AUTOMATED SYNTHESIS OF MULTI-LEVEL COMBINATIONAL LOGIC IN CMOS TECHNOLOGY

by

M. E. Hofmann

Memorandum No. UCB/ERL M85/53

1 July 1985

AUTOMATED SYNTHESIS OF MULTI-LEVEL COMBINATIONAL LOGIC IN CMOS TECHNOLOGY

by

M. E. Hofmann

Memorandum No. UCB/ERL M85/53

1 July 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Title Page

AUTOMATED SYNTHESIS OF MULTI-LEVEL COMBINATIONAL LOGIC IN CMOS TECHNOLOGY

by

M. E. Hofmann

Memorandum No. UCB/ERL M85/53

1 July 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ACKNOWLEDGMENTS

Professor Richard Newton, my research advisor, provided the inspiration for this work. I wish to acknowledge his contribution first.

Over the course of the research described in this thesis I have had the opportunity to work with and ask the help of many people. They are not too numerous to mention. I want to thank Rick Spickelmier, Tom Quarles, and Peter Moore for general advice, programming help, and those things which "only take five minutes".

Several discussions with Professor Hugo DeMan provided useful insight into the design of NORA CMOS circuits. For work on specific problems I want to acknowledge several people: Peter Moore spent many hours on the ALU design and hacked the logic-level simulator to handle CMOS gates. (I also thank Peter for his uncanny ability to prove mathematical theorems in the time it takes him to go from a seated position to the blackboard.) Richard Rudell contributed to my knowledge of Boolean minimization problems and quickly added new features to the ESPRESSO program that helped in the evaluation of various minimization strategies. Ron Gyurcsik wrote a program to do least-squares fits and one to aid in MOSFET threshold voltage calculations. Ron also helped in the formulation of the charge redistribution equations. In the design of the ALU chip I wish to thank Dave Wallace for design of the input latches, John Zapisek for design of the destination logic, and B.K. Bose for design of the input and output pads. Thanks to their diligence the chip worked on first silicon. On the dynamic CMOS test chip I want to thank Joan Pendleton for work on the DRC rules file and for the design of the input and output pads. Chuck Kring contributed many hours in lab helping me benchmark this chip and Kok Chang ran a multitude of SPICE2 simulations.

The dynamic CMOS test chip was fabricated at XEROX PARC. Ben Pugh and Bridget Scamporrino at XEROX provided wafer processing details and test parameters. Their assistance in chip fabrication is gratefully acknowledged. Partial funding by Tektronix, Inc.,

Digital Equipment Corporation, and DARPA under grant N00039-83-C-0107 is also acknowledged.

I want to thank Jeff Burns in his editorial capacity for reading numerous drafts of this thesis. I would like to thank Jeff Burns, Ron Gyurcsik, Ken Keller, Grace Mah, Karti Mayaram, Tom Laidig, Deirdre Ryan, and Chris Marino for being regular guys. They are a credit to this universe and I think they are really swell. If it weren't for people like these life would be far less entertaining.

I want to thank my parents, Garda and Peter. They provided support and encouragement and gave me good advice when I wasn't in the most optimistic of moods. More than anyone else, they provided me with a reason and a goal for undertaking this project.

Finally, I would like to thank myself. I'm glad now that I stuck with it, though on the whole, if I had it to do over again, I think I'd rather move up to the Yukon and shoot moose. In finishing this work it seems only fitting and proper to quote the immortal philosopher and observer of goings-on, F. Flintstone: "Yabba-Dabba-DOO!"

AUTOMATED SYNTHESIS OF MULTI-LEVEL COMBINATIONAL LOGIC IN CMOS TECHNOLOGY

Mark Eric Hofmann

Ph.D.

Department of Electrical Engineering
and Computer Science

Sponsors: Tektronix, Inc.,
Digital Equipment Corporation

Signature 
Richard Newton
Committee Chairman

ABSTRACT

A framework for the synthesis of combinational logic functions in a dynamic CMOS technology is presented. The input to the package of programs, which may be run as a *pipeline*, is a set of Boolean equations. The synthesis package generates mask-level geometries as output. Circuit optimizations for both speed and area have been developed. This approach compares favorably with other automated synthesis systems, such as PLA-based methods, in terms of circuit delay and layout area.

Different technologies and design styles for the implementation of combinational logic have been studied. Static and dynamic circuits have been characterized and extensively simulated. Two experimental chips were fabricated to further examine dynamic CMOS circuits. One of the test chips, a 32-bit ALU, is being used as part of a VLSI RISC microprocessor. The test measurements show that many dynamic circuits will have *charge redistribution* problems unless precautions are taken in design. Algorithms have been developed which partition complex circuits so that charge redistribution problems are avoided.

The regular structure generated by the pipeline is an extension of Weinberger arrays. The use of the *Domino* design style allows the construction of complex gates. Rather than perform a two level expansion on the Boolean equations, the framework maintains a multi-level expression hierarchy. This hierarchy is implemented in a multi-level matrix which can be topologically compacted. The algorithms for compaction presented in this dissertation allow simple-column and multiple-row *folding* with external constraints. The folded *connectivity* matrix is translated to the mask level by a *context-based* tiler. The tiler reads from a tile library and is process independent.

Table of Contents

CHAPTER 1

Introduction and Review of Previous Work

Much of the circuitry in a VLSI design may be cast in a regular, or array-based, form and thus may be generated automatically. However, blocks of complex combinational logic often require hand layout because they are not structured; the time spent on this portion of the design is often the most significant part of the project [latt81]. The aim of the work presented in this dissertation was to explore methods of automating the design of complex logic functions. In addition to reducing the time between circuit conception and circuit fabrication, automated methods of circuit design and layout decrease the possibility of design error, ease the overhead of circuit modification and often lead to efficient testing strategies.

1.1. Need for Automation of Combinational Logic

A typical VLSI chip is comprised of a relatively small number of different sections. In almost any VLSI design there will be sections of RAM and ROM. There will likely be a processing section, for example an ALU in a microprocessor architecture. In addition, signal buffering, conditioning circuitry, and control logic are required. In general, a processor may be divided into two broad sections: control and datapath. The control section consists of complex combinational logic and a small amount of storage circuitry. The datapath section includes ROM, RAM, an ALU, and intermediate storage latches. Even in a highly structured chip design, such as the reduced instruction set CMOS SOAR processor [patt81], over 30% of the chip area was used explicitly for control purposes [mari85]. On the other hand the combinational part of the control logic contributes only 10% of the total device count. The disparity in these figures reflects the patently irregular nature of combinational logic. *Irregular* in this sense means that the pieces of combinational logic do not fit together well

and a substantial amount of routing is required to connect them.

So-called *random* or *arbitrary* combinational logic used in a custom design has a replication factor close to one. *Replication factor* is defined as the number of times the same cell is used or placed in the design. The replication factor of the datapath is typically skewed by on-chip ROM or RAM. For SOAR the RAM replication factor is over 2300. SOAR is a 32-bit bitslice machine, and therefore, even excluding RAM, the replication factor is over 32 since storage latches and buffers are used several times in each bitslice. For the control logic of the SOAR chip common latch, buffer, and inverter library cells were used wherever possible. Still, almost all of the control logic is implemented using 14 PLAs. Even though this logic may make up only a small part of the total device count, it represents a significant amount of chip area and often the majority of distinct, designed cells. A major portion of the design time is expended in layout of this logic.

1.2. Goals of Current Research

The goal of this research was to create a framework to study the automatic generation, optimization, and layout of arbitrary, multi-level combinational logic. As explained in Section 1.4, the input to the package presented in this dissertation is an optimized, multi-level combinational logic description. The logic function may have been hand-optimized or optimized automatically [bray84b] [rude85] for logic compactness. The goal of the work described here is to optimize the function for both speed and area, taking into account electrical considerations. Since the process-critical parts of the package read from *technology* files, changes in processing parameters, within the given design style and technology, can be easily accounted for. The actual layout generation is performed by a design-rule-independent tiling program which means that the program does not have to be modified as design rules change.

The synthesis framework has been implemented as a pipeline of CAD programs that allows a circuit designer to specify combinational logic at a high level and produce an efficient circuit realization at the mask level. A *pipeline* of programs is a set of programs

that may be used either separately or as a package. If used as a complete package, the output of one program drives the next without need for user intervention. A program or package of programs that generates a particular type of cell automatically is called a *module generator* [newt81]. In Figure 1.1 the stages in the implemented pipeline, known as MAMBO, are shown. The names of the associated tools in the pipeline are given in parentheses.

In the pipeline implementation a *technology*, *electrical design style*, and *layout method* must be chosen. Large-scale digital designs are typically implemented in a MOS technology because of its superior packing density. For this project the CMOS technology has been chosen because it is possible to design low power, dense circuits in CMOS. The choice of *design style* is an implementation-level decision. The relevant design parameters are the speed of a basic gate, area consumption, and ease of automated layout, topological optimization, and Boolean minimization. After extensive simulation, a mixed static and dynamic, clocked design style was chosen. The term *layout method* applies to the geometric level of design. The layout method used in this project is a generalization of Weinberger arrays. It has the advantage of guaranteeing a regular, structured layout.

1.3. Organization of this Dissertation

This dissertation has nine chapters. The remainder of the introduction provides a review of previous work in the area of automated synthesis of combinational logic modules. Contrasts between static and dynamic CMOS design styles are drawn in Chapter 2. The advantages and deficiencies of each style, along with their best application areas, are described. Detailed results of simulations of static and dynamic circuits in a specific CMOS process are presented in Chapter 3. Results from a test chip constructed to examine problems with dynamic circuits are also described. Chapter 3 concludes with results from a 32-bit ALU test chip designed in the Domino style. In Chapter 4 an overview of the stages in the MAMBO synthesis system is presented. The objective and constraints at each synthesis step are stated and the initial parsing and logic optimization phases are described

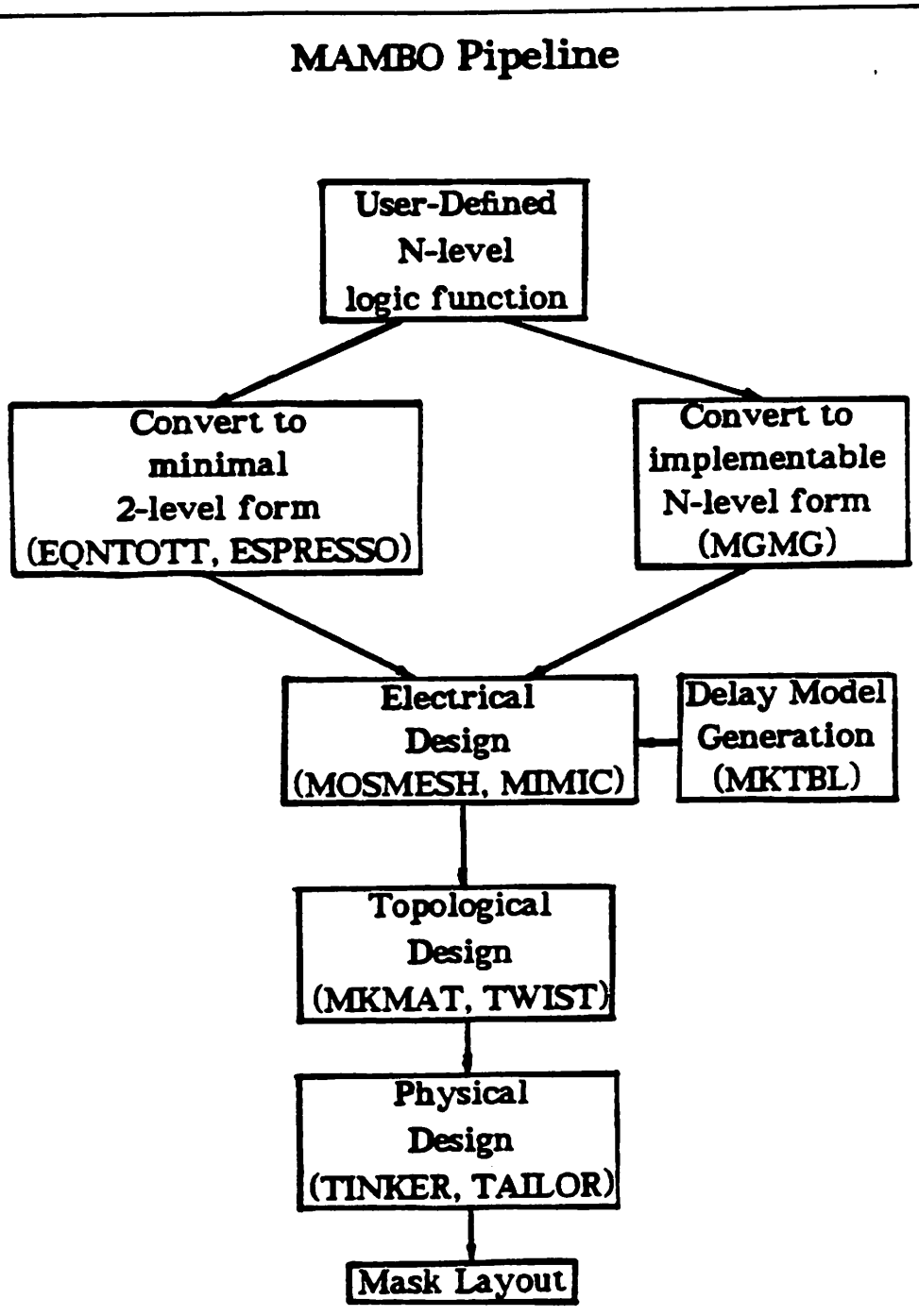


Figure 1.1: Stages in the MAMBO Automated Logic Synthesis System

in detail. In Chapter 5 the tradeoffs involved in partitioning large, dynamic, combinational circuits are explained. Circuits may be partitioned according to several criteria to result in reduced delay and greater ease of layout. In Chapter 6 algorithms for the area

optimization of matrix structures are explored. The current structure is contrasted with previous work on folded, tiled PLA structures. Chapter 7 contains an overview of layout tiling methods and a comparison between tiled and routed methods. The separation of electrical and geometrical rules is examined. A comparison of layouts and delays of several PLAs from the SOAR chip with multi-level dynamic implementations is presented in Chapter 8. Conclusions and directions for further work are presented in Chapter 9.

1.4. Comparison with Previous Work

The combinational logic synthesis problem can be broken up into four parts as shown in Figure 1.2.

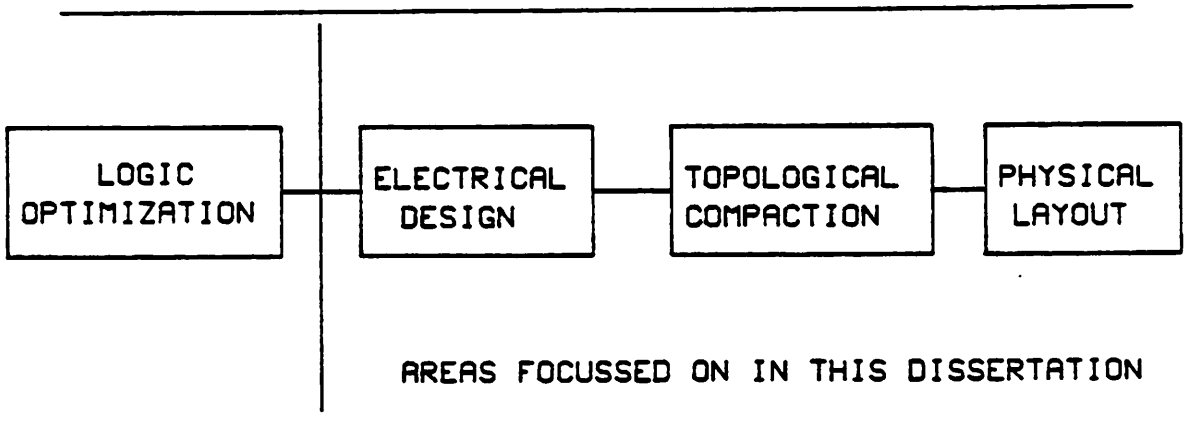


Figure 1.2: The Four Steps in Logic Synthesis

The first step, logic optimization, is a multi-faceted problem. The goal of logic optimization is to reduce circuit complexity in some manner so that the optimized circuit requires less chip area. The front-end program in the MAMBO pipeline, MGMG, will perform simple logic optimization if requested, however it is assumed that the logic expressions input to the MAMBO package are already in a logically optimized form. Two methods of logic optimization are reviewed here. The first method reduces circuit complexity by partitioning the function. The methods of functional and hierarchical partitioning are described. The second method is logic optimization by multi-level Boolean minimization.

1.4.1. Logic Optimization by Circuit Partitioning

A circuit may be partitioned according to the function it performs. That is, if the designer knows something in particular about the function he wishes to implement he may be able to use this to advantage in circuit generation. An example circuit where such techniques are useful is the *exclusive-or* (*XOR*) function. In an n-input XOR the function has the value 1 *if and only if* an odd number of inputs are 1. The following example is taken from [flei75]. A calculation is carried out on the number of bits it would take to represent an XOR function on 16 inputs, using different decoding schemes. For the case of one decoder with 16 inputs the XOR maps into a single column with 2^{16} bits. This represents the completely decoded case, where each bit (a *1* or a *0*) indicates the function output. This could be mapped into a single circuit. The other extreme is to employ 16 1-bit decoders. Each decoder produces two outputs, the input variable and its inversion. Now there are just two possible cases per decoder times 16 decoders or $2^1 \times 2^4 = 2^5 = 32$ bits per column. However 2^{15} columns are required, since there are 2^{15} ways of representing two variables across 16 inputs. This case, which could be implemented by partitioning a single PLA into 16 smaller arrays, is clearly the worst case.

There are intermediate solutions and these are tabulated in Figure 1.3.

Number of decoders	Inputs per decoder	Total number of bits
1	16	$2^{16} = 65,536$
2	8	$2^{10} = 1024$
4	4	$2^9 = 512$
8	2	$2^{12} = 4096$
16	1	$2^{20} = 1,048,576$

Figure 1.3: Total number of bits to implement XOR

The best case turns out to be four 4-input decoders. This situation could be realized by four small PLA circuits. In this example, the output of each decoder produces $2^4 = 16$ lines. Since four decoders are used this is a total of 4×2^4 or 64 bits per column. Since each decoder deals with only four inputs there are just 2^3 minterms per PLA. Thus the total bit count across all columns of all PLAs is just $4 \times 2^4 \times 2^3$ or 512 bits overall.

In this presentation the routing between the component circuits has been neglected. This can represent a significant portion of the circuit area. An example circuit which illustrates the interconnection routing problem is presented at the end of this section.

Another partitioning method to reduce total cell area is to implement a hierarchy of cells. This approach does best when the functions are complex and heavily interdependent. This method is similar in effect to the approach employed by the current work, where a multi-level form of the input Boolean expressions is retained. The following example is taken from [ayre79]. Here the designer wishes to implement a 16-bit counter. It can be implemented as single circuit in two-level logic. A schematic representation of a single PLA implementation is shown in Figure 1.4. PLA area is 32,000 units.

