1993

Dynamic logic synthesis with application to self-timed pipelines.

Hong Ming. Chan
University of Windsor

DYNAMIC LOGIC SYNTHESIS WITH APPLICATION TO SELF-TIMED PIPELINES

by

Hong Ming Chan

A Thesis
Submitted to the Faculty of Graduate Studies through the
Department of Electrical Engineering in Partial Fulfillment
of the Requirements for the Degree of
Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

1992

ISBN 0-315-83061-1

Name Hong Ming, Chan

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 — 0544 — UMI

SUBJECT TERM — SUBJECT CODE

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

This thesis describes a new method of designing multiple output dynamic logic suitable for an automatic synthesis procedure. A new cascode voltage switch logic synthesis method is derived with examples demonstrating the procedures. The procedures are summarized into 3 reduction rules. This method is modified to synthesize multiple output domino logic. A companion algorithm for handling "Don't care cases" is also developed. Another algorithm for transforming a non-planar circuit into a planar circuit for use in automatic layout synthesis is presented. An alternate method of realizing cascode voltage switch logic is developed. It is a semi-custom cell design method. The cell uses almost one half the number of the transistors used in a full tree implementation. All of the new synthesis procedures are automated by a program written in PROGRAPH and C. A SRT self-timed divider is implemented to demostrate the use of the new procedures. It is implemented in 3μm CMOS technology.

ACKNOWLEDGMENTS

I would like to express my sincere thanks and appreciation to Dr. G. A. Jullien and Dr. W.C. Miller and Dr. S. Bandyopadhyay for their tremendous support and guidance throughout the progress of this thesis. I give my special thanks to Bruce Erickson for helping me to meet the deadline of layout submission. In addition, I would also like to thank Andy Hung for helping me to draw graphs included in this thesis, Sam Lai and Henry Chan for rescuing the mask layout file. Thanks must also go to all of my friends who stay in school at night. Last but far from least, I would like to thank my parents.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 CONTROL SCHEMES

Traditional VLSI design discipline uses a centralized control scheme. All activities are usually synchronized and scheduled in a central control sub-system. This control center has control wires linking different parts of the system. The communications between sub-systems are also synchronized by this control center. Figure 1.1 shows the conceptual diagram of this centralized control scheme. As the density of the VLSI chip increases, the amount of circuitry that can be packed into one chip also increases. The operating frequency also increases. The management of this control network becomes very difficult since more sub-systems are to be controlled. To solve this problem, a distributed local control scheme can be used, as shown in figure 1.2.

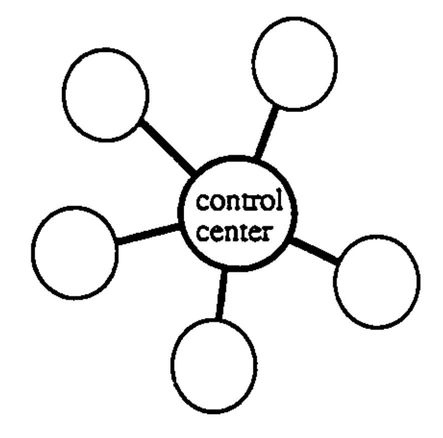


Figure 1.1: Centralized control

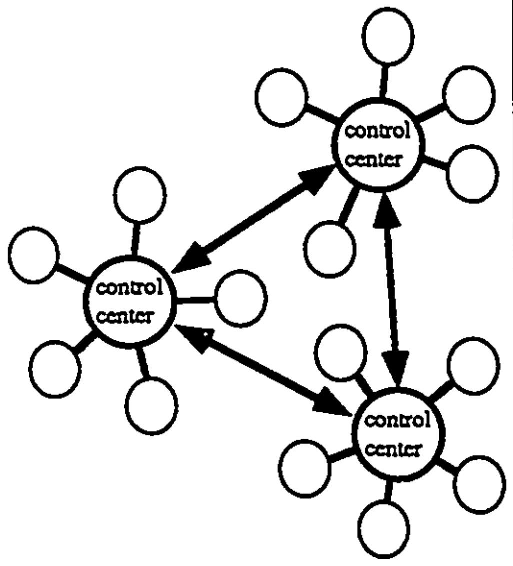


Figure 1.2: Distributed local control

Each control center is responsible for only its neighbors. Between control centers, they communicate using a handshaking asynchronous control scheme which is independent of the operating frequency. This approach has been proposed[1]. By further extending this concept, a complete asynchronous system design discipline is possible. This is called a self-timed system[2]. There are a vast range of control methods possible, between these end-points, as shown in figure 1.3

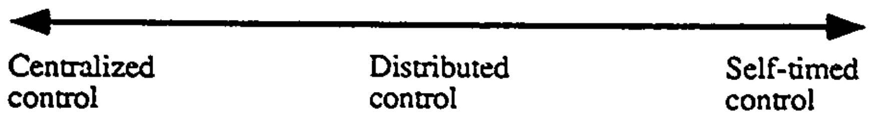


Figure 1.3: Different control schemes

One extreme end is the traditional centralized control. The other extreme end is the total asynchronous self-timed system. This self-timed system control leads to the study documented in chapter 2. In the realization problem of the self-timed functional block, cascode voltage switch logic is found to be the candidate for the function block realization. The research of the new synthesis method for cascode voltage switch logic leads to the development of a more general class of switching tree. Another prospective of switching tree viewing from the design hierarchy decomposition is presented in the next section.

1.2 DESIGN HIERARCHY

VLSI design is hierarchically decomposed into levels of sub-systems. This is necessary for a complex design task since over tens of thousands of transistors are integrated into one system. Figure 1.4 shows a typical hierarchical decomposition. Integration of two or more levels of design hierarchy into one will reduce the final area of the chip. It is because some redundancy wasted in partitioning the design is saved; however, this level binding technique will also increase the design efforts involved. The extra time is used in managing the more complex sub-system blocks in which each of them is combined from two or more blocks. The switching tree[3] is a design technique that combines two levels of design hierarchy into one. Those levels are the gate level and the one above it. The final design decomposition is shown in figure 1.5.

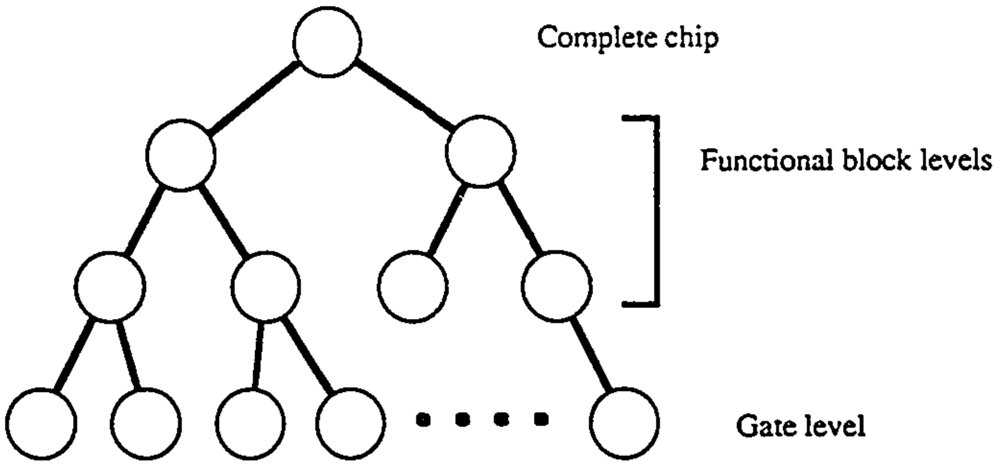


Figure 1.4: Typical design hierarchy

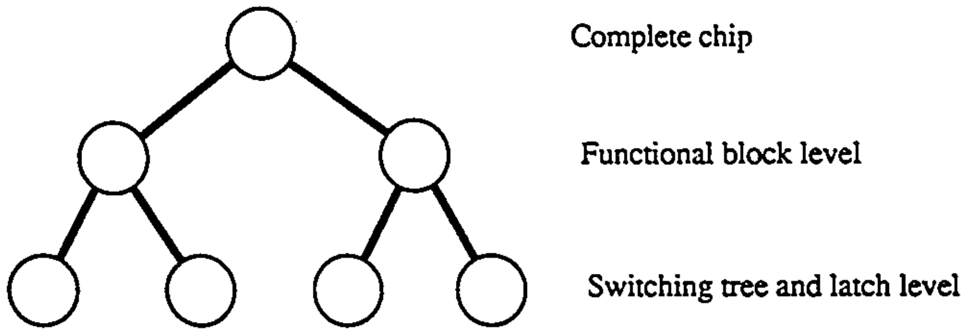


Figure 1.5: Design hierarchy with switching tree

1.3 THESIS ORGANIZATION

This thesis consists of three main parts: the self-timed pipeline[7], multiple output dynamic logic design[6] and SRT[4][5] divider implementation. The main contribution from the author is the derivation of algorithms for multiple output dynamic logic design.

Chapter II gives an overview of the self-timed pipelines. The basic structures are presented and the techniques of analysis are introduced. Different methods of performance improvement are also examined.

Chapter III gives an introduction to dynamic logic design and its problems. The traditional as well as the new proposed design methods of cascode voltage switch logic and multiple output domino logic are examined. An algorithm for handling "don't care cares" is also proposed. A transformation algorithm will be derived with examples. Lastly the synthesis program is presented.

Chapter IV contains an introduction to the traditional division method along with an examination of SRT division. Its implementation in VLSI is presented with simulation results.

Chapter V concludes the thesis with a summary and a few suggestions for possible future research directions.

CHAPTER 2

SELF-TIMED PIPELINE

2.1 INTRODUCTION TO SELF-TIMED PIPELINE

Pipelining is a common high speed information processing architecture used in digital signal processing, arithmetic computation, and machine instruction processing. The basic idea of this architecture is similar to a factory assembly line. Each worker on the line is specialized to work on a small portion of the product. The unprocessed parts are fed in the front of the line and the products are generated from the tail. It is a crucial job for the pipeline designer to create an efficient pipeline architecture. The issues involve communication method and processing element design.

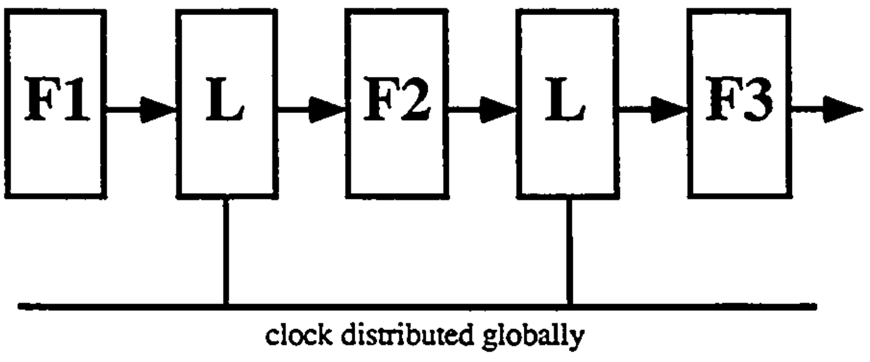


Figure 2.1: Block diagram for synchronous pipeline

Generally, pipelines are classified into two types: synchronous pipelines and self-timed pipelines. The synchronous pipeline, shown in figure 2.1, consists of combinational function blocks separated by latches. The clock signal is distributed to each latch from a common clock generating source. The self-timed pipeline, shown in figure 2.2, consists of adjacent self-timed logic stages with no connecting latch. The communication between two consecutive stages is through bi-directional handshaking control signals. No global clock is required to be distributed to each stage.

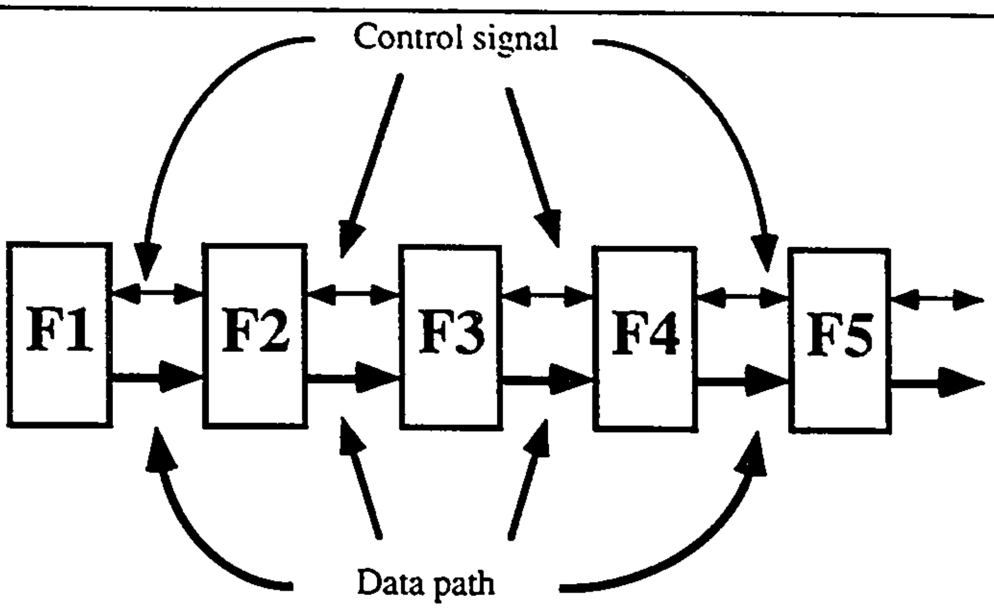


Figure 2.2 Block diagram for self-timed pipeline

A typical block diagram for a clocked system is shown in figure 2.3. Each function block has a delay block attached to the clock distribution line. The delay value for each delay block varies from block to block. The resistance and capacitance of the distribution wire, along with clock loading capacitance, parameterizes the delay block. These differential delay blocks create a well known problem in clocked systems called clock skew. System designers are forced to take a very conservative approach to compensate for this delay. In a multi-phase clock system, the problem can be solved by extending the non-overlapping time; however, the computation time remaining will be reduced.

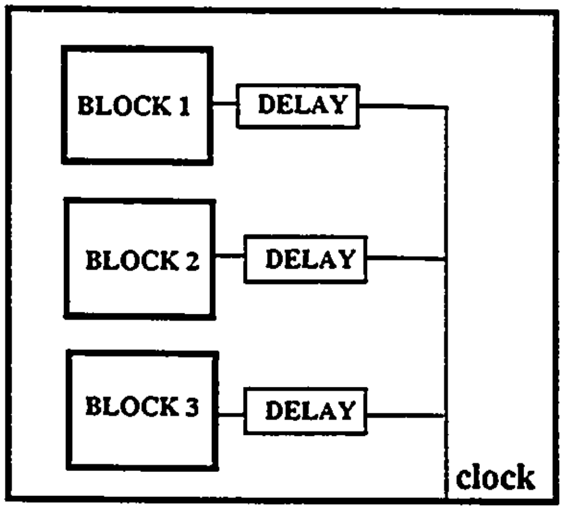


Figure 2.3: Block diagram for clocked system

Another solution is by add

extra delay blocks to equalize the delay generated from clock loading and clock distribution wires. The cost of this method is the extensive design and simulation effort required. Also, it is highly technology dependent. In other words, moving the design from one fabrication technology to another becomes costly. The third method is the total elimination of clock distribution wires by adopting a self-timed system design discipline. Since the self-timed system does not have a global clock, the clock distribution problem does not arise.

The efficiency of a pipelined system is measured by the throughput and the latency. The latter is the time taken to travel from head to tail of the pipeline while throughput is the number of data processed per unit time. For a synchronous pipeline, as shown in figure 2.4, the total latency is equal to the sum of the delay through each block. The following equation describes the relationship:

Total latency=F1+F2 ...... Fn +n *D
where D is the delay through the latch L

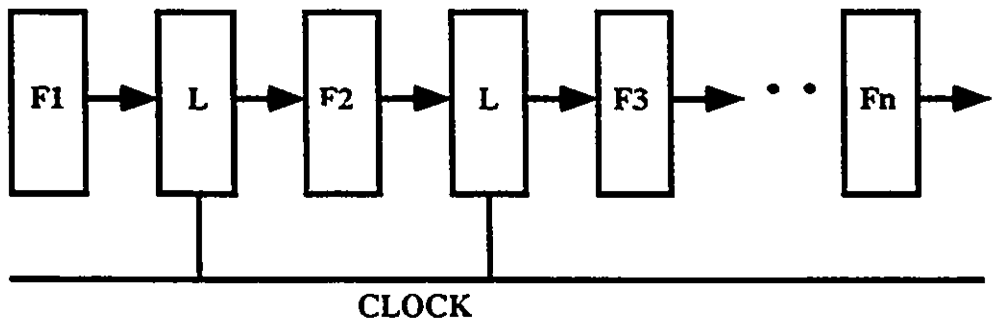


Fig 2.4: Block diagram for the synchronous pipeline

The throughput is the reciprocal of the clock period.

Throughput=1/clock period
where clock period= MAX(F1+D, F2+D, ... Fn+D)

The actual delay time for each of the function blocks may be different. In this case the differences between the actual and the maximum block delay are wasted for each function block.

A better approach is the self-timed pipeline shown in figure 2.5. Total latency can attain the sum of the function block delay under special conditions. Since the self-timed pipeline stage processes the data as soon as the data becomes valid, the throughput is always larger than that of a synchronous one. The handshaking delay is shielded from the calculation of total latency, but not the throughput.

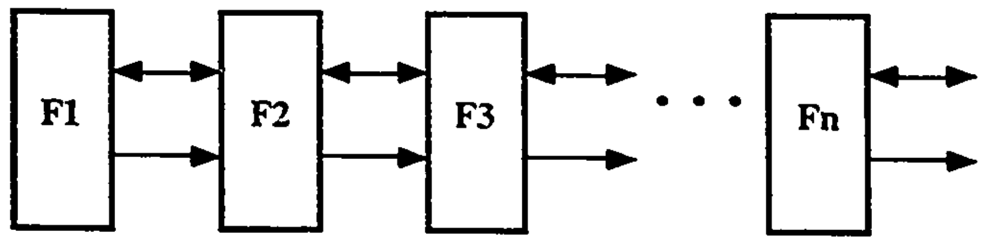
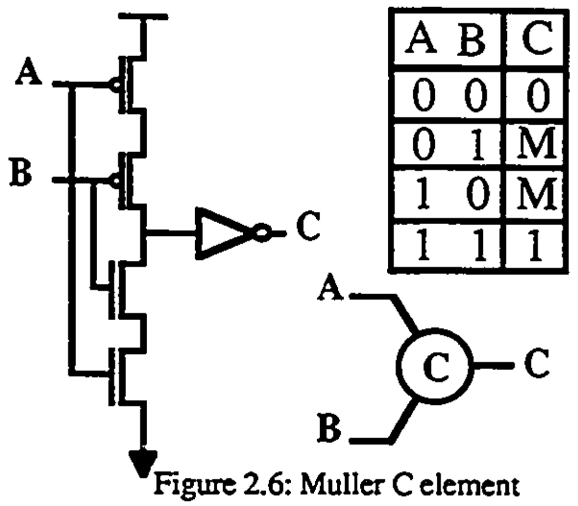


Figure 2.5: Block diagram for the self-timed pipeline

In a self-timed system, the timing criterion is probabilistic delay, as opposed to worst case delay in the synchronous system. This property shifts the traditional practice of minimizing the worst case delay of the system to the probabilistic delay. Environmental conditions also impact on the use of self-timed systems. Traditional design strategy focuses on conservative specifications to ensure the the system works in the expected environment. If the system is placed in some extreme conditions, the system may fail. In the self-timed system, the sub-systems change their operating speed for the existing condition. For example, temperature may cause a synchronous system to fail easily, while a self-timed system is more robust with respect to the change of temperature.

2.2 BASIC ELEMENT

The Muller C element[8] is a basic circuit used in self-timed systems. A static version of the static element is described in reference[9]. The schematic diagram is shown in Figure 2.6. The truth table shows that



A	B	C
0	0	0
0	1	M
1	0	M
1	1	1

Figure 2.6: Muller C element

when both inputs are equal then the output equals the input. When the inputs differ, the output is the previous stored value. The output changes only when both inputs change to the same state. An multi-input Muller C element can be constructed from the basic 2 input Muller C element as shown in figure 2.7

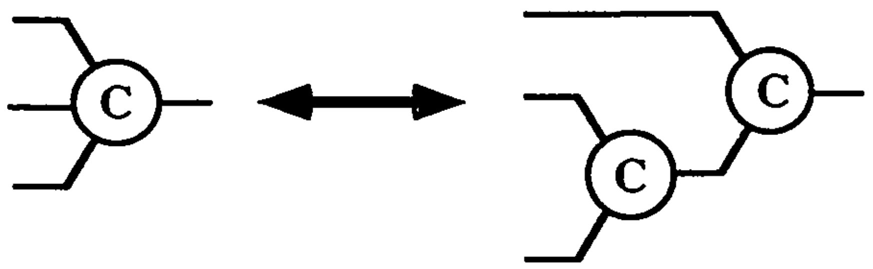


Figure 2.7: Construction of a multiple-input Muller C element

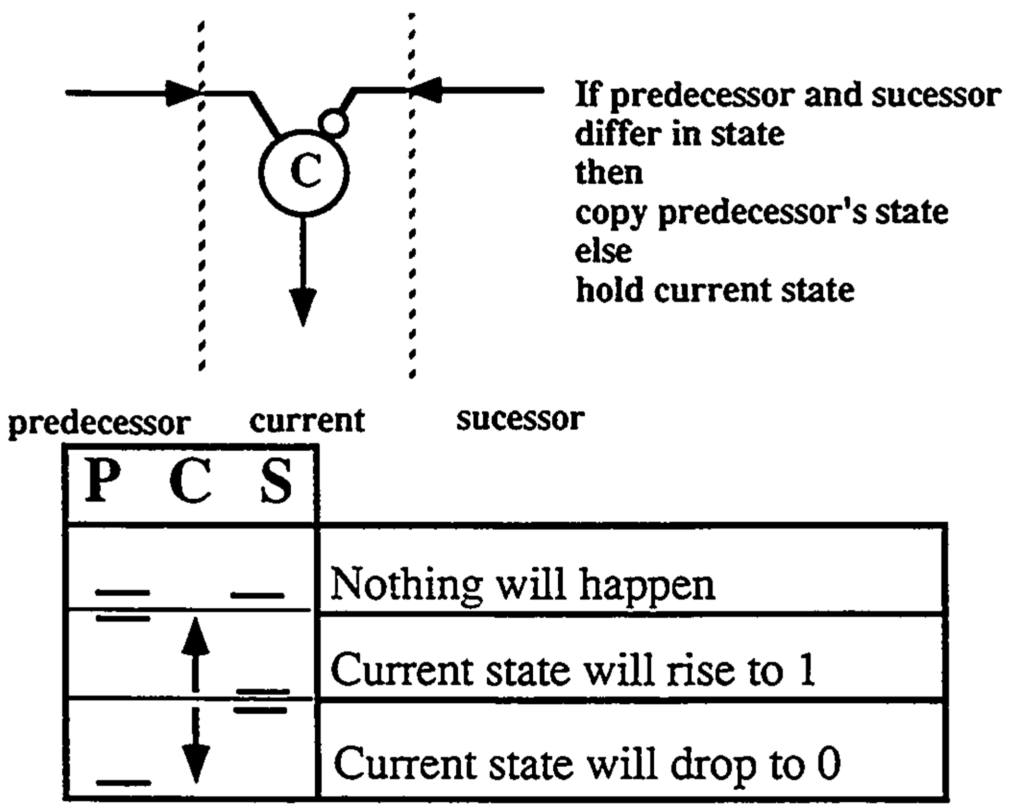


Figure 2.8: Use of the C-element in a self-timed circuit

An experimental circuit is used to demonstrate the use of the C element[9]. Figure 2.8 shows a C element with one terminal connected to an inverter. The operation is that if the predecessor's and successor's states are different then the predecessor's state is copied to the current state. Figure 2.8 clearly shows this relationship.

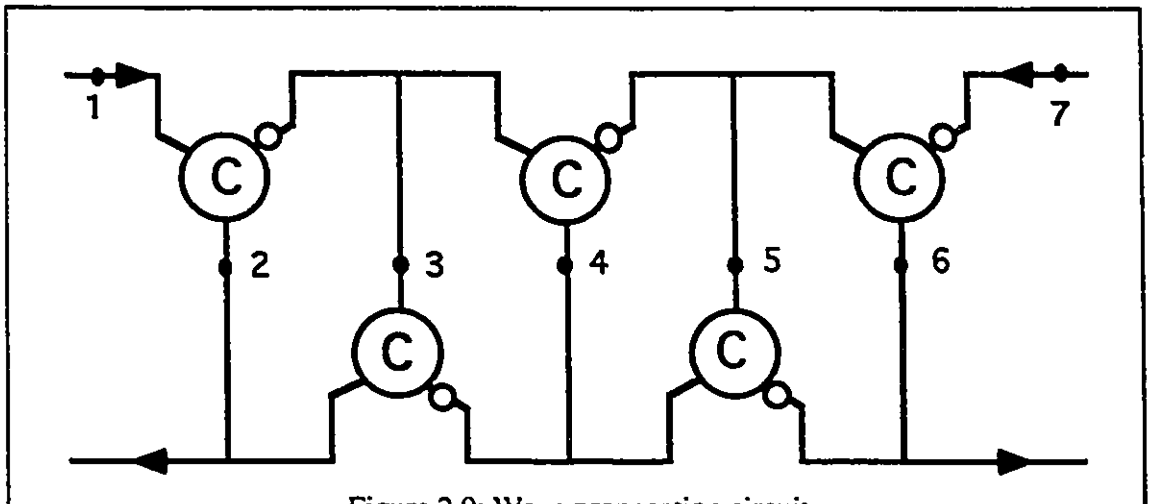


Figure 2.9: Wave propagation circuit

Five blocks of C elements, with inverters, are cascaded together to form a circuit (figure 2.9) with the functionality shown in figure 2.10. Figure 2.10 represents a snapshot of the signals at the tapped points with the top one being the earliest. When a "1" logic state is fed to a circuit, it propagates, wave-like, to the end. This characteristic will bc explored later when we create a self-timed pipeline.

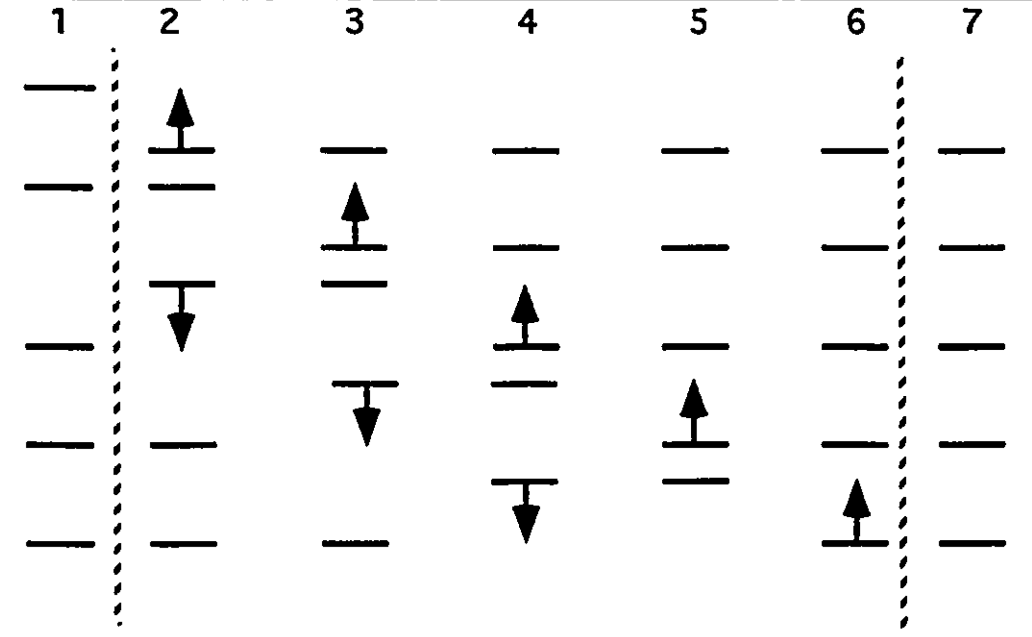


Figure 2.10: Snapshot of the signals in the wave propagation circuit

2.3 CONTROL SIGNALS

Self-timed pipelines can be classified by the method of block to block communication. There are three types of common communication methods: transition signaling[9], 4 phase dual rail signaling[2] and 2 phase dual rail signaling[10]. Similar to clocked systems, which use clock signal to indicate the validity of the output signals, self-timed circuits use special signaling methods to indicate the validity of the output signals.

2.3.1 TRANSITION SIGNALING

Transition signaling uses transitions rather than levels to represent an event. Both rising transitions or falling transitions are called events. The absolute voltage level of the signal is irrelevant to the meaning of the signal. For instance, let a pair of wires be used to transmit one bit of data. To transmit a logic zero, the first bit is complemented. If later a logic one is sent, the second bit is complemented.

For signal manipulation of the transition signal, basic components, similar to standard digital logic gates, are available (figure 2.11).

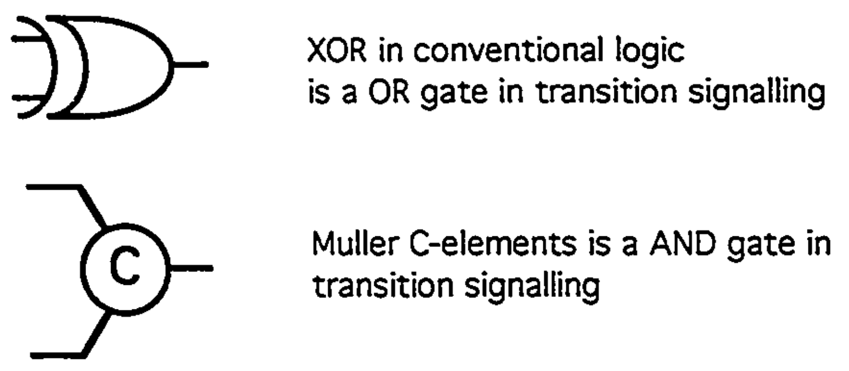

XOR in conventional logic
is a OR gate in transition signalling
C
Muller C-elements is a AND gate in
transition signalling

Figure 2.11: Signal manipulating elements for transition signal

2.3.2 4 PHASE DUAL RAIL SIGNALING

4 phase dual rail signaling represents each bit of information on one pair of wires. If the two wires are at logic '0', this denotes an invalid data token. When the first wire goes high, this represents logic zero. The two wires have to go back to level zero (invalid

data token) before transferring another data token. To represent logic one, the second wire should be set to high. The table below summarizes the representation.

X^T	X^F	
0	0	Invalid
0	1	Logic 0
1	0	Logic 1
1	1	not used

Table 2.1: 4 phase dual rail encoding scheme

A full adder (figure 2.12) will have 3 pairs of wires for inputs and 2 pairs of wires for outputs. Th: full adder starts the calculation when all input signals are valid signals. The completion of the calculation can also be detected from the output terminals.

