flag = 0;

  for(i = 0; i <= *height; i++) {
    if(uj[i] < 0.) {
      active_index[i] = 0;
      flag = 1;
    }
  }
  return flag;
}


void solveuj(int *height, int *active_index, float
*minusgrad, float *uj) /* solve for uj */
{
  int i;

  for(i = 0; i <= *height; i++) {
    if(active_index[i] == 1) uj[i] = minusgrad[i];
```

```
    else uj[i] = 0.0;
  }
}


int mulztg(int *height, int *active_index, float *gradient,
float *ztg) /* calculate z^Tg */
{
  int i, k = 0;

  for(i = 0; i <= *height; i++) {
    if(active_index[i] == 0)  ztg[k++] = gradient[i];
  }

  return k;
}


float lnsrch(int *active_index, int *colum, int *fanout,
float *oldwidth, float *funval_old, float *gradient, float
*sj, float *newwidth) /* find the new width set and active
set for less delay value by varying sigma */
{
  int i, l, iter, flag = 0;
  float sigma, sigmahat, funvalue;

  for(i = 0; i <= *colum; i++) {
      if(sj[i] > EPS) {
        sigma = (MAXW - oldwidth[i]) / sj[i];
        if (flag == 0) {
          sigmahat = sigma;
          flag = 1;
        }
        else if (flag == 1) {
          if(sigma < sigmahat) {
            sigmahat = sigma;
```

```
            l = i;
          } /* end of if(sigma) */
        } /* end of if(flag) */
      } /* end of if(sj) */;
  } /* end of for(i) */

  sigma = 1 < sigmahat? 1 : sigmahat;

  for(iter = 0; iter < ITMAX; iter++) {
    for(i = 0; i <= *colum; i++)  newwidth[i] = oldwidth[i]
+ sigma * sj[i];
    funvalue = delayfunc(colum, newwidth, fanout, gradient);
    if(funvalue < *funval_old) {
      break;
    }
    else sigma /= 2.0;
  }

  if(sigma == sigmahat)
    active_index[l] = 1;

  if(funvalue >= *funval_old) {
    for(i = 0; i <= *colum; i++) newwidth[i] = oldwidth[i];
    funvalue = *funval_old;
  }

  return funvalue;
}


void hessianfunc(int *height, float *oldwidth, float
*newwidth, float *oldgradient, float *newgradient, float
hessian[][20]) /* calculate Hessian of gate delay function
*/
{
  int i, j;
```

```
  float delta[20], gama[20], hdelta[20], fac, fae, fat[20];

  for(i = 0; i <= *height; i++) {
    delta[i] = newwidth[i] - oldwidth[i];
    gama[i] = newgradient[i] - oldgradient[i];
  }

  for(i = 0; i <= *height; i++) {
    hdelta[i] = 0.0;
    for(j = 0; j <= *height; j++)
      hdelta[i] += hessian[i][j] * delta[j];
  }

  fac = fae = 0;
  for(i = 0; i <= *height; i++) {
    fac += gama[i] * delta[i];
    fat[i] = gama[i] - hdelta[i];
    fae += fat[i] * delta[i];
  }

  for(i = 0; i <= *height; i++) {
    for(j = 0; j <= *height; j++) {
      hessian[i][j] += (fat[i] * gama[j] + gama[i] *
fat[j]) / fac - (fae / (fac * fac)) * gama[i] * gama[j];
    }
  }
}


void gradientfunc(int *height, float *width, int *fan_out,
float *gradient) /* calculate Gradient of delay function */
{
  int i, j, colum;
  float gate_width, resist[20], capacit[20], delay, interm,
sumc, sumr;
```

```
  colum = *height;

  gate_width = (*fan_out / 3) * INV_WIDTH;

  for(i = 0; i <= colum; i++) {
    resist[i] = RSH / width[i];
    if(i == 0)
      capacit[i] = FACTOR * (width[i] + WP) + CONSTANT +
CGTA * MASKCHANNEL * gate_width;
    else
      capacit[i] = FACTOR * (width[i] + width[i - 1]) +
CONSTANT;
  }

  for(i = 0; i <= colum; i++) {
    interm = 0;
    for(j = 0; j < i; j++) {
      interm += capacit[i] * resist[i];
      delay += interm;
    }
  }

  for(i = 0; i < colum; i++) {
    sumc = 0;
    sumr = 0;
    for(j = 0; j<= i; j++) sumc += capacit[i];
    for(j = colum; j < i; j--) sumr += resist[i];
    gradient[i] = 2 * FACTOR * sumr - resist[i] * sumc /
width[i] + FACTOR * resist[i];
  }

  sumc = 0;
  for(i = 0; i <= colum; i++) sumc += capacit[i];
  gradient[colum] = FACTOR * resist[colum] - resist[colum]
* sumc / width[colum];
}
```

```
float snrm(int *n, float *sx) /* compute square root and
adjust for scaling */
{
    /* Initialized data */

    static float zero = (float)0.;
    static float one = (float)1.;
    static float cutlo = (float)4.441e-16;
    static float cuthi = (float)1.304e19;

    /* System generated locals */
    int i__1;
    double ret_val, r__1;

    /* Builtin functions */
    double sqrt();