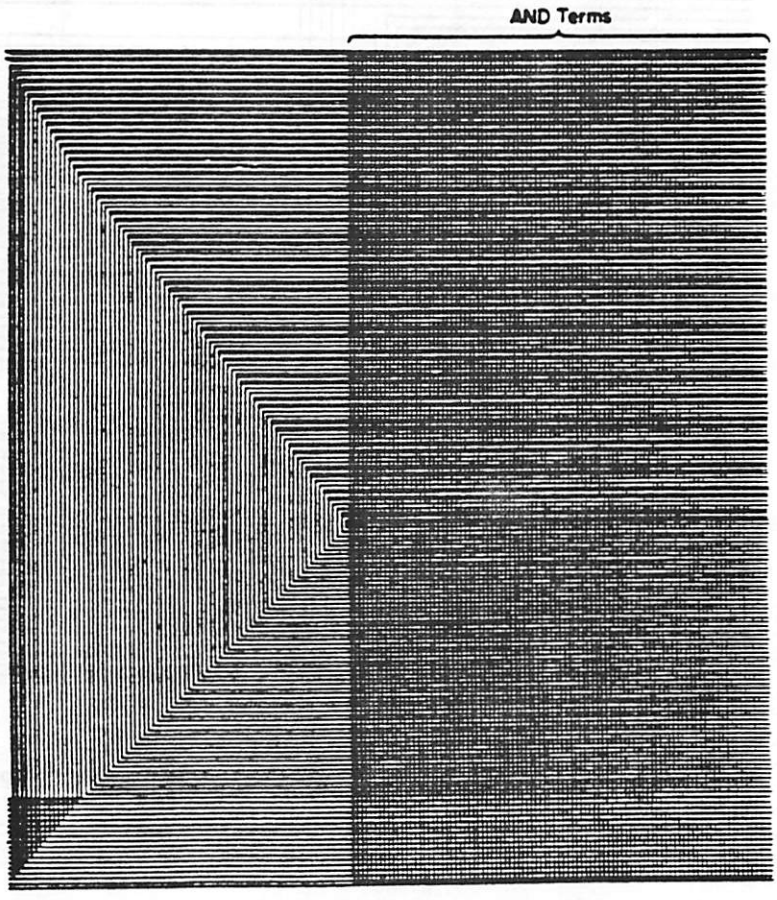


Figure 1.4: 16-bit Counter as a Single PLA

However, by introducing additional logic stages, so that the circuit is now AND-OR-AND-OR, the overall area of the counter can be reduced. The total area for the two-deep implementation is 11,500 units.

The decomposition can be continued until the minimum branching factor of two is reached. In this case, if each bit of the 16-bit counter is handled by a single PLA, a five-level representation can be obtained. A schematic representation of such an implementation is presented in Figure 1.5. Total area of the circuit is 7,000 units.

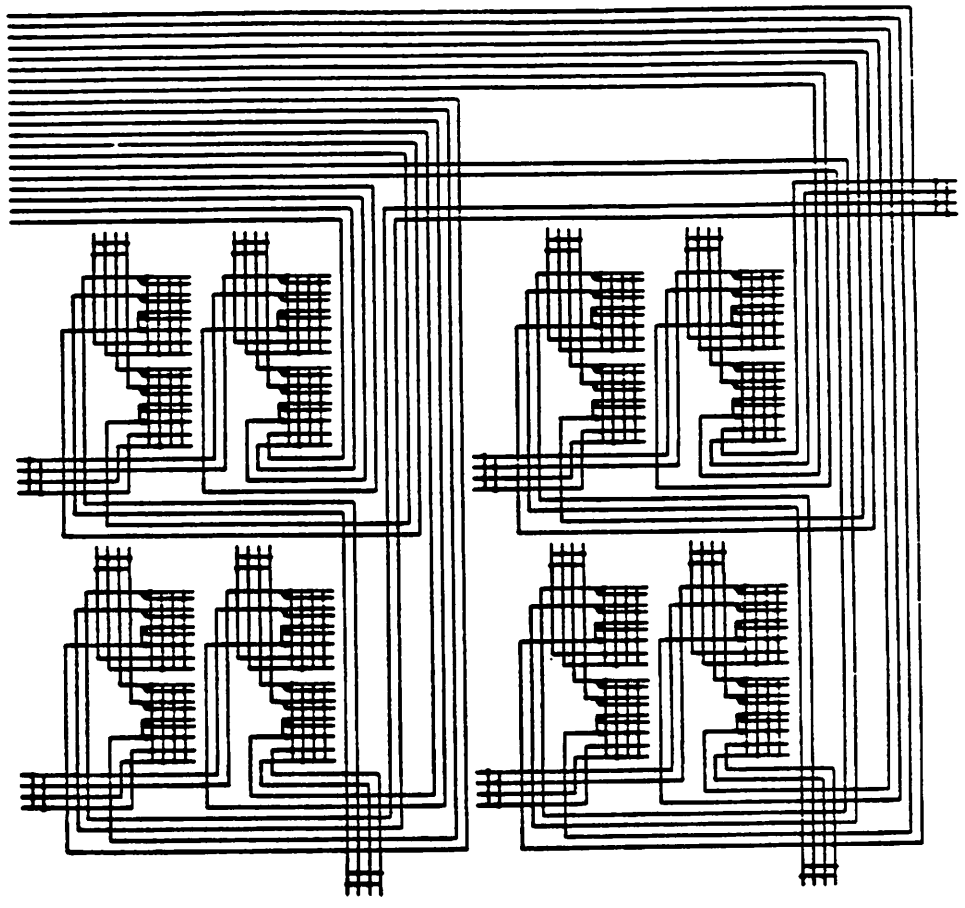


Figure 1.5: 16-bit Counter as Five Levels of PLA

Because total cell area has been reduced significantly, the speed of the counter, although not reported, would increase, other factors remaining equal. It is not clear from [ayre79] whether or not the additional routing area for the PLA interconnections was taken

into account. It would appear from the diagrams that routing area has been factored into the area calculation. In general, as circuit function is broken up into more and more levels of logic, the ratio of routing area to cell area increases. The total area may, in fact, increase in absolute terms. The amount of space dedicated to routing in the last two examples appears significantly greater than the amount of space occupied by the PLA module area. The curve shown in Figure 1.6, from [sans81], shows that as the area of individual cells, for example PLAs, decreases in size and complexity proportionally more area is taken up by interconnect.

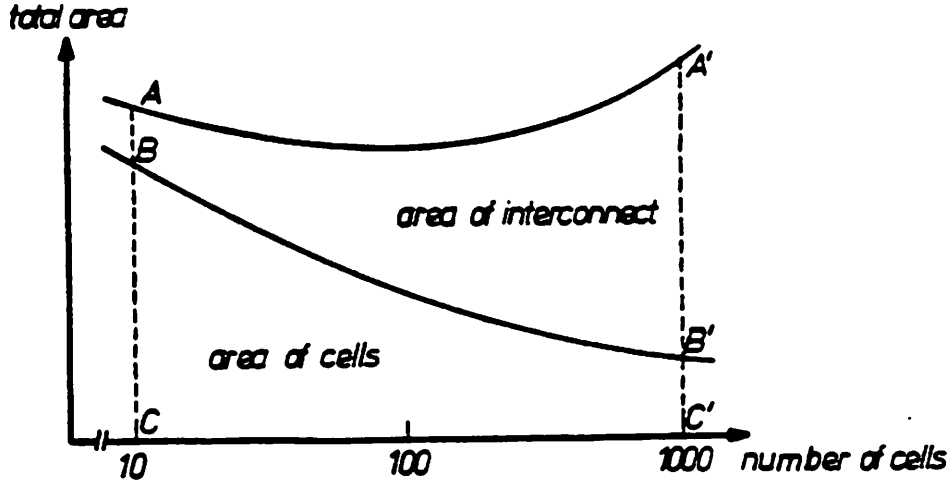


Figure 1.6: Ratio of Interconnect Area to Cell Area as Fragmentation Increases

In the extreme, total chip area may increase even though there are fewer device placements.

1.4.1.1. Example Interconnection Problem in Partitioning

When a larger block is partitioned by either of the two methods just presented inter-block routing becomes an issue. Layout schemes which route by block abutment, therefore, can result in a substantial area savings. A partitioned 32-bit ALU implemented as six PLAs has been compared with a bitslice approach [sout82]. Four 8-bit PLAs, based on the highly optimized, compact design presented in [schm80] form the core of the NMOS

design. Two other carry-lookahead PLAs are employed for speed. The floorplan of the 32-bit ALU-PLA layout is shown in Figure 1.7.

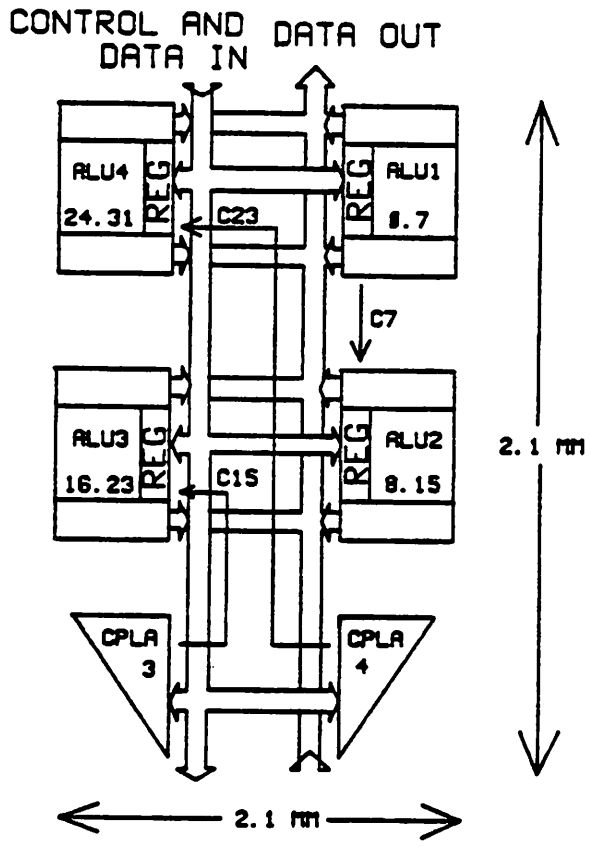


Figure 1.7: Floorplan of Partitioned ALU-PLA

The bitslice approach was based on a single cell per bit design. Carry generation was accomplished by a Manchester-type carry chain which is able to bypass 4-bit sections of the ALU for fast carry propagation.

Both designs were fabricated in a single layer metal, polysilicide process. A comparison of the two designs is shown in Figure 1.8.

Attribute	PLA	Bitslice
Worst-case Speed	$22ns$	$35ns$
Power	$200mW$	$125mW$
Total Area	$4.2mm^2$	$1.0mm^2$
Routing Area	$1.8mm^2$	-

Figure 1.8: Comparison of PLA and Bitslice Techniques

A significant portion of the PLA design, $1.8mm^2$, or 43%, is taken up in routing area. This is because it is very difficult, if not impossible, to generate partitioned PLAs that route by abutment. Presumably the bitslice design takes up less space not only because the slices connect by abutment, but also because they are more regular in structure. The speed advantage of the PLA is due largely to the full carry-lookahead: the tiling approach itself does not compromise circuit speed. The comparison indicates that a layout scheme which routes by abutment gives a more compact result. This result is especially important in multi-level circuits where the amount of intercell routing is large.

1.4.2. Logic Optimization by Boolean Minimization

Rather than apply special knowledge about a circuit to minimize its area, direct multi-level Boolean minimization can be employed to reduce device count and hence circuit area. Several algorithms have been published [bray82] [risc82] [bray84b] which deal with fast, heuristic methods for both decomposition and factorization of Boolean expressions. *Decomposition* is a technique for discovering common subexpressions in a system of (two or more) Boolean expressions. *Factorization* is a similar technique, used to rearrange a single expression. Brayton and McMullen [bray82] outline an algorithm which simplifies a set of functions until they are "relatively prime" by successive substitution of new variables for common subexpressions.

Before examining the process of decomposition, it is useful to define some terms. Two expressions are said to be *relatively kernel free* if they have no kernels in common. A *kernel* of an expression is a *cube free primary divisor*. A *cube* is a set c of literals such that if Boolean variable x is an element of c, $\bar{x}$ is not in c. A *literal* is a Boolean variable

or its negation. An expression is said to be *cube free* if the only cube *evenly* dividing the expression is 1. Function g divides function f *evenly* if $(f / g)g = f$. The *product* and *division* operators are defined for f *orthogonal to* g. Functions f and g are said to be *orthogonal* if none of the literals of f are in g. The *primary divisors* of an expression f are those cubes c which divide f, *i.e.* $f / c \neq \phi$. For example, if $f = EF(A + BC)$ and $g = (A + BC)$ then $f / g = EF$ hence g *divides* f and, in this case since $(f / g)g = f$, g divides f *evenly* and $(A + BC)$ is a *primary divisor* of f.

Decomposition is applicable to sets of Boolean expressions, while factorization applies to single functions. It is the decomposition methods that are useful for matrix optimization. Decomposition is a two step process. First, common subexpressions, consisting of two or more cubes, are extracted from of a set of functions until the expressions are relatively kernel free. At this point expressions can, at most, share a single cube. In the second step, these are located and extracted also. The result is that the only common divisors are single literals: all global commonality has been discovered.

Brayton and McMullen call the first step *distillation* and the second step *condensation*. Both steps involve simplification of an expression by extracting a common subexpression: each step is repeated until no common subexpression can be found among any pair of expressions.

Both steps require a selection heuristic. In distillation the object is to find a pair of kernels K, K' such that at least 2 cubes are common, for K, K' not in the same function. In the condense algorithm a pair of cubes c, c' must be found such that at least 2 literals are common, for c, c' not in the same function. The effectiveness of these steps depends on the selection heuristic. Rather than search for all kernels, one can define the *level* for a kernel and then restrict the search to all kernels at a given level. The level of a kernel is recursively defined. Level 0 kernels are all kernels in which no literal appears twice. Kernels $\mathbf{K}^{n+1}(f)$ are those kernels, not including f itself, which are kernels of $\mathbf{K}^n(f)$. By this definition the complete kernel set, $\mathbf{K}(f)$, is the union of all levels n of $\mathbf{K}^n(f)$.

The table in Figure 1.9. from [bray82]. shows an example of 0- and 1-level partial kernel decomposition.

Decomposition Level	Transistor Count
None	2750
0-Level	1928
1-Level	1786

Figure 1.9: Example of Boolean Decomposition

Partial decomposition is supplemented by an additional collapsing step after the condensation algorithm. This extra step is useful because some kernel terms may not have multiple instances. in other words they appear only once in the set of Boolean functions. In this case *back substitution* into the kernel list allows the discovery of complex subexpressions and reduces the number of separate subexpressions. As a practical consideration the complement of the extracted kernel is also computed: it may form part of the subexpression as well.

The result of using the above approach is a smaller number of gates and perhaps a decrease in the total number of Boolean variables. Gate reduction comes from elimination of duplicate function implementation. The number of Boolean variables in a set of logic expressions is the sum of input. output. and intermediate variables. The number of input and output variables remains fixed. The number of intermediate variables will be reduced if it is possible to collapse subexpressions. The logic-optimized circuit is thus both smaller and denser than the original circuit.

1.4.3. Implementation of Logic-Optimized Circuits

After logic optimization. the combinational circuit proceeds through the stages of electrical. topological and physical design— the topics of this dissertation. Combinational logic may be realized in many different ways. The structured forms of layout include Weinberger arrays [wein67]. storage/logic arrays or *SLA*s [pati79]. and gate matrices [lope81]. These methods are termed *tiled* methods because connection between cells is by

abutment, just like tiling a floor. In comparison, there are *routed* schemes where interconnection between blocks of logic is performed by a router. Hand layout or the layout of partitioned PLAs are examples of routed approaches. Another common technique for implementing combinational logic is the "standard cell" approach. [souk81]. In a standard cell system simple functions are performed by each of many different cells. The cell collection makes up a library. Each cell in the library has the same height but a variable width.[1] The height constraint allows for the construction of rectangular routing channels to interconnect cells. The cells are selected from the library on the basis of their function and are placed in rows, perhaps by an automated placement program, based on the number and position of their inputs and outputs. The cells are then routed automatically. A typical standard cell layout is shown in Figure 1.10 [dunl83].

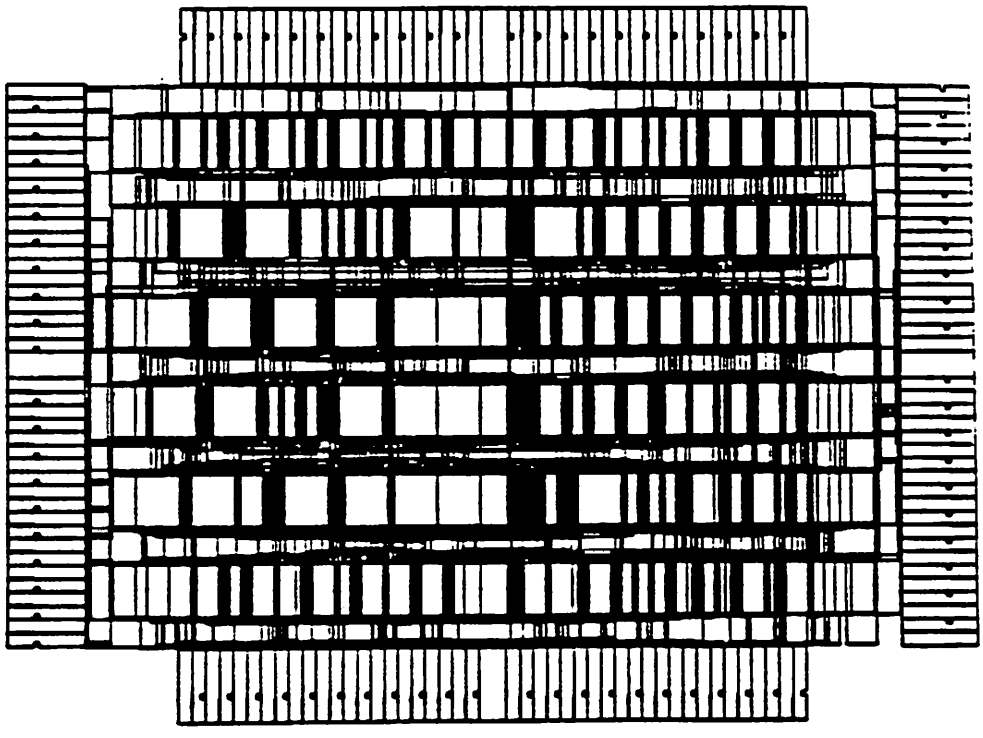


Figure 1.10: Automated Standard Cell Layout

[1]Recently standard cell systems have been developed which permit both variable height and variable width cells.

Other tools that work from high-level circuit descriptions are the MACPITTS program [sout83] and work on silicon compilation at IBM [bray84a]. The MACPITTS program generates Weinberger arrays for the control structures and is an example of a tiled method. The IBM approach is based on a form of Dynamic CMOS logic known as differential cascode voltage switched logic (DCVS). In this routed method, compact function cells are constructed and an automated program performs the cell interconnection.

1.5. PLA Design

The most successful structured approach to date is based on a two-level circuit representation. It is possible to represent any combinational circuit in two-level form (*i.e.* product-of-sums, sum-of-products) [nagl75]. The classical implementation of such a representation is the *Programmable Logic Array* or (*PLA*) [carr72]. Because research into PLA generation and optimization is well advanced it is reviewed in detail here. PLAs are often the approach of choice for combinational logic design because they are easy to machine generate and, for small circuits, give good speed due to their two-level nature. In practice, however, PLAs which implement functions of many input and output variables (*e.g.*, 20 to 100) tend to be slow. This is a direct result of their large size. Such PLAs are large because they provide the possibility for every input term, or its complement, to take part in every product term and, therefore, to influence every output. These PLAs often have a correspondingly large number of product terms which adds to the worst-case circuit delay. Large PLAs have high source capacitances on product term lines. Unless special precautions are taken, there will also be large *IR* drops on input and output signal paths [mah84].