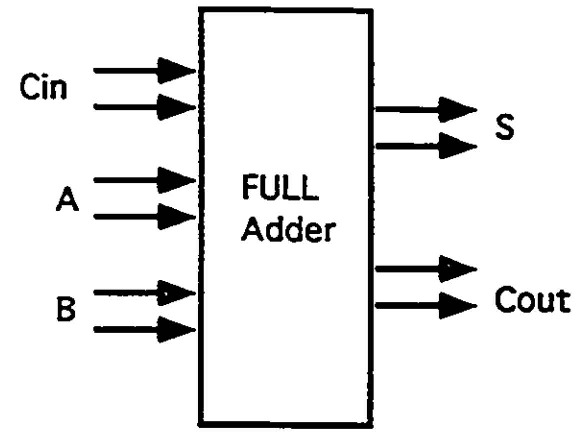


Fig 2.12: Full adder using 4 phase dual rail signaling

2.3.3 2 PHASE DUAL RAIL SIGNALING

The disadvantage of using 4 phase dual rail signaling is the necessity of returning the signal to invalid states(0 0) between valid logic signals. The invalid state functions as a delimiter for valid dates in a sequence of signals. Without it, the receiving end cannot properly recognize two bits of data which have the same logic state. Whenever new logic information has to transmit to another block, an invalid state has to be transmitted first to reset the receiving block. This resetting takes up some of the time available for computation. An alternate method of signaling, called 2 phase dual rail signaling[10], also uses two wires to represent logic. Unlike 4 phase dual rail signaling, however, it does not need an invalid state between logical information transfer. Data is encoded into two phases: even or odd. Each data in a stream must have the opposite phase data preceding it. Table 2.2 illustrates the relationship.

Z^1	Z^0	Phase	State
0	1	ODD	0
1	0	ODD	1
1	1	EVEN	1
0	0	EVEN	0

Table 2.2: 2 phase dual rail encoding scheme

If Odd phase data is fed to a function block, Odd phase output data will be created. Even phase data will generate Even phase output. The design should be carefully crafted to ensure this rule is being followed. Since transition signaling and 2 phase Dual rail signaling take comparatively more hardware to build than 4 phase-dual rail signaling, the 4 phase dual rail is chosen as the signaling method used in this thesis.

2.3.4 COMPLETION DETECTOR

To generate 4 phase dual rail signals, dynamic cascode voltage switch logic (figure 2.13) is a good candidate for function block implementation.

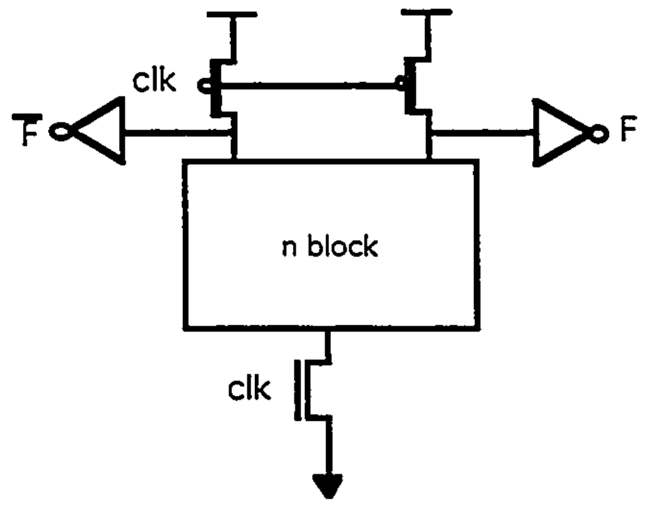


Figure 2.13: Dynamic cascode voltage switch logic

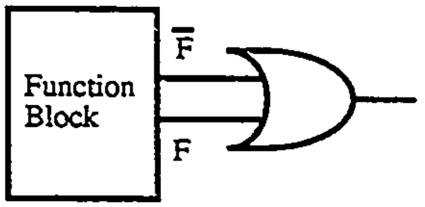


Figure 2.14: Completion signal

The $\overline{F}$ and F represent the non-complement and complement output of the logic gate. When the function block is in the pre-charging state, both outputs drop to zero. This characteristic matches well with the 4 phase dual rail signaling method. Since it is a dynamic logic family, it shares the same behavior with other members in this family. The behavior inherited from the dynamic logic family has fast evaluation time and area efficient VLSI implementations. The output state of this logic family is not determined from the voltage level of the output terminal. It is determined from the choice of which terminal experienced a low to high transisition. This makes this logic family logically complete. Unlike domino logic which can not implement inverting logic, the DCVSL can implement any function in one block. Also, the completion of evaluation can be detected from the output terminals alone. When both output terminals stay at logic level zero, the evaluation is not complete. For single output functions, only an OR gate is needed to generate the completion signal, as shown in figure 2.14.
For multiple output functions, the C element, introduced earlier, is used to form the completion signal (figure 2.15).

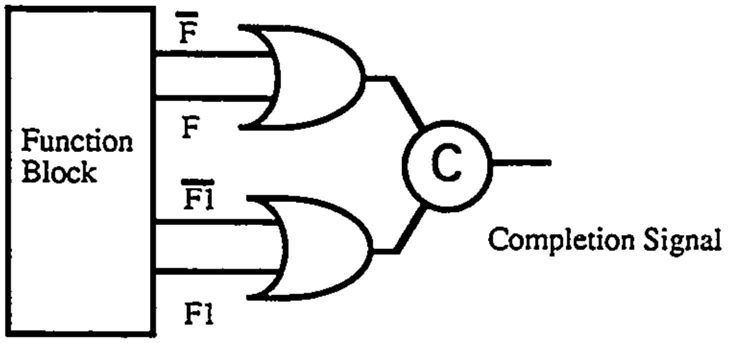


Figure 2.15: Completion signal generation for multiple output function

2.4 SELF-TIMED PIPELINE CONFIGURATIONS

2.4.1 PC0 CONFIGURATION

After examining the function block, C element and completion detector, a few basic pipeline configurations will be explored in this part of the chapter.

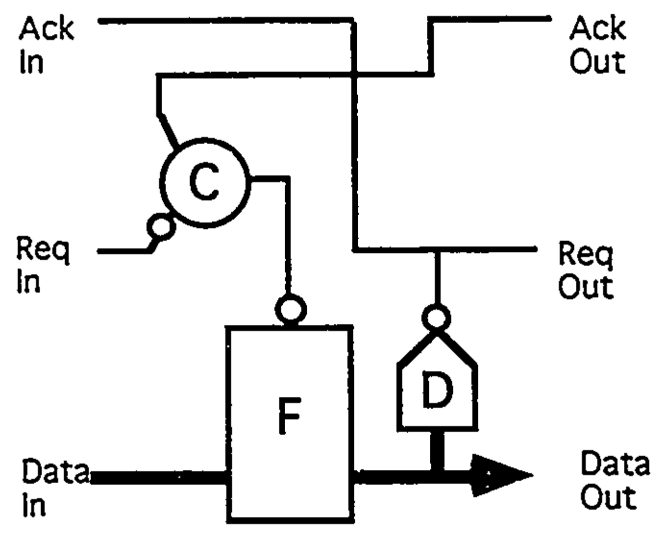


Figure 2.16: One stage of the PC0 pipeline

One stage of the PC0 pipeline[7] is shown in figure 2.16. F is the dynamic cascode switch logic block. D is the completion detector. AckIn and ReqOut become zero when F completes the evaluation and they become one when F is in the reset state (all output terminals stay at zero). If AckOut is one and RegIn is zero then F will start to evaluate. When Ackout is zero and Reqin is one then F will start to pre-charge. Table 2.1 summarizes the relationship

Event	Conseqence
Function block completes the evaluation	Ackin=0, Reqout=0
Function block completes the reseting	Ackin=1, Reqout=1
Ackout=1, Reqin=0	Activate the function block evaluation phase
Ackout=0, Reqin=1	Activate the function block pre-charging phase

Table 2.3 The signal dependency of the PC0 stage

By cascading 3 stages of pipeline blocks, a pipeline is constructed as shown in figure 2.17. When F1 is in the evaluation phase and F3 is in the pre-charge phase, F2 will start to evaluate. This mechanism is responsible for the wave front propagation. When F1 is in the pre-charging phase and F3 is in the evaluation phase, F2 will start to pre-charge. This second mechanism is responsible for the wave crest propagation. These two mechanisms combined together can push a wave to propagate from the pipeline head to the tail. Between each consecutive stage, there is no explicit latch. This is because dynamic logic is used as the function block. If the input to the function block is invalid, the function block will still retain the previous output value.

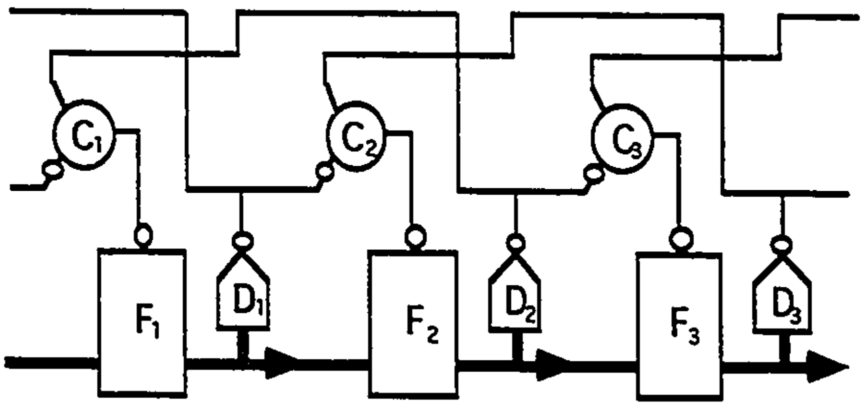


Figure 2.17: PCO pipeline with three stages

2.4.2 PC1 CONFIGURATION

The first modification is to add dummy stages to hold the data. Fig 2.18 shows one combined stage. The C latch is shown in figure 2.19

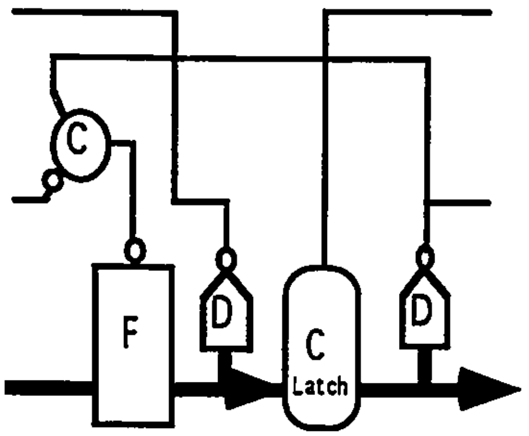


Figure 2.18: One stage of PC1 pipeline

The new configuration is called PC1[7]. The "1" used in "the word PC1" represents one extra latch. Since a PCO pipeline with n stages, at least n/2 stages are needed to hold the reseted stages. It is because consecutive unreseted stages need a reset spacer in between to seperate them out. At most n/2 stages are used to process data. However the data is dynamically moving forward in the pipeline. Some extra stages will be wasted for the dynamic effect. By adding some dummy stages(C latches) to hold the data, a pipeline can hold more data than before. If the complexity of

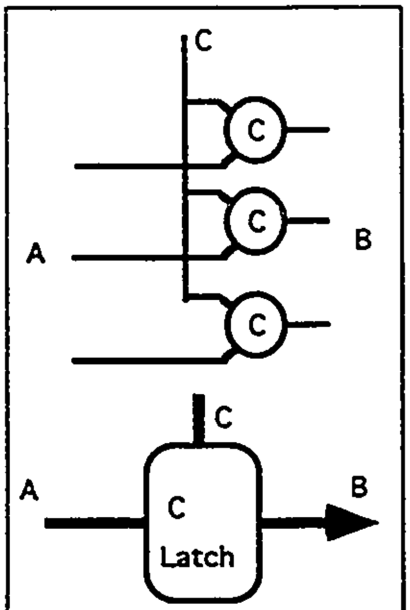


Figure 2.19: C latch

the function block is much higher than the latch, then the performance improvement over the original PC0 configuration is greater.

2.4.3 PS0 CONFIGURATION

The delay of a C element is in the critical path of a forward moving data token. If C elements are eliminated from the design, then the forward moving data can propagate faster. By carefully analyzing the PC0 configuration, the C element is found to be redundant and it can be replaced with a single wire, shown in figure 2.20; this modification is termed the PS0 configuration. The rising transition of the C element is not important when dual-rail signaling is used. The input data prevents each stage from evaluating until its input become valid. Each stage's predecessor resets faster than the stage's successor evaluates, and the falling transition of the C element is always triggered by the falling of the signal from the successor completion detector. Therefore, it can be replaced with a single wire connecting the successor's completion detector to the resetting input of the function block.

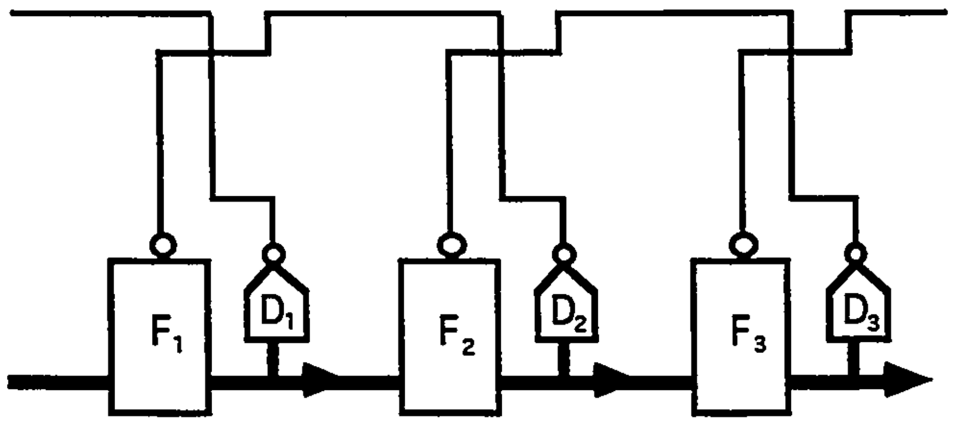


figure 2.20: PS0 pipeline with three stages

2.4.4 COMPLEX DATAPATHS

Non-linear pipelines with arbitrary topology can also be constructed. The main basic building blocks for these are datapath merging and datapath splitting configurations[7] as shown in figure 2.21a and figure 2.21b.

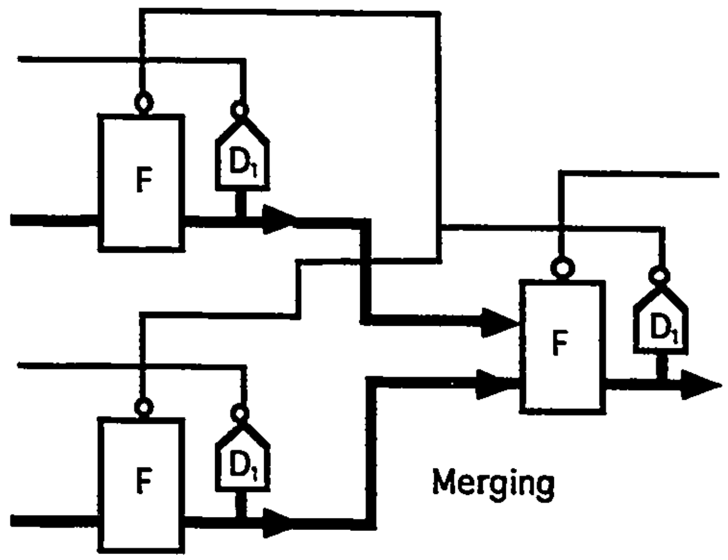


Figure 2.21a: Datapath merging structure

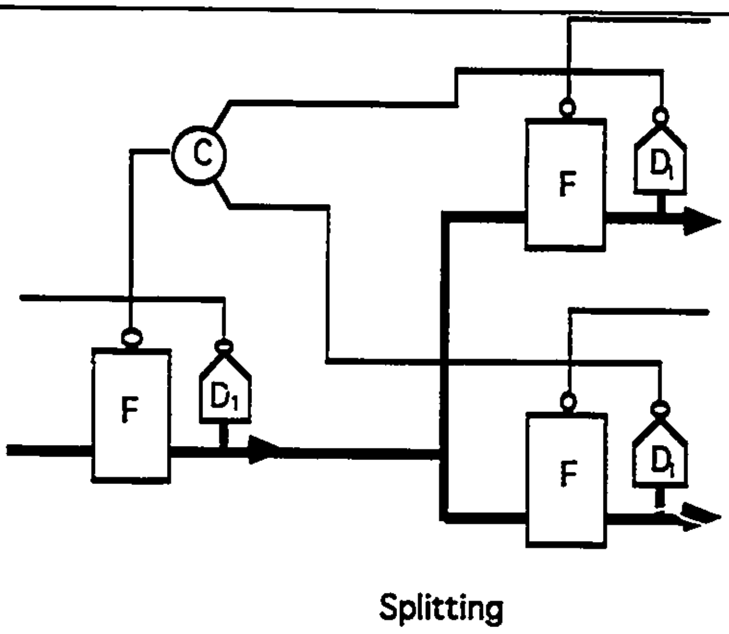


figure 2.21b: Datapath splitting structure

2.5 ANALYSIS METHODS

2.5.1 PERFORMANCE PARAMETERS

The performance of a synchronous pipeline can be easily determined from the clock frequency and the number of stages contained in the pipeline. However, for a self-timed pipeline, it is not an easy task. The delay of each component and configuration determine the performance. Before introducing different performance parameters for a self-timed pipeline, some basic concepts have to be understood. A token consists of a data and a reset spacer; otherwise, it is called a bubble. A token flows forward, while a bubble flows backward. Tokens need bubbles to travel forward. The supply of bubbles can limit the performance; the supply of tokens and local control mechanism also determines the performance. A list of performance parameters is shown in table 2.4.

Performance parameters

Lf	forward latency -the delay from new valid data outputs at one stage to new valid data output at the following stage. -It can be measured by observing a data token flowing forward through an empty pipeline.
Lr	reverse latency -the delay from the acknowledgement of a stage's output to the acknowledgment of its predecessor's output. -It can be measured by observing the bubble flowing backward through an empty pipeline.
P	minimum local cycle time. -analogous to minimum clock period for a traditional synchronous pipeline.
T	throughput -It is the rate at which the input and output must deliver and consume tokens respectively to keep a pipeline flowing at its maximum capacity.
S	static spread $S=2/(H+1)$ where H is the number of extra latches in each stage. S=number of stages needed to hold one data token (one data and one spacer).

Table 2.4: Performance parameters for self-timed pipeline

2.5.2 ANALYSIS BY DATA DEPENDENCY GRAPH

The performance can be analyzed by using a data dependency graph. Each node in the graph represents a component delay. The up-arrow and down-arrow refer to the rising transition and the falling transition respectively. From the PC0 configuration (Figure 2.17), the data dependency can be traced to create the data dependency graph (figure 2.22).

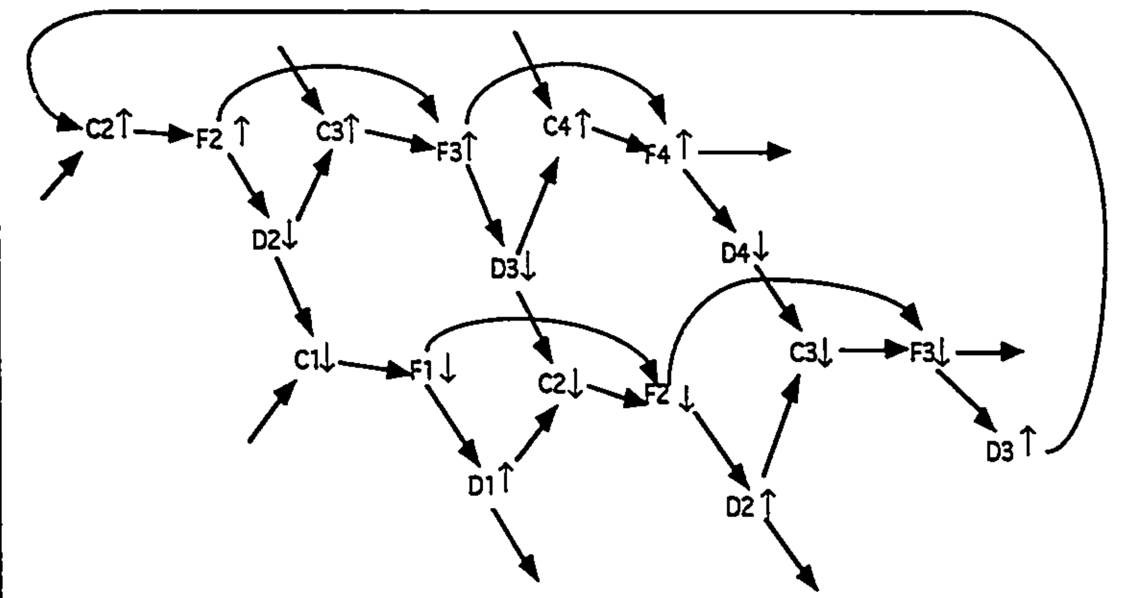


Figure 2.22: Data dependency graph for PC0 configuration

After the data dependency graph[7][]11] is drawn, some performance parameters can be extracted. To find the forward latency, the whole graph is traced to find the longest path from a transition to the same kind of transition with the next highest index. For example $F_2\uparrow \rightarrow D_2\downarrow \rightarrow C_3\uparrow \rightarrow F_3\uparrow$ is the longest path meeting the restriction. Delays for $F_2\uparrow$, $D_2\downarrow$ and $C_3\uparrow$ are added up to get the forward latency. Reverse latency is one half the value of the delay through the longest path from a transition to the same kind of transition with an index 2 lower than the current index. The local cycle time is the delay of the longest cyclic path.

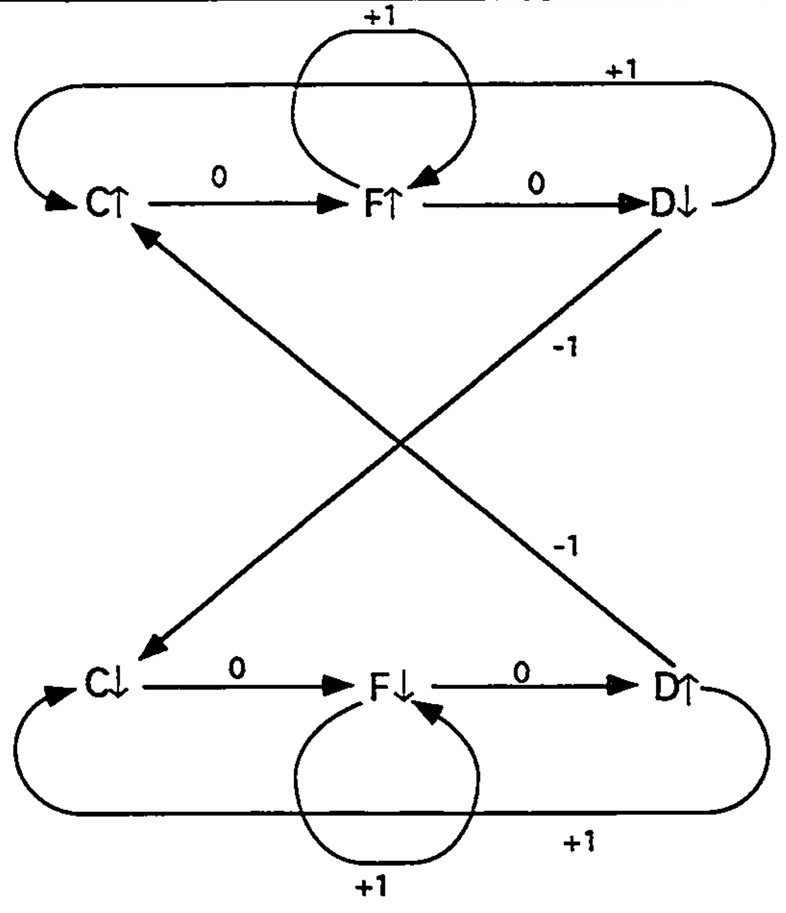


Figure 2.23: Folded dependency graph for PC0

Observing the data dependency graph in figure 2.22, the graph can be easily recognized to have a repeated structure. It is due to the homogeneous structure of the pipeline. Each stage is identical to every other stage. By making use of this property, the data dependency graph can be re-organized into a more compact form called a folded data dependency graph[7]. One example for this folded graph for the PC0 structure is shown in figure 2.23. Each node in the folded dependency graph represents the delay for component transition. The number attached to the directed edge is the stage offset. A number zero attached to the directed edge means that the component delays at both ends head are in the same stage. Between D↓ and C↑, an offset of 1 is placed. The meaning is that D↓ is first activated follwed by the activation of C↑ of the next stage. Equations 2.1 to 2.3 describe the forward latency, reverse latency and minimum local cycle time.

$$L_f = \underset{\text{with } 0 < \sum_{i \in path} w_i}{\text{maximum over all non-repeating cyclic paths}} \left[\frac{\sum_{i \in path} t_i}{\sum_{i \in path} w_i} \right] \tag{2.1}$$

$$L_r = \underset{\text{with } 0 > \sum_{i \in path} w_i}{\text{maximum over all non-repeating cyclic paths}} \left[\frac{\sum_{i \in path} t_i}{- \sum_{i \in path} w_i} \right] \tag{2.2}$$

$$P = \underset{\text{with } 0 = \sum_{i \in path} w_i}{\text{maximum over all non-repeating cyclic paths}} \left[\sum_{i \in path} t_i \right] \tag{2.3}$$

The corresponding methods used to find performance parameters by dependency graph and folded dependency graph are conceptually equivalent. The methods are structurally different, but the underlying concepts remain the same. The folded dependency graph can only be used for homogenous pipelines. If the pipeline contains different function blocks for its stages, the normal dependency graph should be used. By applying equation 2.1 to 2.3 to the PC0 folded data dependency graph, the following equations are generated.

$$L_f = \max\left[t_F\uparrow + t_D\downarrow + t_C\uparrow\right] \tag{2.4}$$

$$L_r = \frac{1}{2}\left[t_F\uparrow + t_D\downarrow + t_C\downarrow + t_F\downarrow + t_D\uparrow + t_C\uparrow\right] \tag{2.5}$$

$$\begin{aligned} P = {} & t_F\uparrow + t_D\downarrow + t_C\downarrow + t_F\downarrow + t_D\uparrow + t_C\uparrow \\ & + 2 \bullet \max\left[t_F\uparrow + t_D\downarrow + t_C\uparrow, t_F\downarrow + t_D\uparrow + t_C\downarrow\right] \end{aligned} \tag{2.6}$$

By making some assumptions, the equations can be greatly simplified. The assumption includes: the rise and fall time of completion detectors are equal; the rise and fall time of Muller C elements are equal; the reseting delay is larger than the evaluation delay for function blocks; the delay of C element is less than the reseting delay of function blocks. The equations (2.7 -2.10) describe the assumptions.

$$t_D = t_D\uparrow = t_D\downarrow \tag{2.7}$$

$$t_C = t_C\uparrow = t_C\downarrow \tag{2.8}$$

$$t_F\downarrow \leq t_F\uparrow \tag{2.9}$$

$$t_C \leq t_F\downarrow \tag{2.10}$$

Using these assumptions, equations 2.1 to 2.3 can be reduced to the following simpler equations.

$$L_f = t_F\uparrow + t_D + t_C \tag{2.11}$$

$$L_r = t_D + t_C + \frac{1}{2}\left[t_F\uparrow + t_F\downarrow\right] \tag{2.12}$$

$$P = 3t_F\uparrow + t_F\downarrow + 4t_D + 4t_C \tag{2.13}$$

Applying equation 2.1 to 2.3 to various PC and PS self-timed pipelines, a set of equations describing the performance of pipelines is derived. The results are shown in table 2.4. The numbers are co-efficients of the equation. For example PC0 cycle time has 3 for $t_F\uparrow$, 1 for $t_F\downarrow$, 4 for t_C and 4 for t_D. This is actually the equation $3t_F\uparrow + t_F\downarrow + 4t_C + 4t_D$.

Config	Cycle time				Forward latency			Reverse latency			
	$t_F\uparrow$	$t_F\downarrow$	t_C	t_D	$t_F\uparrow$	$t_C\uparrow$	$t_D\downarrow$	$t_F\uparrow$	$t_F\downarrow$	t_C	t_D
PC0	3	1	4	4	1	1	1	0.5	0.5	1	1
PC1	2	0	4	4	1	2	1	0.5	0.5	2	2
PC2	1	1	4	4	1	3	1	0.5	0.5	3	3
PC3	1	1	4	4	1	4	1	0.5	0.5	4	4
PS0	3	1	0	2	1	0	0	0.5	0.5	0	1
PS1	2	0	2	2	1	1	0	0.5	0.5	1	2
PS2	1	1	2	2	1	2	0	0.5	0.5	2	3
PS3	1	1	2	2	1	3	0	0.5	0.5	3	4

Table 2.5: Performance parameters for various pipelines

From table 2.5, it can be seen that the PS family always has better performance than the PC family. By adding latches to the stages, the cycle time is reduced. However the forward latency and reverse latency are increased. After adding two latches, both the PC and PS family saturate. This means that adding more latches brings no more improvement to the cycle time. If the function block reset time is not much faster than its evaluation time, then adding two extra latches is not better than adding one. Another observation is in the PS0 family; the forward latency only equals to the function block evaluation time. The effect looks like a pure combinational function block delay.

Config	Big F Block		Small F Block	
	$t_F\downarrow = 0$		$t_F\downarrow = t_F\uparrow$	
	$t_C = t_D = 0$		$t_C = t_D = t_F\uparrow$	
	L	P	L	P
PC0	1	3	3	12
PC1	1	2	4	10
PC2	1	1	5	10
PC3	1	1	6	10
PS0	1	3	1	6
PS1	1	2	2	6
PS2	1	1	3	6
PS3	1	1	4	6

Table 2.6: Effect of changing function block size on the performance parameters

To get an overview of the forward latency and local cycle time range, a table with values normalized to the function block evaluation time is constructed. Table 2.6 shows different performance values for large and small function blocks. For large function blocks, it is assumed that (tC=tD=0) with reset time equal to zero. This is valid because the evaluation time is comparatively larger than other delay times. For small blocks, all delay times are assumed to be equal (tC=tD=tF). This is also a valid assumption as all component delays are comparatively close to each other.

2.6 CONSTRAINTS OF DIFFERENT OPERATION REGIONS

A pipeline is a subset of a more general class of structure called a ring structure. A ring is a looped pipeline with N equal to the number of stages it contains and G the number of function block evaluations required. For a general iterative problem, N is a fraction of G. It takes $\frac{G}{N}$ number of loops to obtain the final result. A simple pipeline is a sub-class of a general pipeline with G equal to N. To use the ring, or the pipeline, more efficiently, multiple tokens are allowed to circulate inside the ring. Let K be the number of tokens that can be packed into the ring where S is the static spread equal to $\frac{2}{H+1}$. H is the number of extra latches in each stage. For example, PC0 has no extra latch so H=0. This $\frac{N}{S}$ value is called token capacity. The number of bubbles that can be found in a ring is $2\frac{N}{S-K}$. The factor of 2 is due to the fact that each missing data token leaves 2 bubble spaces.

Pipeline performance is evaluated by latency and local cycle time. For the ring structure, similar parameters are present which are called total latency (λ) and total cycle time (ϕ). The total latency is the time a token takes to finish computation and leave the ring structure. The total cycle time is the time between consecutive tokens fed to the ring. There exists a relationship between these two parameters. The equation

$$\lambda = K\phi \tag{2.14}$$

governs this relationship, and is intuitively simple. Since there are K tokens circulating in the ring and ϕ is the consecutive token feeding time, K times ϕ is the time for a token to stay in the ring to keep that K tokens circulating. Since T= $\frac{1}{\phi}$, equation 2.15 holds.

$$\lambda = \frac{K}{T} \tag{2.15}$$

2.6.1 PERFORMANCE GRAPH

The performance of a ring can be determined from the performance graph[7] shown in figure 2.24. The axes are K and G, which are the number of tokens and stages in the ring. By controlling the number of data tokens fed to the ring and the number of stages contained in the ring, the operating point can be moved to different regions of the performance graph.