    /* Local variables */
    static float xmax, scale;
    static int next, i, j, ix;
    static float hitest, sum;

    sum = zero;
    scale = one;
    hitest = cuthi / (float) (*n);

/* begin main loop */
    for (i = 0; i < *n; i++)
    {
        r__1 = sx[i] / scale;
        if (fabs(r__1) > cutlo)
        {
            if (fabs(r__1) >= hitest)
            {
               xmax = fabs(r__1);
               scale *= xmax;
```

```
                sum = sum / r__1 / r__1 + one;
              }
              sum += r__1 * r__1;
            }
            else if(fabs(r__1) == zero) continue;
            else
            {
              sum = sum / r__1 / r__1 + one;
              sum = sum * r__1 * r__1;
            }
        }

/* end of main loop. */

        ret_val = scale * sqrt(sum);
        return ret_val;
} /* snrm */


void solve(int *row, int *col, float zhz[][20], float *mzg,
float *p) /* solve for p[i] which is the factor of s[j] */
{
        int i, j, k, n, m;
        float r, e[20], c;
        n = *row;
        m = *col;

        for(k = 0; k < n; k++) {
          for (i = 0; i < n; i++) {
            if (i == k) continue;
            r = zhz[i][k] / zhz[k][k];
            for (j = k+1; j < m; j++) zhz[i][j] -= r *
zhz[k][j];
            mzg[i] -= r * mzg[k];
          }
        }
```

```
    for(i = 0; i < n; i++) {
      p[i] = mzg[i] / zhz[i][i];
    }
}


void main() /* the main body */
{
  float stepsize, constraint[20][20], width[20],
newwidth[20], gradient[20], minusgrad[20], ztg[20], uj[20],
sj[20], oldgradient[20], oldwidth[20], activeset[20],
hessian[20][20], norm, funval, funval_new;
  int i, j, iter, active_index[20], colum, row_ztg, flag,
fanout;

/* initialize widths */
          printf("Please input the height of the tree:");
          scanf("%d", &colum);
          printf("Please input the fanout of the tree:");
          scanf("%d", &fanout);

  width[0] = MINW;
  width[colum] = MAXW;
  stepsize = (MAXW - MINW) / colum;
  for(i = 1; i < colum; i++)
    width[i] = width[i-1] + stepsize;

/* initialize active_index */
  for (i = 1; i < colum; i++)  active_index[i] = 0;
    active_index[0] = 1;
    active_index[colum] = 1;

/* initialize hessian */
  for (i = 0; i <= colum; i++) {
    for (j = 0; j <= colum; j++)
      hessian[i][j] = 0.0;
```

```
    hessian[i][i] = 1.0;
  }

/* calculate objective function initial value and gradient*/
  funval = delayfunc(&colum, width, &fanout, gradient);

/* minimization main loop over the iterations */
  for (iter = 0; iter < ITMAX; iter++) {
/
* calculate zTg */
    row_ztg = mulztg(&colum, active_index, gradient, ztg);

/* normalize zTg */
    norm = snrm(&row_ztg, ztg);
    if(norm < EPS) {    /* solve -gj = Atj uj for uj */
      for(i = 0; i <= colum; i++)  minusgrad[i] = -
gradient[i];
      solveuj(&colum, active_index, minusgrad, uj);

      flag = checkuj(&colum, active_index, uj);

      if(flag == 0) {  /* we have the optimal solution */
        for(i = 0; i <= colum; i++) printf("width[%d] =
%f\n", i, width[i]);
        for(i = 0; i <= colum; i++) printf("gradient[%d] =
%f\n", i, gradient[i]);
        printf("delay = %f", funval);
        printf("delay = %f", norm);
        exit(0);
      }
    } /* end of if(norm) */

 /* determine new direction sj */
   direction(&colum, active_index, gradient, hessian, sj);
```

```
/* determine new widths */
      for (i = 0; i <= colum; i++) {
        oldgradient[i] = gradient[i];/* save old gradient */
        oldwidth[i] = width[i];  /* save old width */
      }
      funval_new = lnsrch(active_index, &colum, &fanout,
oldwidth, &funval, gradient, sj, width);
printf("new function value = %e\n", funval_new);

      funval = funval_new;

      hessianfunc(&colum, oldwidth, width, oldgradient,
gradient, hessian);
for (i = 0; i <= colum; i++) {
  printf("active_index[%d]= %d\n", i, active_index[i]);
  for (j = 0; j <= colum; j++) printf("hessian[%d][%d]=
%f\n", i, j, hessian[i][j]);
}

  } /* end of for(iter) */

        for(i = 0; i <= colum; i++) printf("width[%d] =
%f\n", i, width[i]);
        printf("delay = %e", funval);

} /* end of mindelay() */
```

Vita Auctoris

Jinghong (June) Wang was born on June 5, 1965 in Beijing, China. She graduated with the Bachelor of Engineering from Tsinghua University, Beijing, China in 1988. After graduation she worked for three years at the Institute of Microelectronics, Tsinghua University, Beijing, China, in ASIC design tool development and microprocessor analysis. During the course of her Master's study, she worked for a year at Semiconductor Insights Inc., Kanata, Canada, analyzing microcontrollers and DRAM devices. In the meantime, she also received a special gift - her son David Yicheng Luo.