1.5.1. PLA Compaction by Folding

Not long after the first PLA generation programs were introduced [glas80] [hofm80] [land82] it was recognized that PLA density could be increased by topologically rearranging the input and output blocks or *planes* of the PLA. Functionally the PLA remains

unchanged. however some of the unused placement sites have been discarded. This topological rearrangement is called *folding*. Folding compacts the PLA by taking advantage of the fact that though all inputs may contribute to a given product term. and all product terms may contribute to a given output. it is very rare that such fully connected terms exist. Therefore. in what is known as *simple* folding two inputs can share the space formerly occupied by a single input. This can be done likewise for output and product terms. In *multiple* folding more than two input. output. or product terms are collapsed into the space of a single term.

Early work on PLA folding theory and implementation was carried out by [hach82] and by [hofm80]. In [luby82] the optimal PLA folding problem was shown to be NP-complete. Therefore. heuristics are employed to generate fast. near-optimal compaction. Several early folding heuristics were shown to be near-optimal only for certain classes of PLAs and an exhaustive search algorithm using branch-and-bound techniques was found useful on small (*e.g* <20 inputs/outputs) or dense PLAs [hofm80]. The program runtime proved prohibitive for large. sparse structures.

While many of the early folding programs provided significant area reduction they often did so at the expense of increased external routing.[2] Specifically. the designer had no control over the placement of input and output signals. This meant that while the area of the core planes of the PLA was reduced the overall area of the PLA. with the interconnection routing taken into account. might actually have been worse. More recent work by De Micheli [demi84] [demi82] addresses the folding problem with input and output constraints in detail. De Micheli presents a set of heuristics for both constrained and unconstrained multiple folding. Running in a constrained mode. area reductions appear to be about 20% less (referenced to original area at 100%) than their unconstrained compacted counterparts [demi84]. This work represents the current state-of-the-art in PLA compaction.

[2]The BLAM program [hofm80] was able to reduce a large UC Berkeley RISC I processor control PLA by 40%. The compacted version of the PLA was not used because input signal routing from the surrounding circuitry to the PLA was difficult using the layout techniques available at that time.

1.5.2. PLA Compaction by Block-Partitioning

Partitioning a circuit into independent pieces, that is subcircuits which have distinct inputs and outputs, will reduce circuit area. Partitioning a PLA in this manner is in effect performing *block diagonalization* on the original PLA. For example, in the case of an AND plane with n inputs and m product terms, the unpartitioned plane occupies $O(n \times m)$ area. In the partitioning limit, if each input contacts only a single product term, the AND planes of the m fully partitioned PLAs total $O(n)$ in area. This type of block-partitioning is employed by the SMILE program described in [demi83].

1.5.3. Deficiencies of PLA Compaction Approaches

While folding and partitioning produce area-efficient circuits, which is one of the key parameters in the assessment of module generators, such programs only indirectly address the problem of circuit delay. The fan-in of the PLA planes may be reduced by compaction and this may indirectly decrease the critical path delay, but no delay optimization is provided by folding or partitioning itself. The work in this dissertation recognizes that timing analysis and delay optimization are crucial points in high-performance circuit design. In contrast to the PLA-based combinational logic systems, the work presented here takes as its primary goal the optimization of critical path delay. Topological compaction is performed after delay optimization and care is taken not to degrade circuit performance.

1.6. Summary of Results

The target specification of this research was to build a tool that produces delay-optimized circuits which are also compact in layout area. This goal has been met. Typical worst-case circuit speeds, as a result of delay optimization, are around a factor of two faster than comparable optimized PLA implementations of the same logic function. Of the examples tested, circuit speedups ranged from 1.2 to 2.5 times a PLA with the greater speedups seen on the more complex circuits. The tradeoff with speed is circuit area. The straightforward automated tiling scheme used in MAMBO produced structures typically

twice the size of comparable PLAs, though the range was from 1.5 times for large PLAs to 5 times for small circuits. For particular circuits MAMBO designs are both faster and smaller than comparable PLA implementations. These results show that the optimized multi-level approach yields circuits with a speed advantage at a cost in area. The designer may tune the synthesized circuit for the most favorable area-speed tradeoff. The multi-level approach is most effective for complex combinational circuits where it yields circuits more than twice as fast as comparable PLAs with a only a small increase in circuit area. Detailed comparisons of combinational logic synthesized using the MAMBO pipeline appear in Chapter 8.

CHAPTER 2

Comparison of Static and Dynamic CMOS Circuits

2.1. Design of Static CMOS Logic

To provide active pullup and pulldown of logic signals in static CMOS design, the circuit function is duplicated by n- and p-channel devices. The two groups of devices are logic duals of one another. The example circuit of Figure 2.1 realizes the NAND function of five inputs.

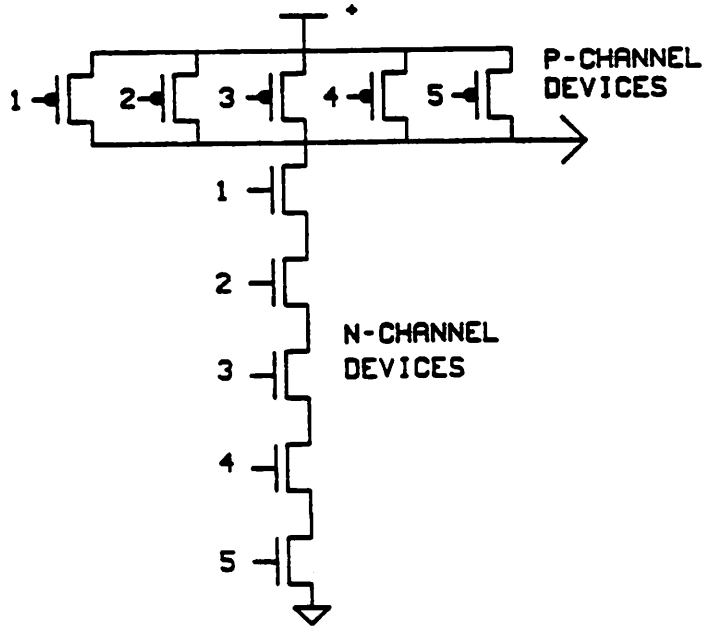


Figure 2.1: Static CMOS NAND Function of 5 Inputs

A conducting path is opened between GND and the output node when all five inputs are asserted high. By contrast, a path between V_{DD} and the output node exists when any one of the p-channel devices is asserted low. Each input signal drives two gates, an n-channel and a p-channel device.

2.1.1. Ratioing N- and P-Devices

For the NAND gate, at least one input signal must be low to drive the output node high. To drive the output low, all input signals must be high. Since both states are driven, it is important that the rise and fall times be approximately equal if overall circuit performance is to be optimized. It is standard practice to ratio the FETs to obtain this goal. There are two factors which exert first-order effects on gate delay. For particular values of V_{GS} and V_{DS}, the MOSFET current is directly proportional to K, a transconductance parameter, where

$$K_n = \frac{W_n}{L_n}\mu_n C_{ox} \tag{2.1a}$$

and

$$K_p = \frac{W_p}{L_p}\mu_p C_{ox} \tag{2.1b}$$

for the n-channel and the p-channel devices, respectively. Here it is assumed that the thickness of gate oxide is the same for both devices thus C_{ox}, the gate oxide capacitance per unit area, is identical for both devices. W and L represent the width and length of the MOSFET active area, respectively. The surface mobility of electrons in the p-type substrate, μ_n, is typically two to three times higher than the surface mobility of holes in the n-type substrate, μ_p. The exact ratio depends on a variety of factors, including substrate doping, and therefore depends on whether an n-well, p-well, or *twin-tub* technology is used. The second factor is the effective $\frac{W}{L}$ ratio of "on" devices between V_{out} and V_{DD} or between V_{out} and GND. By varying W and L of the series and parallel connected devices in Figure 2.1, both of these factors can be overcome. In the SPICE2 [nage75] simulations that follow it is assumed that $\mu_n = 2.5\mu_p$.

2.1.2. Fanout Loading Calculations

Since static CMOS gates consist of functions duplicated in p- and n-logic, each input signal must drive both an n- and a p-device. The first-order input gate capacitance of

such a static circuit is given by:

$$C_{in} = C_{ox}(W_n L_n + W_p L_p) \tag{2.2}$$

For an inverter, the p-channel device contributes about twice as much gate capacitance as the n-device. Therefore, a single inverter load in static CMOS is equivalent to approximately three inverter loads in NMOS technology and, in general, dynamic CMOS circuits show a smaller input capacitance than their static CMOS counterparts [pret85].

2.1.3. Design of NAND and NOR in Static CMOS

The logic functions NAND and NOR are realized more simply than AND and OR in static CMOS. This is because the unimplanted n-device turns on when V_{GS} is large positive, hence when the gate is high and the source is grounded. The p-device turns on when V_{GS} is large negative, hence when the source is tied to V_{DD} and the gate is low. That is, for both device types, when the input voltage increases the output voltage decreases. This contributes the basic inverting component. A positive logic NAND is fashioned by placing n-devices in series and p-devices in parallel. A positive logic NOR is built by placing the p-devices in series and the n-devices in parallel.

In practice, there is a bias toward building static CMOS gates in NAND form. The two factors which affect device sizing, series chain length and mobility differences, tend to cancel in the design of a NAND gate. For the NOR case the two factors are multiplicative, resulting in a large disparity in p- and n-device sizes, especially for high fan-in gates. The large p-devices add capacitance to the output node and consume circuit area. It is for this reason that some circuits, for example the gate matrix designs in the BELLMAC chip, [kang83a] have been built using only NAND gates.

2.1.4. Worst-Case Delays for Static NAND Gates

For a NAND configuration the worst-case delay time is T_{PHL}, the amount of time it takes the output to go from high to low. This definition is illustrated in Figure 2.2 (from [hodg83]). The delay time is given by:

$$T_{PHL} = \frac{C_T (V_{OH} - V_{OL})}{2 I_{D(avg)}} \tag{2.3}$$

where the total capacitance is given by:

$$C_T = N_O C_G + N_I C_{GDp} + N_I C_{GDn} + (N_I - 1) C_{GSn} + K_{eq} [N_I C_{DBp} + (2N_I - 1) C_{DBn}] \tag{2.4}$$

In Equation 2.4 N_O represents the gate fanout. I_D is inversely proportional to N_I, the number of inputs:

$$I_D = \frac{K / 2 (V_{GS} - V_t)^2}{N_I} \tag{2.5}$$

K is the process-dependent transconductance parameter.

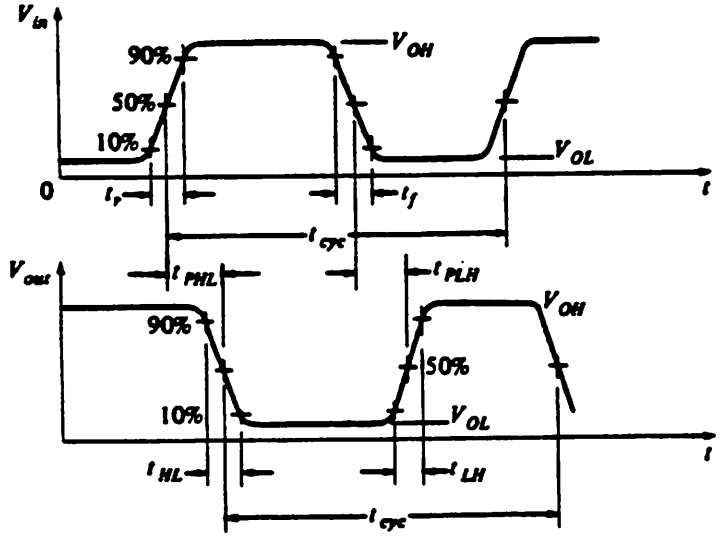


Figure 2.2: Delay Time Calculation for Static NAND

2.1.5. Relative Placement of Gate Input Signals

Consider a high fan-in NAND gate. The speed at which the gate makes the $1 \rightarrow 0$ transition depends on the speed at which all the n-devices react. By placing the fastest switching FET closest to the output node, the output voltage will begin to fall and the gate will begin to make the transition, even before the more distant transistors have fully turned on. It is important that the designer take advantage of this information in circuit layout. This concern applies only to the n-devices since the p-devices are in parallel:

hence there is no "closest" transistor to the output node.

2.1.6. Static CMOS Speed

Pullup and pulldown times for best and worst case SPICE2 simulations for a 5-input AND gate (NAND with inverted output) are tabulated in Figure 2.3. In the worst-case analysis a single input was switched, in the best-case simulation all inputs were switched in parallel. The sizes of n- and p-channel devices were adjusted to achieve roughly equal rise and fall times in the worst case. As with other simulations described here, the circuits were laid out and all parasitic capacitances were extracted and accounted for in the simulations. The SPICE2 models used for simulation appear in Appendix A.

5-Input AND	L_p / W_p	L_n / W_n	risetime (ns)	falltime (ns)
Best case	3/6	3/10	7.3	2.9
Worst case	3/6	3/10	9.8	10.0

Figure 2.3: Delay Times for 5-Input Static CMOS AND Gate

2.2. Design of Dynamic CMOS Logic

The design of dynamic circuits in CMOS offers considerably more layout flexibility than static methods. Dynamic methods rely on charge storage for correct operation. The most commonly employed dynamic methods use *two-phase* clocking (one clock). In the initial or *precharge* phase, output nodes are precharged to either logic high or low. In the second phase, called the *evaluate* phase, output nodes either remain stable or are allowed to make a single, unidirectional transition. With this latter constraint circuit "glitches" are avoided. This is important in charge transfer circuits, because once a node is improperly discharged it cannot return its valid state until the next precharge.

Dynamic circuits offer an advantage over static designs: fewer devices are needed in most cases since circuit function is not duplicated. Removal of the dual network makes design-for-testability easier [gonc83]. The logic used to realize the function, called the *core*, may be of either n- or p-devices. In the examples that follow n-cores are most

often used.

Dynamic circuit design also has several disadvantages. Some of these drawbacks are summarized:

- The electrical design of dynamic circuits is unquestionably more difficult than in the static case.
- The circuits must be simulated to determine the necessary precharge time.
- Dynamic circuits potentially suffer from a charge redistribution problem on the output node.
- It is especially important to consider the ordering of "internal" versus "external" signals to address this charge problem.
- Certain dynamic design styles are not logically complete.
- Dynamic circuits utilizing two different clocks can have circuit races if not carefully designed.

In the following sections each of these topics is considered with regard to the two most commonly used dynamic design techniques.

2.2.1. Dynamic CMOS with Domino Logic

The most basic dynamic design is a style termed *Domino* after Krambeck. et. al. [kram82]. This style involves the use of a single clock, denoted by ϕ. This clock controls a p-channel and an n-channel device: the former between circuit logic output and V_{DD}, the latter between the circuit logic and GND, as shown in Figure 2.4. In addition, a static inverter is required at the output of the gate. The Domino representation of a 5-input AND circuit is shown in Figure 2.4.

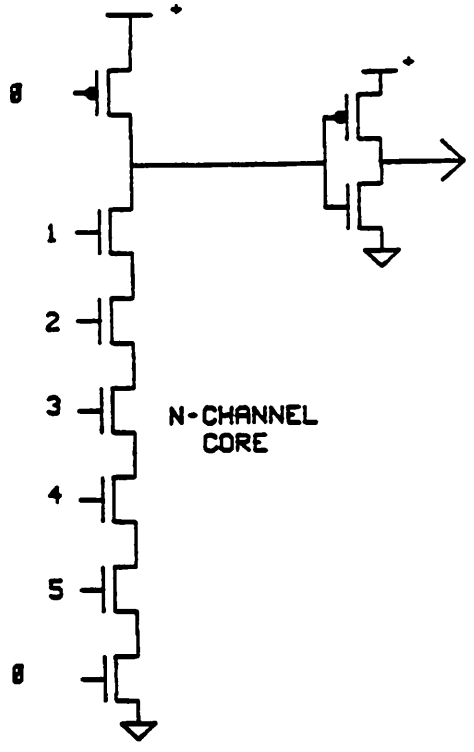


Figure 2.4: CMOS Domino 5-input AND circuit

While a standard (static) gate requires $2k$ devices to realize such a function, where k is the number of inputs, a Domino gate requires $k+4$ transistors. The core is composed of k devices: two additional transistors are gated by the clock, and two devices are required for the static output inverter.

The Domino gate works in the following way: During the precharge phase ϕ is held low. The p-channel pullup device charges the Domino core output node high and the core is isolated from GND. At the end of precharge the core node is at V_{DD}. In the evaluate phase ϕ goes high which connects the core to GND and isolates the core output from V_{DD}. If, during evaluate, inputs to the core devices are asserted such that a path is created between the output node and GND, then the output node makes a single $1 \rightarrow 0$ transition. Otherwise, the output node remains in its precharged (high) state.

This process is then repeated with the next precharge phase. The term "Domino" comes about from the analogy of setting up a line of dominoes (precharging) and then letting the first domino make a transition. This may cause a further transition which may, in turn, cause a succeeding transition and so forth— much like the effect of toppling a chain of dominoes.

2.2.2. Device Sizing in Dynamic CMOS

Dynamic logic is ratioless, which is to say that pullup and pulldown devices are not sized depending on the amount of current they draw, or their relative mobility. The static circuits just examined were ratioed in order to compensate for series stacking and for mobility variations so rise and fall times were about equal. In precharged logic only a single state transition is possible. To save circuit area, minimum devices can be used; to gain speed, wide devices can be employed in the core. The sizing of n-channel and p-channel devices are essentially independent tasks.

2.2.3. Static Inverter Requirement

The need for the static inverter is explained fully in the original paper on Domino logic. Briefly, the inverter is required because the precharge state brings the (n-core) output node high. If there is no inverter, a logic high will turn on the following n-channel device. At the beginning of evaluate, therefore, the n-core of a succeeding gate will be active and may be falsely discharged. This occurs because a momentary connection exists between the output of the driven gate and GND. Once this node is discharged it cannot be recharged until the next precharge phase. To prevent this, all input gates are required to be off at the beginning of the evaluate phase. It is therefore necessary to add an inverter in between precharged (high) nodes and driven n-channel (active high) devices.

2.2.4. Distinction Between External and Internal Signals

It is important in a dynamic design style, like Domino, to draw the distinction between an *internal* and an *external* signal. An internal signal is one which comes from a previous Domino gate. An external signal comes from a static source such as a latch. Internal signals must be "off" at the beginning of the evaluate phase. An external signal, however, is expected to be *stable* at the beginning of the evaluate phase— whether it is on or off. That is, internal signals are stable and off during precharge and may make a single transition during evaluation. External signals are expected to stabilize before the end of precharge and remain in their final state during evaluation. Thus, another consideration in the calculation of the precharge interval is the length of time it takes external signals to stabilize.

In dynamic circuits relative signal placement is also important. Since external signals must be stable before the beginning of evaluation, they are placed closest to the output node. The internal signals come next and their placement depends on the order in which they switch. Note that if a series external signal is "off" during evaluation the state of any internal signals does not matter, since the external signal blocks any output node transition.

2.2.5. The Charge Redistribution Problem in Dynamic Circuits

It has already been pointed out that in dynamic circuits care must be taken to avoid improper discharge of a precharged node. Charge may also be lost from the output node due to the phenomenon of charge redistribution. This problem is most evident in circuits with a high fan-in of largely internal signals. Figure 2.5 illustrates the problem.