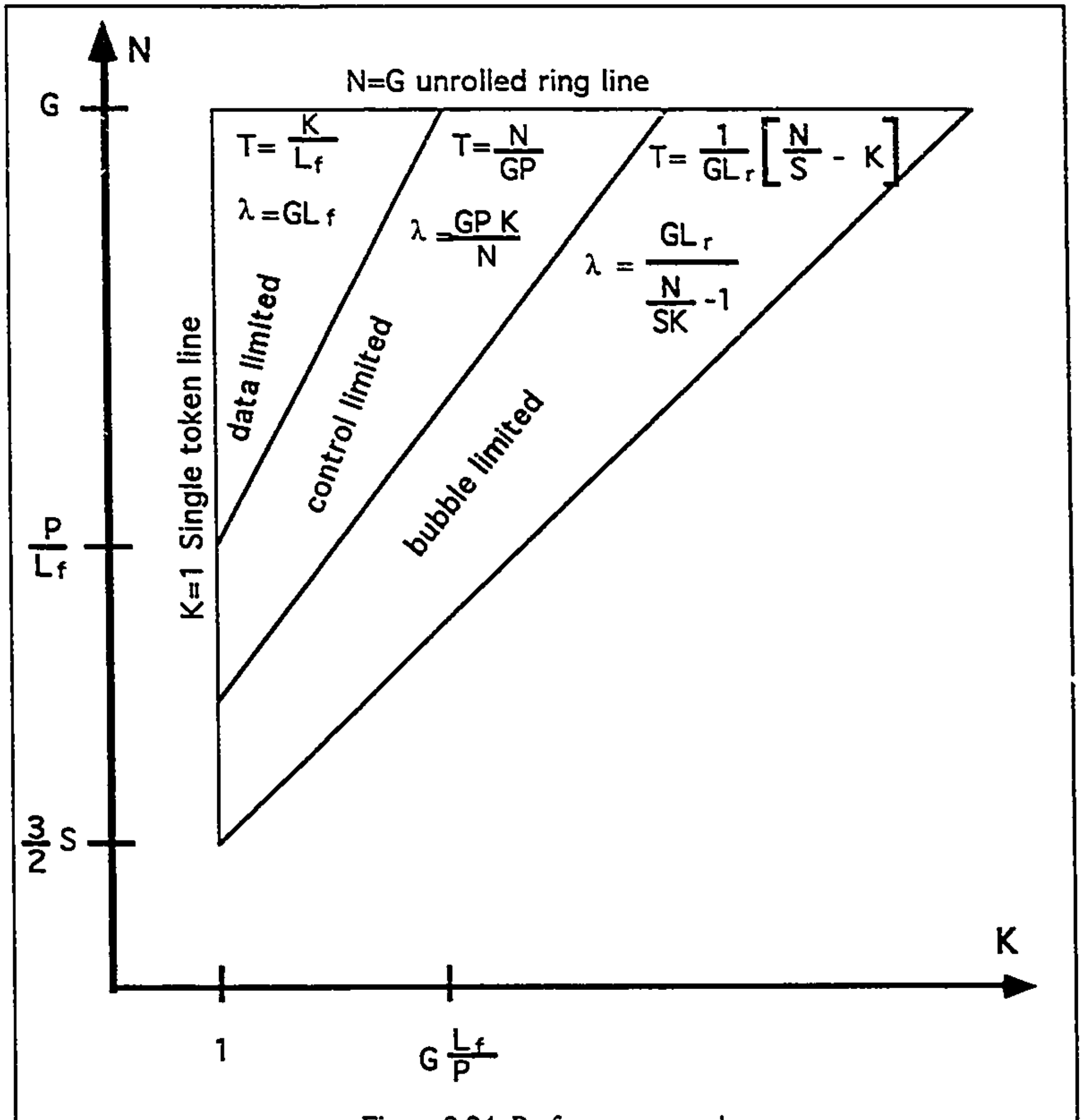


Figure 2.24: Performance graph

The data limited region is the region where the performance is limited by too few data tokens introduced to the graph. The ring is not fully utilized. Therefore the total latency is

$$\lambda = GL_f \tag{2.16}$$

Substituting this equation to equation 2.15, this equation $T = \frac{K}{G L_f}$ is derived. For bubble limited region, the performance is limited by an insufficient number of bubbles in the ring. The data tokens are jammed in the ring. They can only move forward if they have bubbles to consume, however in these regions, there are too many data tokens, therefore very few spaces are left for bubbles. The throughput equals the number of bubbles multiplied by the data token moving rates.

$$T = 2\left(\frac{N}{S} - K\right) \cdot \frac{1}{2G L_r}$$

$$T = \frac{1}{G L_r}\left(\frac{N}{S} - K\right) \tag{2.17}$$

$\frac{1}{G L_r}$ is the bubble moving rate. The rate of the data token is 2 times less because a bubble moves two spaces backward to make a data token move one space forward. Substituting equation 2.17 into equation 2.15, we obtain:

$$\lambda = \frac{G L_r}{\frac{N}{SK} - 1} \tag{2.18}$$

In the control limited region, the limiting factor is the local cycle time. Therefore the throughput equation is

$$T = \frac{N}{GP} \tag{2.19}$$

and by substituting it into equation 2.15. equation 2.20 is generated

$$\lambda = \frac{GPK}{N} \tag{2.20}$$

Equating 2.16 and 2.20, the intersecting line between data limited region and control limited region is found to be

$$N = K\frac{P}{L_f} \tag{2.21}$$

This line is called the "zero-overhead" line. Operations that fall in this line give the lowest latency value and the best throughput rate. There are another two interesting operation lines; the single token line and unrolled ring line. The single token line (K=1) denotes only one token circulating in the ring. This line intersects with the zero-overhead line at $N = \frac{P}{L_f}$. This value gives the minimum number of stages for constructing a zero-overhead single token self-timed ring. The unrolled ring line (N=G) is actually the operation line for pipeline. It intersects with the zero-overhead line at $K = G\frac{L_f}{P}$. This value gives the number of data tokens that should be fed to a pipeline to get maximum throughput and lowest latency value.

By making some assumptions, an estimation of the number of stages that a ring should have in order to attain zero-overhead performance is derived. For a ring with large function blocks, it is assumed that the blocks reset in zero time compared to their evaluation times. Completion detectors and C elements are also assumed to require zero delay. This is a valid assumption, since the transistor chain in the n block discharges much more slowly than the single p-transistor pre-charges. Another reason is that cascaded function blocks evaluate in series but reset in parallel. For a ring with small function blocks, function reset times, function evaluation times, completion detector delays and C element delays are assumed to be equal. According to these assumption, a table is generated to show the P/X ratio for different configurations(table 2.7). It can be seen that PC0 takes 3 stages for a ring with large function blocks and 4 stages for a ring with small function blocks to circulate a single token without overhead. For a multiple token ring, the number of stages equals the number of the stages for a single token multiplied by the number of tokens. The range in PS0 is 3 for large function blocks and 6 for small function blocks. All other sizes of function blocks will require 3 to 6 stages for each token.

pipeline configuration	Big F block	Small F block
	$t_F\downarrow = 0$	$t_F\downarrow = t_F\uparrow$
	$t_C = t_D = 0$	$t_C = t_D = t_F\uparrow$
PC0	3	4
PC1	2	2.5
PC2	1	2
PC3	1	1.67
PS0	3	6
PS1	2	3
PS2	1	2
PS3	1	1.5

Table 2.7: Effect of changing function block size on the total stage number

2.7 PERFORMANCE IMPROVEMENT TECHNIQUES

While satisfying the zero-overhead constraint, there are some techniques which can further improve the performance. A few of them will be presented in the following sections.

2.7.1 DECREASING GRAIN SIZE

Decreasing the grain size of the ring will increase the limit of throughput. Figure 2.25 shows two pipelines with a large grain size pipeline at the top and small grain size pipeline at the bottom. Both pipelines perform the same function. The difference is only in the grouping of function blocks in each stage. The large grain size pipeline groups three function blocks in each stage, while the small grain size pipeline has only one function block per stage. The effect of changing grain size is shown in figure 2.26. The large grain size pipeline has a smaller maximum throughput rate than the small grain size pipeline. The resultant increment in performance is obvious since the small grain size has more stages and more spaces for storing data tokens. This is analogous to adding extra latches to the stages to increase the performance. These small grain pipelines can accommodate more data tokens at any time instance without breaking the zero-overhead constraint. However, reducing the grain size, in other words breaking the pipeline into more stages, will increase the hardware overhead. Also, the number of stages can be reduced if the grain size is chosen to be small. It is because more stages are available and these extra stages can be eliminated by allowing data tokens to circulate in the ring more cycles than before.

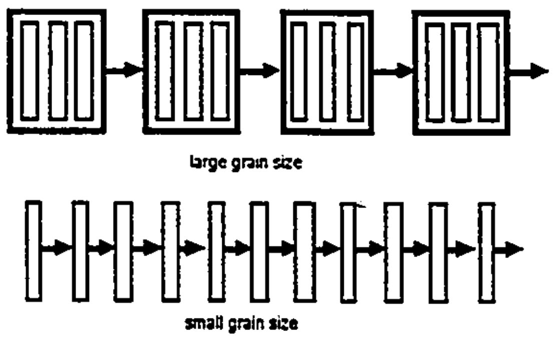


Figure 2.25: Pipelines with Large and small grain size

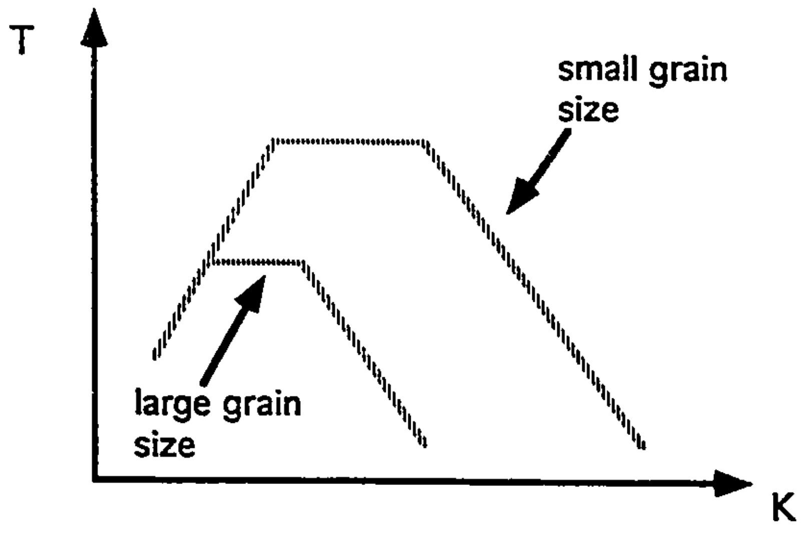


Figure 2.26: Effect of grain size on throughput rate

The tradeoff is between grain size, number of stages and area. Area can be minimized while satisfying the zero-overhead constraint. An example is presented for the PS0 configuration. First of all, the zero-overhead constraint is stated

$$N \geq K\frac{P}{L_f} \tag{2.22}$$

Next, the P and Lf from Table 2.4. are substituted. It becomes

$$N \geq K\left[3+\frac{t_F\downarrow+2t_D}{t_F\uparrow}\right] \tag{2.23}$$

And then it is assumed that function blocks reset in a single delay, let g_F be the measure of grain size which is assumed to be $\frac{t_F\uparrow}{t_F\downarrow}$ After substitution of g_F into the equation. It becomes

$$N \geq K\left[3+\frac{1}{g_F}\left[1+\frac{2t_D}{t_F\downarrow}\right]\right] \tag{2.24}$$

And area is proportional to (g_F+aD)N, where aD is the relative size of the completion detector. This area equation is minimized with respect to g_F. The following two equations describe the conditions of zero-overhead with minimized area for PS0 configuration.

$$\frac{N}{K}=3+\sqrt{\frac{3\left[1+\frac{2t_D}{t_F\downarrow}\right]}{a_D}} \tag{2.25}$$

$$g_F = \sqrt{\frac{a_D}{3}\left[1 + \frac{2t_D}{t_F\downarrow}\right]} \qquad (2.26)$$

2.7.2 SHIFTING FUNCTION BLOCKS

Another way to improve the local cycle time is to reduce the delay time for the completion detector. The complexity of the completion detector can be reduced if the data path width from completion detector to tap is reduced. This can be accomplished by moving the thin data path out of each stage as in figure 2.27. After moving them to the interior, the external stage to stage data path width is changed to 5 bits. This will reduce the complexity of the completion detector and thus reduce the local cycle time.

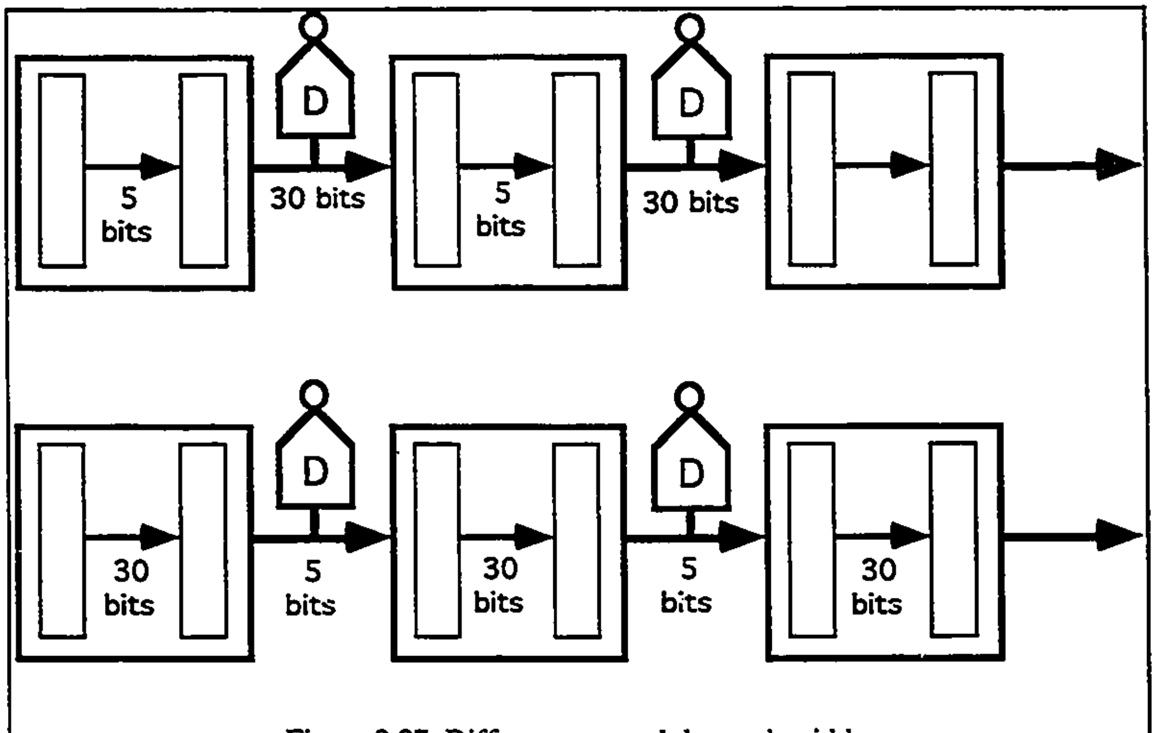


Figure 2.27: Different external datapath widths

2.7.3 SELECTIVE TAPPING OF DATAPATHS

Another similar method is selective tapping of datapaths. Although from a theoretical viewpoint, all the wires in the data paths should be tapped, some tricks can be

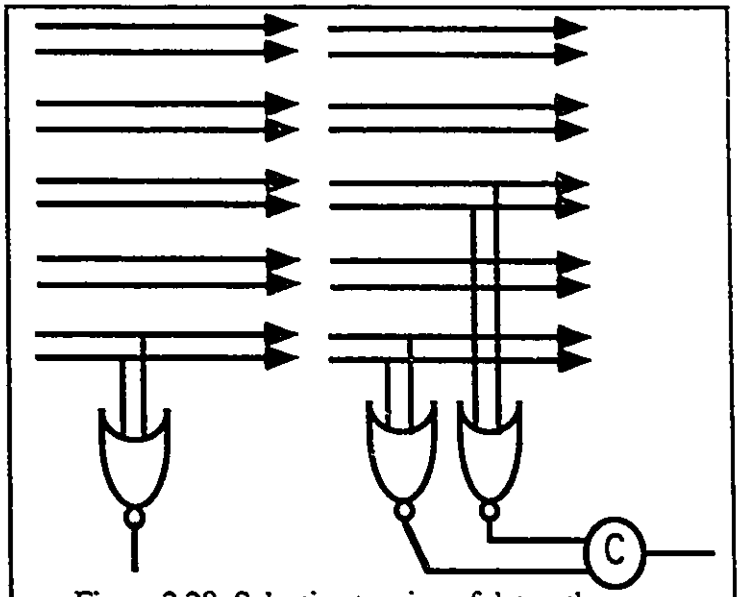


Figure 2.28: Selective tapping of datapaths

used to reduce the tapping by analyzing the circuit to determine the most critical wires for tapping. As seen in the left side of figure 2.28, one critical pair is found and therefore one group of tapping is necessary. The right one has two pairs of possible critical wires, so the tapping is applied to two groups of wires.

2.7.4 COMPLETION DETECTOR PLACEMENT

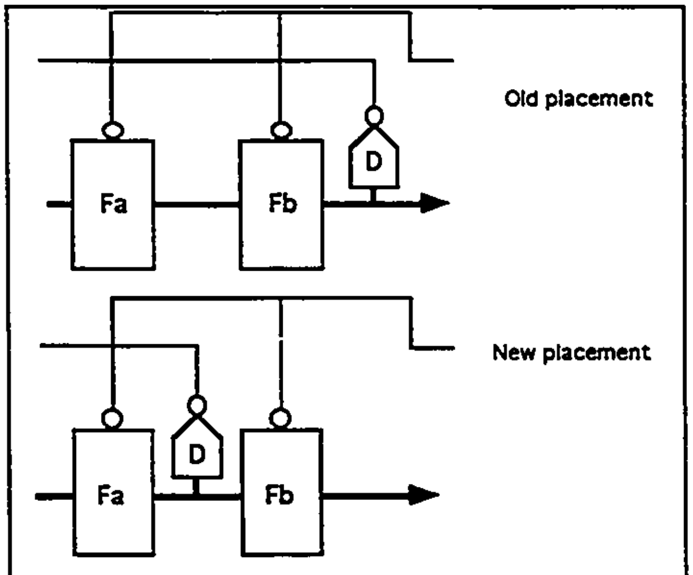


Figure 2.29: Different placements of completion detector

The last method to increase the performance is moving the placement of the completion detector. If a stage contains multiple cascade function blocks, the completion detector can be moved to the output of the first function block. That will make the completion detector delay overlap with the function block delay as in figure 2.29.

2.8 SUMMARY

Various problems in constructing synchronous pipelines have led to the study of self-timed pipelines. The study begins with identification of advantages of self-timed pipelines over traditional synchronous pipelines. The Muller C element is examined with its structure and its application in wave propagation circuit. Signaling methods are studied afterward. They include transition signaling, 4 phase dual rail signaling and 2 phase dual rail signaling. Composition and use of completion detector are also examined. After all of the basic background information has been provided, the PC0 self-timed pipeline is constructed, followed by PC1 self-timed pipeline constructed from adding extra latches to PC0 pipeline. The PS0 self-timed pipeline is derived from PC0 self-timed pipeline by eliminating the C elements. Methods of constructing complex datapaths are also introduced. Analysis methods are introduced by explaining the five performance parameters. These parameters can be obtained by analyzing the data dependency graph which is derived from the pipeline delay values and topology. The repeated structures in the data dependency graph are used to simplify the graph into a folded data dependency graph. The results of extracting the performance parameters are organized into tables. Another graph showing different regions of operation is constructed from the performance parameters derived previously. These regions are data limited region, control limited region and bubble limited region. the throughput and latency of pipelines operating in these regions are also derived. Zero-overhead line is identified as the line intersecting between data limited region and control limited region. At the end of this chapter for the self-timed pipeline, various techniques of performance improvement are discussed. They include decreasing grain size, shifting function blocks, selective tapping of datapaths and changing the completion detector placement.

CHAPTER 3

SWITCHING TREE SYNTHESIS

3.1 INTRODUCTION TO DYNAMIC LOGIC DESIGN

When designing digital logic circuits, either static or dynamic logic cells may be used as building blocks. As long as the power supply is operating, static logic will keep the logic level constant if the inputs are constant. Dynamic logic needs periodic charging to maintain the valid logic state. It has two different operating phases called pre-charge and evaluation. The pre-charge phase is used to charge an internal parasitic capacitor such that the output is biased to a valid logic state. In the evaluation phase, the parasitic capacitor may discharge depending on the input signal. Only at the end of the evaluation phase is the output considered valid. The dynamic logic family also decreases the power dissipation because there is never a conducting path between V_{DD} and V_{SS} at any time instance. NMOS logic has high power consumption as there is a conducting path in some time instance. The cost in area is smaller for dynamic logic than static logic. Static logic needs both an NMOS network and a PMOS network for implementation; however, dynamic logic needs only one of the MOS networks for a single gate implementation. Dynamic logic is generally faster than static logic for there is less input capacitance associated with dynamic logic.

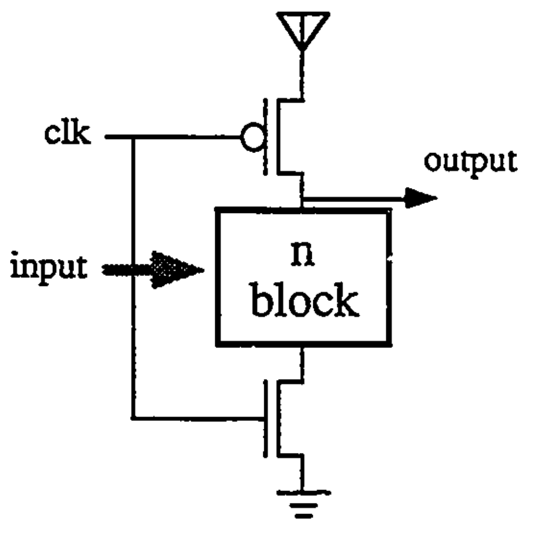


Figure 3.1: Dynamic CMOS logic

Dynamic CMOS logic, as shown in figure 3.1, consists of a PMOS and a NMOS clock transistor together with an n-logic network block. The n-logic may create a conducting path depending on the input signal. Direct cascading of two dynamic logic cells is not feasible.

Figure 3.2 shows two cascaded dynamic blocks. When the clock is low, both precharging nodes are charged to V_{DD}. Afterward, the clock is changed to V_{SS} which switches the logic gates into the evaluation phase. The first dynamic logic will start to evaluate, however the second dynamic logic will get the output of the first dynamic logic as input. This input is invalid since the first dynamic logic has not yet completed evaluation. This invalid input may accidentally discharge the evaluation node of the second dynamic logic block. To solve this problem, multiple phase clocking[12] can be used. Adding extra delay, as shown in figure 3.3, is also a solution. This extra delay value is highly technology dependent and the accuracy is largely controlled by technology parameters; therefore, it is not a practical solution.

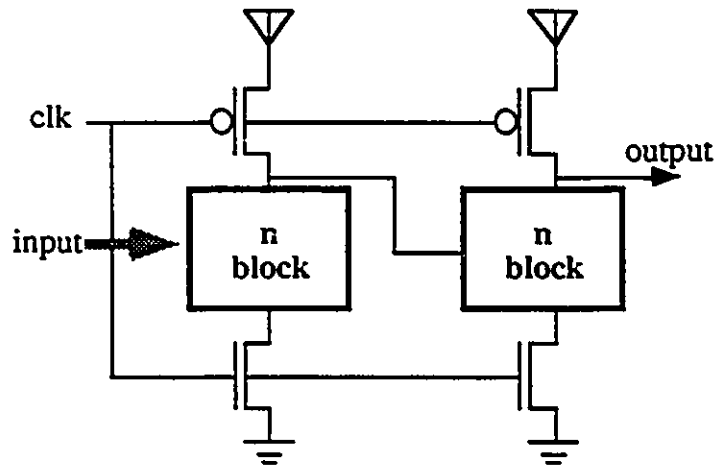


Figure 3.2: Cascading two dynamic logic structures

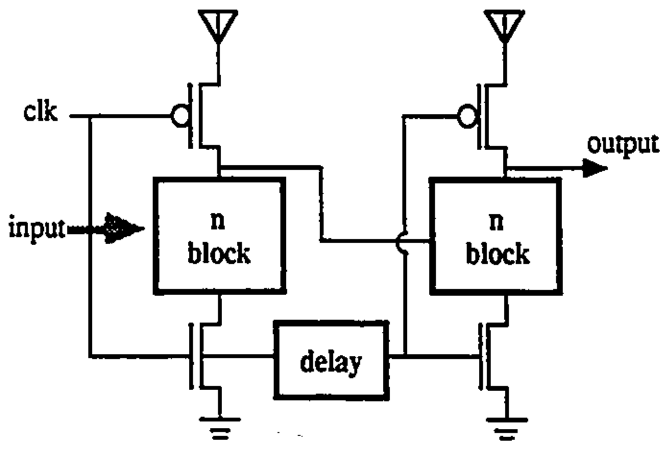


Figure 3.3: Adding extra delay to solve race problem

3.1.1 DOMINO LOGIC

Another approach is through modifying circuit structure. Figure 3.4 shows a domino logic gate[13]. It is simply a dynamic logic gate cascaded with a CMOS static inverter. By doing so, the race problem in cascading dynamic logic can be solved. The inverter inverts the pre-charging node signal and during the beginning of the evaluation phase, all outputs of domino logic are low and therefore no driven domino logic gate will accidentally discharge. Another advantage of this logic is that the inverter acts as a buffer which increases the driving force of the gate. Domino logic has the disadvantage of not being logically complete. Only non-inverting logic functions can be implemented with this logic family. The reason is the uni-directional logic switching property. For a domino logic gate, the output can only change from zero to one or stay at zero, but it cannot change from one to zero. Since the domino logic belongs to the class of the dynamic logic family, it shares some of disadvantages with dynamic logic, such as the leakage problem of the evaluation node. Leakage may cause the evaluation node to discharge to a faulty state.

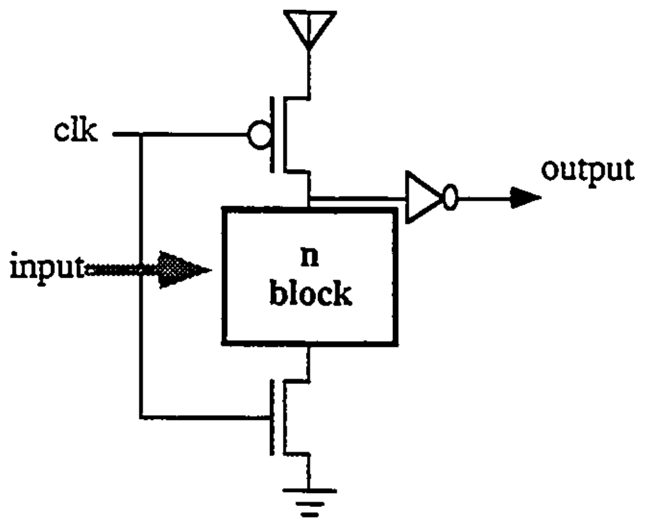


Figure 3.4: Domino Logic

Figure 3.5 shows two solutions for this problem: adding weak transistor to continuously recharge; adding feedback transistor to retain the correct logic level.

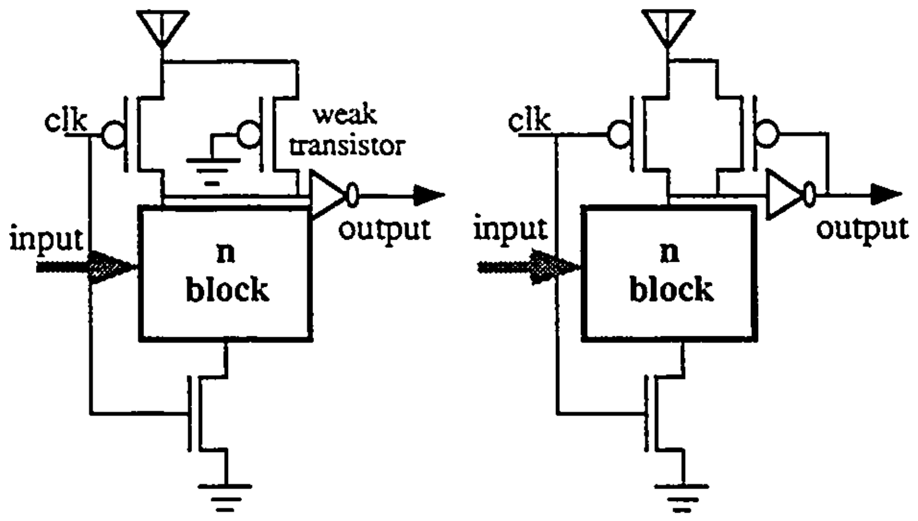


Figure 3.5: Solutions for solving leakage for dynamic logic

3.1.2 CASCODE VOLTAGE SWITCH LOGIC

Domino logic is restricted to non-inverting gates which may cause trouble in logic synthesis. To overcome these problems, cascode voltage switch logic[14] is introduced. This new logic family is logically complete and complementary outputs are provided. Figure 3.6 shows two CVSL logic structures.

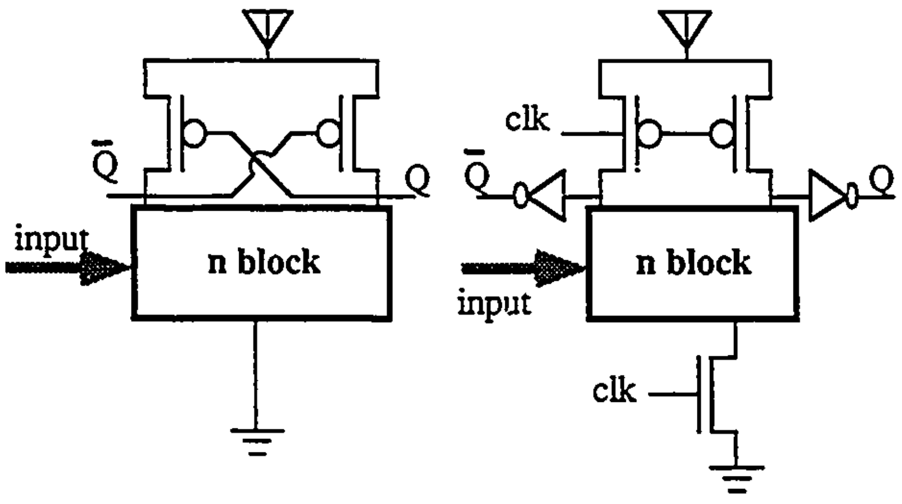


Figure 3.6: Static and dynamic cascode voltage switch logic

The left one is a static gate with a push-pull load and the right one is a dynamic gate with a clock signal connecting both PMOS transistors. Because both complementary and non-complementary outputs are provided, an inverter in this logic family is merely two wires. By reversing the two output terminals of any gate, the complementary function of the gate is generated. An AND gate is identical to a NAND gate. An OR gate is identical to a NOR gate. We need only AND gates and OR gates in this logic family to realize any Boolean function. Another advantage of this family is that the completion of evaluation can be observed from the output, which is not available in other logic families. Whenever a complementary output value pair appears at the output terminals, the evaluation phase is complete and the output values are valid.