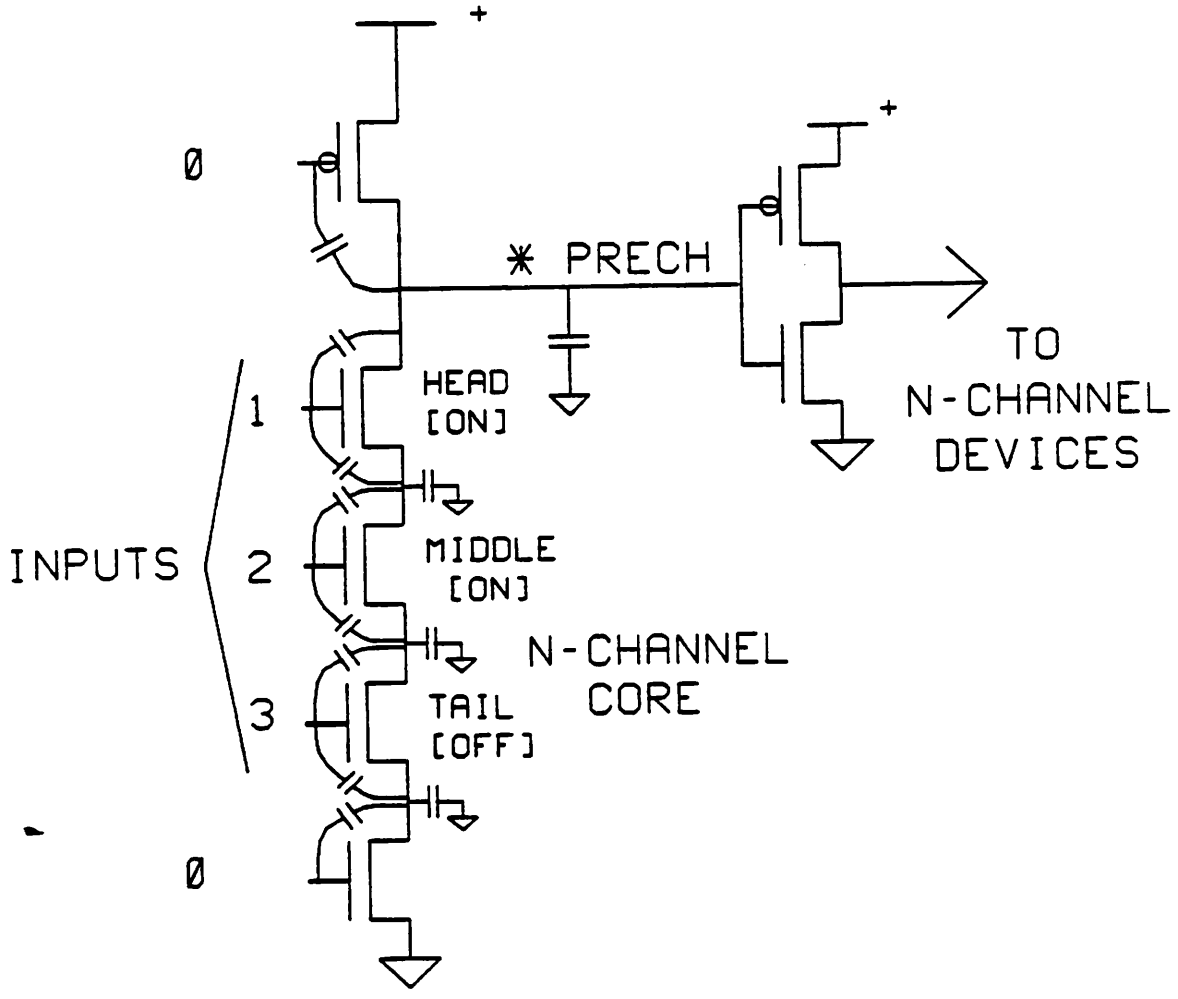


Figure 2.5: The Charge Redistribution Problem

Assume all signals are internal. If, during evaluation, the two devices closest to the output node turn on but the third device remains off, the output node should remain high. However, when the internally driven FETs turn on, charge stored on the output node capacitance flows into the source/drain capacitances of the n-devices. The output charge is thus split across several nodes. If the collective source/drain capacitances are of the same order as the gate capacitance of the static inverter, the charge lost from the precharged node may cause the voltage there to drop and the static inverter to make a false transition.

The charge redistribution problem can be quantified and a simple criterion for the onset of charge redistribution is now defined. This criterion is used by the MOSMESH program, described in Chapter 4, to determine the presence of a charge redistribution problem. The precharged capacitance (*prech*) lies on the node designated by an asterisk in Figure 2.5. All other capacitances represent parasitics (*para*). A charge redistribution (*CR*) problem is defined to exist if:

$$\frac{prech + (head)\times para}{prech + (head + middle)\times para} < \frac{V_{TH}}{V_{DD}} \tag{2.6}$$

where:

head: number of drain nodes from the core devices which touch the prech node

middle: number of source/drain nodes from the core devices which are not *head* and which do not touch the grounded pulldown device gated by the clock

V_{TH}: the switching threshold of the output buffer/inverter

V_{DD}: positive power supply rail

The buffer capacitances which contribute to the prech capacitance are:

$W \times L_p \times C_{OXp} + W \times L_n \times C_{OXn} +$ (gate capacitance)

$C_{GDOp} \times W_p + C_{GSOp} \times W_p +$ (s/d overlap cap)

$C_{GDOn} \times W_n + C_{GSOn} \times W_n +$ (s/d overlap cap)

$C_{GBOn} \times L_n + C_{GBOp} \times L_p$ (gate/bulk overlap cap)

The pullup device also contributes to the prech capacitance:

$C_{jp} \times Area_{pullup} + C_{JSWp} \times Perimeter_{pullup} +$ (bulk capacitance)

$C_{GDOp} \times W_{pullup}$ (overlap cap)

The parasitic contributions come from two sources: from junction capacitances of the source and drain to the substrate and from source/drain overlap capacitances. Note that the parasitic capacitances contributed by the head devices add to the prech capacitance. The capacitances given below are per source or drain region per device.

$C_{Jn} \times Area_{core} + C_{JSWn} \times Perimeter_{core} +$ (core bulk cap)

$C_{GSOn} \times W_{core}$ (s/d overlap cap)

A breakdown of the parasitic contributions is shown in Figure 2.6.

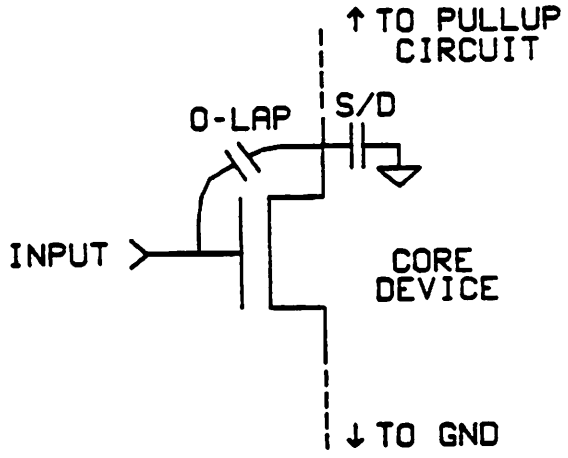


Figure 2.6: Parasitic Contributions

There are two solutions to the CR problem, both involve keeping the ratio of output node capacitance to parasitic input capacitance large. The first method is to make the static inverter larger. This has the advantage that the circuit can now drive a large fanout. Adding capacitance slows the switching time, however. Small circuit area, one of the advantages of dynamic designs, is also compromised. The second approach is to decrease the size of the core devices. If these devices are of minimum size, however, they represent a higher resistance and thus cause the gate to switch more slowly. Thus, without significantly modifying the circuit, the only solution to the redistribution problem requires slowing down the circuit. This is a drawback which both Domino style circuits and NORA circuits, described in Section 2.3, suffer.

2.2.6. Domino Circuits Are Not Logically Complete

The addition to the dynamic circuit of the interstage inverter means that Domino gates are either AND or OR in function. Neither configuration can produce an inversion and

hence the Domino family is not logically complete by itself. This drawback can be overcome by moving the inversions to the last Domino stage. The last stage has no restrictions on its output, assuming it drives standard static CMOS logic. This moving of inversions, also called *bubble pushing*, can be accomplished by expression-tree transformation. The software tool MGMG, described in Chapter 4, can transform an arbitrarily deep expression tree to AND—OR form.

2.2.7. OR Gates Preferred in Dynamic CMOS

In contrast to static CMOS, where device ratioing for mobility plays a role, the practical gate for implementing dynamic circuits is the OR function (n-core). Since the single p-device is only active in precharge, it is not affected by the n-core devices (which are only active in evaluate). It is preferable to arrange the n-devices in parallel to reduce the resistance path between output node and GND. Parallel n-devices give the OR function. By creating the same device configuration with a p-core in place of the n-channel devices, the AND function is constructed. Typically, Domino design is n-core only.

The pure n-core OR function and the pure p-core AND function have the additional advantage over their counterparts of being immune to the charge redistribution problem. This is because of the single device between output node and clocked device; there are no internal nodes for redistribution.

2.3. Dynamic CMOS Design Using NORA

A style of dynamic CMOS design which utilizes both n- and p-core gates in an alternating pattern has been described by Goncalves and DeMan [gonc83]. This style is more complicated than the Domino approach, which it is based on. NORA logic uses two clocks, commonly denoted by ϕ and $\bar{\phi}$. The name NORA comes from *NO RA*ce logic. A NORA circuit is shown in Figure 2.7.

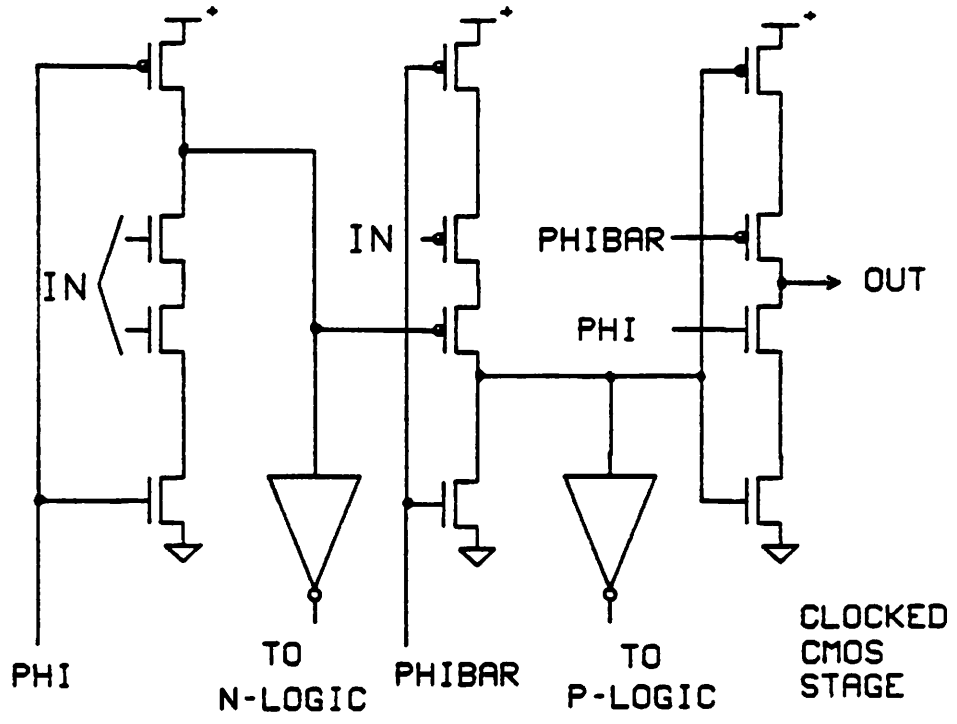


Figure 2.7: A Simple NORA Circuit

The first gate in this figure resembles a Domino gate except that the interstage inverter is not used. NORA obviates the need for such inverters by requiring a p-core gate follow an n-core one. Alternatively, an inverter could be used to connect two n-core gates in succession in which case NORA reduces to the Domino style. By eliminating the need for an interstage inverter, both n- and p-core gates can be built realizing the NAND and NOR functions. Since negations can also be realized it would appear at first glance that NORA designs are logically complete. This is not the case. As a result of the construction rules to preserve the racefree properties of NORA, explained below, the designer realizes no greater design flexibility from NORA circuits than from standard Domino designs. However, because the static inverter is not required, the overhead for a NORA circuit is reduced: NORA gates require only $k+2$ devices to implement a k-input, single-output function.

Like Domino logic, NORA utilizes a precharge and an evaluate phase. Unlike Domino, two clock phases are employed. This allows pipelining of interconnecting stages but also

introduces the potential problem of circuit races due to skewed clocks. In Domino logic circuit races are not possible since all gates are controlled from the same clock. There *is* a problem of clock distribution about a large circuit, a problem common to all dynamic designs.

2.3.1. The C²MOS latch in NORA logic

By employing a C²MOS (*clocked CMOS*) latch between pipelined circuit stages one can guarantee that races are avoided. This latch was first proposed and analyzed in [suzu73]. Figure 2.8a shows the latch in its most common configuration.

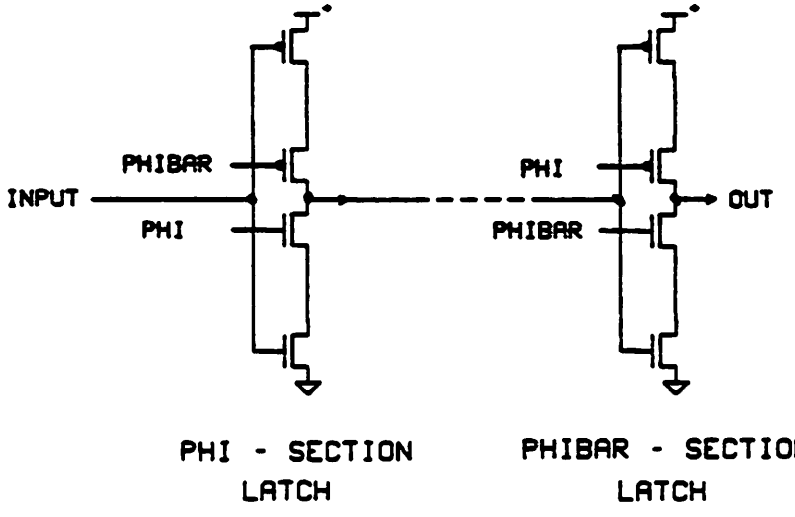


Figure 2.8a: C²MOS Latch

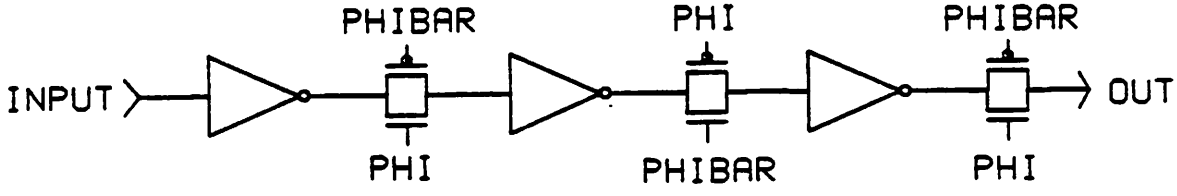


Figure 2.8b: Standard CMOS Transmission Gate

This latch, which uses four transistors, is used in place of the CMOS transmission gate, shown in Figure 2.8b. In Figure 2.9 SPICE2 simulation waveforms are shown for a shift register using first a pair of simple pass gates and then a pair of C²MOS latches. It can be

seen that the former configuration is sensitive to clock skew. As skew is increased logic levels are compromised. This occurs because in the skew period the transfer gate is neither on nor off, therefore logic levels are undefined. The skew period is that interval when ϕ and $\bar{\phi}$ are both 1 or both 0. By comparison, the clocked latch is always in a defined state. Note that the clocked CMOS latch inverts its input data.

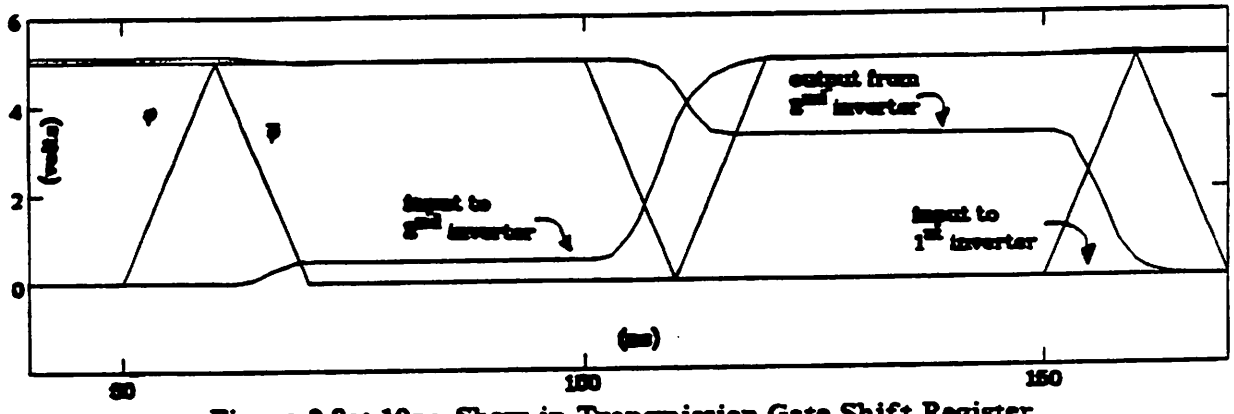


Figure 2.9a: 10*ns* Skew in Transmission Gate Shift Register

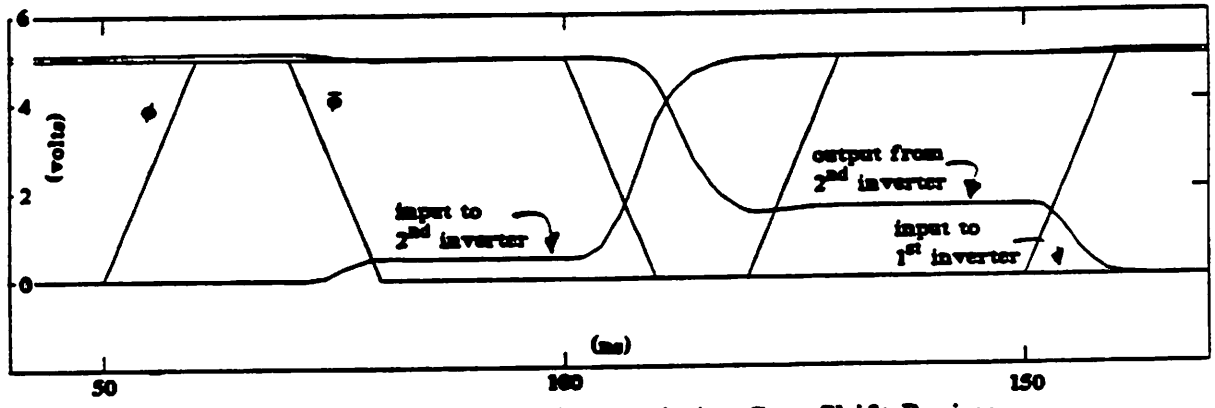


Figure 2.9b: 20*ns* Skew in Transmission Gate Shift Register

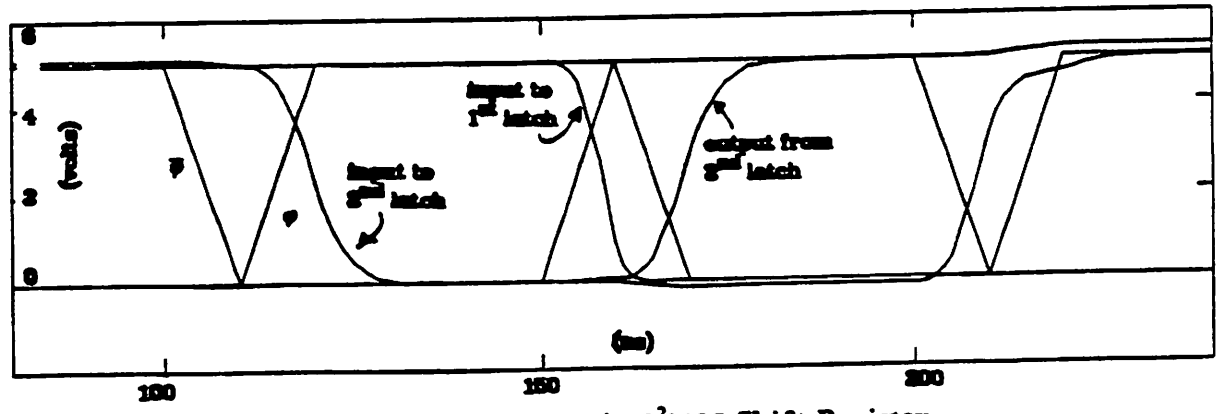


Figure 2.9c: 10*ns* Skew in C^2MOS Shift Register

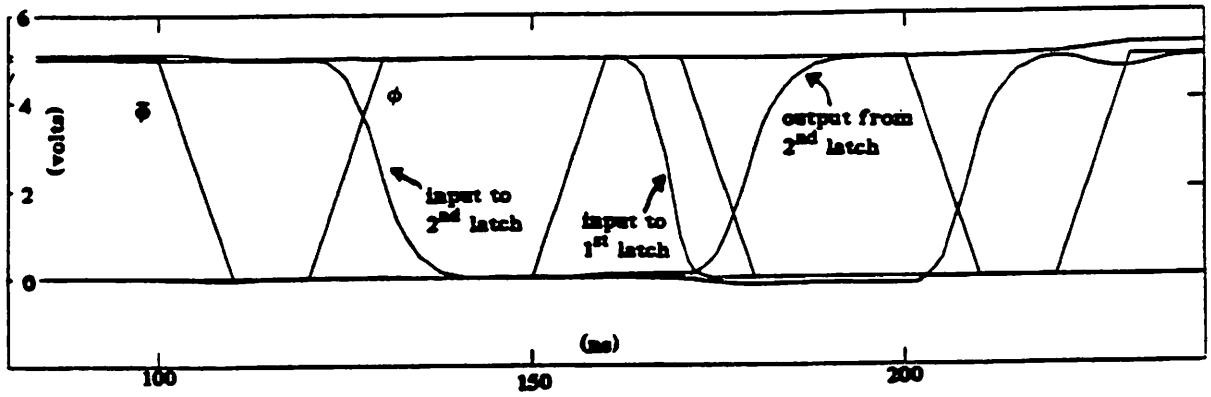


Figure 2.9d: 20*ns* Skew in C^2MOS Shift Register

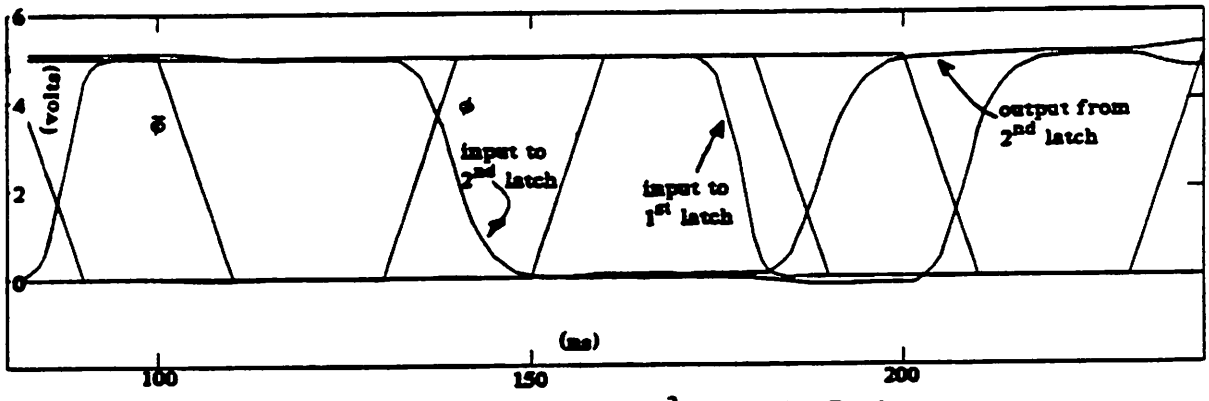


Figure 2.9e: 30*ns* Skew in C^2MOS Shift Register

The operation of the clocked CMOS latch is shown in Figure 2.10.

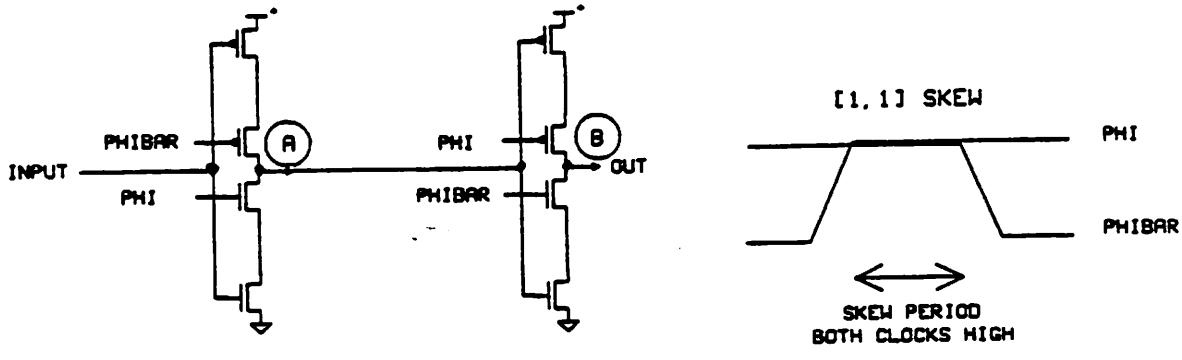


Figure 2.10: Operation of C^2MOS Latch

If a 0 is shifted into the first latch, then it is only possible for node B to take a $1 \rightarrow 0$ transition if node A makes the $0 \rightarrow 1$ transition and $\overline{\phi} = 0$ and $\overline{\phi} = 1$. $\overline{\phi}$ must equal 0 in

order for node A to make the transition high, but it must be 1 so that node B can make the transition low. Since $\bar{\phi}$ can only be either on or off such a skew problem cannot affect the latched signals. This example of skew is called *(1,1) skew* because it is analogous to the pass gate example with both ϕ and $\bar{\phi}$ high. A similar potential race condition exists in the case where ϕ and $\bar{\phi}$ are both low. This is called *(0,0) skew*. Again, the operation of the C^2MOS latch is immune to these skew conditions.

Unfortunately, it is still possible to have skewed-clock-induced races in C^2MOS circuits. If the V_t's of the complementary devices are mismatched then there exists a voltage range, $V_{\cdot}$, where the n-device is beginning to turn on and the p-device is not yet off, and vice versa. Notice in Figure 2.10 that ϕ drives an n-channel device in the first latch, but a p-channel device in the second latch. This insures that the latches work on opposite phases. However, it also means that the conditions $\phi = 0$ and $\phi = 1$ are not mutually exclusive. It is still possible to have clocked feedthrough between latches. The SPICE2 simulation result given in Figure 2.11 shows this effect when V_{tp} is −2.0V and V_{tn} is +0.5V. The C^2MOS latch is immune to clock skew but is still sensitive to V_t mismatches.

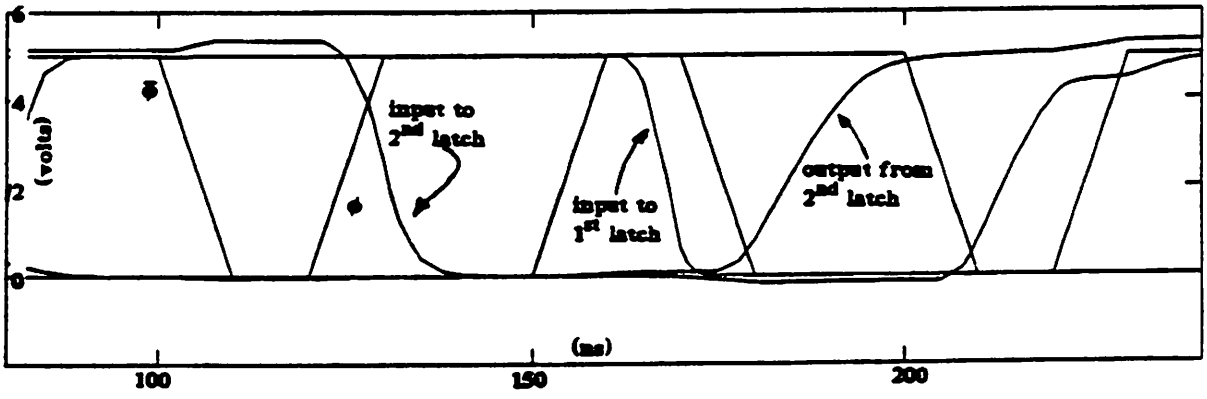


Figure 2.11: 20*ns* Skew in C^2MOS Shift Register with V_t Mismatch

2.3.2. Pipelining NORA stages

In the NORA style stages are coupled via the clocked CMOS latches. Each stage, which may be composed of an arbitrary number of n- and p-core gates, is clocked by ϕ and $\bar{\phi}$ signals. ϕ sections are interleaved with $\bar{\phi}$ sections to create a continuous pipeline. A ϕ

section is in precharge when $\phi = 0$ and $\bar{\phi} = 1$. A $\bar{\phi}$ section precharges under the opposite conditions. This allows the stages to be connected as shown in Figure 2.12 [gonc83].

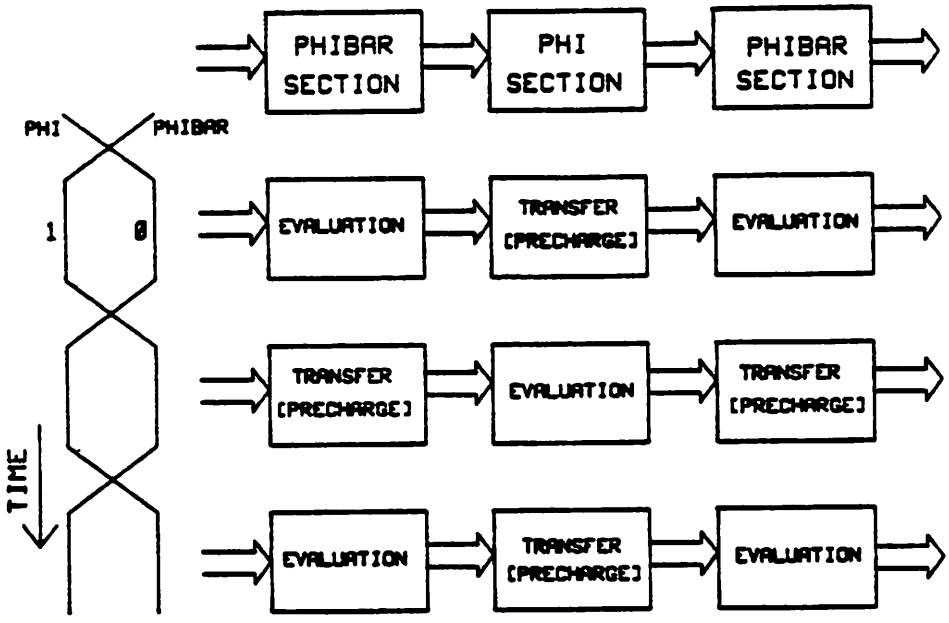


Figure 2.12: Pipelined NORA Stage

By using this technique, a result becomes available at the end of each evaluation phase, thus at the end of each clock cycle. It is assumed that the ϕ and $\bar{\phi}$ sections are of roughly equal complexity and, therefore, precharge and evaluate times will equal each other. In NORA, half the clock cycle is spent in precharge and half in evaluation. Domino logic, which is not pipelined, does not require an even split between precharge and evaluation phases. It is important in speed comparisons with static logic to factor in the precharge or setup time of dynamic circuits.

2.3.3. Construction Rules to Preserve Racefree Properties

In order to preserve the "racefree" properties of NORA conferred by the C^2MOS latch a number of construction rules must be observed. As summarized by Goncalves and DeMan [gonc83] the rules are:

(1) There is an even number of static inversions between the last dynamic stage of a section (ϕ or $\bar{\phi}$) and the C^2MOS output latch. (Precharge racefree.)

(2) There is at least one dynamic block placed in such a way that there is an even number of inversions between this dynamic block and the C^2MOS input latch. (Evaluation racefree.)

or

The total number of (dynamic and static) inversions between the input and output C^2MOS latches is even. (Evaluation racefree.)

These constraints are in addition to the requirement of a static inverter to eliminate the Domino internal delay problem.

The intent of these rules is to insure that within each (ϕ or $\bar{\phi}$) section the data signal is *controlled by only one clock*; it is independent of the opposite phase clock. The data signal must be immune from races in both precharge and evaluation phases. For the precharge phase consider the converse of the first rule: assume a single (static) inversion between the last precharged node and the output latch. For the n-core device in a ϕ section shown in Figure 2.12 the precharge node goes high in precharge as a result of a ϕ-controlled precharge FET. The output of the static inverter goes low and information on the latch output could be lost if the p-channel clocked FET, controlled by $\bar{\phi}$, was clocked late with respect to the ϕ-controlled precharge. This configuration depends on both clocks. Without the inversion (an even number of inversions) the input to the latch goes high. Now for the latched information to be lost the ϕ-controlled n-channel device would have to turn on. But both the precharge device and the controlling latch device are gated by the same clock. This clock is low in ϕ-section precharge by definition and therefore latched information is preserved.

When the clock is high, the ϕ-section is in evaluation. Consider the converse of the second evaluation racefree constraint: assume that between the output latch from a $\bar{\phi}$ section and the output latch from the next ϕ section there is a single static inversion. This configuration is shown in Figure 2.13.

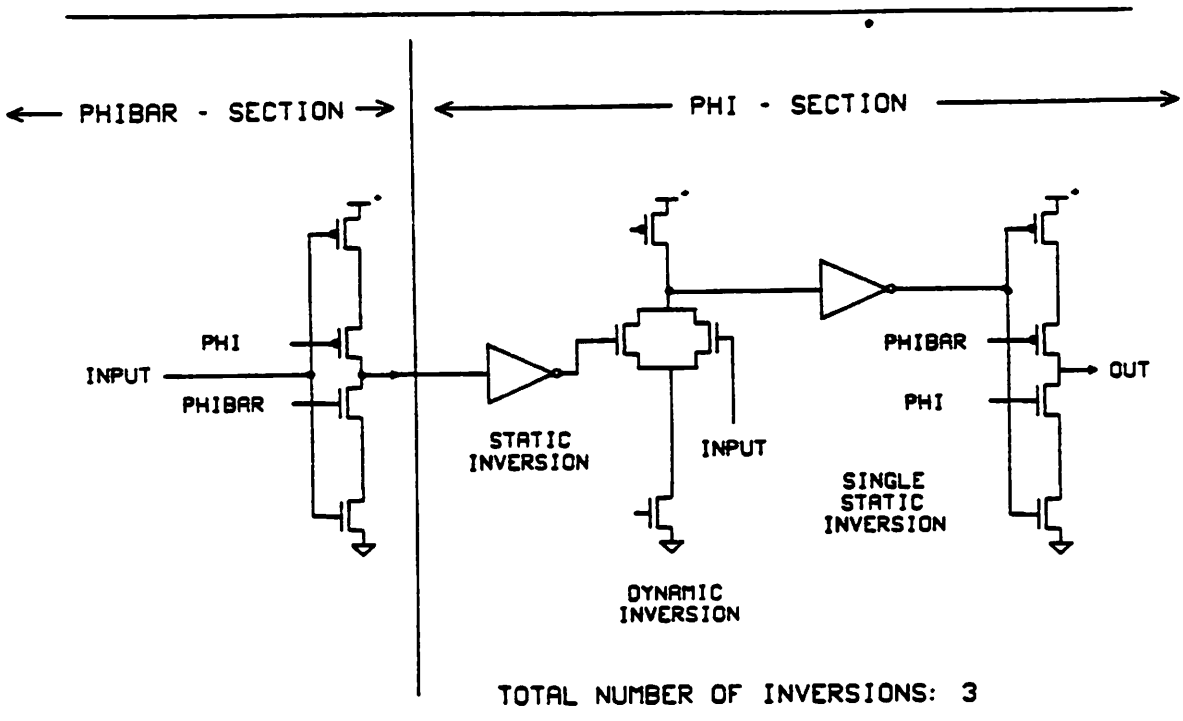


Figure 2.13: ϕ—section with Single Static Inversion (Contradiction of NORA rule 2.)

In ϕ-evaluation, ϕ goes high: if the output from the inverter is also high then information on the output latch may change because the n-channel clocked device gated by ϕ is on. But if the inverter output is high, the input must be low. The logic 0 was the result of the n-channel clocked device of the previous ($\bar{\phi}$ section) latch being on. This device is gated by $\bar{\phi}$. Again, if skew exists between the two clock phases, latched information may be lost. With an even number of interstage inversions it is only possible for one latch to be "on", allowing its output to reflect changes at its input. Because the ϕ and $\bar{\phi}$ latches are active on opposite phases, the same clock that activates the ϕ latch deactivates the $\bar{\phi}$ latch. The racefree properties are preserved because data transfer depends only on a single clock.

The first evaluation racefree rule guarantees that the racefree properties are preserved by assuring that a dynamic block is gated by the same clock that gates the input C^2MOS latch, as illustrated in Figure 2.14.

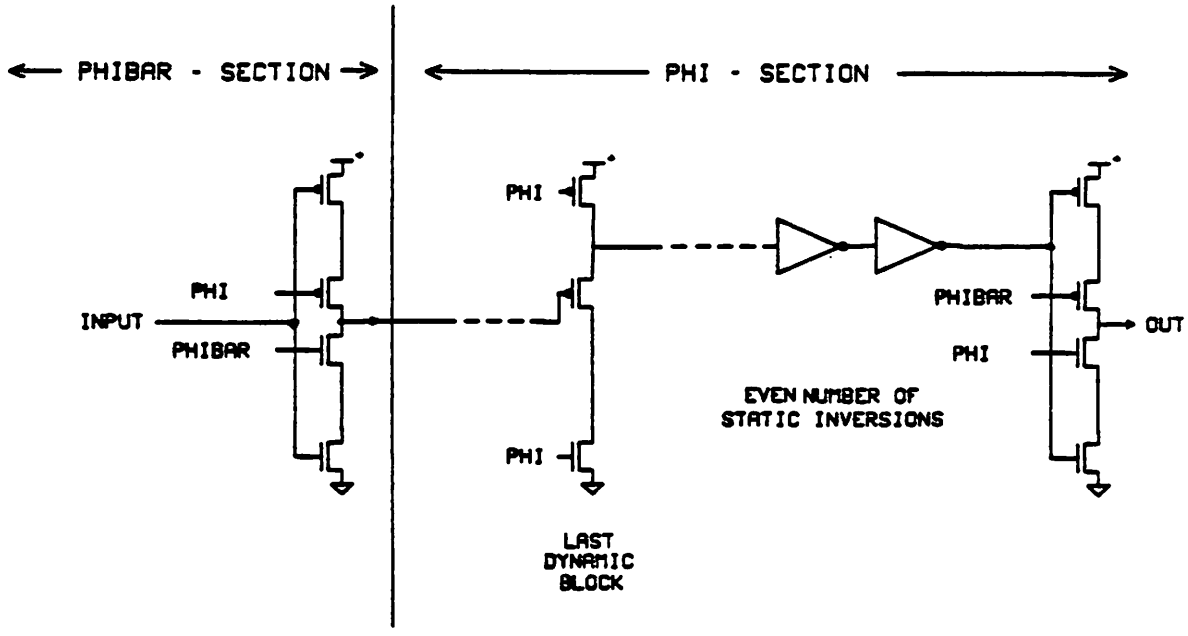


Figure 2.14: Precharge Racefree Rule: Even number of static inversions between last dynamic stage and output latch.