3.1.3 CHARGE SHARING PROBLEM

The major problem of using dynamic logic is the charge sharing problem. It occurs when an input to the function block is high and turns on an n-block transistor as shown in figure 3.6. This creates paths to the internal nodes consisting of drain and source capacitance. This capacitance, together with the evaluation node capacitance, shares the charge originally stored only in the evaluation node. The equation below shows the new voltage at the evaluation node.

$$Vp^{*} = \frac{C_p}{C_i + C_p} Vp \tag{3.1}$$

Vp	Pre-charge node voltage
Vp^{*}	New pre-charge node voltage
C_i	Internal capacitance
C_p	Pre-charge node capacitance

The voltage will be lowered to Vp^{*} according to equation 3.1. Rather than pre-charging the evaluation node, some internal nodes can be selectively pre-charged by adding an extra pre-charge transistor[12] as shown in figure 3.7. The selection of internal nodes has to be verified by simulation.

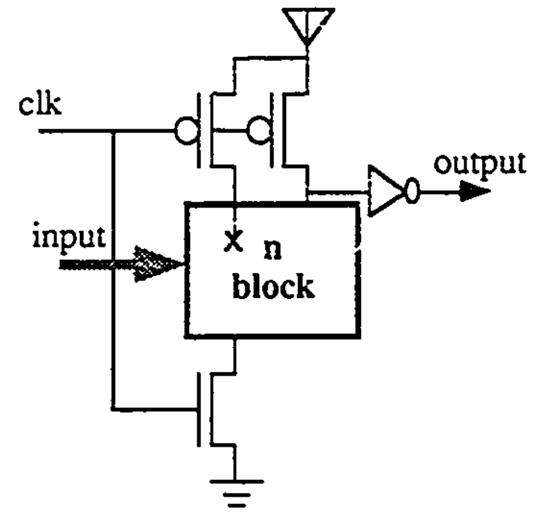


Figure 3.7: Pre-charging internal node

Another solution is to ensure that the input signals are stable before entering the evaluation phase. This can be achieved by delaying the clock signal to the function block, as in figure 3.8. By doing so, the internal nodes can be pre-charged in the pre-charging phase.

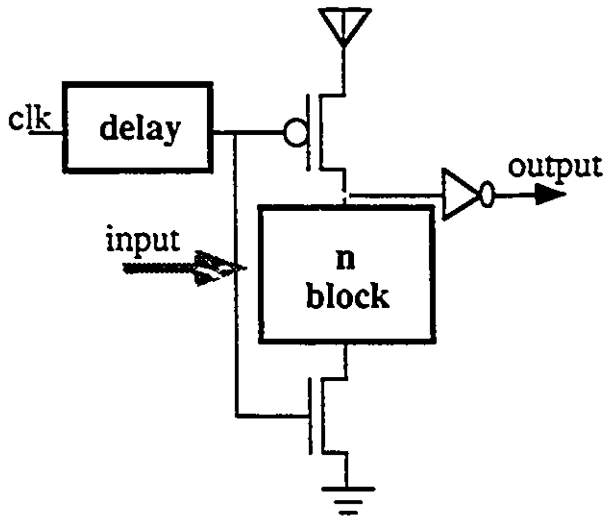


Figure 3.8: Delaying the clock signal to solve the charge sharing problem

By increasing the capacitance of the pre-charging node, the problem can also be solved. An extra capacitor connecting the pre-charge node can increase the capacitanc. or indirectly by using a larger inverter in the domino logic. The feedback transistor configuration used in solving the leakage problem can partially solve this problem. However, it is not a very efficient method because the charge sharing takes place in a very short time while the feedback transistor takes a longer time to re-supply the charge.

3.2 CASCODE VOLTAGE SWITCH LOGIC SYNTHESIS

The traditional method of realizing Boolean function block is based on switching circuit theory[17] which uses AND, NAND, OR, and NOR gate as the basic construction blocks. However, due to the increasing availability of VLSI ASIC design and advancement of computer aided design technology, classical switching network theory[17] which uses switches as the basic elements can be used. These switches are translated into MOSFETs in MOS fabrication technology. A method based on graph theory has been developed for MOS circuit design called switching network logic[21]. Hwang[6] has develop a 32 bits CMOS adders based on multiple-output domino logic. Chu[15] has developed a design procedure based on the Karnaugh map and the Quine-McCluskey method for cascode voltage switch logic design. Recently Jullien[3] has developed a minimization procedure based on a transistor tree for dynamic logic block.

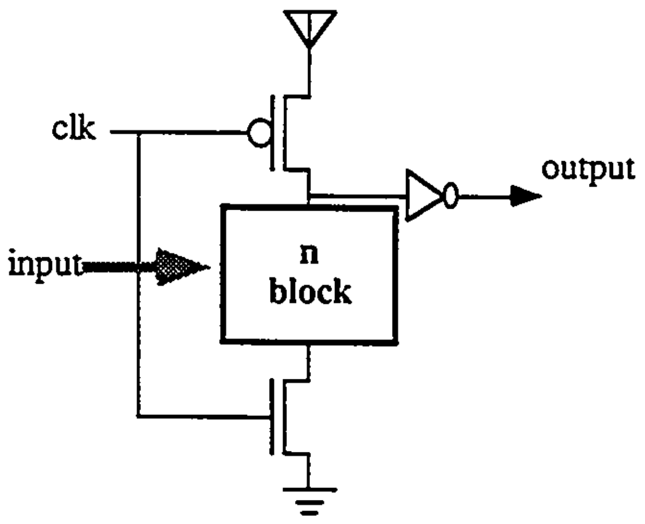
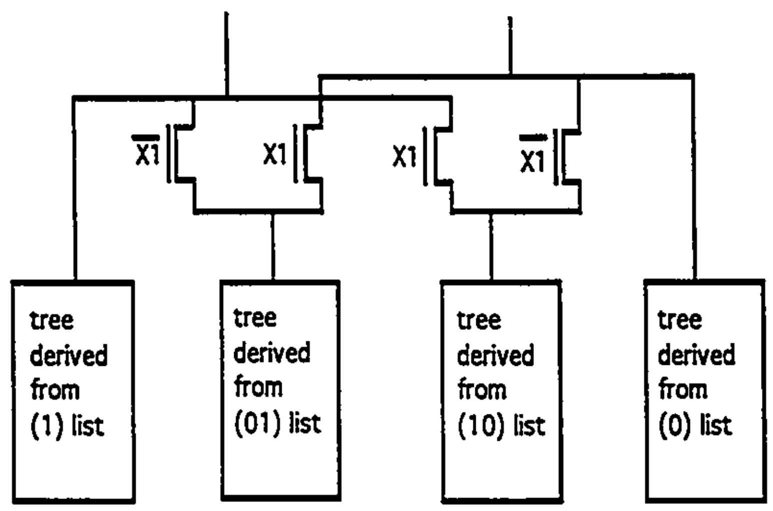


Figure 3.9: n block used in domino logic

Switching network design concerns the topological structure of a transistor network(n) as shown in figure 3.9. This n-block is the NMOS transistor network for dynamic logic design. One or more evaluation nodes are connected to this n-block with a switching pathway connecting these evaluation nodes and the bottom node conditionally turned on and off depending on the state of the input signal. This n-block design can be divided into two sub-problems: the connectivity design and the physical layout design. The connectivity design concerns the transistor connectivity only without considering the physical placement of the transistors. It is appropriate to think of this as a netlist design. The next phase is the physical layout design. This step takes a network as the starting point and the final product is a physical layout description which conforms to the target fabrication technology. The connectivity design is a technology independent procedure while the physical layout design is highly technology dependent. However, some variation can be seen in this hierarchical division of problems. For example, the connectivity design can take the layout style adopted in the physical layout as a constraint parameter and produces a net-list with concern to the final layout design.

3.2.1 TRADITIONAL DESIGN METHOD

The traditional method of cascode voltage switch logic design is based on the K-map and tabular method introduced by Chu[15]. These methods basically divide the n block into four sub-trees as shown in figure 3.10



$\overline{X1}$
X1
X1
$\overline{X1}$
tree derived from (1) list
tree derived from (01) list
tree derived from (10) list
tree derived from (0) list

Figure 3.10: Traditional method of CVSL synthesis

and the designer derives the tree structures based on Boolean functions generated from the K-map or tabular method. Afterwards, the designer merges the sub-trees by finding common structures between all sub-trees. The designer needs some intuitive skill to partition the structure into four sub-trees. It is not an easy task to choose a variable to partition the structure such that the final transistor count is small. The merging part of the procedure is also a problematic step, since merging may create an unexpected conducting path which may cause incorrect evaluation. As the number of input variables increase, the design procedure becomes very time-consuming and the result may not be very efficient. In the following section, a new design procedure which gives very good results, is presented; this procedure is easily automated by computer. Even without a computer, the procedure is quite efficient to apply.

3.2.2 NEW DESIGN METHOD

Any n-block for cascode voltage switch logic can be implemented with a full tree having n levels. For example, the Boolean function, as shown in figure 3.11, is to be implemented. The output column of the truth table is copied to figure 3.12. All of the wires with zeros and ones are connected respectively. After connecting all the zeros and ones wires, there are two terminals at the top of figure. 3.13 and one terminal at the bottom. This three terminal block is the n block for the cascode voltages switch logic. By merging transistors in the n block, the transistor count of the n-block can be significantly reduced. The method for reducing the block is based on the algorithms proposed by Bryant[16]. In this paper, he presents a new data structure for representing Boolean functions and a set of algorithms for manipulating them. The basic element in his data structure representation is a two way decision block commonly used in the tree data structure representation. Its hardware equilvant is realizied by using two transistors proposed by the author.

a	b	c	F
0	0	C	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 3.11: Example Truth Table

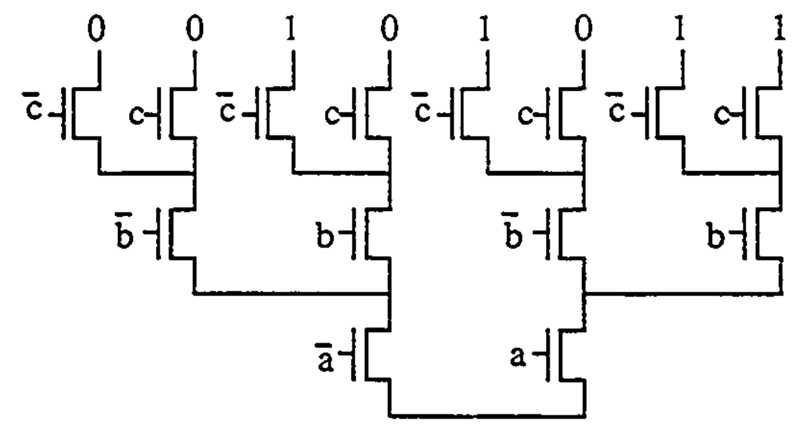


Figure 3.12: Full tree of transistors with terminals assigned with logic values.

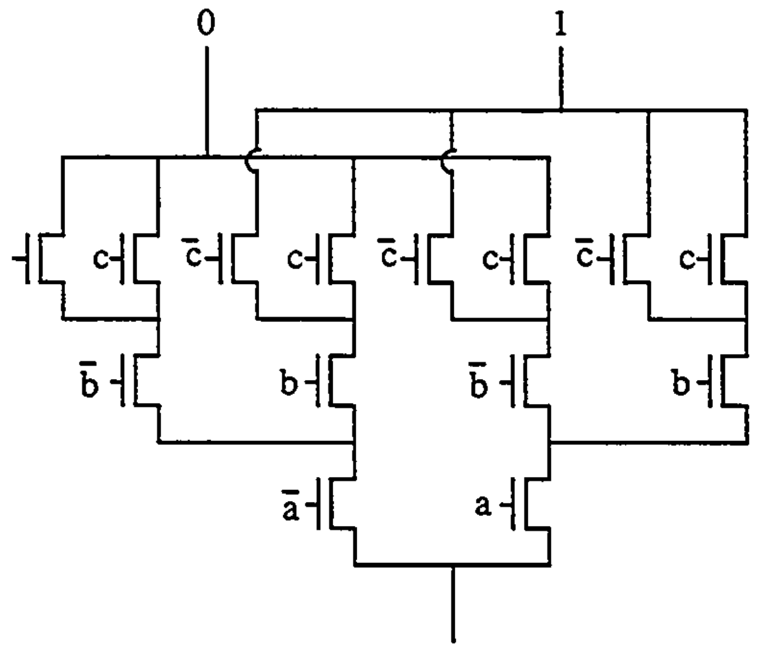


Figure 3.13: Full tree of transistors with terminals wired

The basic element for this method is a circle symbol representing two transistors as shown in figure 3.14. This circle is a three terminal structure with two terminals at the top and one at the bottom. The variable beside the circle is the input variable connected to the gates of the two transistors inside the circle. A node number is usually attached to the circle for indexing purposes.

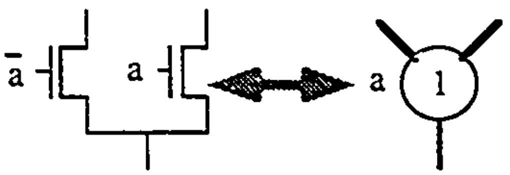
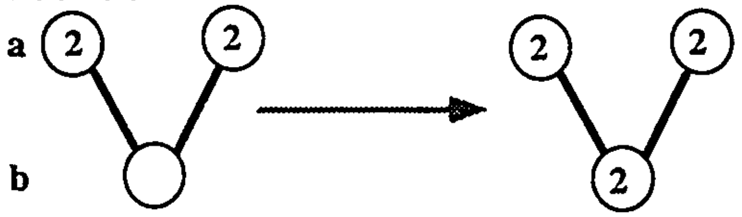


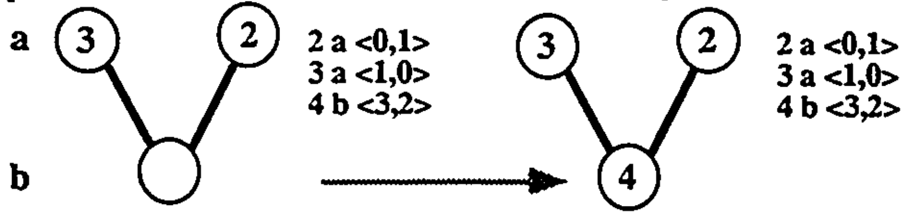
Figure 3.14: equivalent circuit for a circle symbol

Another aspect of the method is a set of rules for assigning the node number to each circle. Table 3.1 show the rules. This procedure is started with an empty key table which grows when a new key is assigned to a node. After every node has a node number attached to it., the key table will be used for constructing the merged tree diagram. This merged tree diagram is then mapped to the transistor network by substituting the circles with the corresponding transistors.

Rule 1: If the left and the right node numbers are equal, the parent node number will be equal to the children's node number.



Rule 2 If the left and the right node numbers can be found in the key table, the recorded parent node number will be substitute to the current parent node.



Rule 3 If the left and the right node numbers cannot be found in the key table, a new parent node number will be created and recorded in both the key table and the current parent node.

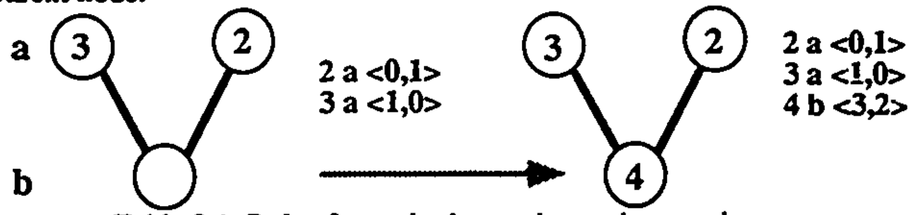


Table 3.1: Rules for assigning node numbers to the tree

Figure 3.15 shows a truth table for the sum bit of a full adder. The output column is copied to the top of the figure 3.16 which shows a full tree inverted.

a	b	c	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3.15
Full adder truth table

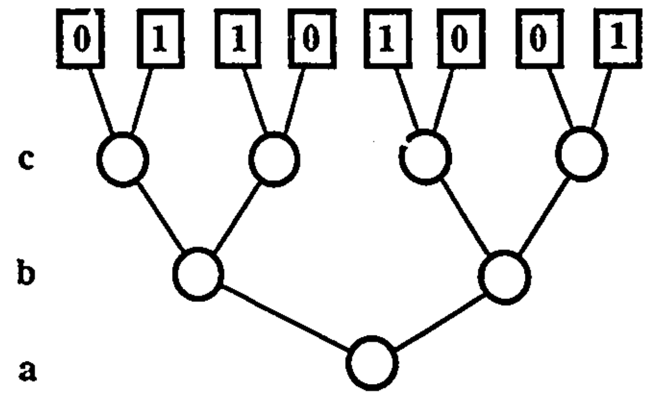
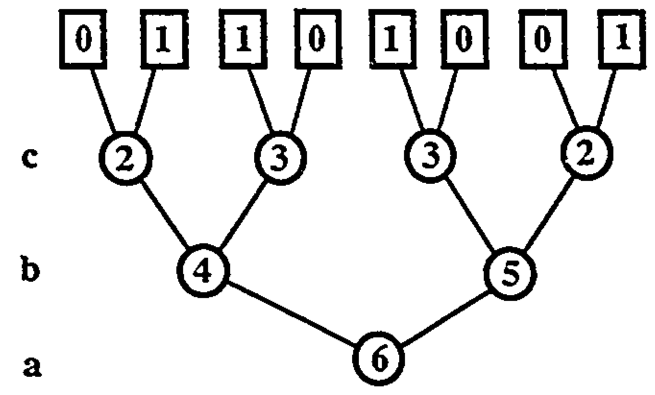


Figure 3.16: Full tree with sum bit column of the full adder truth table

The terminals are connected to the rectangular boxes containing the output values of the sum bit. The circles are assigned with node numbers according to the rules presented in table 3.1. The final tree with assigned node numbers is shown is figure 3.17.



2	c	<0,1>
3	c	<1,0>
4	b	<2,3>
5	b	<3,2>
6	a	<4,5>

Figure 3.17: Full Tree with assigned node numbers

The key table is then used for constructing the merged tree diagram as shown in figure 3.18.

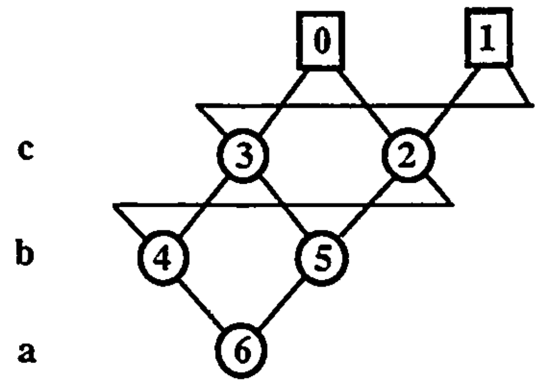


Figure 3.18: Merged tree for sum bit of full adder

The first row of the key table 2c<0,1> corresponding to a circle with 2 as the node number, placed at the row c. Its left wire is connected to 0. Its right wire is connected to 1. Each row is mapped to the merged tree diagram one after another. After the merged tree is available, the transistor connection diagram for the n-block is constructed by substituting each circle with the corresponding transistors. The final n-block is in figure 3.19.

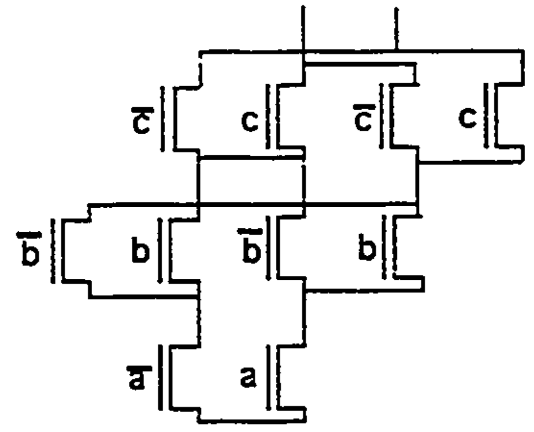


Figure 3.19: CVSL implementation of sum bit generation for full adder (n-block only)

Connected with the pre-charge transistor, ground switch and inverter buffer, it is shown in figure 3.20.

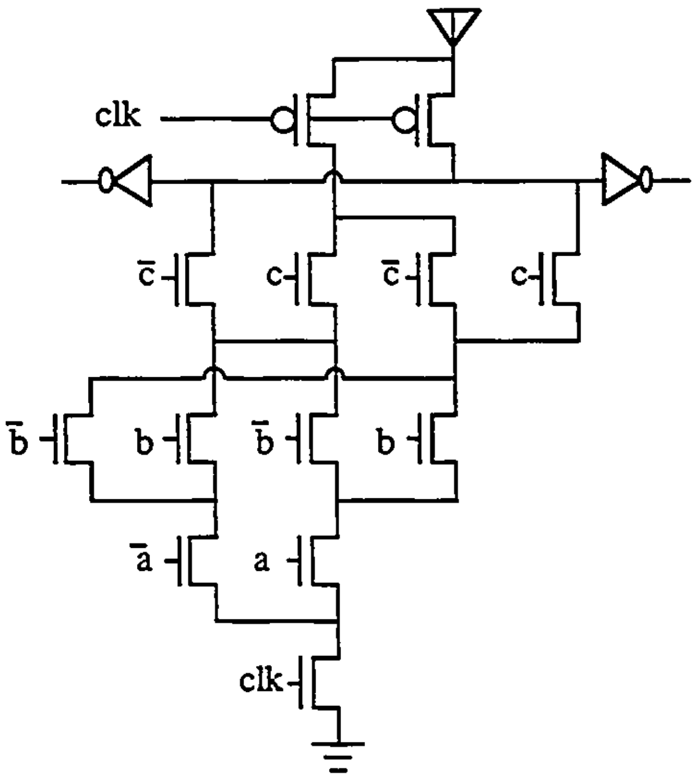


Figure 3.20: Dynamic cascode voltage switch logic implementation of the sum bit

A few examples are shown in figure 3.21 and figure 3.22. The variable ordering will significantly affect the resultant transistor count. Figure 3.23 shows two circuits of the middle bit of a 2 bit adder with a good and a bad variable ordering. In order to get a circuit with least transistor count, enumeration of all combinations of variable ordering should be incorporated into the synthesis procedure. For an n input variable function, there are n! combinations to try.

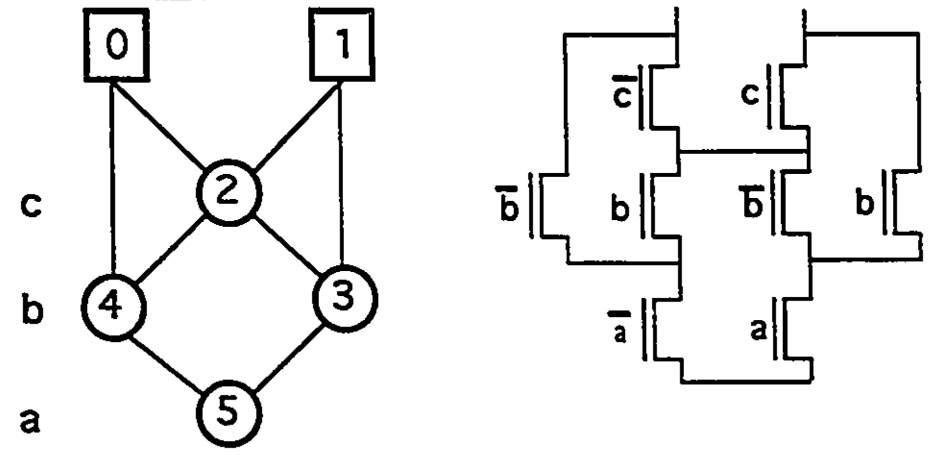


Figure 3.21: CVSL n-block of the carry bit generation for full adder

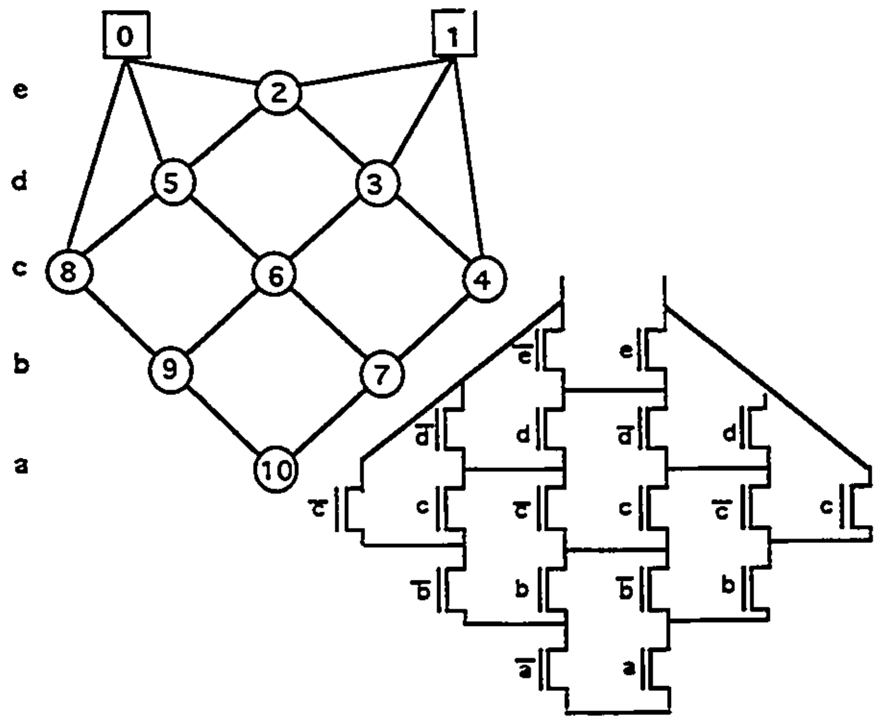


Figure 3.22: CVSL implementation of 5 bit majority function

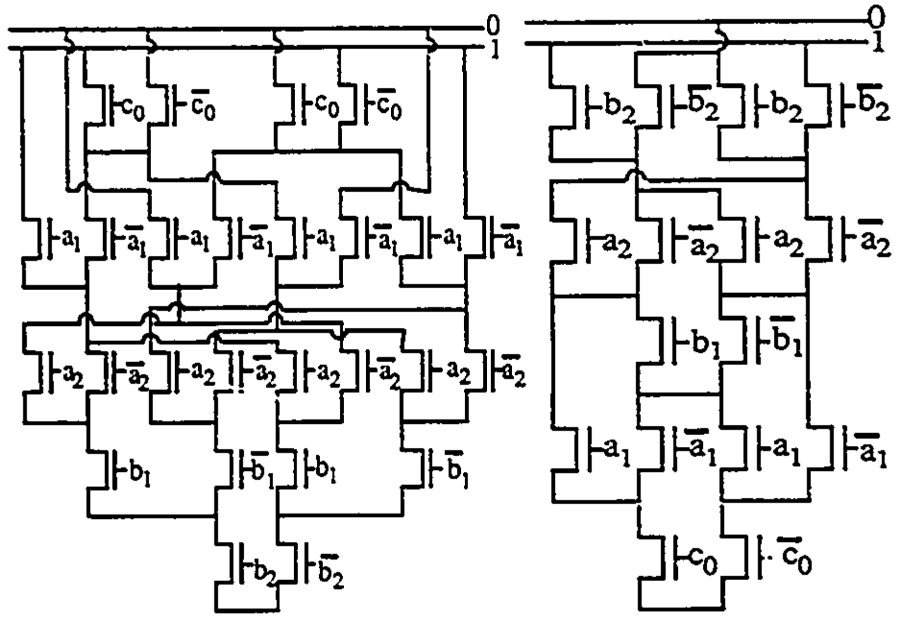


Fig 3.23: Two implementations of the middle bit of a 2 bit adder.

3.2.3 SEMI-CUSTOM CELL DESIGN

The n-block design method proposed previously is suitable for full custom VLSI design. Here, a new semi-custom CVSL programmable cell is proposed. As mentioned before, any Boolean function of n variables can be implemented with a full tree of n levels. A full tree of 3 levels is shown in figure 3.24.

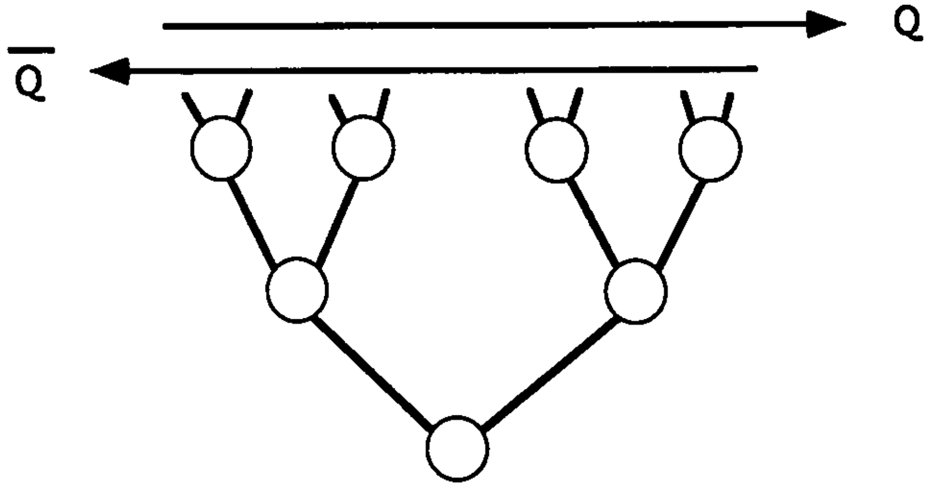


Figure 3.24: Full tree implementation for CVSL

The terminals of the top row circles can be connected to either Q or $\bar{Q}$ depending on the truth table of the target Boolean function. This approach takes a lot of transistors compared with the reduced tree method. However, some tricks can be applied to the full tree to reduce the transistor count by almost half.

First, observing the circles at the top row of the full tree diagram, there are only four possible connections for each top row circle. They are shown in figure 3.25a. Also, two of the possible connections can be reduced to a single wire only as shown in figure 3.25b.

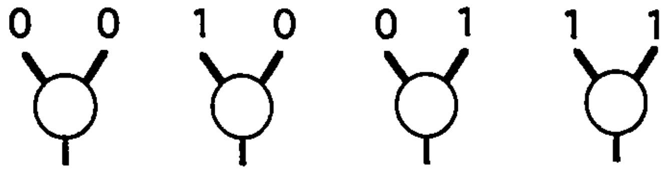


Figure 3.25a: All possible combinations of circle symbol at the top level

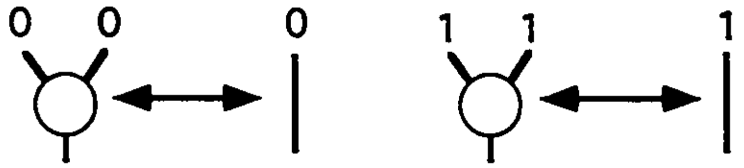


Figure 3.25b: Circle to wire transformation

By pre-encoding these four possible connections, the top row of any full tree can always be replaced with 2 circles. The wire connection part is moved one level into the tree as shown in figure 3.26. Since the top row of any full tree has the largest number of circles than the sum of all other rows, the encoded tree has significantly less transistors than the full tree. The full tree has 2^{n+1}-2 transistors where n is the number of input variables. The encoded tree has 2^n+2 transistors. As n increases to a large value, the percentage of saving will become almost 50%.

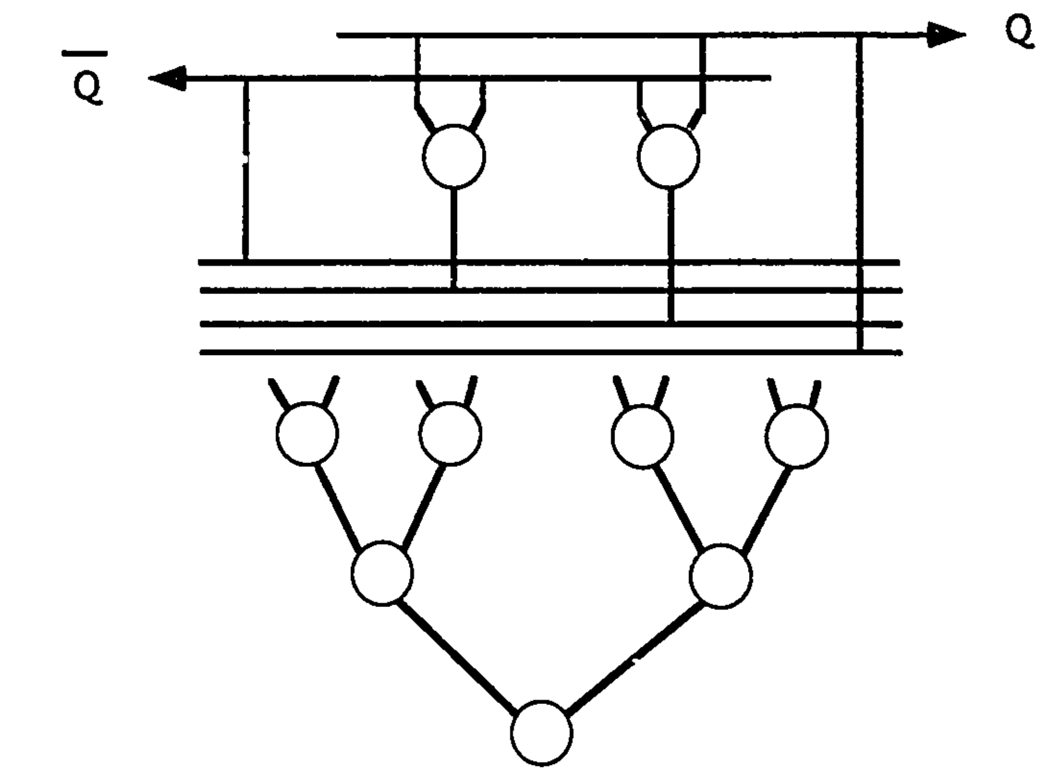


Figure 3.26: One level encoded semi-custom cell

3.2.4 COMPARISON

To get an overview of the advantage of using the new reduced tree and the encoded tree, a table (table 3.2) is shown below for comparison. All the data are generated from a computer optimization program. It can be seen that the new reduced tree method always generates the best solution.

Transistor count table for different tree structures

Cell function	New S-tree	Full tree	encoded tree
5 bits Majority	18	62	34
5 bits XOR	18	62	34
5 bit Prime detector	20	62	34
mod3 detector	22	62	34

Table 3.2

3.3 MULTIPLE OUTPUT DOMINO LOGIC DESIGN

In the previous section, a new cascode voltage switch logic design method has been proposed. This method generates circuits with both complement and non-complement outputs . To implement multiple output cascode voltage switch logic, this procedure is applied to each output function separately. The cascode voltage switch logic can be seen as two domino logic gates grouped together. However, for the new design method proposed, the two separate n-blocks are merged together. If two can be merged, more blocks should possibly be merged. This multiple-block merging technique is called the Multiple-Output Domino Logic (MODL) synthesis method. MODL synthesis is not a new concept. It has been used in Hwang's work[6]. Their Model of synthesis of MODL is shown in figure 3.27 which restricts $f = f_1 f_2$. A more complete method of MODL synthesis is proposed in Jullien's work[3]. In that paper, the MODL synthesis is treated as a tree reduction problem by using a few reduction rules. However, the order in which the reduction rules are applied will greatly affect the result. Also, after each rule is applied, a checking of correctness is necessary since applying a rule may alter the implementation of the original Boolean function. A more efficient method is proposed in the next section. This method generates an unique transistor network for each specific variable ordering as opposed to other methods. Checking of correctness is unnecessary for this synthesis method. Reduction by hand is also feasible. The final layout area is smaller than by using other methods if th full custom layout style is adopted.

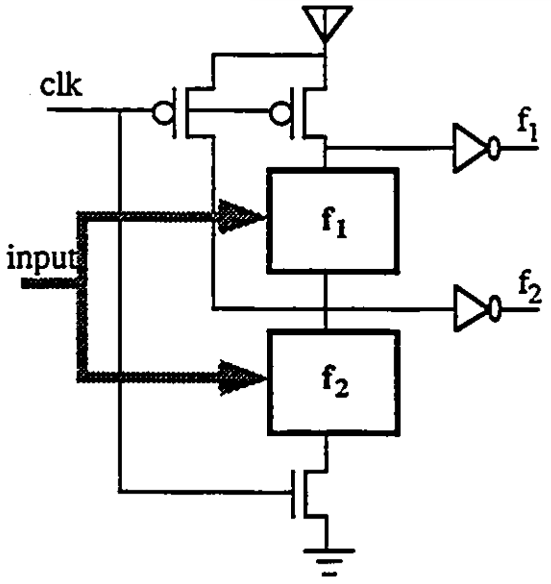


Figure 3.27: Traditional model of MODL

3.3.1 THE NEW METHOD

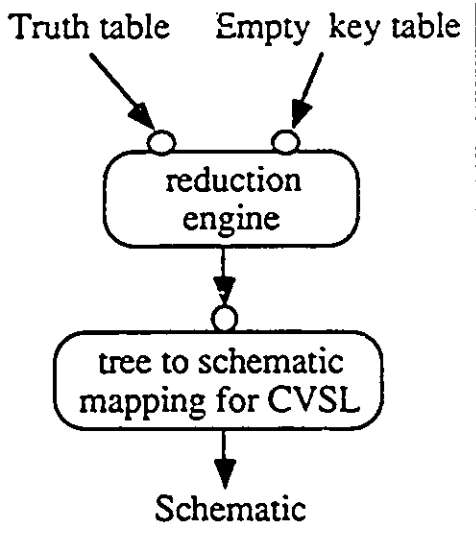


Figure 3.28: CVSL synthesis

The new synthesis method for MODL is a modified algorithm of the new CVSL synthesis method introduced earlier. Before the MODL method is described, a birds-eye view of the CVSL algorithm is presented first. Figure 3.28 shows a block level abstraction of the CVSL algorithm. The truth table and the empty key table are supplied to the reduction engine. This reduction engine applies the rules to the full tree and then each circle in the tree is assigned a node number. Following this, a merged tree description is formed in a key table

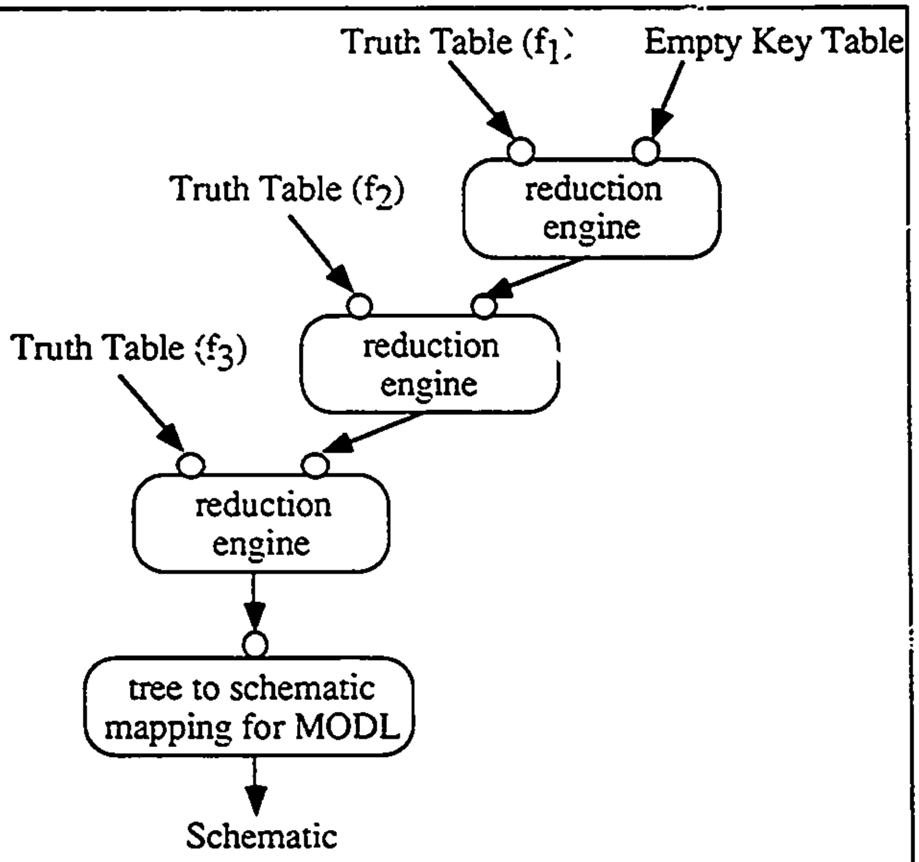


Figure 3.29: MODL synthesis

format. This key table goes through a mapping procedure by transforming each circle into two transistors. This part is accomplished in the "tree to schematic mapping" block. The result is the schematic for the n-block. The block diagram of the MODL synthesis procedure is shown in figure 3.29. The reduction engine used in MODL synthesis is same as the one used in CVSL synthesis. The tree to schematic mapping procedure is different from the one used in CVSL. The block diagram (figure 3.29) is for the three outputs function (F_1,F_2,F_3). By cascading more reduction engines, the synthesis procedure for any number of outputs is possible. For MODL, the n-block, as shown in figure 3.30, is the result. Since the zero terminal has no use in MODL, we eliminate those transistors in the n-block which are attached to the zero terminal. An example of the majority function is given in figure 3.31 The n-block on the left side is generated from the new CVSL algorithm. By eliminating the three transistors connected to zero terminal, an n-block for domino logic is derived as shown on the right side. The number of transistors can be further reduced by replacing the two circled transistors with two wires.

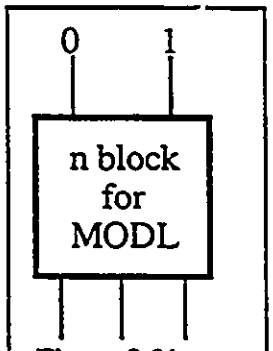


Figure 3.30
N block for MODL

Since the zero terminal has no use in MODL, we eliminate those transistors in the n-block which are attached to the zero terminal. An example of the majority function is given in figure 3.31 The n-block on the left side is generated from the new CVSL algorithm. By eliminating the three transistors connected to zero terminal, an n-block for domino logic is derived as shown on the right side. The number of transistors can be further reduced by replacing the two circled transistors with two wires.

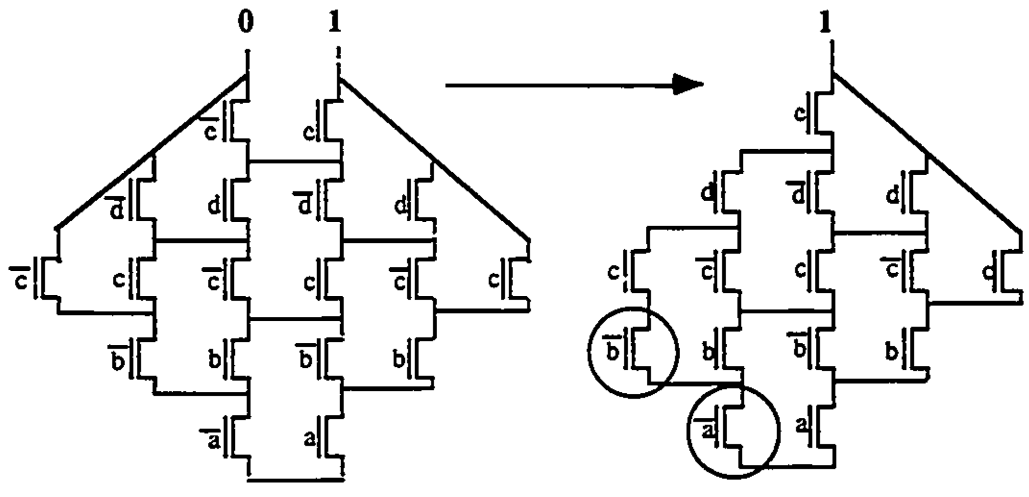


Figure 3.31: CVSL and domino logic implementation of Majority function

After eliminating the redundant transistors and the zero terminal, the n-block will be inverted before connected to the rest of the circuitry, as shown in figure 3.32.

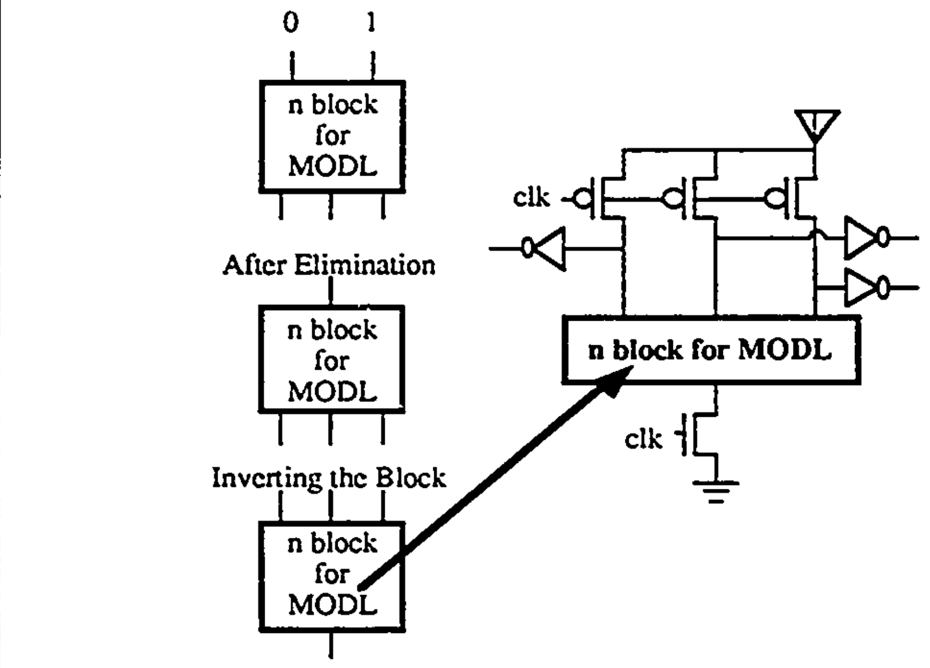


Figure 3.32: The steps required after the merged tree generation

This simple elimination and inversion is the difference between CVSL and MODL in the "tree to schematic mapping" procedure. An example of a 2 bit adder is shown in figure 3.33.

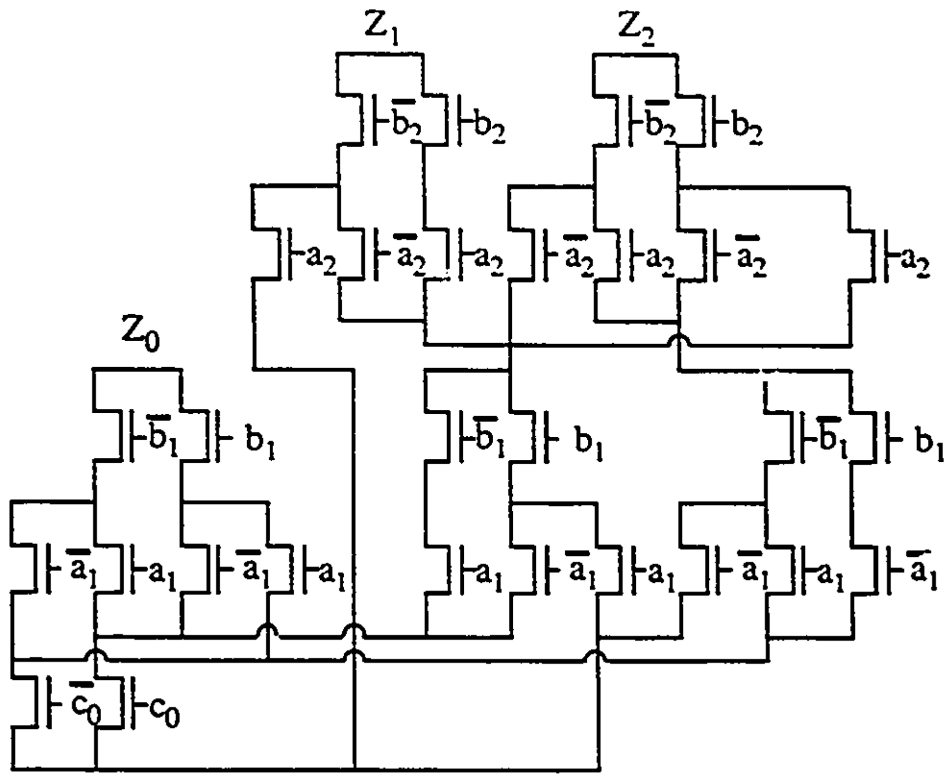


Figure 3.33: MODL implementation of 2 bit adder

3.4 DON'T CARE CASE

In a real world application, not all digital circuits have fully specified minterms as the requirement. Some applications, for instance BCD to 7 segment decoder, MOD 7 multiplier, BCD code related circuits and some control circuits do not have fully filled truth tables. In other words, they have some "don't care cases". These don't care cases can be zeros or ones. This degree of freedom gives us choices for further reduction.

One easy way to make use of this freedom is enumeration of all possible combinations to the "don't care cases" and afterwards, each truth table will pass through the reduction process. The one which produces a minimum transistor count is the final selection. For n variables with m don't care cases, there are $2^m n!$ trees to reduce. n is usually less than 8; however m can be any number between 0 to 127 for functions with less than 8 variables. If a function has many don't care cases, the number of combinations for reduction will lead to unacceptable computation time.

Because of this problem, a new algorithm is developed to handle the don't care cases more efficiently. The don't care case algorithm basically consists of two main parts. The first part substitutes the don't care cases in the truth table with ones and zeros. The second part performs the tree reduction. The reduction algorithm is either the CVSL reduction engine or MODL reduction engine depended on the target application. The algorithm is shown as below

```
Generate all variable ording combinations
Repeat
    put the output columns of the truthtable into "logic"
    logic<-dontcare(logic)
    tree<-reduction engine(logic)
Until all variable orderings are used
get the tree with the least number of transistors
```

"logic" is a representation of a truth table. Only the output columns are recorded in the "logic". For example a full adder has eight rows for the possible minterms and two column for the sum and carry bit outputs. These two columns are recorded as below

$$< 0 > \quad < 01101001 >$$
$$< 1 > \quad < 00010111 >$$

The "don't care" procedure substitutes each don't care case from "X" to "1" or "0". It is a recursive procedure which basically does pattern matching. The underlying concept behind this procedure is the assumption that more transistors can be reduced if the merging occurs closer to the top of the tree. It is a valid assumption since a sub-tree closer to the top of the tree generally consists of more transistors than the one farther down. It can be visualized from figure 3.34.

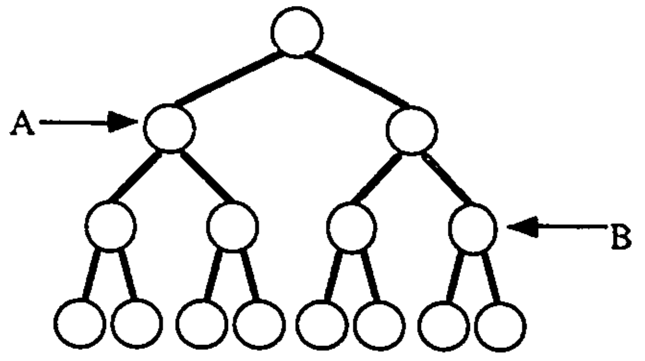


Figure 3.34: Full tree of circle symbol

Sub-tree A consists of more transistors (circles) than sub-tree B. The algorithm first tries to do merging by assigning ones and zeros to the truth table. If the trees cannot be merged, then the algorithm divides the each tree by half, and repeats the procedure until all don't care cases are bound to fixed binary values. The algorithm is shown below.

```
dontcare (logic)
        logic <- bindlogic (logic)
        IF allfixed (logic)
                RETURN logic
        ELSE
                aftersplit = split (logic)
                tobebind = dontcare (aftersplit)
                logicout = rebind (tobebind, logic)
                RETURN logicout
        END
```

"bindlogic" is a procedure for finding the possible merging of vectors. Two or more vectors can be merged if each corresponding element in the vectors can be bound together. An "X" can be bound with "X", "1" or "0". A "0" or "1" can be bound with itself or "X". Table 3.3 shows all the possible results.

Input	Result
x 0	-> 0
x 1	-> 1
x x	-> x
0 1	-> can't be bound.

Table 3.3: functionality of the "bindlogic"

An example for "bind logic" is given below.

input		output	
0	< 0x10x1xx >	< 0,1 >	< 00100110 >
1	< 00100x10 >	< 2 >	< 100101x0 >
2	< 100101x0 >		

vectors 0 and 1 can be merged together.

"Allfixed" is a procedure for testing the existence of "don't care cases" in the vector table. If the vector table contains any number of "don't care cases", the result will be false, otherwise the result will be true.

Split is a procedure for splitting the vector table into a larger one. Each vector will be split into two vectors. for example

input		output	
< 0,1 >	< 00100110 >	< 0 >	< 0010 >
< 2 >	< 100101x0 .	< 1 >	< 0110 >
		< 2 >	< 1001 >
		< 3 >	< 01x0 >

"Rebind" is a procedure for mapping the split vectors table back into the original one. For example

	argument 1		argument 2	
input	< 0,1 >	< 0010 0110 >	< 0 >	< 0010 >
	< 2 >	< 1001 01x0 >	< 1 > < 3 >	< 0110 >
			< 2 >	< 1001 >
output	< 0,1 >	< 0010 0110 >		
	< 2 >	< 1001 0110 >		

By considering don't care cases, the number of transistors used in a module 7 multiplier is reduced from 56 to 52. This is only one specific example which shows only a small improvement. The number of transistors saved may be larger for other applications.

3.5 WINDSOR SYNTHESIZER AND PLANAR SWITCHING TREE

The new CVSL or MODL synthesis procedure produces a network as a result. The next step of the logic implementation is the physical layout of the netlist produced in the synthesis procedure. This step can be simply accomplished by full custom layout. It is a very time-consuming task and an error-prone activity. Another approach is automation of this step by computer. Windsor synthesizer (WINSYN)[19] is such a program for mapping transistor network to physical layout.

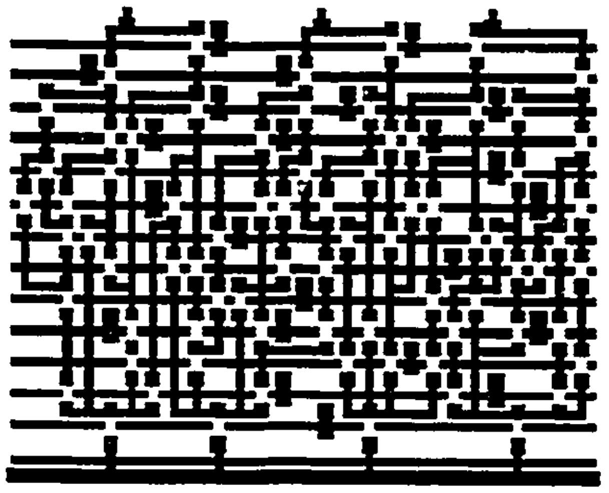


Figure 3.35 WINSYN Layout

Figure 3.35 shows a layout produced from the WINSYN for double-metal CMOS technology. The horizontal lines running across the layout are metal2 wires and polysilicon wires overlapped together. The function of these metal2 wires is for minimizing the resistance. The long polysilicon wires have a large resistance. Tapping some points in the line to metal2 will reduce the resistance. The polysilicon wires are the gate inputs for all transistors. The transistors are interconnected by metal1 wires. By traversing the netlist, the layout can be produced very easily. The only drawback with this approach is due to 2 dimensional wiring restriction. Only metal1 is used for interconnection, therefore the netlist is restricted to a planar structure. No crossing wire is permitted. However, the synthesis procedures will generate both planar and unplanar netlist. To overcome this problem, a new procedure is derived to transform a unplanar merged tree to a planar merged tree.

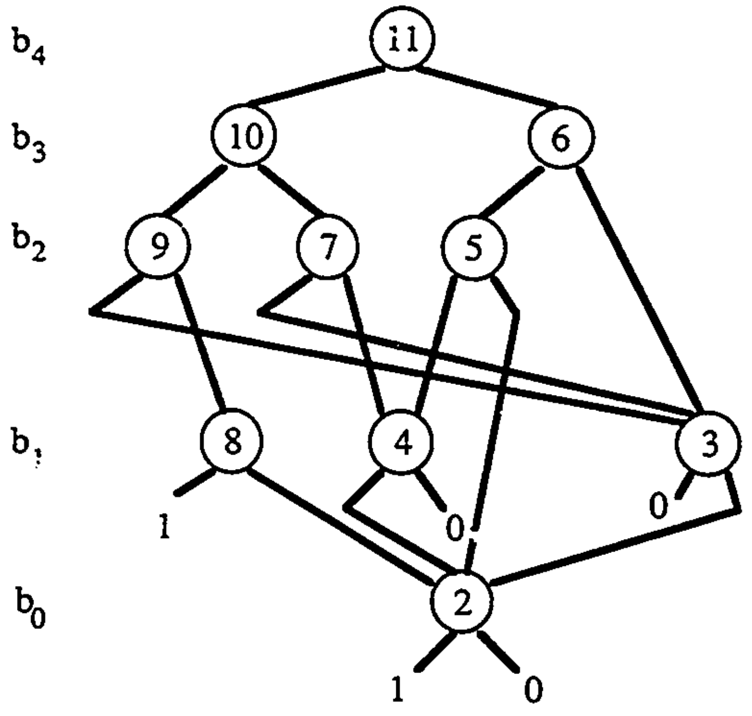


Figure 3.36: Merged tree diagram for prime detector

The procedure is better illustrated by example. Figure 3.36 shows a merged diagram for a prime detector. Its function is to determine whether, or not, the input value is a prime. The algorithm builds nodes row by row. The root node is re-created first and then its right and left children are re-created as shown in Figure 3.37. Node 10 will create node 9 and 7. Node 6 will create node 5 and 3; however, the location of node 3 is not in the same row as 9, 7 and 6. This special case is handled

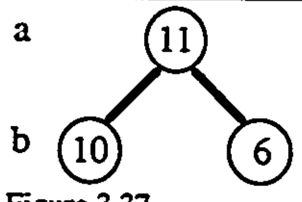


Figure 3.37
Step 1 of transformation

by creating a virtual node 3 in the row that it is not supposed to be. This is shown in figure 3.38.

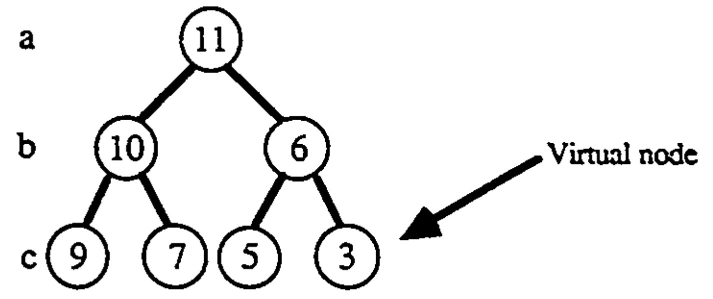


Figure 3.38: Step 2 of transformation

Procedure continues after this step. The next row is created as shown in figure 3.39.

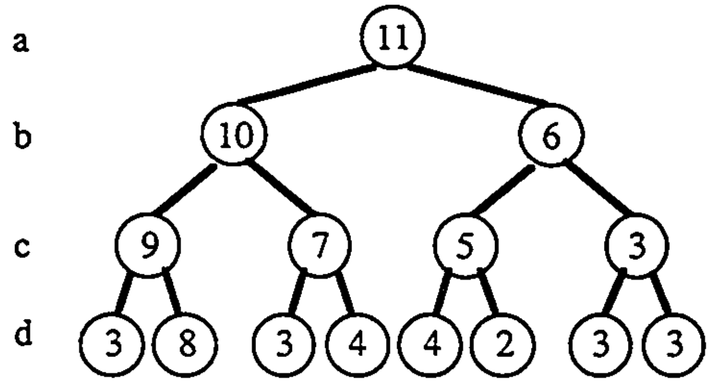


Figure 3.39: Step 3 of transformation

Virtual nodes will create children with the same node number as their parent. This node 3 in level c is a virtual node, therefore it creates children with the same node number as their parent. In row d, there are nodes with the same node number. By exchanging the right and the left children, the number of nodes can be reduced as shown in figure 3.40. The exchanged parent node is marked with a double circle.

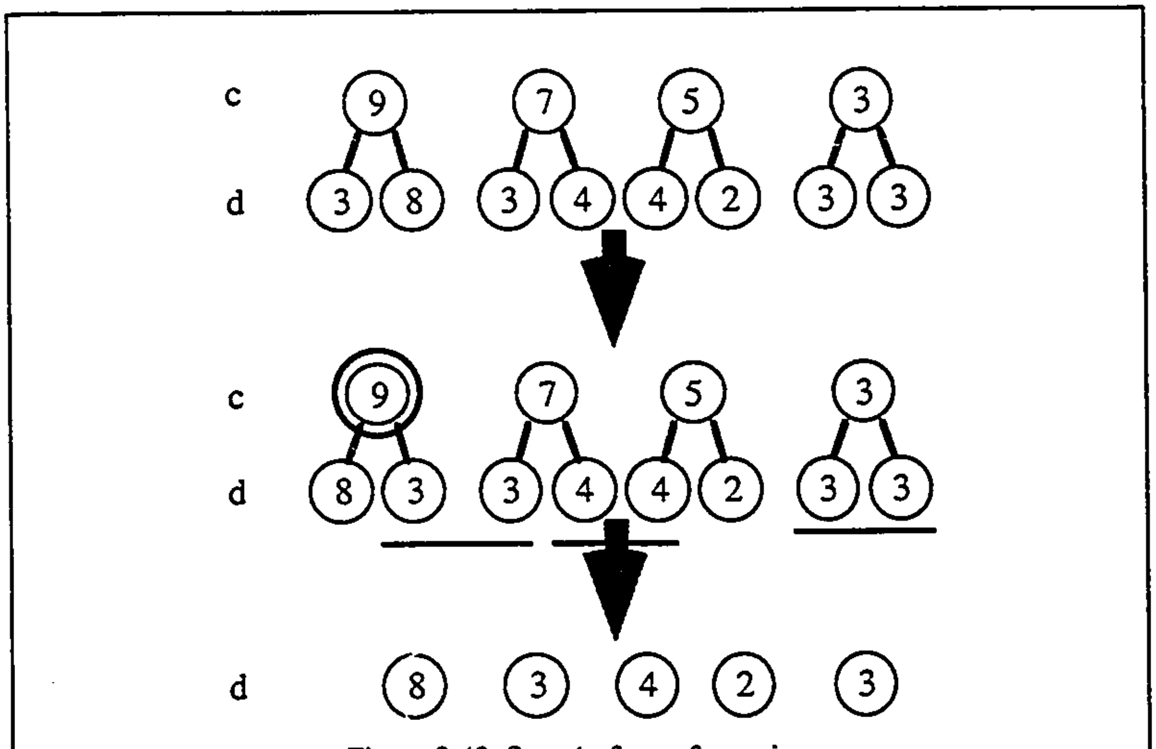


Figure 3.40: Step 4 of transformation

This new row is substituted into the tree as shown in figure 3.41

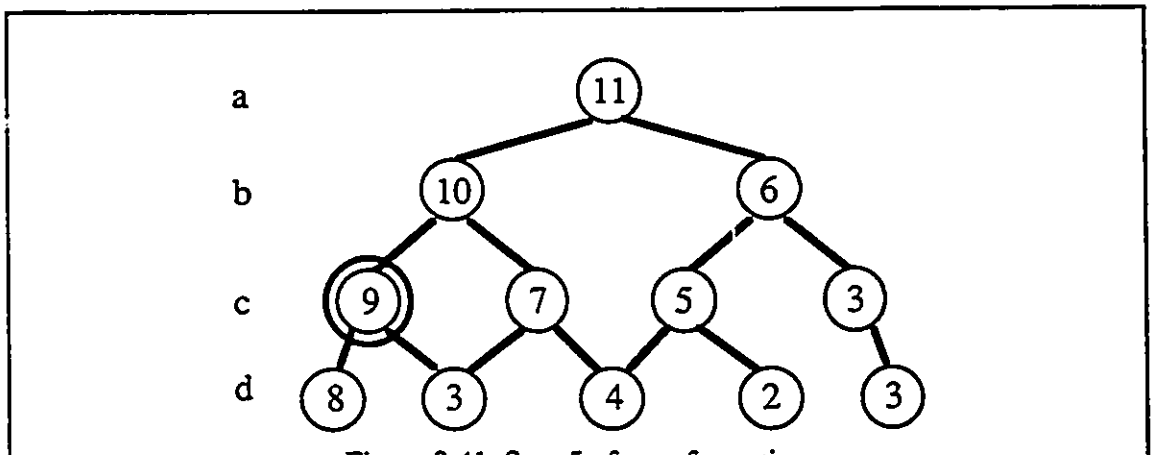


Figure 3.41: Step 5 of transformation

Another row is created as shown in figure 3.42.