If this dynamic block is the last dynamic block of the section then it must be followed by an even number of (static) inversions- as required by the precharge constraint. If the dynamic block is not the last block then it is followed by an even or odd number of dynamic blocks. If there are an odd number more blocks, then the total number of inversions is even, which is the second evaluation racefree rule. If the number of succeeding dynamic blocks is even, then there must be an even number of inversions between this logic block and the C^2MOS output latch. This again assures that data signals depend only on a single clock.

It is often possible to change a circuit which violates these constraints into one which obeys the rules. Three possibilities are suggested [gonc83]:

1) A static inversion can be converted into a dynamic one.

2) A static inversion can be converted to a C^2MOS latch.

3) The static inverter can be placed after the C^2MOS latch.

2.4. Special Design Considerations in Dynamic CMOS

In this section three aspects of CMOS design particular to dynamic logic are examined. First, inverter buffer sizing is considered with regard to noise margin, delay, and charge redistribution. Second, the effect of an added static pullup to the precharge node is described. Finally, a stacking scheme is examined which can often speed up a high fan-in series gate. Such circuits are typically slow and, therefore, speedup is especially important.

2.4.1. Noise Margins in the CMOS Inverter

A static inverter is employed to connect stages in Domino and NORA logic. The noise margin of this inverter is of particular importance in the design of dynamic circuits. The high and low noise margins used in this analysis are defined as:

$$NM_L = V_{IL} - V_{OL} \tag{2.7a}$$

and

$$NM_H = V_{OH} - V_{IH} \tag{2.7b}$$

respectively. Following a standard text, for example [hodg83], the points V_{IH} and V_{IL} on the voltage transfer curve are defined by $dV_{out} / dV_{in} = -1$. By solving for this derivative and using the result in the appropriate drain equation the noise margin parameters can be obtained.

The drain current of a device in saturation is given by

$$I_D = \frac{k}{2}(V_{GS} - V_t)^2 \tag{2.8a}$$

where $V_{DS} \geq V_{GS} - V_t$. The drain current in the linear region is

$$I_D = \frac{k}{2}[2(V_{GS} - V_t)V_{DS} - V_{DS}^2] \tag{2.8b}$$

where $V_{GS} \geq V_t$ and $V_{DS} \leq V_{GS} - V_t$. To find V_{IH} and V_{OL}, assume the n-channel device is in the linear region and the p-channel device is in saturation. Thus $I_{DN(lin)} = I_{DP(sat)}$ or

$$\frac{K_n}{2}[2(V_{IH}-V_{tn})V_{OL}-V_{OL}^2] = \frac{K_p}{2}(V_{DD}-V_{IH}-V_{tp})^2 \tag{2.9}$$

In this equation and those that follow the absolute value of V_{tp} is used. K_n is kW_n / L_n and K_p is kW_p / L_p. Solving this equation for V_{IH} yields

$$V_{IH} = \frac{\pm[2K_p(V_{DD})-K_n(V_{OL})(2K_p(V_{tp}+V_{tn}))+(K_n-K_p)V_{OL}]^{1/2}}{K_p} \tag{2.10}$$

$$\frac{+K_p(V_{tp})-K_n(V_{OL})-K_p(V_{DD})}{K_1}$$

Solving the dV_{out} / dV_{in} equation for a slope of -1 yields

$$V_{IH} = \frac{2V_{OL}+V_{tn}+(K_p / K_n)(V_{DD}-V_{tp})}{1+(K_p / K_n)} \tag{2.11}$$

By equating these two expressions and iterating, a solution can found simultaneously for V_{IH} and V_{OL} Results for various pullup/pulldown ratios are given in Figure 2.15 for the worst-case SPICE2 model. It can be seen that the NMOS device is, in fact, in the linear region while the PMOS device is saturated.

To find V_{IL} and V_{OH} assume the n-channel device is in saturation and the p-channel device is in the linear region. Thus $I_{DP(lin)} = I_{DN(sat)}$ or

$$\frac{K_p}{2}[2(V_{DD}-V_{IL})(V_{DD}-V_{OH})-(V_{DD}-V_{OH})^2] = \frac{K_n}{2}(V_{IL}-V_{tn})^2 \tag{2.12}$$

Solving this equation for V_{IL} yields:

$$V_{IL} = \frac{\pm[(V_{OH}-V_{DD})(2K_n V_{tn}+(K_p-K_n)V_{OH}-(K_p+K_n)V_{DD})]^{1/2}}{K_n} \tag{2.13}$$

$$\frac{-K_n V_{tn}-K_p(V_{OH}+V_{DD})}{K_n}$$

Solving the dV_{out} / dV_{in} equation for a slope of -1 yields:

$$V_{IL} = \frac{2V_{OH}-V_{DD}-V_{tp}+(K_n / K_p)V_{tn}}{1+(K_n / K_p)} \tag{2.14}$$

By equating these expressions and solving as before, values for V_{IL} and V_{OH} are obtained. Results for various pullup/pulldown ratios are given in Figure 2.15. It can be seen that the NMOS device is in saturation while the PMOS device is in the linear region of operation.

A factor related to noise margin is the $V_{in} = V_{out}$ point, sometimes called V_{TH} or the *logic threshold voltage*. This point occurs between V_{IL} and V_{IH}, when both devices are in saturation, thus $I_{DN(sat)} = I_{DP(sat)}$ or

$$\frac{K_n}{2}(V_{in} - V_{tn})^2 = \frac{K_p}{2}(V_{DD} - V_{in} - V_{tp})^2 \tag{2.15}$$

V_{TH} values are summarized in Figure 2.15.

Noise Margins for Selected Device Ratios (in volts)								
L_n / W_n	L_p / W_p	V_{OL}	V_{IL}	V_{IH}	V_{OH}	NM_L	NM_H	V_{TH}
3/4	3/24	0.38	2.96	3.55	4.25	2.58	0.70	2.93
3/4	3/10	0.50	2.05	2.99	4.56	1.55	1.57	2.50
3/4	3/8	0.52	1.84	2.85	4.62	1.32	1.77	2.39
3/16	3/24	0.53	1.58	2.66	4.70	1.05	2.04	2.25
3/14	3/12	0.53	1.12	2.31	4.85	0.59	2.54	1.98

Figure 2.15: Noise Margins versus Device Ratios

2.4.1.1. Effects of Device Ratioing on Noise Margin

For the static CMOS inverter case it is preferable to have symmetric noise margins and to place V_{TH} midway between V_{OL} and V_{OH}. For the SPICE2 parameters used in the simulations in this report, where μ_n is 2.5 times μ_p, it can be seen that ratioing devices to cancel mobility differences also gives symmetric noise margins and a good V_{TH}. This is shown in the second line of Figure 2.15. This device ratio also guarantees approximately equal rise and fall times. It is the *ratio* of pullup to pulldown, not their absolute size, that determines noise margin.

In dynamic design there is the additional constraint of signal degradation due to leakage and charge redistribution. Also, precharged dynamic gates are unidirectional in nature, making either a $0 \rightarrow 1$ or $1 \rightarrow 0$ transition. Therefore best performance is obtained when noise margins are not equal.

Unfortunately, these additional constraints argue for opposing constructions. Consider an n-core dynamic circuit, precharged high. The accompanying buffer-inverter (if used) has its output driven low. The important transition is $0 \rightarrow 1$ at the output. To

obtain least delay, the p-channel device should be wide to bring the inverter output up quickly. This corresponds to a narrow NM_H. In theory this is fine, since the input node is precharged, via a p-channel device, to V_{DD}. However, if operating frequency is low or the dynamic gate is composed largely of internal signals, the precharged node voltage may dip. If NM_H is made too small the buffer may make a false transition. Therefore, a smaller p-channel device seems better. In addition, if a long, static pullup device supplements the dynamic precharge, a slower-acting inverter allows the static pullup to recover from charge redistribution or leakage.

It has been found by simulation that a smaller p-channel device and thus a wider NM_H is preferable. Circuits with potential charge redistribution problems cannot take advantage of the unidirectional transition of precharged logic. By making the p-channel device smaller, and thus slower, the n-channel device, often enlarged for added capacitance, can also be reduced in size. The result is a circuit with delay equivalent to a symmetric inverter, but which consumes less area. Inverters with ratios shown in the last two lines of Figure 2.15 were used in various test circuits with severe redistribution problems.

In circuits where a charge redistribution problem is known not to exist, for example in n-core NOR circuits or circuits with largely external signals, the p-channel device is widened to decrease delay. In this case the ratio of device widths is determined by the lower limit on the NM_H band that the designer wishes to tolerate. The lowest frequency of operation now determines the noise margin.

2.4.2. Addition of Static Pullups to Precharged Nodes

In [kram82] and [gonc83] it is pointed out that one can add high L/W static pullup devices to the precharge node of dynamic gates. This helps with low frequency operation where charge may begin leaking from a node. The static pullup device may be either p-channel (gate connected to GND) or n-channel (gate connected to V_{DD}), depending on whether the core is n- or p-type. A further reason for using static pullup devices is reduction of the charge sharing problem. In Domino logic, the output node is isolated from

the precharge node by an inverter which has some associated delay. If the devices are sized properly, the static pullup device can begin pulling the precharged node up, after a false trigger, before the inverter reacts fully. This requires careful ratioing since the designer wants the inverter fast on the one hand to react to solid transitions, but slow on the other hand to allow the pullup to aid on false transitions.

2.4.3. Speedup of Dynamic Logic By Gate Stacking

Finally, it is possible to speed up the operation of dynamic circuits by pyramiding or stacking devices. In Figure 2.16 a schematic of two 5-input AND gates is shown.

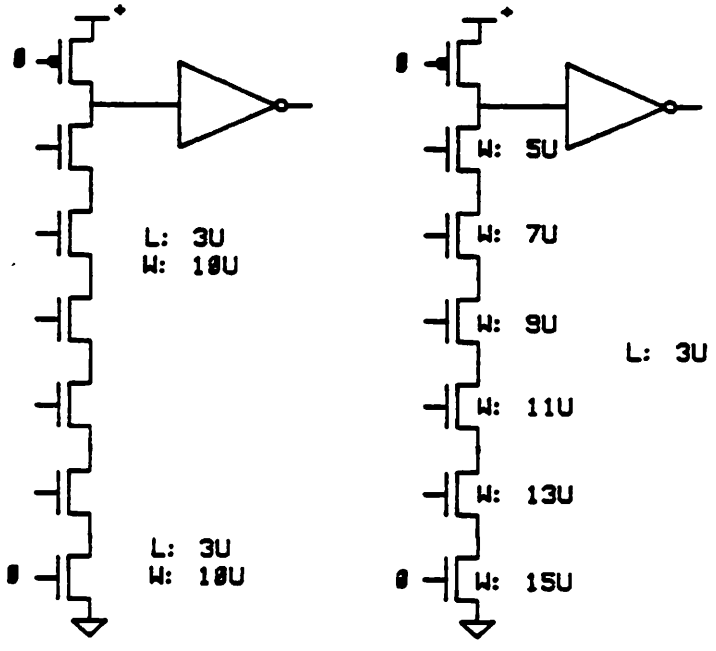


Figure 2.16: Comparison of Standard and Stacked AND gates

The gate on the left has six 10μ FETs in the n-core. The gate on the right has six FETs in the n-core with widths (from bottom to top) of 15, 13, 11, 9, 7, and 5 microns. The total source/drain area is identical for the two cases. The stacked circuit is slightly faster under worst-case conditions, as shown in Chapter 3. A circuit configuration similar to that shown on the right was employed in the design of the BELLMAC-32A and -32B, successors

to the original BELLMAC-32. This stacking technique was first mentioned by Shoji [shoj82]. Note also that the stacking technique reduces the charge redistribution problem, since the FETs closest to the output have the least capacitance. This allows the inverter follower to be smaller and accounts for part of the speedup seen. The main reason that the stacked circuit is faster, according to Shoji, is that the decrease in capacitance more than offsets the increase in device resistance due to narrower channels [shoj85].

2.5. Conclusions

Standard static CMOS is easier to design in than dynamic CMOS and is quite fast. NORA circuits require fewer devices to realize a given circuit function than static or Domino CMOS. They require more distinct signals, namely ϕ and $\bar{\phi}$, but in general consume less area. By stacking devices and proper ratioing of inverters and static pullups Domino and NORA circuits can be made faster than static CMOS designs. However, dynamic circuits must be precharged: static circuits have no such requirement. Dynamic designs have potential race problems due to clock skew. The addition of the clocked latch helps relieve this problem but does not eliminate it. In addition, the constraints placed by the NORA design style on the designer may prove cumbersome and yield relatively complex circuits for simple functions. Therefore, for simple circuits, standard static CMOS is best. Where complex circuit function is involved, for example in combinational logic or an ALU, and clock signals are readily available and needed by other on-chip circuits, dynamic Domino or NORA logic should be considered. While the design constraints required for dynamic implementation make manual design a complex task, these constraints can be accounted for in a computer program. By using computer generation of dynamic logic, the speed and area advantages of this approach can be achieved without the risk of electrical design problems.

CHAPTER 3

Simulation and Measurement of Static and Dynamic Circuits

The first stage in the design of a software tool for automated generation of integrated circuits is the characterization of the circuit building blocks which will be used to create the larger circuit. In this chapter a basic building block, the 5-input AND function, is used as a benchmark circuit to compare various static and dynamic CMOS implementations introduced in the previous chapter. Circuit performance is simulated using parameter values extracted from the MOSIS 3μ CMOS p-well process [mosi82]. These values have been translated into SPICE2 *level*=2 MOS model parameters and used in simulations. The SPICE2 models appear in Appendix A. In the second part of this chapter measured results of a dynamic CMOS test chip, fabricated in a 2μ n-well process, are presented. Charge redistribution effects and circuit delay times are reported and correlated with predicted values. The chip measurement procedures are explained fully in Appendix B. In the closing portion of this chapter, the design and fabrication of a Domino 32-bit ALU, which is part of a microprocessor chip, is described. This ALU was designed in the same style as that employed by the automated synthesis tools used in the MAMBO package.

3.1. Range of Circuits Simulated

The simulations described here address two main points: First, how does a straightforward static circuit compare in performance to a standard dynamic circuit as fan-in varies? Second, keeping fan-in constant, how do various extensions to dynamic circuits affect their performance relative to the static circuit and the basic dynamic implementation? To solve the charge redistribution problem examined in Chapter 2 the dynamic circuits must be modified. The effects of such modifications are also presented here.

3.2. Rationale for Choice of Benchmark Circuit

The basic benchmark is a 5-input AND function. The AND function was chosen over the OR because circuit delay varies strongly with increased AND-gate fan-in. This variation is not as strong with CMOS OR gates. A 5-input gate was chosen because device counts are nearly equal for most static and dynamic designs at this point and, hence, area should be roughly equal. The static AND gate requires 12 devices, the basic dynamic version uses 9 devices. The layout of the dynamic circuit needs additional area because of the clocked gates.

An attempt was made to examine the functions that are most troublesome to realize as high speed circuits. Most common dynamic logic design methods involve stringing together, either in series or in parallel, the even functions AND and OR. Since the speed of a dynamic logic gate depends on the length of its worst-case path from output node to voltage rail, the OR gate was not examined. In a dynamic OR gate the worst-case path is always through two FETs— the paralleled input FETs and the clocked gate. Thus the OR gate is always faster than the AND gate. In addition the OR gate has no charge redistribution problem because only one input device separates the core output from the clock. There are no parasitic source/drain capacitances. AND gates present problems and are examined exclusively here.

3.3. Simulation Technique

All circuits were simulated using the *level*=2 SPICE2 models. The core of both the static and dynamic circuits were laid out according to the 3μ design rules. Many of the modified circuits were laid out as well. The capacitances of all circuits were extracted and corrections were made where shortcomings with the current extraction tools were known. Where possible in the simulations initial conditions were established by cycling the circuit. When this proved impractical, due either to convergence problems or excessive simulation time, initial conditions were forced by use of the SPICE2 .ic option. All circuit inputs were conditioned by passing them through a minimum size inverter to decouple them from ideal

sources. The fanout of all simulated circuits was three inverters. Where multiple clocks were necessary, one clock was chosen as primary and assumed to come from an ideal source and all other clocks were derived from it.

3.4. Standard Static CMOS Benchmark

A schematic diagram of the static CMOS benchmark circuit appears in Figure 3.1. A layout plot of a static CMOS AND gate appears in Figure 3.2. The n-channel devices have a width of 10μ. It was found by simulation that roughly equal rise and fall times were obtained when the p-channel devices had a width of 6μ. This counteracts for a mobility mismatch between the two types of devices in the range of 2.5 - 3. The 5-input NAND gate drives a static inverter. The n-channel device is 4μ, the minimum width allowed by the design rules. The p-channel device is 8μ, again ratioed in width to give roughly equal rise and fall times.

The delay test for this circuit consisted of two parts. First the pulldown time was measured. The four inputs of the series devices nearest the output were driven on. The

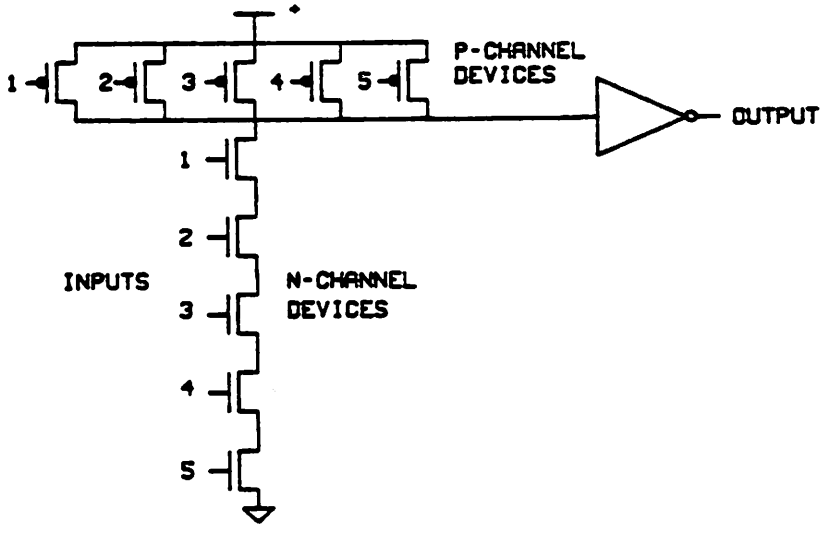


Figure 3.1: Schematic Static CMOS Circuit

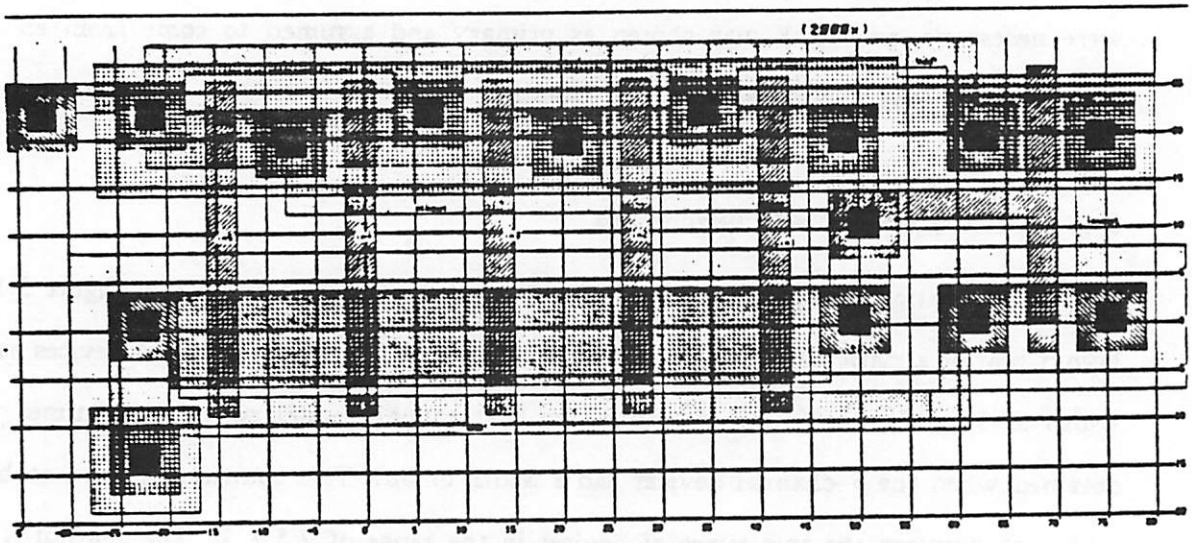


Figure 3.2: Layout of Static CMOS Circuit

input to the bottom device was initially off. This means that its counterpart p-device is on. The output node is charged high through this path. In addition the source/drain capacitances of the four n-channel devices closest to the output are charged high. After these nodes are charged, the fifth input goes high and the series output node begins to drop only after the series-leg nodes have discharged. The resulting delay time represents the worst-case pulldown time for the gate.

Pullup time is measured by first discharging all the series n-channel devices. This is achieved by driving all inputs initially high, so that the output from the series leg is low. The input to the fifth, or bottom, n-channel device is then switched off and the p-channel counterpart is switched on. Thus the NAND gate output is brought high by the a single p-device. This FET must charge the output node which now represents the collective source capacitances of the p-devices and the source/drain capacitances of four of the n-channel devices. The resulting delay time represents the worst-case pullup time for the NAND gate. Delay time is measured between the 2.5V level of the falling input signal and the 2.5V level of the falling output inverter.

In like manner, 2-input and 9-input static CMOS AND gates were simulated. In all cases the width of the n-channel devices was kept constant at 10μ. This value was used in the simulation of the dynamic circuits described later. In order to keep rise and fall times roughly equal, the p-devices of the 2- and 9-inputs gates were modified. Figure 3.3 summarizes the rise and fall times found for the static circuit as a function of number of inputs. The width of n- and p- devices is also listed.

Rise and Fall Times for Static CMOS AND Gates				
Fan-in	Risetime (ns)	Falltime (ns)	PD width (μ)	PU width (μ)
2	4.2	4.1	10	15
5	9.8	10.0	10	6
9	23.4	19.8	10	4

Figure 3.3: Delay Times for CMOS AND Gates

3.5. Dynamic CMOS Benchmark

The operation of dynamic circuits was examined in detail in Chapter 2. A schematic diagram of a basic dynamic circuit performing the 5-input AND function is shown in Figure 3.4.

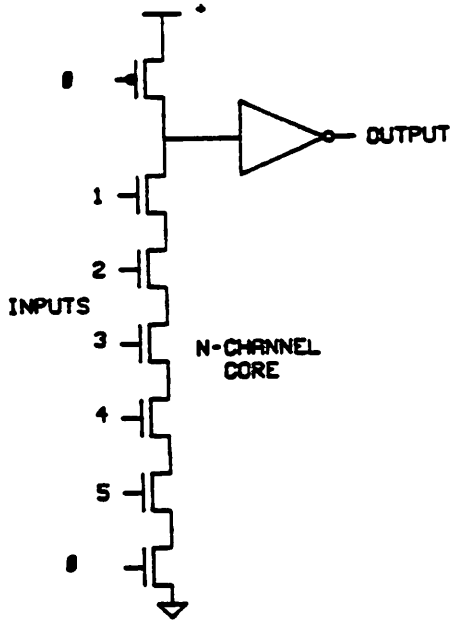


Figure 3.4: Schematic of Basic Dynamic Circuit

A layout of the basic circuit is shown in Figure 3.5.

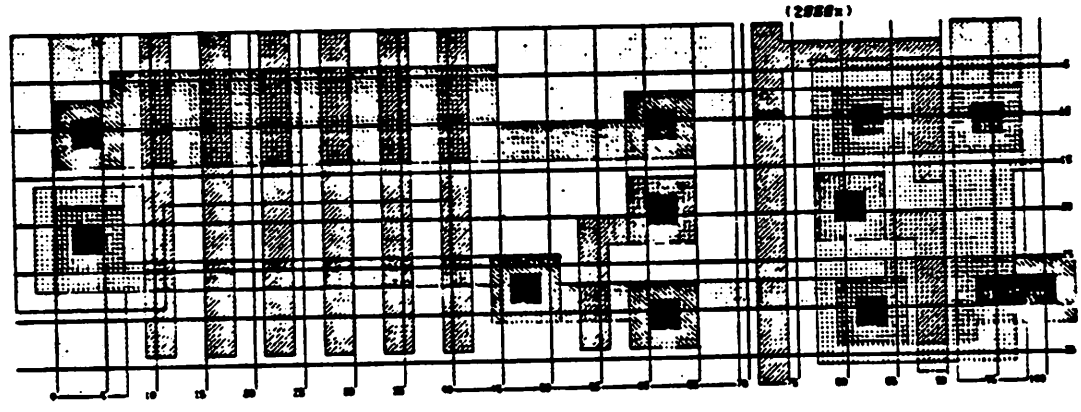


Figure 3.5: Layout of Basic Dynamic Circuit

The basic dynamic configuration was evaluated for two implementations. A circuit employing 10μ-wide devices in the n-core was laid out to compare dynamic and static speeds. A 4μ-wide n-core series chain was also simulated to provide an area-compact benchmark circuit. These two sets of simulations are used to bracket the design space for

a typical dynamic circuit.

3.5.1. Test Regime for Dynamic Circuits

Since these dynamic circuits are precharged high (*i.e.* the output after the static inverter is low) the only transition time to measure is the $0 \rightarrow 1$ risetime at the inverter output. Dynamic circuits, however, may suffer from charge redistribution and the performance in this regard must also be verified.

3.5.1.1. Dynamic Speed Test

For the speed test all input signals are assumed to be *external*. External signals may change during precharge and must be stable in evaluate. For a worst-case analysis of an externally-driven gate, it is assumed that all inputs are high so that all n-channel devices are on. During precharge the p-channel clocked device is on and deposits charge on the core output node as well as on the source/drain capacitances of all the core devices. When the clock signal goes high, indicating the beginning of the evaluate phase, a path is opened between the core output node and GND. The capacitance of the output node as well as all source/drain nodes must drain off before the output node goes low. The static inverter output is driven high. This represents the worst-case delay through the dynamic gate. Delay is measured from the 2.5V value of the rising ϕ signal and 2.5V signal of the rising inverter output node.

3.5.1.2. Dynamic Charge Redistribution Test

For this test all signals are assumed to be *internal*. Internal signals are those signals which are driven by the outputs of other dynamic gates. Such gates are stable and off during precharge; they may turn on during evaluate. All source/drain core nodes are grounded through use of the initial condition option in SPICE2. The core output node is then precharged by bringing ϕ low. Because all inputs are off, internal source/drain capacitances remain at ground potential. When ϕ goes high at the beginning of evaluate, the

charge stored on the output node begins to redistribute through the core devices. If all but the bottom input now turn on, the worst-case charge redistribution case occurs. The charge on the output node is split between four source/drain capacitances. If no modification to the circuit is made this will result in a *false trigger*, an unwanted transition of the core output node from high to low, causing the inverter output node to be driven high. In order for a circuit to pass the charge redistribution test the voltage on the output node should not rise above a specified value, taken to be 0.3V for this technology.

3.5.2. Speed Comparison of 4μ- and 10μ-Wide Dynamic Circuits

Four micron- and ten micron-wide n-core dynamic AND gates were laid out. It was assumed that all signals were external so that no charge redistribution problem exists. If all inputs were internal, simulations show that a charge redistribution problem would exist for even a three-input AND gate. The results of the simulations are summarized in Figure 3.6 below and in graphic form in Figure 3.7. The 4μ-wide devices represent a compact circuit while the 10μ-wide devices are meant to give an indication of the speed of dynamic gates where area is not a primary consideration.

Risetimes for 4μ- and 10μ-Wide Dynamic AND gates		
Fan-in	Risetime (ns) 4μ-wide	Risetime (ns) 10μ-wide
2	7.4	4.5
5	13.3	7.7
9	19.8	12.2

Figure 3.6: Risetimes for Dynamic AND Gates

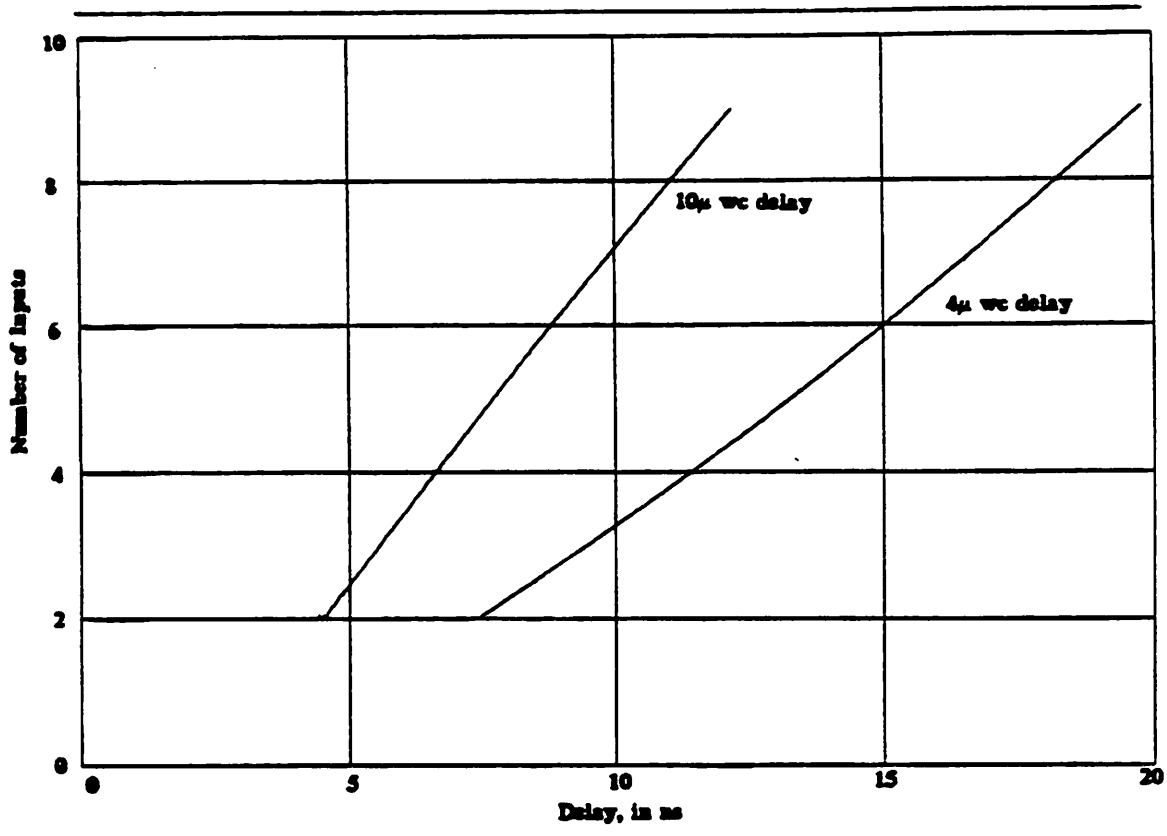
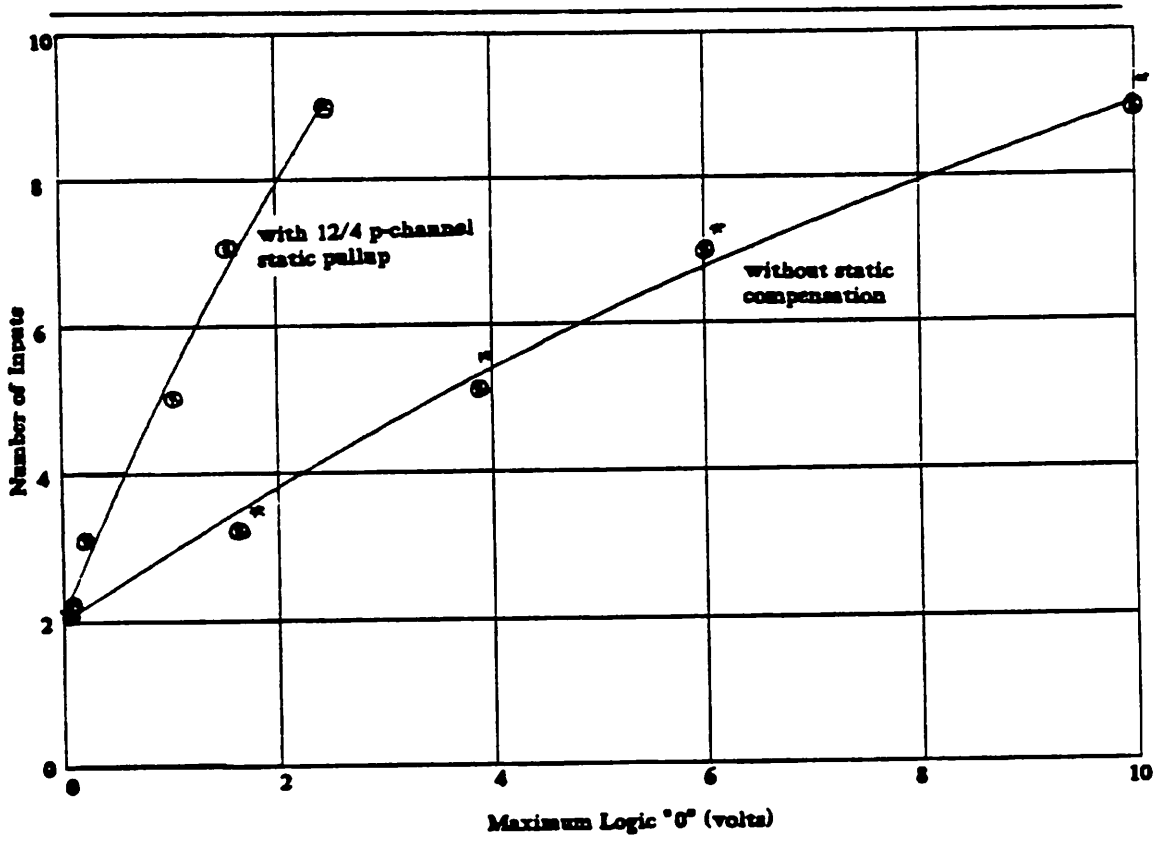


Figure 3.7: Comparison of 4μ- and 10μ-Wide Devices With No Charge Redistribution Problem

3.6. Simulation of Dynamic Circuits with Charge Redistribution

If not enough of the inputs of a given gate are external and can therefore be placed at the top of the input core to reduce charge redistribution, then it is necessary to compensate for the problem in another way. There are two possible solutions. First, the ratio of capacitance of parasitic source/drain nodes to the output node can be altered— either by making the input devices smaller or the output static inverter bigger. Both of these modifications result in a slower-switching circuit. The second solution is to add a static device to pull up the precharged node. In the n-core case a long, weak p-channel device with grounded gate is added to the core output node. It has been found by simulation that a 12μ-long, 4μ-wide p-channel device is more than sufficient to counter the worst-case

charge redistribution problem. Figure 3.8 presents a graph showing how the charge redistribution problem is cured by addition of the weak p-channel pullup.



*Value at 100ns after switching; output voltage still increasing

Figure 3.8: Comparison of Output Voltage Due to Charge Redistribution With and Without Pullup Compensation

Because the p-device is always on it opposes the pull down of the core output, hence when the node *should* fall there is some additional delay. However, this circuit can be optimized to remove the extra pulldown time by ratioing the devices in the output inverter. Figure 3.9 shows the worst-case speed characteristics for circuits with charge redistribution problems. The highest voltage that the inverter output reaches on false trigger is also listed. The values in this figure are overly conservative because they assume that all inputs are at once internal and external. These values therefore give a loose bound on worst-case circuit performance.

Risetimes for 10μ-Wide Dynamic AND Gates with Charge Redistribution		
Fan-in	Risetime (ns)	Peak Voltage False Trigger (V)
2	5.2	0.003
5	9.2	0.105
9	15.5	0.244

Figure 3.9: Risetimes for Dynamic Gates with Charge Redistribution

A comparison between different cases of standard dynamic AND gates is presented in graphical form in Figure 3.10. The leftmost curve represents a "best" case where no charge redistribution problem exists and the externally-driven signals happen to all be in the off state. This means that switching is rapid because only the precharged node itself has to be discharged: all parasitic source/drain capacitances are already at ground. The middle curve has already been presented in Figure 3.7 and represents the worst case for externally-driven inputs. The rightmost curve is derived from the data presented in Figure 3.9.

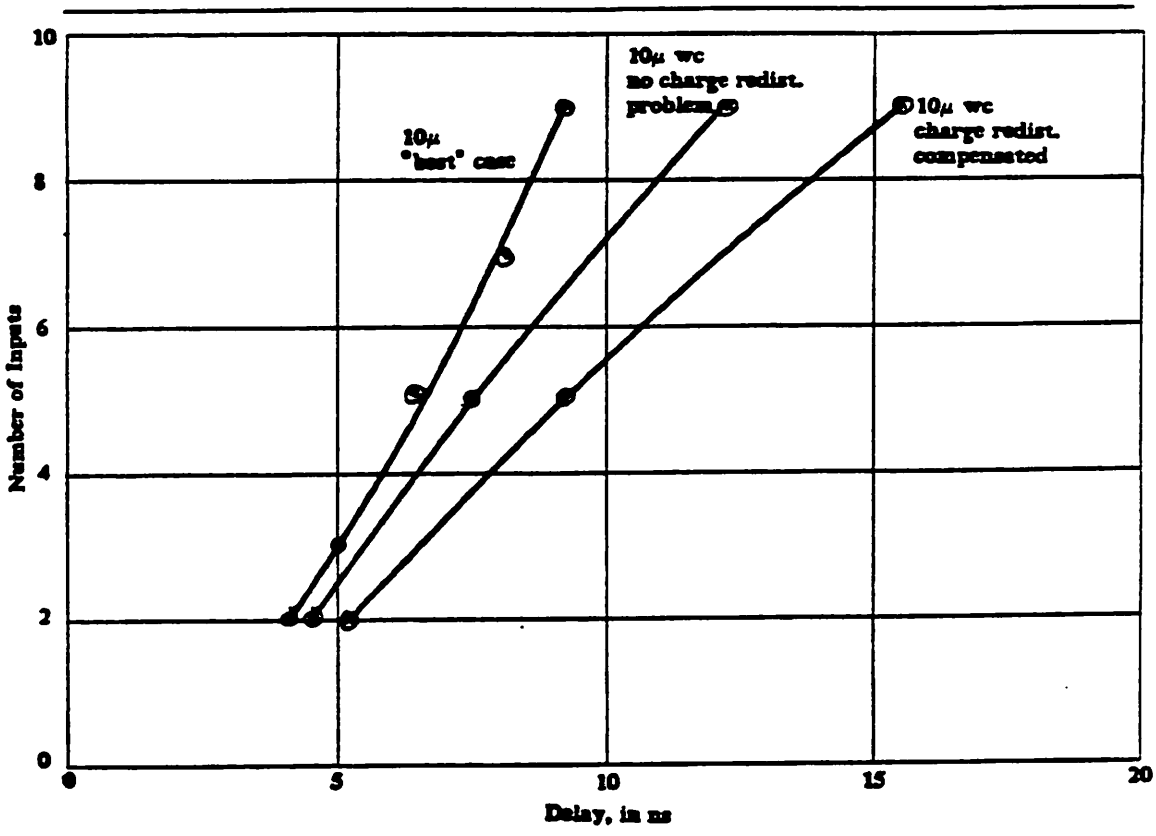


Figure 3.10: Comparison of 10μ-Wide Dynamic AND Function Under Various Input Situations

The lower-bound basic dynamic circuit can now be compared with the standard static version of the AND gate examined previously. The comparison is given in Figure 3.11. Note that the dynamic circuit now consists of 10 devices (due to the static pullup) while the static circuit requires 12. Because the dynamic circuit requires a clock signal. some additional routing is required which makes the circuits roughly equal in area. Both circuits can be sped up by increasing device widths.