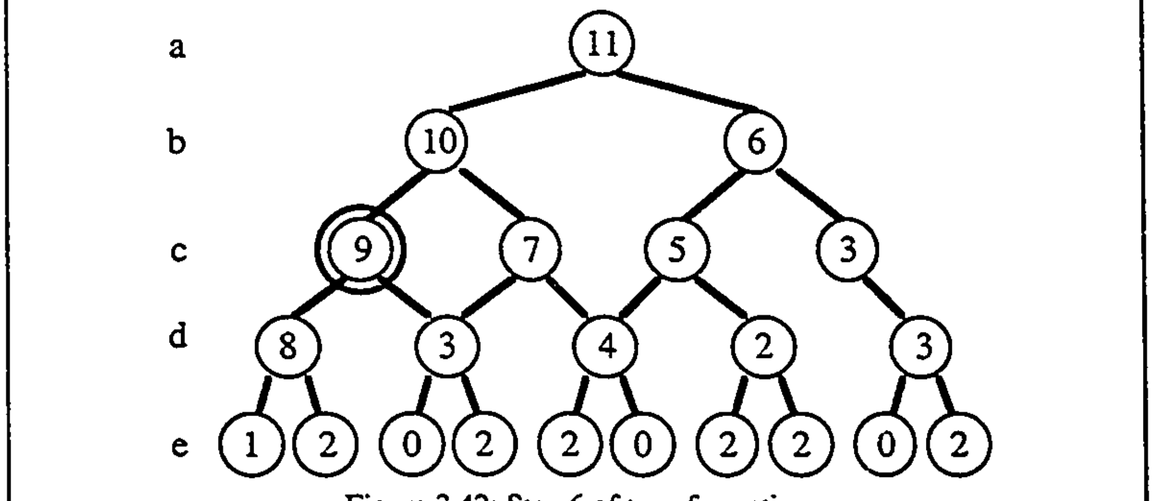


Figure 3.42: Step 6 of transformation

After flipping and merging, the tree becomes as shown in figure 3.43

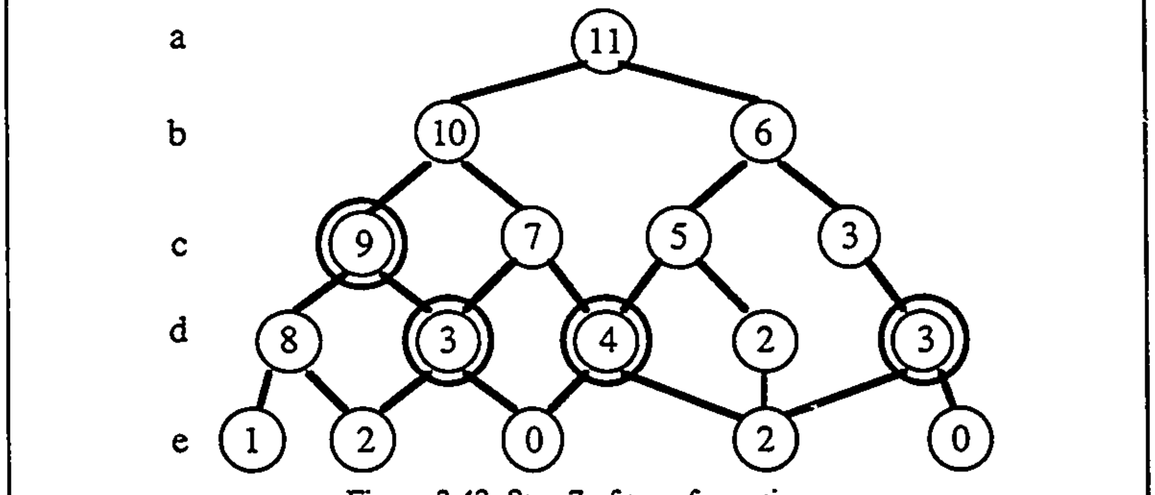


Figure 3.43: Step 7 of transformation

The last row is created as shown in figure 3.44

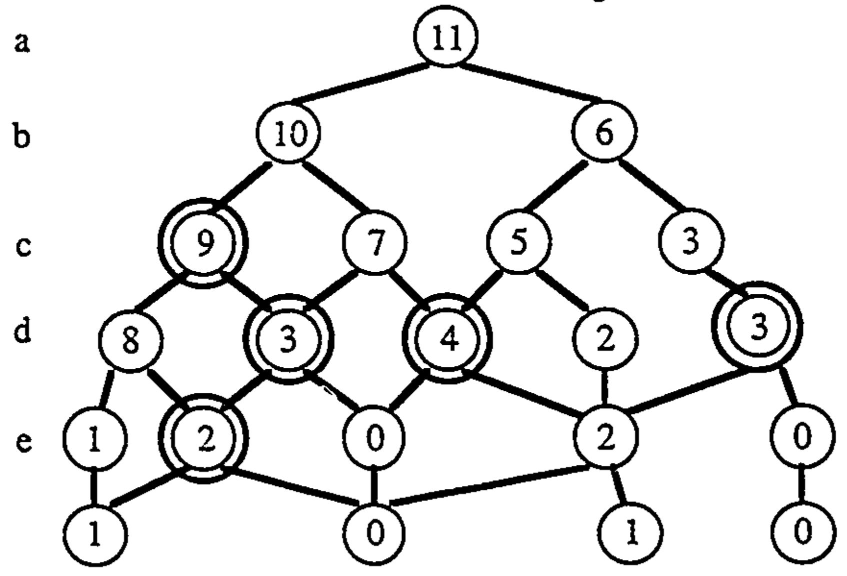


Figure 3.44: Step 8 of transformation

The next step is the elimination of redundant circles from the planer tree. Those circles with single bottom wires are eliminated by substituting them with wires. The final merged tree diagram is shown in figure 3.45

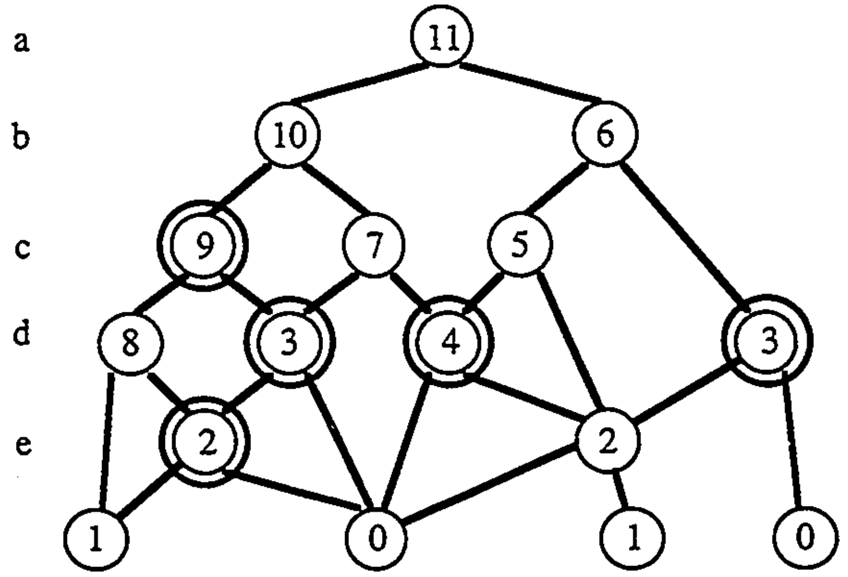
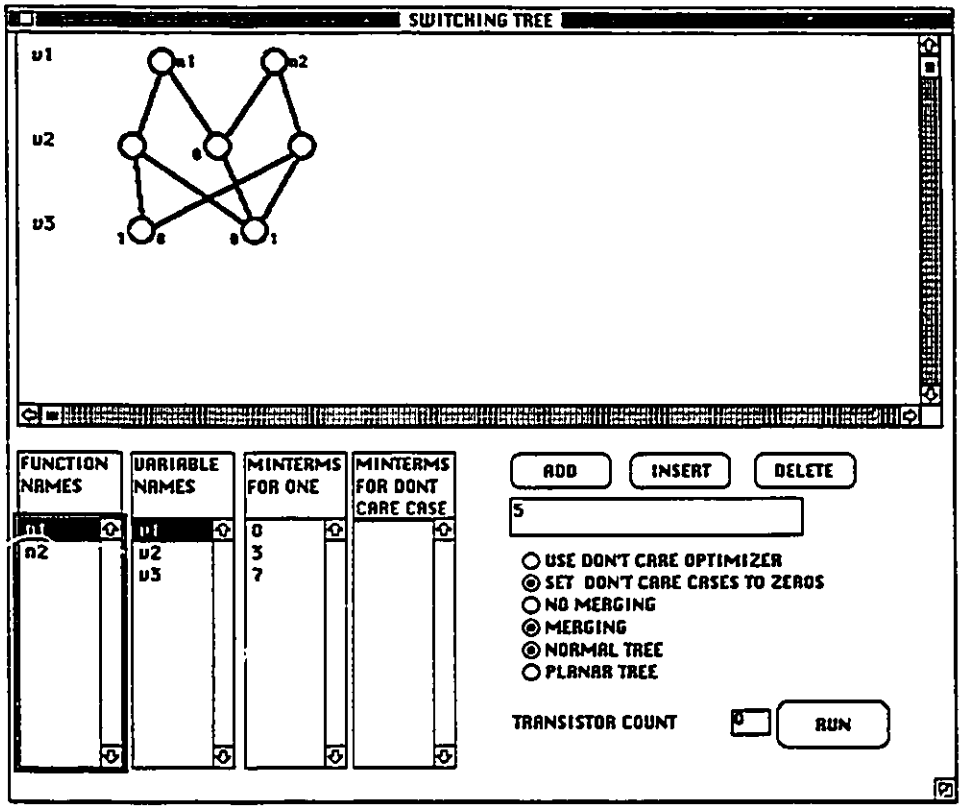


Figure 3.45: Step 9 of transformation

The new planer switching tree takes only 4 more transistors than the original unplanar tree. However, the area of the layout generated from WINSYN is not determined by the transistor count, rather it is determined by the number of branches of the switching tree.

3.6 SYNTHESIZER DESIGN

All of the algorithms introduced earlier are automated with a program written in PROGRAPH[20] and C[21] running on a Macintosh. The C part of the program is responsible for the reduction procedures and all the time-consuming steps. The PROGRAPH code is used as the graphical user interface (GUI) because of the ease of effort in GUI programming. The GUI is shown in figure 3.46. MODL and CVSL synthesis can be selected by toggling the merging option. The "don' tcare" procedure is activated by selecting the "don't care" option. The planar tree option is for planar switching tree generation. After the truth table and options are entered, pressing the RUN bottom will produce a merged tree diagram in the graphic window.



FUNCTION NAMES	VARIABLE NAMES	MINTERMS FOR ONE	MINTERMS FOR DONT CARE CASE
n1	v1	0	
n2	v2	3	
	v3	7	

5

○ USE DON'T CARE OPTIMIZER
◉ SET DON'T CARE CASES TO ZEROS
○ NO MERGING
◉ MERGING
◉ NORMAL TREE
○ PLANAR TREE

TRANSISTOR COUNT 0

Figure 3.46: graphic user interface for the synthesizer

3.7 SUMMARY

The switching tree synthesis part of this thesis is mostly the author's own contribution. This chapter starts with an introduction to dynamic logic design. The race problem is discussed which leads to the examination of domino logic. The cascode voltage switch logic is also examined for solving the logical incompleteness of the domino logic. The cause and solutions of the charge sharing problem are discussed. The solutions include pre-charging internal node, delaying the clock signal, increasing the evaluation node capacitance and adding an feedback transistor. After the introduction part, a new cascode voltage switch logic synthesis method is proposed with examples demonstrating the procedures. The procedures are summarized into 3 reduction rules. An alternate method of realizing cascode voltage switch logic is proposed. It is a semi-custom design method as opposed to the full custom design required for the new synthesis procedure. Only a few connections are needed to program a semi-custom cell. The cell uses almost one half the number of the transistors used in a full tree implementation. By utilizing the new cascode voltage switch logic synthesis method, a new multiple output domino logic synthesis technique is derived. A companion algorithm for handling "don't care cases" is also proposed. Its underlying concept and procedures are explained. The WINSYN synthesizer requires a planar switching tree as input, therefore a new planar transformer is proposed with examples. At the end of the chapter for switching tree, a synthesizer designed for tree reduction is presented.

CHAPTER 4

IMPLEMENTATION OF DIVIDER

4.1 CONVENTIONAL DIVISION METHOD

The mathematical division operation is similar to the multiplication operation in the use of the iteration operator. Multiplication algorithms iterate the use of addition, whereas division algorithms use subtraction algorithm instead. Most division algorithms are either modification or extensions of restoring and non-restoring division methods[22]. The restoring algorithm is another name for the pencil and paper division method that is usually taught in grade school. The procedure involves repeated subtraction of a shifted divisor from the dividend to find the digits of the quotient. An example is given below to show the procedure.

$$\frac{6.936}{3} = 2.312$$

The actual steps are decomposed as below

$$\begin{array}{r} 6\,9\,3\,6 \\ \underline{-3\,0\,0\,0} \\ 3\,9\,3\,6 \\ \underline{-3\,0\,0\,0} \\ 9\,3\,6 \\ \underline{-3\,0\,0\,0} \\ -2\,0\,6\,4 \end{array}$$

Three subtractions are needed to give a negative remainder. Two subtractions are the maximum number of subtraction to give a positive remainder. Therefore the first digit is 2. Before the division is continued, the negative remainder is restored to positive value by adding the shifted divisor. The word "restoring" of "restoring division method" comes from this step of restoring the remainder to a positive value.

```
 -2064
 +3000
 -----
   936
```

Subtractions are repeatedly applied to the remainder until a negative remainder appears.

```
   936
  -300
  ----
   636
  -300
  ----
   336
  -300
  ----
    36
  -300
  ----
  -264
```

Three is the maximum number of subtractions to give positive remainder. Therefore the second digits is 3. After restoring the remainder to a positive value, the process continues as before. Finally, the list of digits will be 2312 which are the digits for the quotient.

In the non-restoring division method, both subtraction and addition are used. The subtraction is used first to give a negative remainder. The number of subtractions becomes the first digit. Negative remainder is multiplied by the radix which is 10 for the decimal system. The addition operations are repeatedly applied to this remainder until a positive value returns. The number of additions becomes the second digit with an overbar indicating that it is a negative digit. The process continues with subtraction and addition until the remainder is zero or the number of required digit is fulfilled. An example below illustrate the process

```
  6936
 -3000
 -----
  3936
 -3000
 -----
   936
 -3000
 -----
 -2064
```

First digit is 3

```
                     -20640
                   +   3000
                     -17640
                   +   3000
                     -14640
                   +   3000
                     -11640
                   +   3000
                      -8640
                   +   3000
                      -5640
                   +   3000
  -2064               -2640
x    10            +   3000
 -20640                 360
```

Second digit is $\bar{7}$

```
                      3600
                     -3000
   360                 600
  x 10               -3000
  3600               -2400
```

Third digit is 2

```
                     -24000
                   +   3000
                         ..
                         ..
  -2400                  ..
x    10                  ..
 -24000                   0
```

Fourth digit is $\bar{8}$

The answer is $3.\bar{7}2\bar{8}$

This pseudo-quotient can be easily converted to the normal quotient by separating the positive part and negative part

$$\begin{tikzcd}
 & \text{Positive part} \quad 3.020 \\
3.\bar{7}2\bar{8} \arrow[ur] \arrow[dr] & \\
 & \text{Negative part} \quad 0.708
\end{tikzcd}$$

By subtracting the negative part from the positive part , the quotient can be derived.

```
 3.020
-0.708
 2.312
```

The sign of the digit depends on whether addition or subtraction is used. Adding or subtracting depends on the sign of remainder. The table below summarizes the relationship.

Addition	->	negative digit
Subtraction	->	positive digit
positive remainder	->	subtraction
negative remainder	->	addition

If (r-1) subtractions are used without changing the remainder sign then (r-1) is recorded as the quotient digit.

Table 4.1: Rules for restoring division

For a binary pseudo-quotient, there is a simple algorithm for converting to normal signed digit representation. The table below shows the steps involved.

Step 1	Convert all $\bar{1}$ to 0
Step 2	Shift 1 bit to the left and a "1" is shifted into the LSB
Step 3	Complement the MSB

Example	$1\bar{1}11\bar{1}$	
	After step 1	10110
	After step 2	101101
	After step 3	001101

Table 4.2: Steps for converting binary pseudo quotient to normal signed digit

4.2 SRT DIVISION

SRT division[4][5] is originated from the three initial proposers. They are D. W. Sweeney of IBM, J. E. Robertson of the University of illinois and K. D. Tocher of imperial College. Their last names become the name of the division method. This division method uses a redundant digit set for quotient bits and they can be determined from an estimation of divisor bit and remainder bits. Not all of the bits are required at all times for quotient digit determination.

The result of a division algorithm is a list of quotient digits representing the quotients Q

$$Q = \sum_{i=0}^{n-1} q_i r^{-i} \qquad (4.1)$$

r	radix
Q	quotients value
qi	quotient digit
n	number of quotient digit

For normal irredundant division with quotient digit set (0....,r-1), a quotient value Q can only be represented by a unique combination of quotient digits. To correctly determine the partial remainder at each computational stage, the entire previous partial remainder has to add or subtract to or from the divisor. This addition or subtraction operation takes time proportional to the number of partial remainder digits. By using an irredundant quotient digit set, quotient digits can be determined by inspecting a portion of partial remainder digit and divisor digits. This works because a single quotient value can be represented with a small sequence of irredundant quotient digits. Previous inaccuracies generated can be compensated with subsequent quotient digits. The partial remainder, quotient digit and previous partial remainder have the following relationships:

$$R_{i+1} = r R_i - D q_i \qquad (4.2)$$

qi: quotient digit
Ri: partial reminder of stage i
D: radix
r: radix

The dividend is initialized with rR0. Redundant quotients are in the set {-p,..0..p} where p is restricted to be $\frac{r}{2} \leq p \leq r-1$ according to reference [23]. By inspecting a few bits of partial remainder and divisor, the quotient digit can be chosen. However, there are restrictions to the number of bits that should be examined. Three diagrams are used to determine the number of bits used for inspection. They are the Robertson diagram, Taylor diagram and Colored Taylor diagram. Figure 4.1 show a Robertson diagram for the quotient digit set {-1,0,1} of radix 2 division.

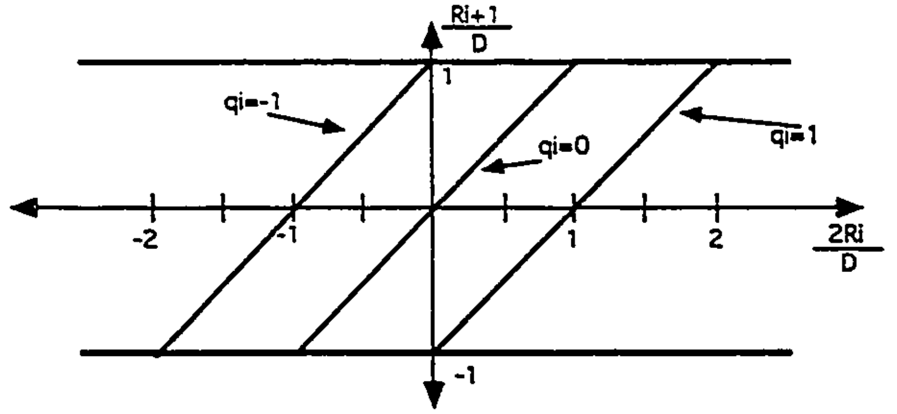


Figure 4.1 Robertson diagram for quotient digit set {-1,0,1} of radix 2 division

From equation 4.2

$$R_{i+1} = r R_i - D q_i$$

This equation is divided by D

$$\frac{R_{i+1}}{D} = r \frac{R_i}{D} - q_i \tag{4.3}$$

$\frac{R_{i+1}}{D}$ is the y axis of the Robertson diagram.

For radix 2, r=2, $\frac{2R_i}{D}$ is the x axis of the Robertson diagram. The lines are different possible quotient digits. The boundary for the y axis is ±1 computed from equation

$$\left|\frac{R_{i+1}}{D}\right| \le \frac{P}{r-1} \tag{4.4}$$

according to reference [23]. For example, if $\frac{2R_i}{D} = 0.5$, there are two possible choices of quotient q_i=0 and q_i=1. If $\frac{2R_i}{D} = 1.5$, there is only one possible choice which is q_i=1

The Taylor diagram is derived from the Robertson diagram by considering divisors and dividends that are both normalized to the range 1 to 2. The graph is shaded to show different quotient digit validation regions. The region shaded by more than one quotient digit pattern are valid for both quotient digits. Therefore, more than one choice is possible in those overlapping regions. Figure 4.2 shows a Taylor diagram for radix 2.

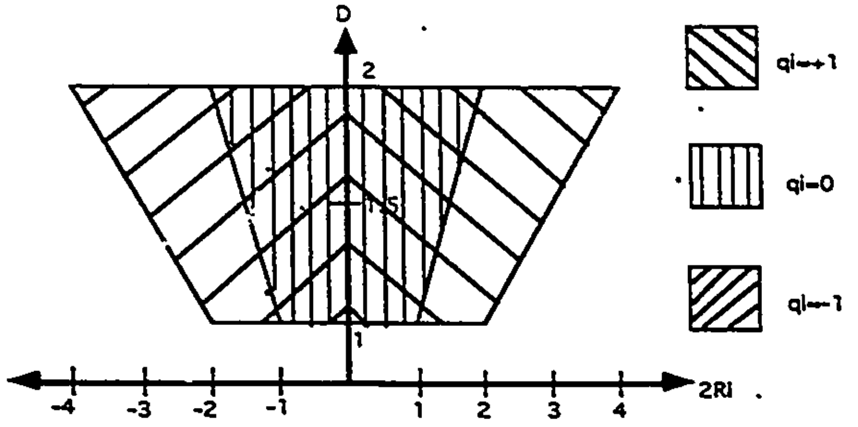


Figure 4.2 Taylor diagram for quotient digit set {-1,0,1} of radix 2 division

The colored Taylor diagram is constructed by painting with a rectangular paint brush on the Taylor diagram. Each rectangular paint applied to the diagram must fit wholly inside a valid quotient digit region. The vertical size of the paint brush is the maximum value of the unexamined bits of the divisor, the horizontal size is the sum of the maximum value of the unexamined bits of the partial reminder. The grid spacing for the paint brush is the tolerances of the divisor and partial remainder. The following equations describe the sizes for the painr brush and grid spacing.

grid height = $2^{(\text{-DivisorBits})}$
brush height = $2^{(\text{-DivisorBits})}$
LeftBits = $2+\left\lceil \log_2\left(\frac{pr}{r-1}\right)\right\rceil$
grid width = $2^{(\text{LeftBits-RBits})}$
brush width = $2^{(\text{LeftBits-RBits})}+2^{(\text{LeftBits-CBits})}$
where

r:	radix
p:	the maximum quotient digit in the quotient digit set {-p,. . .,0,. . .,p)
LfetBits	the number of partial remainder bits to the left of the binary point
DivisorBits	the number of divisor bits examined
RBits	the number of partial remainder sum bits examined
CBits	the number of partial remainder carry bits examined

The largest the paint brush can be, the smaller the number of bits is needed to be examined. It is found that no divisor bit should be examined and 3 bits of the partial remainder should be examined as shown in figure 4.3

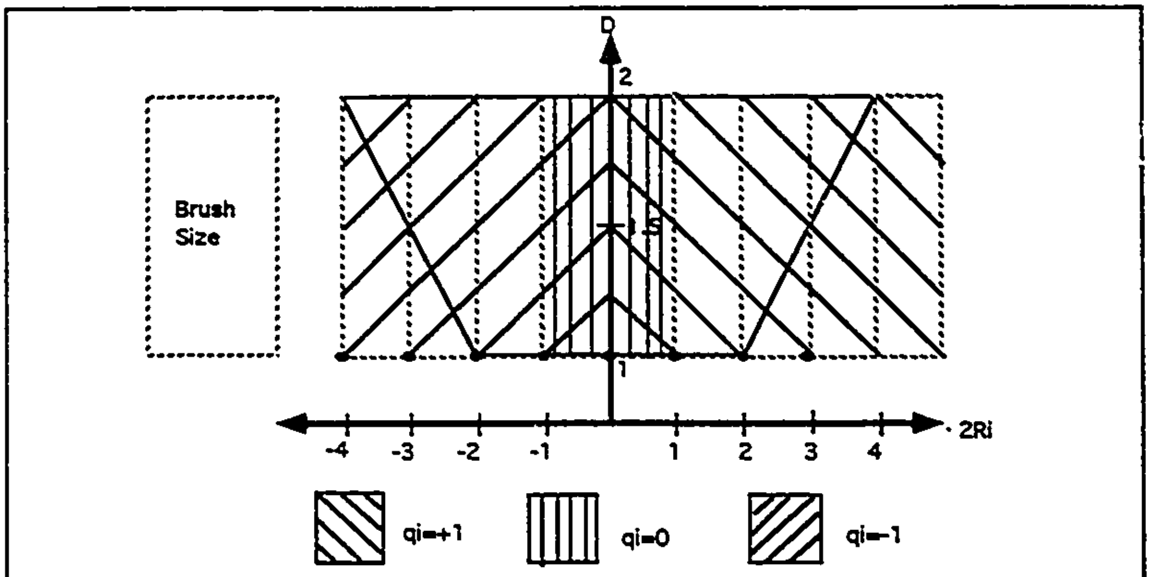


Figure 4.3 Colored Taylor diagram for quotient digit set {-1,0,1} of radix 2 division

4.3 SYSTEM LEVEL ORGANIZATION

Each step of SRT division returns a quotient digit and a partial remainder as results. This partial remainder is fed to next step for computing the next quotient digit and the next partial remainder. This iterative computation algorithm is suitable for homogenous pipeline implementation. In order to minimize the delay time between each quotient result, a self-timed approach is adopted for implementation; the self-timed pipeline can be constructed to meet the "zero-overhead" constraints. The entire computation delay is equivalent to the pure combinational delay of processing blocks. Figure 4.4 shows a block diagram of the implementation of the self-timed divider. There are five stages of processing blocks for the iterative procedure. These five stages are cascaded together to form a ring structure. Each stage has a quotient shift register attached. These registers are constructed using C elements and NOR gates. The remainder register and the compare block are designed for early termination of iteration. If the remainder has remained unchange after one loop of iteration, the subsequent quotient digits will repeat the same pattern again. The early termination of iteration will save a great deal of computation time.

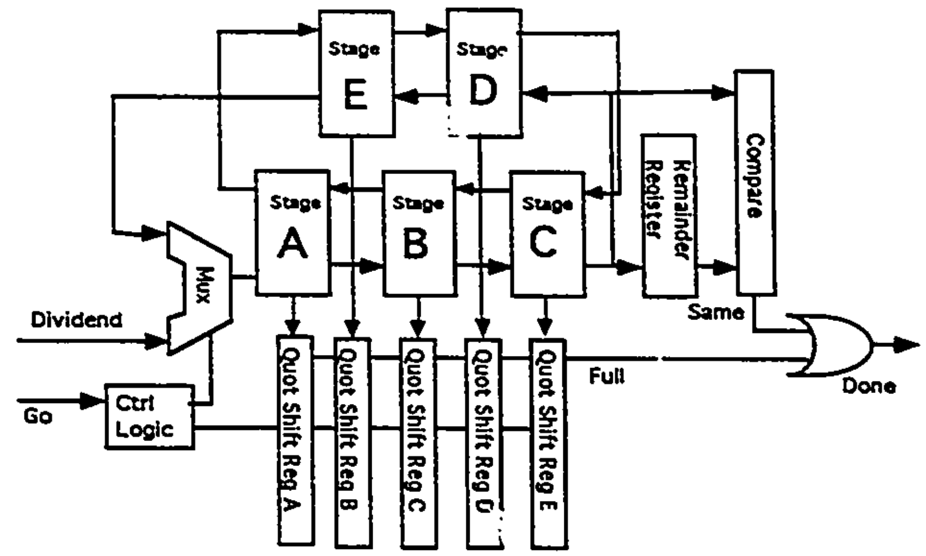


Figure 4.4 Complete block diagram for the self-timed divider

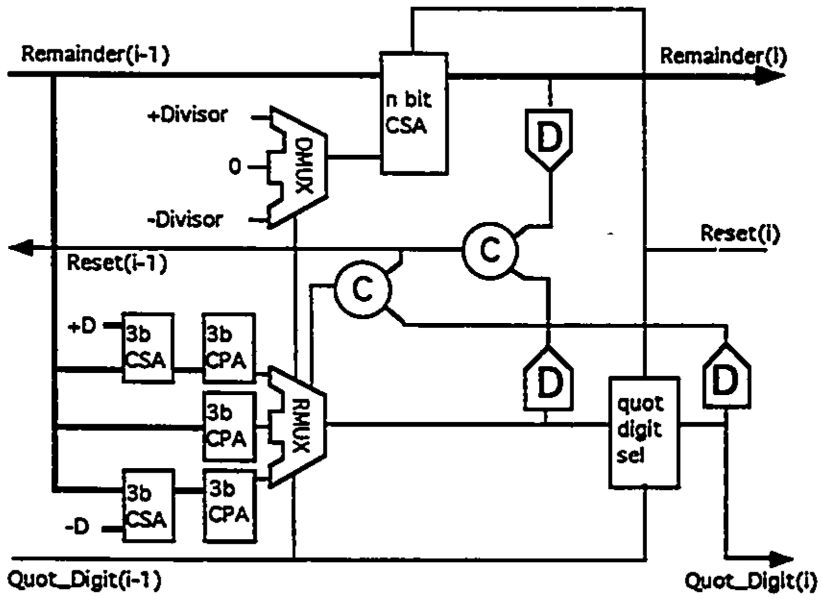


Figure 4.5: The block diagram for one stage of function block

Figure 4.5 shows the block diagram for each stage. The n-bit CSA calculates the current partial remainder from the previous partial remainder and the selected divisor. The selection of divisor is achieved through the use of a multiplexer, controlled by the previous quotient digit. The group of blocks consisting of 3-bit CSAs and 3-bit CPAs is designed for speeding up the computation. The quotient digit selection block needs the current partial remainder for input. However, the current partial remainder is computed if both previous partial remainder and previous quotient digit are available. The group of CSAs and CPAs pre-calculates the 3 most significant bits of current partial remainders. Three possible results are waiting at the input of the multiplexer. As the previous quotient digit becomes available, the selected current partial remainder will be fed into the quotient digit selection. This pre-calculation of partial remainders speeds up the overall computation time.

The quotient digit selection block selects the quotient digit according to the colored-Taylor diagram in figure 4.3. Table 4.3 summarizes the selections

Remainder	-4 -3 -2	-1	0 1 2 3
Quotient digit	-1	0	1

Table 4.3

It is found that when the most negative remainder approximation occurs, the current and the next quotient digit stays at the same value, therefore a force-ahead is implemented. The following equations describe this modified selection scheme.

$q_i=1$, if $p_i>0$ and $F_{i-1}=0$
$q_i=0$, if $p_i=-1$ and $F_{i-1}=0$
$q_i=-1$, if $p_i\leq-2$ or $F_{i-1}=1$
$F_i=1$ if $P_i=-4$

4.4 CIRCUITS AND SIMULATION

All function blocks used in the implementation of the divider are synthesized with the new design procedure. The charge sharing problem is solved by pre-charging selected internal nodes. The selections are verified by spice simulation. Worst case charge sharing conditions are also verified by spice simulation. The functional correctness of the algorithm is verified using the PASCAL language.

All of the circuits are designed using the synthesis procedure in the same manner. An example is given for illustrative purpose. Figure 4.6 shows a schematic for the sum bit of the full adder. Two additional PMOS transistors are used for pre-charging the selected internal nodes. Figure 4.7 shows the mask layout of this schematic.

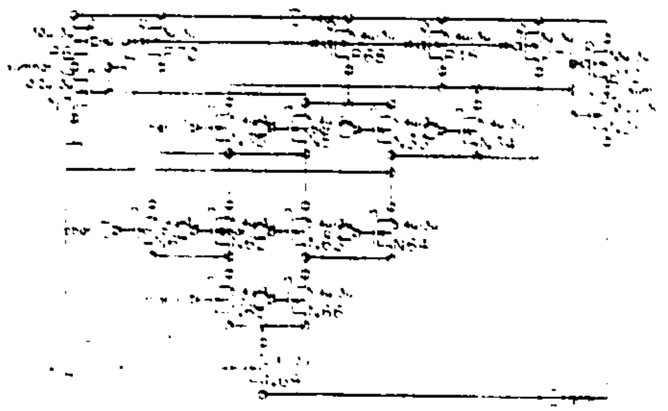


Figure 4.6: Schematic diagram for the sum bit of the full adder

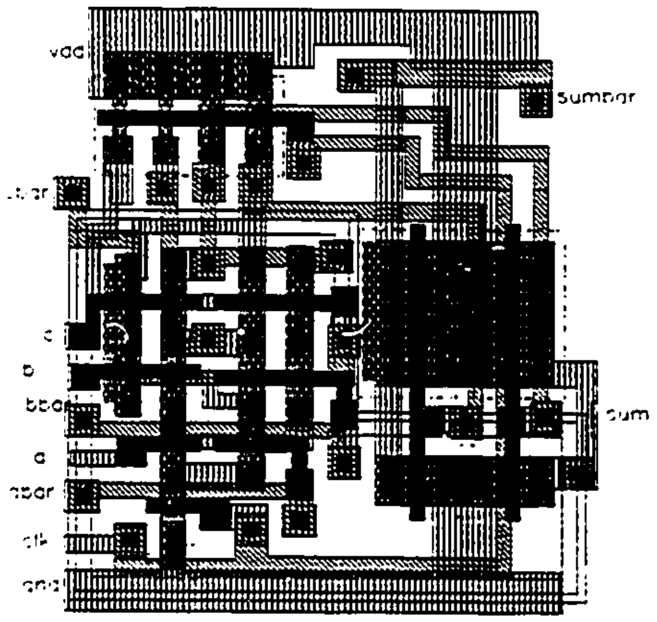


Figure 4.7: Mask layout for the sum bit of the full adder

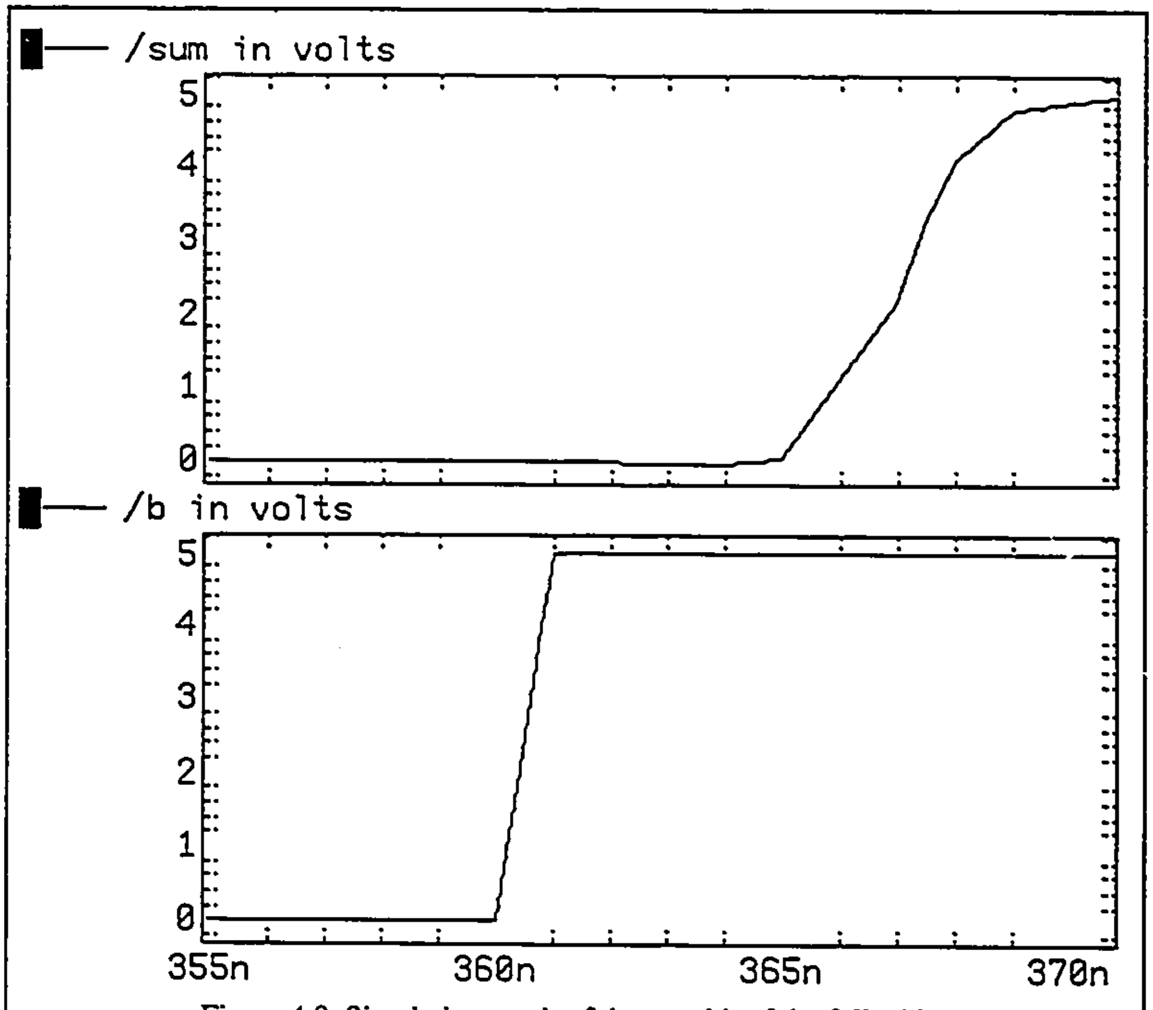


Figure 4.8: Simulation result of the sum bit of the full adder

Figure 4.8 shows the worst case delay HSPICE[25] simulation of the sum bit of the full adder. The worst case delay is 8ns. Since the next computation block will start to evaluate as soon as the output of this block becomes valid, the actual delay time is much shorter than the worst case delay.

Another interesting variation of the synthesis procedure is demonstrated in the synthesis of the quotient selection logic. Table 4.4 shows the truth table of the quotient selection logic. Observing the truth table, it is found that the output columns are mutually exclusive. No two outputs will be "1" at the same time. Therefore, the truth table can be modified as shown in table 4.5. The cascode voltage switch logic synthesis algorithm is applied to this table. The reduced tree is synthesized as shown in figure 4.8

F	P2	P1	p0	q1	q-1	q0
0	0	0	0	1	0	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	0	1	0
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	0	1	0

Table 4.4: Truth table for the quotient selection logic

F	P2	P1	p0	q
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	-1
0	1	0	1	-1
0	1	1	0	-1
0	1	1	1	0
1	0	0	0	-1
1	0	0	1	-1
1	0	1	0	-1
1	0	1	1	-1
1	1	0	0	-1
1	1	0	1	-1
1	1	1	0	-1
1	1	1	1	-1

Table 4.5 Modified truth table of table 4.1

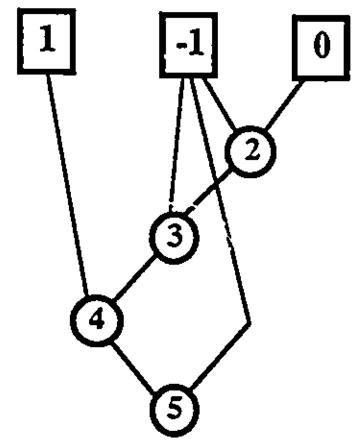


Figure 4.9 Reduced Tree for quotient selection logic

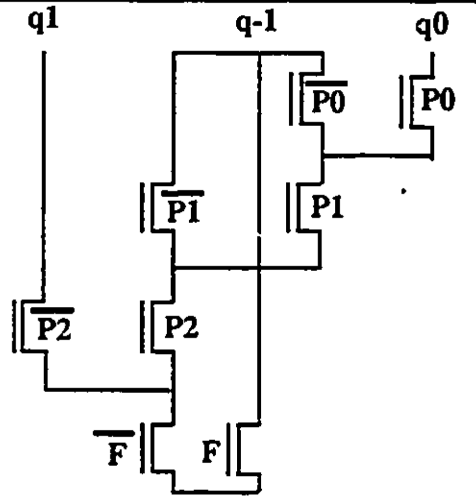


Figure 4.10 Schematic derived from the reduced tree

The schematic is shown in figure 4.9. The rest of the circuits are included in Appendix B

4.5 SUMMARY

This chapter describes the implementation of the divider. It starts with an introduction to the restoring division method. Its disadvantage is overcome by the non restoring division method. A few examples are given to illustrate the procedures. SRT division method is introduced with Robertson diagram, Taylor diagram and Colored Taylor diagram. After all of the basic algorithms are explained, the block level diagrams are presented to show the dataflow and the modification. The circuit part is introduced with the sum bit generation block and its mask layout. Quotient selection logic generation block is synthesized with varied CVSL synthesis method.

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

5.1 CONCLUSION

This thesis is emerged from the study of self-timed pipeline. Different configurations of self-timed pipeline are studied. Communication methods are also studied. Different configurations are evaluated by their performance. It includes the throughput and latency. Their performance are determined by using the data dependency graph and performance graph. By assuming different sizes of function block, the range of performance and no of stages can be estimated. After the study of performance measuring, some methods for improving the performance are investigated. In the study of the function block implementation, cascode voltage switch logic is found to be the choice for realizing the function block. A previous design method is investigated and it is found to be inefficient. Therefore, a research effort is initiated to create a more efficient method. Previous work in the Boolean function manipulation[16] is applied to reduce the transistor trees. This reduction technique together with the new interpretation of the tree node forms a new cascode voltage switch logic design method. After many circuit designs are synthesized by using this method, the bottom row of the reduced tree is found to be narrow in shape and each node in this bottom row falls only into one of the four possible types of nodes. This observation is made use in the realization of a new type of semi-custom cell to reduce the number of transistors by half. Further research into this synthesis method leads to a new multiple output domino logic design method which is only a slight modification of the new CVSL synthesis method. The creation of "Don't care case" algorithm is the result of an investigation of further reducing the transistor count in the Module 7 multiplier implementation. The problem of mask layout of this reduced tree by using a new layout style(WINSYN)[19] becomes the next target of the research effort. The result of this effort is a new algorithm created to transform a non-planar switching tree into a planar one. All algorithms proposed in this thesis is automated by a program written in PROGRAPH[20] and C[18]. To demonstrate the use of the new algorithms, a SRT divider designed and fabricated in CMOS 3μm technology.

5.2 FUTURE DIRECTIONS

5.2.1 LOGIC PARTITION

As discussed in Chapter 1, the switching tree is an effort to combine a few levels of design hierarchy into one level. Within the level of switching tree design, the whole system is partitioned into switching tree blocks and latches. This partition is usually intuitively handled by the designer. Figure 5.1 shows two partition schemes of the same system.

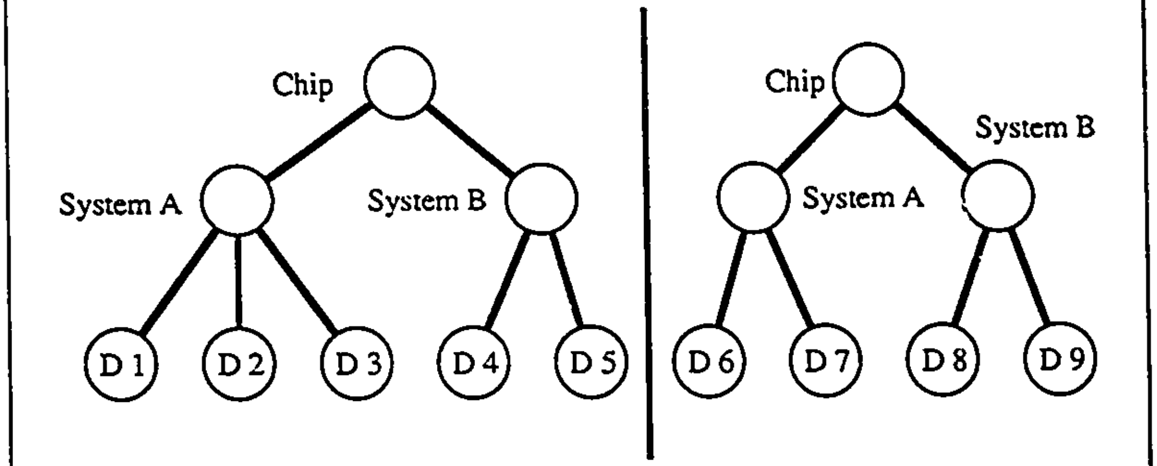


Figure 5.1: Two partition schemes of the same system

The switching tree levels can be partitioned into D_1 to D_5 or D_6 to D_9. Different ways of system partitioning will lead to different performances and costs. An optimal computer generated solution to satisfy the requirements is better than an intuitive designer's solution. A similar effort fcr gate level design has been accomplished[24]. A future research direction for the switching tree may target to the logic partition problem of the switching tree.

5.2.2 IMPROVEMENTS TO SWITCHING TREE SYNTHESIS

More transistors can be reduced by replacing a transistor with a wire. This is possible because the reduction technique introduced previously generates schematic with only single conducting path between the evaluation and the bottom node. Multiple conducting paths are allowed in the schematic design as long as the Boolean function is not changed. An example is already shown in figure 3.31. Two more transistors are eliminated in this circuit by considering multiple conducting paths. The planar switching tree can be more optimized by considering more levels at a time. The method introduced ir chapter 3 considers only one level.

REFERENCES

[1] D. M. Chapiro, "Globally-asynchronous, locally-synchronous systems," Ph.D. Thesis, Stanford University, 1984.

[2] C. Mead, L. Conway, Introduction to VLSI systems, Addison-Wesley: Reading MA, 1980.

[3] G.A. Jullien, W.C. Miller, R. Grondin, Z. Wang, D. Zhang, L. Del Pup, S. Bizzan, 1992, "WoodChuck: A Low-Level Synthesizer for Dynamic Pipelined DSP Arithmetic Logic Blocks." Proceedings of the 1992 IEEE Int. Symp. on Circuits and Systems, (***Invited***), 1, pp. 176-179.

[4] J. E. Robertson, "A New Class of Digital Division Methods," IRE trans. Electronic Computers, vol. EC-7, pp. 218-222, September 1958.

[5] T. D. Tochner, "techniques of Multiplication and Division for Automatic Binary Computers," Quarter J. Mech App. Math., vol. 2, pt. 3, pp. 364-384, 1958.

[6] I. S. Hwang, A. L. Fisher, "Ultra Compact 32-bit CMOS Adders in Multiple-Output Domino Logic," IEEE J. of Solid-State Circuits, vol. 24, no 2, April 1989.

[7] T. E. Williams, "Self-Timed Rings and Their Application to Division," Technical report: CSL-TR-91-482, Stanford University, 1991.

[8] R. E. Miller, Switching Theory. New York: Wiley, 1965.

[9] L. Sutherland, "Micropiplines," Communications of the ACM, vol. 32 no. 6, pp. 720-738, June 1989.

[10] M. Dean, T. Williams, D. Dill, "Efficient Self-timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," Proceedings of the Santa Cruz Conference on Advanced Research in VLSI pp. 55-70, March 1991.

[11] T. A. Chu, "Synthesis of Self-timed Control circuits from Graphs: An example," Proceedings of ICCD, pp. 565-571, Oct. 1986.

[12] N. Weste, K. Eshraghian, Principles of CMOS VLSI DESIGN: A Systems Perspective, Addison-Wesley:Reading MA, 1985.

[13] J. A. Pretorius et. al., "Latched Domino CMOS Logic," IEEE J.S.S.C, pp. 514-512, Jan. 1986.

[14] L. G. Heller, W. R. Griffin, "Cascode voltage switch logic: A differential CMOS logic family," in ISSCC dig. tech. Papers, 1984, pp16-17.

[15] M. K. Chu, D. I. Pulfrey. "Design procedures for differential cascode-voltage switch circuits, " IEEE Trans. Solid-State Circuits. vol. SC-21, pp. 1082-1087, 1986.

[16] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEE Trans. on Comp. vol. 35, pp. 677-691, 1986.

[17] C. E. Shannon, "Symbolic analysis of relay and switching circuits," Trans. Am. Inst. elect. Engrs. 57, 713-723, 1938.

[18] B. W. Kernighan, D. M. Ritchie, The C Programming Language, Prentice Hall, 1988

[19] G. A. Jullien, "Seminar on Windsor Synthesizer", Faculty of Engineering, Department of Electrical Engineering, University of Windsor, May 1992.

[20] The Gunakara Sun Systems , Ltd, Prograph Reference Manual, Halifax, Nova Scotia, Canada, 1990.

[21] M. Y. Wu, W. Shu and S. P. Chan, "A unified theory for MOS circuit design-Switching network logic," Int. J. Electron., vol. 58, no.1, 1-22, 1985.

[22] N. R. Scott, Computer Number Systems and Arithmetic, Prentice Hall, 1985.

[23] T. E. Williams, M. Horowitz, "SRT Division Diagrams and Their Usage in Designing Custom Integrated Circuits for Division," technical report csl-tr-87-326, Stanford, Nov. 1986.

[24] R. Brayton et al., Logic Minimization Algorithms for VLSI Synthesis. Hingham, MA:Kluwer, 1984.

[25] Meta-Software, Inc. HSPICE User's Manual H9001, 1990.

APPENDIX A

C PROGRAM CODE FOR THE SYNTHESIZER

```
#include <stdlib.h>

#define ONE 1
#define ZERO 0
#define DONT 2
#define ERROR -1
#define FINISH -1
#define MERGE 1
#define NONMERGE 0
#define YES 1
#define NO 0
#define MAXY 40
#define MAXPATLEN 40
#define MAXLISTOFLOGIC 50


struct logic {
int x;
int y;
int **table;
int **pattern;
};


struct config {
int top;
int left[100];
int right[100];
int key[100];
};

struct truthtable {
int index;
int *table[7];
};


struct package {
struct config **aconfig;
int noofconfig;
struct logic *alogic;
int seq[10];
int lastseq;
int head[10];
int bestcount;
};

struct logic *logicinput;
struct package *packageoutput;
int *copyvector(int,int *);
int bind(int ,int);
struct logic *copylogic(struct logic *);
void checkonezero(struct logic *);
```

```
int *bindvector(int ,int *,int *);
void pushvector(int *,int *,struct logic *);
struct logic *gencom(struct logic *);
int samepat(int *,int *);
int *bindpattern(int *,int *);
int *copypattern(int *);

void *newmalloc(size_t size) {
void *res;
if ((res=malloc(size))==NULL) { exit(1);}
return res;
}


struct logic *initlogic(int x) {

struct logic *res;
res=newmalloc(sizeof(struct logic));
res->y=-1;
res->x=x;
res->table=newmalloc(sizeof(int *)*MAXY);
res->pattern=newmalloc(sizeof(int *)*MAXY);
return res;
}

int power(int x,int y) {
int res,i;
res=1;
for (i=0;i<y;++i)
res*=x;
return res;
}

int bind(int in1,int in2) {

if (in1==in2) return in1;
else
 if (in1==DONT) return in2;
else
 if (in2==DONT) return in1;
else
 return ERROR;
}



void checkonezero(struct logic *logicv) {
int lenght,height,before,x,y;

lenght=logicv->x;
height=logicv->y;
 for (y=0;y<=height;++y) {
    before=DONT;
```

```
   for (x=0;x<=lenght;++x) {
         before=bind(before,logicv->table[y][x]);
         if (before==ERROR) break;
   }
   if (before!=ERROR) {
        if (before==DONT) before=ZERO;
        for (x=0;x<=lenght;++x)
           logicv->table[y][x]=before;
   }
 }
}

int *bindvector(int x,int *vector1,int *vector2) {
int *res,i;
res=newmalloc((x+1)*sizeof(int));
for (i=0; i<=x;++i)
{res[i]=bind(vector1[i],vector2[i]);
if (res[i]==ERROR) {free(res);res=NULL;break;}
}

return(res);
}


int *bindpattern(int *pat1,int *pat2)
{
int nextstart,i,j,exist,temp[MAXPATLEN],*res;
i=0;
while (pat1[i]!=FINISH) {
   temp[i]=pat1[i];
   i++;
   }
nextstart=i;

i=0;
while (pat2[i]!=FINISH) {
  exist=0;
  for (j=0;j<nextstart;++j)
     if (temp[j]==pat2[i]) {exist=1;break;}
  if (exist==0) {temp[nextstart]=pat2[i];++nextstart;}
  ++i;
 }
res=newmalloc(sizeof(int)*(nextstart+1));
for (j=0;j<nextstart;++j)
res[j]=temp[j];
res[nextstart]=FINISH;
return res;
}

int samepat(int *pat1,int *pat2) {
int i,j,count2;
i=0;
count2=0;
while (pat1[i]!=-1) {
```

```
j=0;
while (pat2[j]!=-1) {
if (pat1[i]==pat2[j]) ++count2;
++j;
}

++i;
}
if ((count2==j)&&(count2==i)) return 1; else return 0;

}


void pushvector(int *pat,int *vect,struct logic *logicout) {


int y,exist,i;
y=logicout->y;
exist=0;
for (i=0;i<=y;++i)
if (samepat(logicout->pattern[i],pat)) {exist=1;break;}
if (exist==0) {
y=++logicout->y;
logicout->table[y]=vect;
logicout->pattern[y]=pat;
}
else {free(pat); free(vect);}

}

int *collect(int *collectpat,int *pat) {
int *res;
res=bindpattern(collectpat,pat);
free(collectpat);
return res;
}


int equalvector(int last,int *vec1,int *vec2) {
int i;
for (i=0;i<=last;++i) if (vec1[i]!=vec2[i]) return(NO);
return(YES);
}

struct logic *gencom(struct logic *logicin) {
struct logic *logicout;
int start,endpoint,i,*collectpat,*pat;
int *tempvector,visited[MAXY],*samevector;
for (i=0;i<=logicin->y;++i) visited[i]=NO;

logicout=NULL;
for (start=0;start<logicin->y;++start) {
 if (visited[start]==NO) {
 samevector=NULL;
 for (endpoint=start+1;endpoint<=logicin->y;++endpoint)
   if (visited[endpoint]==NO) {
     if (equalvector(logicin->x,logicin->table[start],logicin->table[endpoint])) {
     if (samevector==NULL) {collectpat=copypattern(logicin->pattern[start]);
                    samevector=copyvector(logicin->x,
                                   logicin->table[start]);}
        visited[endpoint]=YES;
     collectpat=collect(collectpat,logicin->pattern[endpoint]);}
     else {
     tempvector=bindvector(logicin->x,logicin->table[start],
          logicin->table[endpoint]);
               if (tempvector!=NULL) {
     if (logicout==NULL) logicout=initlogic(logicin->x);
     pat=bindpattern(logicin->pattern[start],logicin->pattern[endpoint]);
     pushvector(pat,tempvector,logicout);
     }
     }}
if (samevector!=NULL) {if (logicout==NULL) logicout=initlogic(logicin->x);
               pushvector(collectpat,samevector,logicout);}}
}
return logicout;
}


struct logic**createlistoflogic(struct logic*logicin) {
int i,j;
struct logic *templogic[MAXLISTOFLOGIC],**temp;
checkonezero(logicin);
i=0;
templogic[i]=logicin;

while ((templogic[i+1]=gencom(templogic[i]))!=NULL)
{
 ++i;}
temp=newmalloc(sizeof(struct logic*)*(i+2));
for (j=0;j<=i+1;++j) temp[j]=templogic[j];
return(temp);
}



int matchcount(int *large,int *small) {
int res,i,j,exist;
res=0;i=0;
while (small[i]!=FINISH) {

j=0;exist=0;
while (large[j]!=FINISH) {
if (small[i]==large[j]) {exist=1;break;}
++j;}

if (exist==1) ++res; else break;
++i;
}
```

```
if (exist==0) return 0; else return res;
}



int *sub(int *large,int *small) {
int co,*res,i,j,exist,temp[240];
co=0;i=0;
while (large[i]!=FINISH) {

j=0;exist=0;
while (small[j]!=FINISH) {
if (small[j]==large[i]) {exist=1;break;}
++j;}

if (exist==0) temp[co++]=large[i];
++i;
}
res=newmalloc((co+1)*sizeof(int));
for (i=0; i<co;++i) res[i]=temp[i];
res[co]=FINISH;
return res;

}




int *copypattern(int *head) {
int i,j,*res;
for (i=0;head[i]!=FINISH;++i);
res=newmalloc(sizeof(int)*(i+1));
j=i;
for (i=0;j>=i;++i)
res[i]=head[i];
return res;
}

int *copyvector(int lastelement,int *head) {
int i,*res;
res=newmalloc(sizeof(int)*(lastelement+1));
for (i=0;i<=lastelement;++i) res[i]=head[i];
return res;
}


int *pickbest(struct logic *logicin,int *vector,struct logic* res) {
int en,i,max,temp,best, *newvector;
en=logicin->y;
newvector=vector;
while (*vector!=FINISH) {
i=0;
max=0;
for (i=0;i<=en;++i) {
```

```
temp=matchcount(vector,logicin->pattern[i]);
if (temp>max) {max=temp;best=i;}
 }
if (max==0) break;
newvector=sub(vector,logicin->pattern[best]);
free(vector);
vector=newvector;
++res->y;
res->pattern[res->y]=copypattern(logicin->pattern[best]);
res->table[res->y]=copyvector(logicin->x,logicin->table[best]);

}
return vector;
}


struct logic *choose(int *vector,struct logic **logiclist) {
struct logic *res;
int i;

res=initlogic(logiclist[0]->x);
i=0;
while (logiclist[i]!=NULL) ++i;
--i;

while (*vector!=FINISH) {

vector=pickbest(logiclist[i],vector,res);
--i;

}
free(vector);
return res;

}

void freelogic(struct logic *logicin) {
int i;
for (i=0;i<=logicin->y;++i) {
free(logicin->table[i]);
free(logicin->pattern[i]);
}
free(logicin->table);
free(logicin->pattern);
free(logicin);
}

void freelistoflogic(struct logic **logicin) {
int i;
for (i=0;logicin[i]!=NULL;++i)
freelogic(logicin[i]);
free(logicin);
}
```

```
struct logic *bindlogic(int *vector,struct logic *test) {
struct logic **listoflogic;
struct logic *bestlogic,*test1;
int *tempvector,i;
test1=test;
tempvector=copypattern(vector);
listoflogic=createlistoflogic(test1);
/*for (i=0;listoflogic[i]!=NULL;++i)
 printlogic(listoflogic[i]);*/
bestlogic=choose(tempvector,listoflogic);
/*printlogic(bestlogic);*/
freelistoflogic(listoflogic);
/*printlogic(bestlogic);*/
return bestlogic;
}

int allfixed(struct logic *logicin) {
int i,j;
for(i=0;i<=logicin->y;++i)
for(j=0;j<=logicin->x;++j)
if (logicin->table[i][j]==DONT) return(0);
return(1);
}
int *copyvectorpart(int start,int endpoint,int *vector) {
int *res,i,j;
res=newmalloc(sizeof(int)*(endpoint-start+1));
for (i=start,j=0;i<=endpoint;++i,++j) res[j]=vector[i];
return res;
}

struct logic *rebind(struct logic *prebind,struct logic *orglogic) {
int i,j,k,loc,realloc;
struct logic *res;
res=initlogic(orglogic->x);
for (i=0;i<=orglogic->y;++i) {
res->pattern[i]=copypattern(orglogic->pattern[i]);
res->table[i]=newmalloc(sizeof(int)*(orglogic->x+1));
}
for (i=0;i<=prebind->y;++i)
for (j=0;(loc=prebind->pattern[i][j])!=FINISH;++j) {
realloc=loc/2;
if (loc-2*realloc==1) {
for (k=0;k<=prebind->x;++k)
res->table[realloc][k+1+prebind->x]=prebind->table[i][k];
}
else {
for (k=0;k<=prebind->x;++k)
res->table[realloc][k]=prebind->table[i][k];
}
}
res->y=orglogic->y;
return res;
```

```
}

struct logic *sprit(struct logic *logicin) {
int tempat[2],i,ii;
struct logic *logicout;
tempat[1]=FINISH;
logicout=initlogic((logicin->x+1)/2-1);
for (i=0,ii=0;i<=logicin->y;++i,ii+=2) {
tempat[0]=ii;
logicout->pattern[ii]=copypattern(tempat);
tempat[0]=ii+1;
logicout->pattern[ii+1]=copypattern(tempat);
logicout->table[ii]=copyvectorpart(0,logicout->x,logicin->table[i]);
logicout->table[ii+1]=copyvectorpart(logicout->x+1,logicin->x,logicin->table[i]);
}
logicout->y=2*(logicin->y+1)-1;
return(logicout);
}

int *newvector(struct logic *logicin) {
int *res,y,i;
y=logicin->y;
res=newmalloc(sizeof(int)*(y+2));
for (i=0;i<=y;++i)
res[i]=i;
res[y+1]=FINISH;
return res;
}

struct logic *dontcare(int *vector,struct logic *logicin) {
struct logic *logicnew,*aftersprit,*tobebind,*logicout,*logicin1;
int *newvec;
logicin1=copylogic(logicin);
logicnew=bindlogic(vector,logicin1);
if (allfixed(logicnew))
   return(logicnew);
else {

         aftersprit=sprit(logicnew);
         newvec=newvector(aftersprit);
         tobebind=dontcare(newvec,aftersprit);
         logicout=rebind(tobebind,logicnew);
         free(newvec);
         freelogic(aftersprit);
         freelogic(tobebind);
         freelogic(logicnew);
         return(logicout);
   }
}

void push(struct config *table,int left, int right, int key) {
int current;
current=++table->top;
```

```
table->left[current]=left;
table->right[current]=right;
table->key[current]=key;
}

int find(int left,int right,struct config *table) {
int res,c;
res=-1;
c=table->top;
while (table->key[c]!=FINISH) {
if ((left==table->left[c]) && (right==table->right[c])) res=table->key[c];
--c;
}
return res;
}


struct config **makecircuit(struct logic *logicin,int selectmode) {
struct config **res;
int x,y,i,j,k,p,left,right,f,newkey;
newkey=2;
if (selectmode!=MERGE) {
res=newmalloc(sizeof(struct config*)*(logicin->y+1));
for(i=0;i<=logicin->y;++i) {
  res[i]=newmalloc(sizeof(struct config));
  res[i]->top=-1;
  push(res[i],0,0,FINISH);
  }
}
else {
res=newmalloc(sizeof(struct config*));
res[0]=newmalloc(sizeof(struct config));
res[0]->top=-1;
push(res[0],0,0,FINISH);
}

x=logicin->x+1;
y=logicin->y;
while (x!=1) {

for (i=0;i<=y;++i) {
p=0;
for (j=0;j<x;j+=2) {
left=logicin->table[i][j];
right=logicin->table[i][j+1];

if (selectmode==MERGE) k=0;else k=i;

if (left==right) logicin->table[i][p]=left;
else
  if ((f=find(left,right,res[k]))!=ERROR) logicin->table[i][p]=f;
else
  {push(res[k],left,right,newkey);logicin->table[i][p]=newkey++;};
p++;
}
if (selectmode!=MERGE) push(res[i],0,0,FINISH);
}
x/=2;
if (selectmode==MERGE) push(res[0],0,0,FINISH);

}
return res;
}

struct truthtable createtruth(int index) {
struct truthtable res;
int norow,i,j,tosa,sa,value;
norow=power(2,index+1);
res.index=index;
for (i=0;i<=index;++i) {
res.table[i]=newmalloc(norow*sizeof(int));
value=0;
sa=power(2,i);
j=0;tosa=0;
while (j<norow) {
res.table[i][j]=value;
j++;tosa++;
if (tosa==sa) {
tosa=0;
if (value==0) value=1; else value=0;
}
}
}

return res;
}

struct logic *changelogic(struct logic *logicin,struct truthtable *con) {
struct logic *res;
int x,y,i,j,k,index,co,acc;
x=logicin->x;
y=logicin->y;
res=initlogic(x);
res->y=y;

for (i=0;i<=y;++i)
res->table[i]=newmalloc((x+1)*sizeof(int));

index=con->index;
co=power(2,index+1)-1;
for (j=0;j<=co;++j) {
acc=0;
for (i=0;i<=index;++i) acc+=con->table[i][j]*power(2,i);
for (k=0;k<=y;++k) res->table[k][acc]=logicin->table[k][j];
}
return res;
}
```

```
void rot (int *initvector,int first,int last) {
int c,i;
c=initvector[first];
for (i=first+1;i<=last;++i) initvector[i-1]=initvector[i];
initvector[last]=c;
}


struct truthtable changetruth(int *convector,int last,struct truthtable *con) {
int i;
struct truthtable res;
res.index=con->index;
for (i=0;i<=last;++i)
res.table[i]=con->table[convector[i]];
return(res);
}


struct logic *copylogic(struct logic *logicin) {
struct logic *res;
int i;
res=initlogic(logicin->x);
res->y=logicin->y;
for (i=0;i<=logicin->y;++i) {
        res->pattern[i]=copypattern(logicin->pattern[i]);
        res->table[i]=copyvector(logicin->x,logicin->table[i]);
}
return res;
}


struct logic *superdontcare(struct logic* logicin) {
struct logic *res,*logic1;
int tempvector[9]={0,1,2,3,4,5,6,7,8};
tempvector[logicin->y+1]=FINISH;
res=dontcare(tempvector,logicin);
return res;
}

int mergecount(struct config **listofconfig) {
int res,i;
struct config *aconfig;
aconfig=*listofconfig;
res=0;
for (i=0;i<=aconfig->top;++i)
if (aconfig->key[i]!=FINISH) {
if (aconfig->left[i]!=0) ++res;
if (aconfig->right[i]!=0) ++res;
}
return res;
}

int nonmergecount(struct config **listofconfig,int total) {
```

```
int res,i,j;
res=0;
for (i=0;i<=total;++i)
for (j=0;j<=listofconfig[i]->top;++j)
if (listofconfig[i]->key[j]!=FINISH) ++res;
res*=2;
return res;
}


void clearpackage(struct package *apackage) {
int i;
for (i=0;i<=apackage->noofconfig;++i)
free(apackage->aconfig[i]);
freelogic(apackage->alogic);
}
struct package *selectconfig(struct config **currentconfig,
                struct logic *logichead,
                struct logic *logicin,
                int* seq,
                int lastseq,
                int selectmode,
                struct package *best) {
int pass,count,i,total;
struct package *res;
pass=0;
if (selectmode==MERGE)
{total=0;
count=mergecount(currentconfig);}
else
{total=logichead->y;
count=nonmergecount(currentconfig,total);}

if (best==NULL)
  {best=newmalloc(sizeof(struct package));pass=1;}
else if (count<(best->bestcount))
   {clearpackage(best);pass=1;}
if (pass==1) {
best->aconfig=currentconfig;
best->alogic=logicin;
best->noofconfig=total;
for (i=0;i<=lastseq;++i) best->seq[i]=seq[i];
best->lastseq=lastseq;
for (i=0;i<=logichead->y;++i) best->head[i]=logichead->table[i][0];
best->bestcount=count;
}
else
{freelogic(logicin);
 for(i=0;i<=total;++i)
 free(currentconfig[i]);
 free(currentconfig);}
res=best;
/*printpackage(best);*/
return res;
```

```
}



struct package *gencon(int *initvector,
            int first,int last,
            struct logic *logicin,
            struct truthtable *con,
            struct package *best,
            int selectmode,
            int enabledont) {
struct truthtable currentable;
struct logic *newlogic,*newlogic1,*newlogic2;
struct config **currentconfig;
int temp[9],i,j;
if (first!=last) {
for (i=0;i<=last;++i) temp[i]=initvector[i];
best=gencon(temp,first+1,last,logicin,con,best,selectmode,enabledont);
for (i=0;i<(last-first);++i) {
rot(temp,first,last);
currentable=changetruth(temp,last,con);
newlogic=changelogic(logicin,&currentable);
free(newlogic->pattern);

newlogic->pattern=logicin->pattern;
/*printcon(temp,last);*/
/*printlogic(logicin);*/
if (enabledont)
  {
  newlogic1=superdontcare(newlogic);
  }
 else
  newlogic1=copylogic(newlogic);
newlogic2=copylogic(newlogic1);
currentconfig=makecircuit(newlogic1,selectmode);
best=selectconfig(currentconfig,newlogic1,newlogic2,temp,last,selectmode,best);
for (j=0;j<=newlogic->y;++j)
free(newlogic->table[j]);
freelogic(newlogic1);
free(newlogic);
best=gencon(temp,first+1,last,logicin,con,best,selectmode,enabledont);
}
}
return (best);
}

struct package *funcom(struct logic *logicin,
int last,int selectmode,int enabledont) {
struct truthtable inittable;
struct package *best=NULL;
int i,initvector[8]={0,1,2,3,4,5,6,7};
inittable=createtruth(last);
best=gencon(initvector,0,last,logicin,&inittable,best,selectmode,enabledont);
```

```
for (i=0;i<=last;++i) free(inittable.table[i]);
return (best);
}




int init(int x,int y) {
int i,*pat;
logicinput=initlogic(x);
for(i=0;i<=y;++i) {
pat=malloc(sizeof(int)*2);
pat[0]=i;
pat[1]=FINISH;
logicinput->pattern[i]=pat;
}
logicinput->y=y;
return x;
}

int startop(int last,int selectmode, int enabledont) {
packageoutput=funcom(logicinput,last,selectmode,enabledont);
return (packageoutput->bestcount);
}


int writetable(int x,int y,int v) {
logicinput->table[y][x]=v;
return v;
}




int readtable(int x,int y) {
return packageoutput->alogic->table[y][x];
}

int readleft(int con,int no) {
return packageoutput->aconfig[con]->left[no];
}

int readright(int con,int no) {
return packageoutput->aconfig[con]->right[no];
}

int readkey(int con,int no) {
return packageoutput->aconfig[con]->key[no];
}

int readtop(int con) {
return packageoutput->aconfig[con]->top;
}
```

```
int readnoofconfig(int x) {
return packageoutput->noofconfig;
}

int readseq(int no) {
return packageoutput->seq[no];
}

int readlastseq(int x) {
return packageoutput->lastseq;
}

int readhead(int no) {
return packageoutput->head[no];
}

int finishop(int x) {
clearpackage(packageoutput);
free(packageoutput);
free(logicinput);
return x;
}


/*
main() {
res=funcom(test,5,MERGE,YES);
}
*/
```

APPENDIX B

CIRCUITS AND MASK LAYOUTS OF THE DIVIDER CHIP

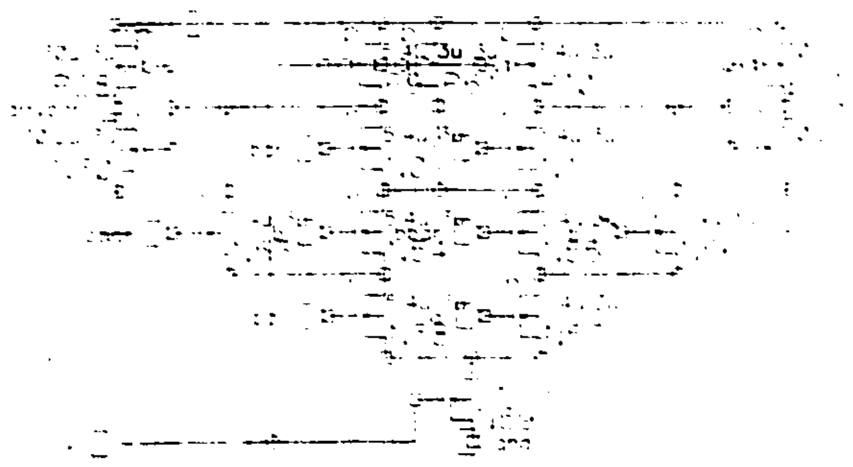


Figure B.1: Schematic diagram for the carry bit of the full adder

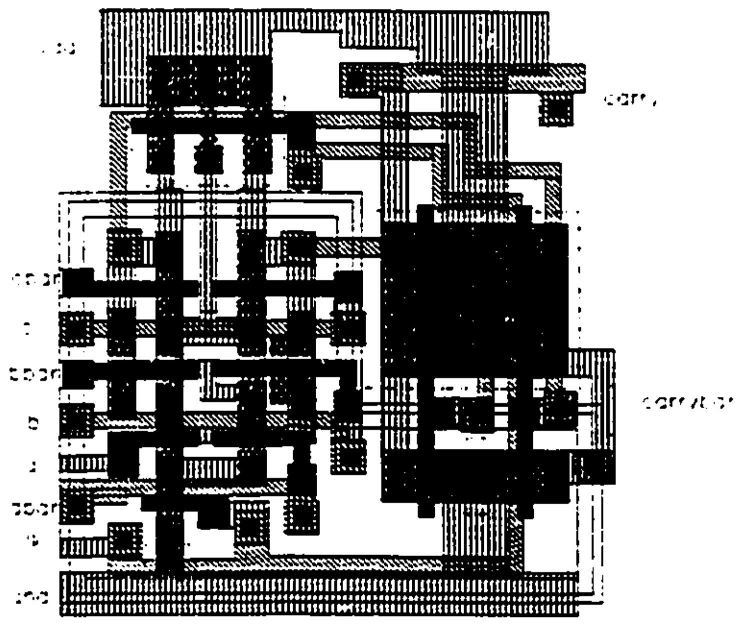


Figure B.2: Mask layout for the carry bit of the full adder

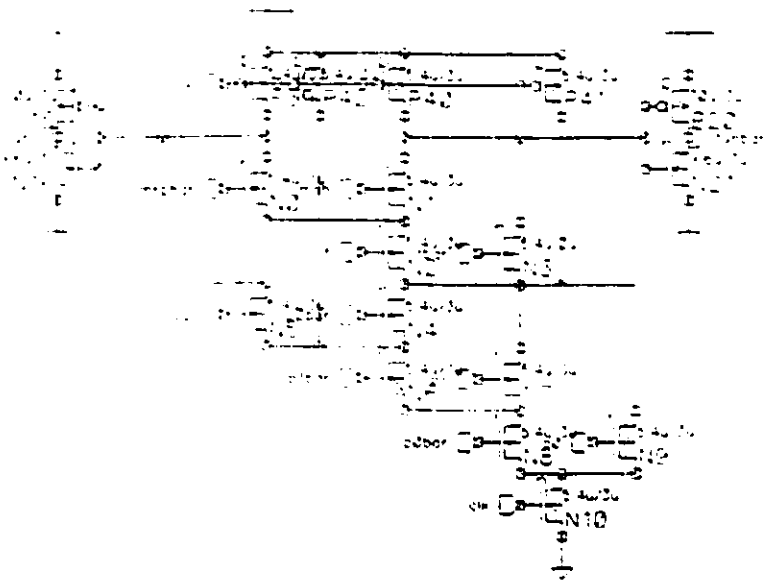


Figure B.3: Schematic diagram for the flag generator

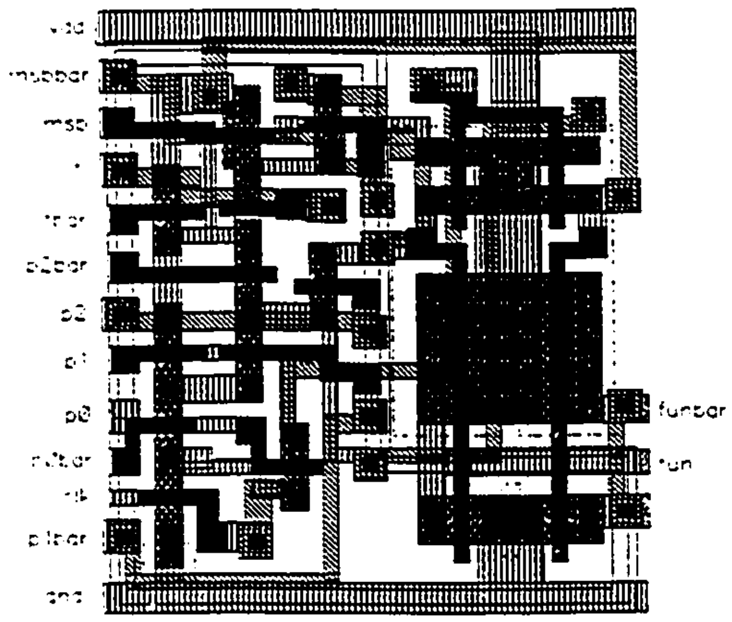


Figure B.4: Mask layout for the flag generator

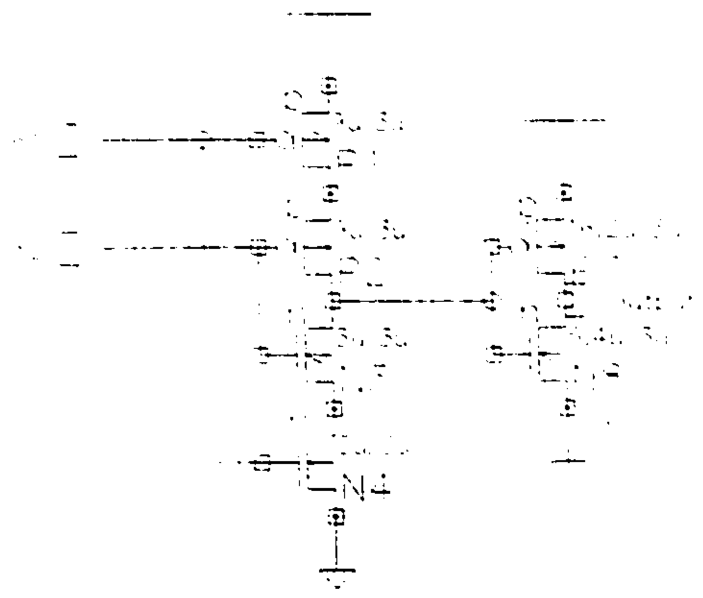


Figure B.5: Schematic diagram for the C element

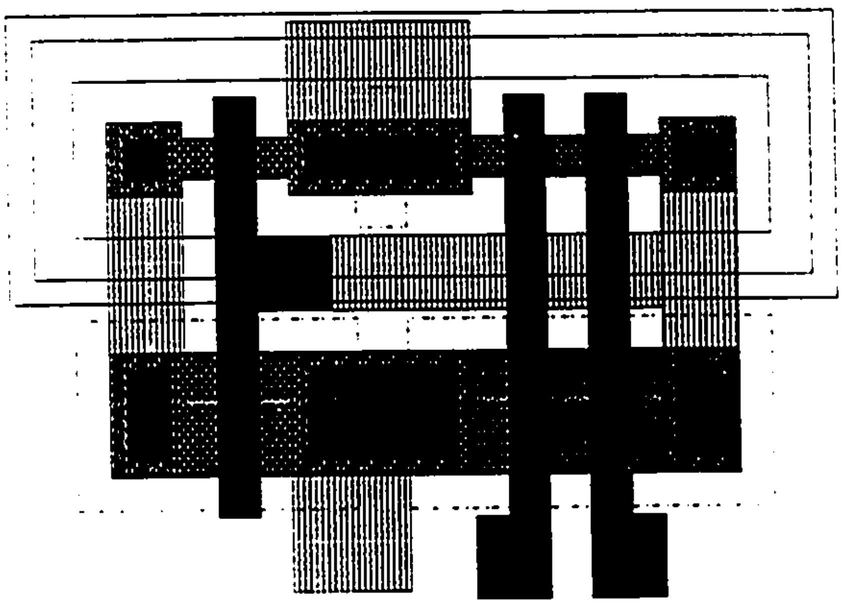


Figure B.6: Mask layout for the C element

Figure B.7: Schematic diagram for the quotient generator

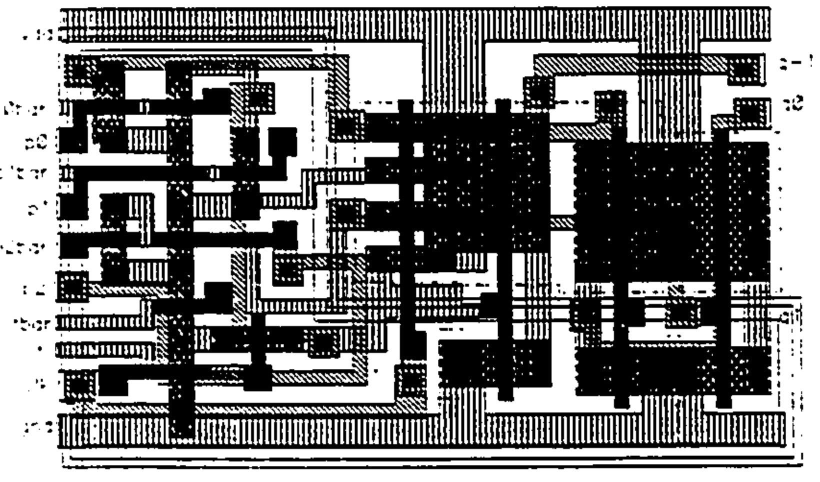


Figure B.8: Mask layout for the quotient generator

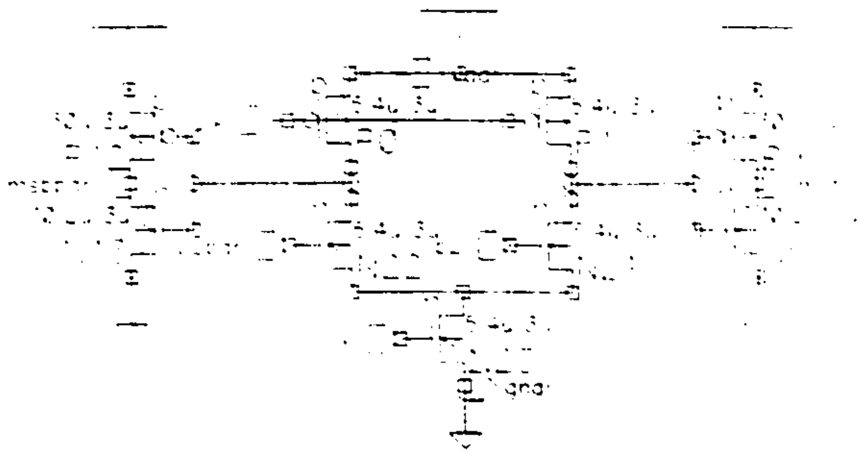


Figure B.9: Schematic diagram for the MSB generator

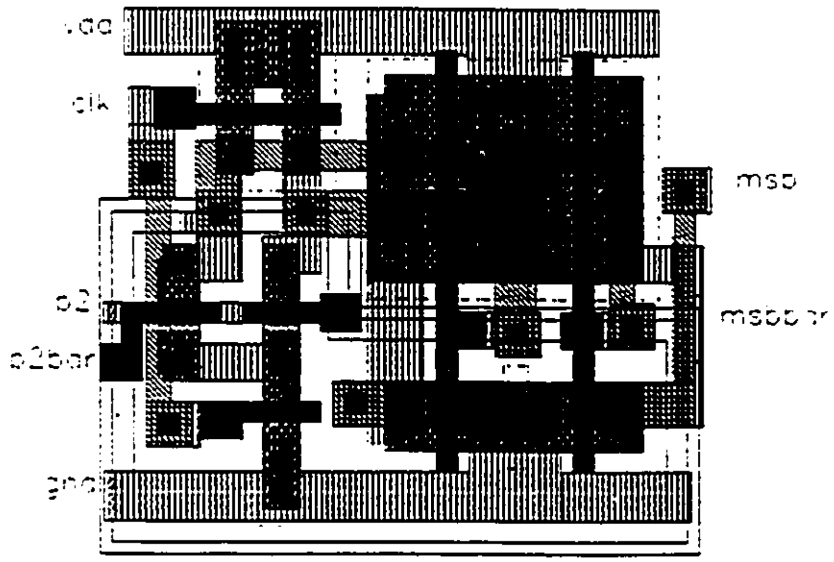


Figure B.10: Mask layout for the MSB generator

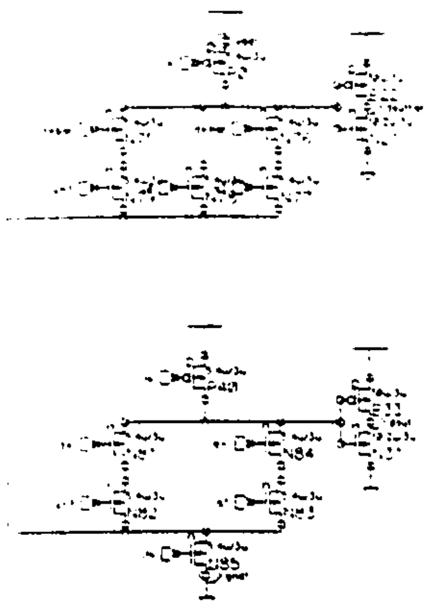


Figure B.11: Schematic diagram for the divisor multiplexor sub-cell

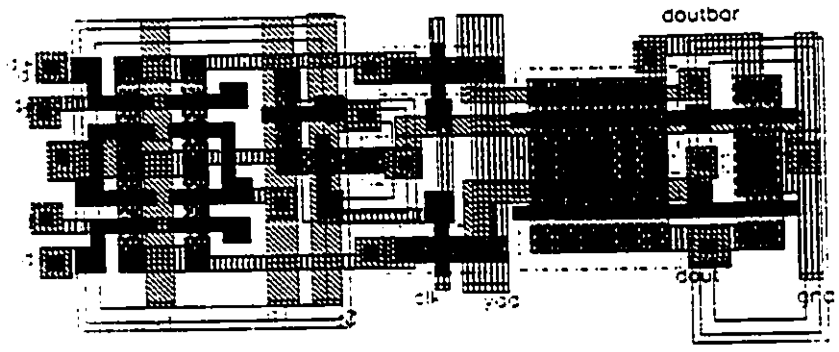


Figure B.12: Mask layout for the divisor multiplexor sub-cell

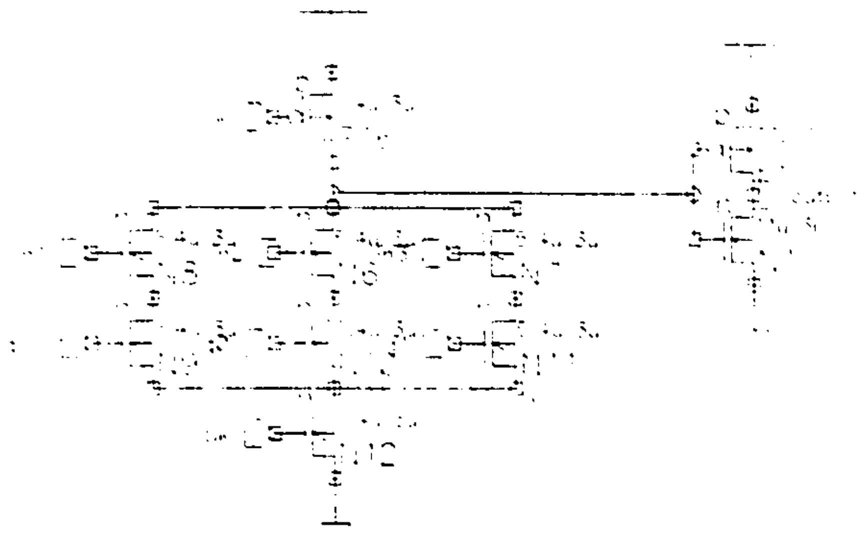


Figure B.13: Schematic diagram for the remainder multiplexor sub-cell

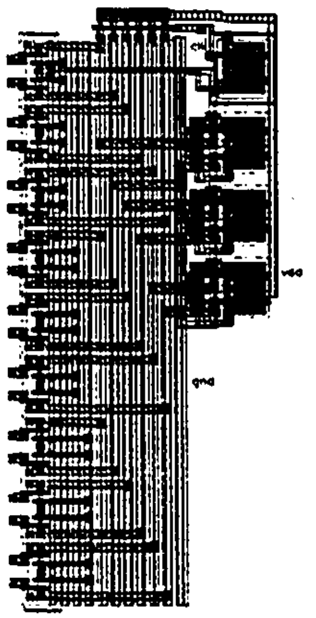


Figure B.14: Mask layout for the remainder multiplexor

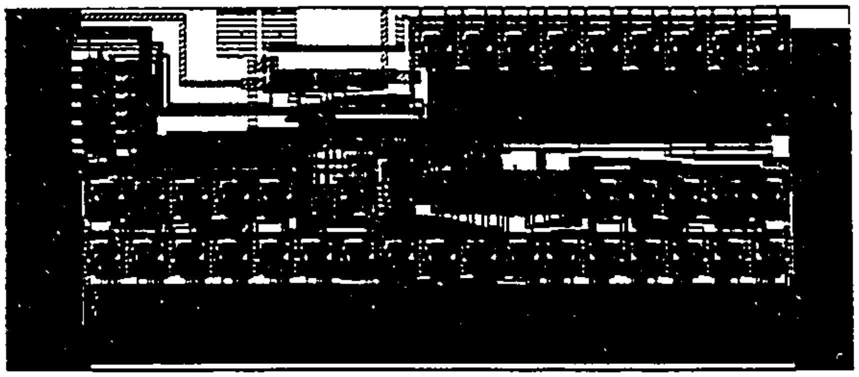


Figure B.15 Mask layout for one stage of pipeline

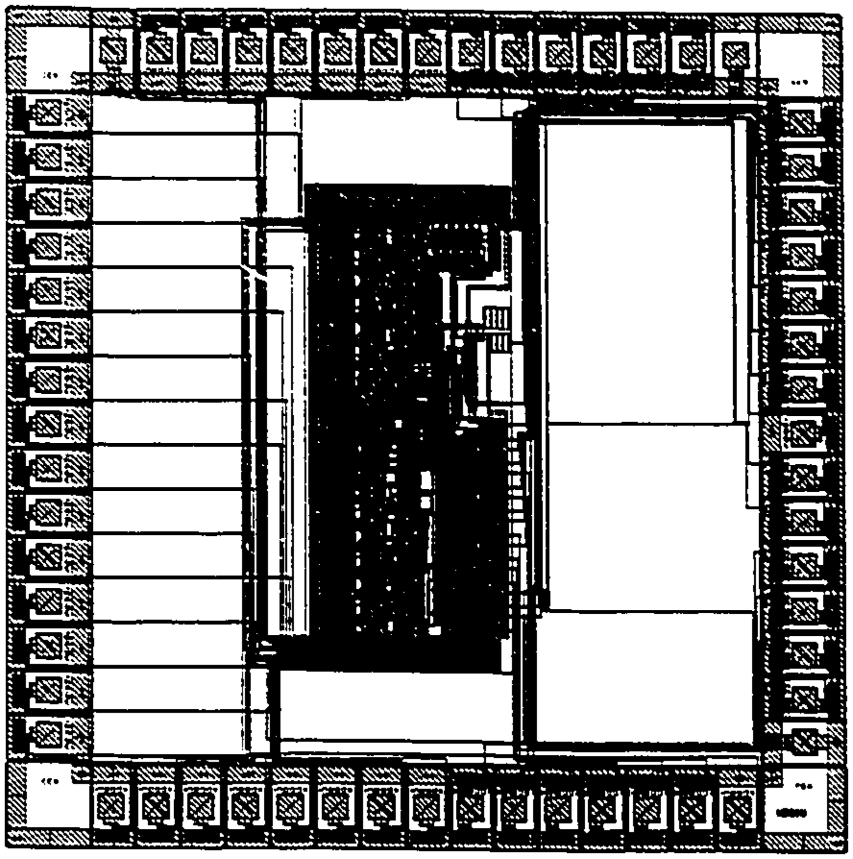


Figure A.14: Mask layout for complete chip

VITA AUCTORIS

NAME:	Hong Ming Chan
PLACE OF BIRTH:	Hong Kong
YEAR OF BIRTH:	1966
EDUCATION	Vincent Massey High School, Brandon, Manitoba 1984 - 1986
	University of Windsor, Windsor, Ontario 1986-1990, B.A.Sc.(Electrical Engineering)
	University of Windsor, Windsor, Ontario 1990-1992, M.A.Sc.(Electrical Engineering)