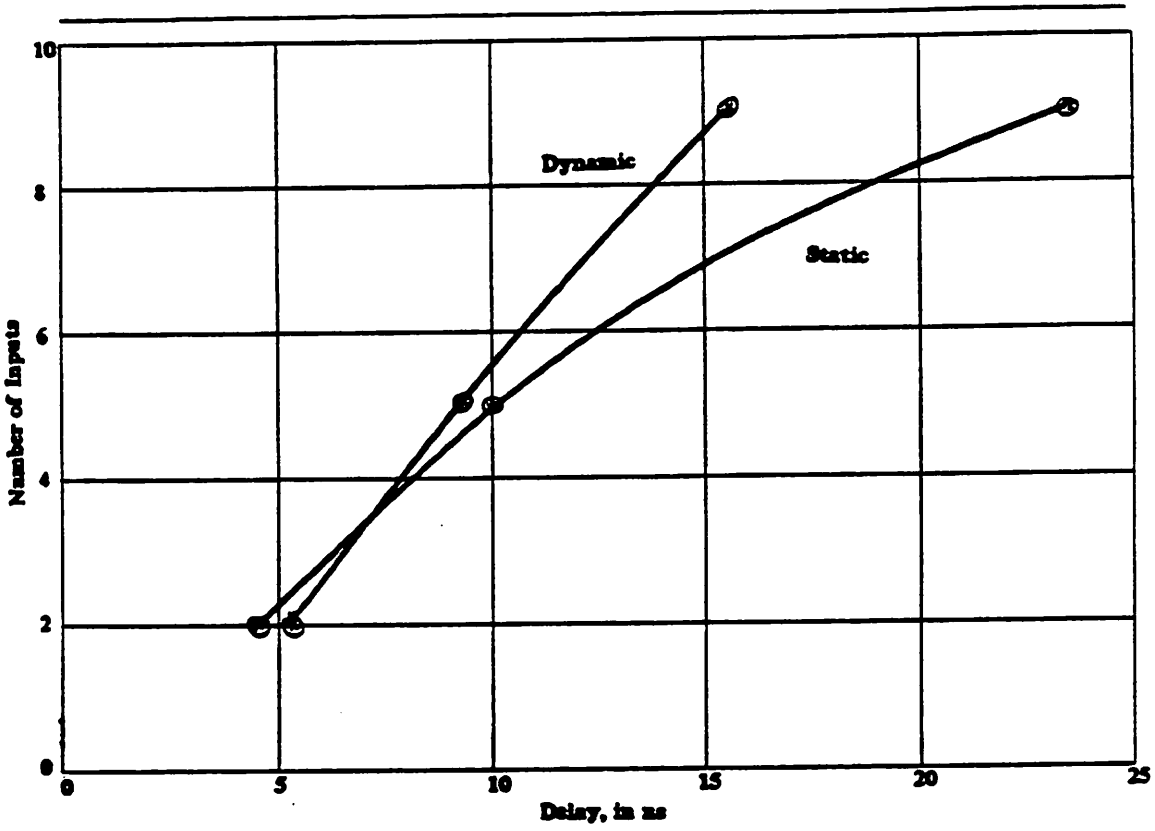


Figure 3.11: Comparison of Worst-case Static and Dynamic AND Functions versus Fan-in

3.7. Comparison of Optimized Dynamic Circuits

In addition to the standard Domino realization of a dynamic gate there are several other ways of building dynamic circuits. The "stacking" or "pyramiding" of gate widths, examined in Chapter 2 and employed in the BELLMAC-32A processor, is one means of speeding up circuit performance. A circuit which has a 10μ gate width on the average is shown in Figure 3.12a.

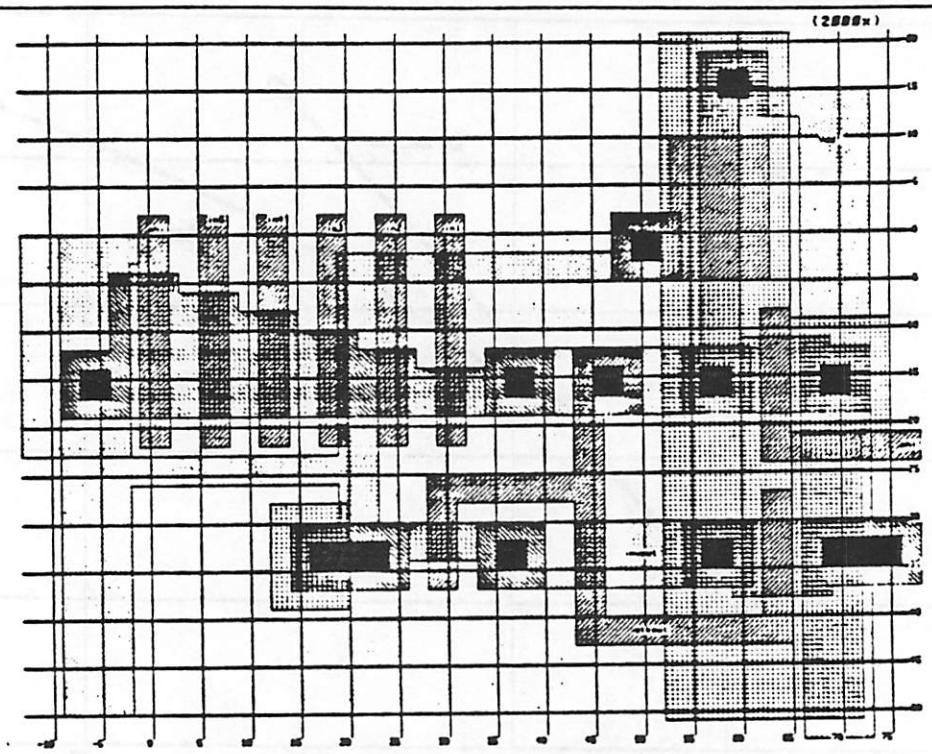


Figure 3.12a: Layout of Stacked-Gate AND

IBM has used a version of the dynamic circuit which has a pseudo-static output stage [hell84]. By feeding back the inverter output to control the weak p-channel pullup, any problems with charge leakage in low-frequency operation are eliminated. Because negative feedback is employed, the circuit tends to switch more slowly on a valid transition at the inputs— the output node tends to resist change. Figure 3.12b shows one realization of the IBM-style circuit.

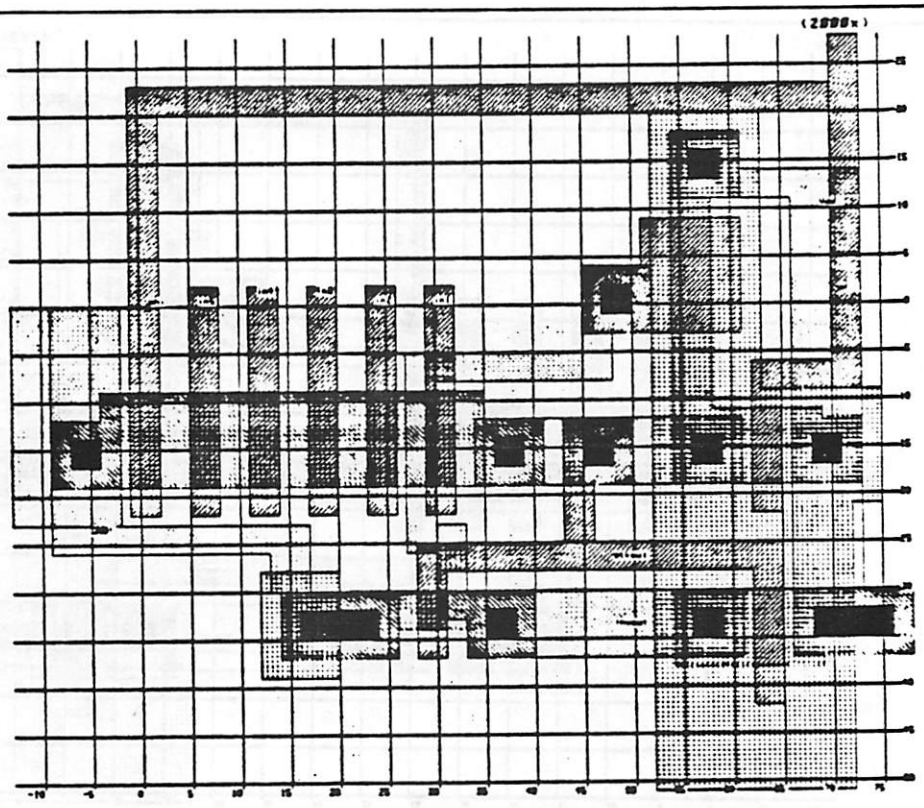


Figure 3.12b: IBM-style Circuit with Positive Feedback

In order to lay out a Domino circuit easily it has been proposed that the n - and p - channel clocked FETs be brought together to alleviate a potential crossover routing problem [newt83]. This "coupled" circuit still takes its output from between the clocked nodes, but now the input devices are below the n -channel ϕ-node. Such a configuration tends to aggravate the charge redistribution problem by introducing another parasitic node between the output node and GND. Figure 3.12c gives a layout of a coupled circuit.

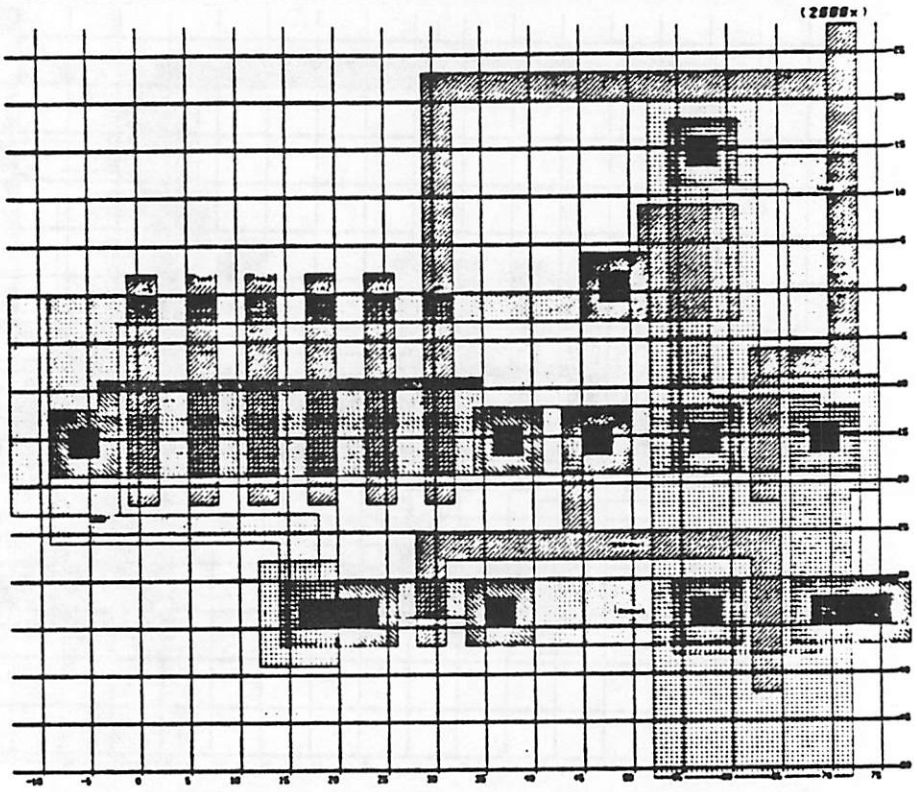


Figure 3.12c: Layout of Coupled-Clock Dynamic Circuit

Figure 3.13 summarizes the simulation results of the previous circuits. In each case the output inverter ratio has been modified so that the p-channel device is 24μ-wide and gives a fast pullup time. It also alters the k ratio of the inverter, raises the logic threshold, V_{TH}, and degrades the high noise margin. Calculations examining these effects were presented in Chapter 2. The calculations showed that symmetric noise margins are not required since the dynamic circuit is unidirectional. Because the circuit is precharged to the positive supply, loss of the high noise margin is not important.

Speed and False Trigger Results for Various Optimized Gates				
Circuit	Speed(ns)	Pullup (L / W)	Inverter Ratio (W_{pu} / W_{pd})	Peak FT Voltage (V)
Static	10.0	—	8/4	—
Domino	7.0	12/4	24/4	0.159
Stacked	6.6	15/4	24/4	0.184
IBM	7.2	12/4	24/4	0.209
Coupled	8.1	12/4	24/4	0.284

Figure 3.13: Comparison of Optimized CMOS Gates

3.8. Delay and Charge Redistribution Measurements from a Test Chip

A test chip was designed to examine the problem of charge redistribution and also to obtain direct measurements of gate speed as a function of number of inputs. The chip was fabricated at an industrial facility in a 2μ, double-metal, n-well process. The test layout contains 2-, 3-, 5-, 7-, 15-, 23-, and 31-input AND gates. It also has a collection of OR gates and a chain of C^2MOS latches cascaded to form a serial shift register.

Two test chips were received, only one of which was functional enough for testing. In this section results from the AND gates on the functional die are presented.

3.8.1. AND Gate Test Circuit

A template of the AND-gate circuit in schematic form is shown in Figure 3.14. In order to ensure conditioned signals, all inputs were buffered from the pads by inverters. The top $n-1$ inputs to the AND gate were coupled together in order to reduce pin count. Almost all delay and charge redistribution tests can be made with the AND gates connected in this configuration. All AND gates shared a common *bottom* signal, and many of the gates shared a common *top* signal as well. This means that the circuits could be accessed in parallel, however only one output was examined at a time. Each gate had an individual output buffer. As shown in Figure 3.14, this was a large p-channel device set up in a source-follower configuration. A p-channel device was used so that the potential of the device well (brought out externally) could be changed as part of the measurement pro-

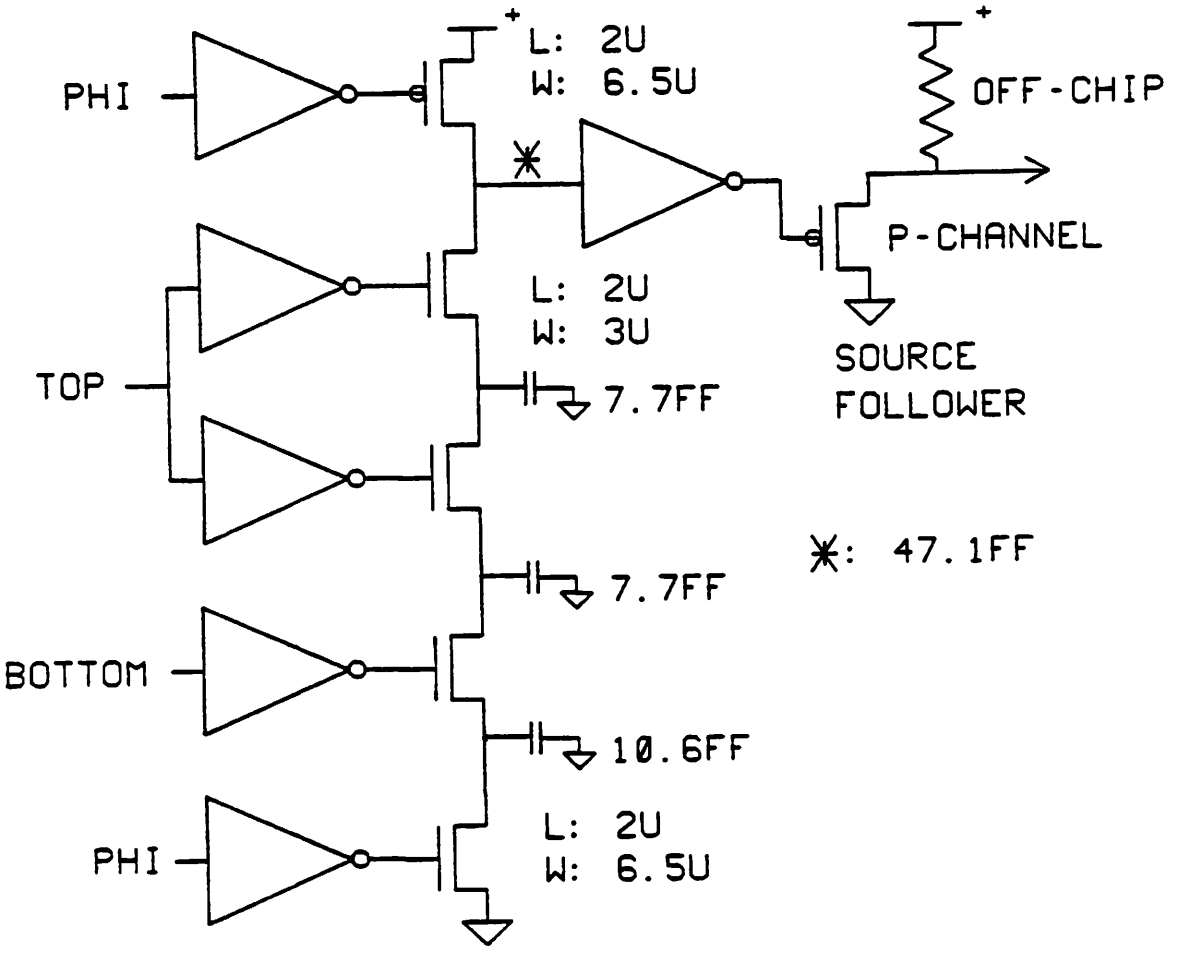


Figure 3.14: Template Schematic of AND gate

cedure. An off-chip resistor was used to allow experimentation with switching speed.

For the AND gate measurements recorded here minimum size devices were used in the n-channel core: minimum size devices are 2μ long and 3μ wide. The clock devices, driven through an inverter, are 2μ long and 6.5μ wide. The conditioning input inverter and the output inverter/buffer on the dynamic gate are made up of a 6μ-wide n-channel device and a 12μ-wide p-channel device. Both devices are 2μ long.

3.8.2. Precharge and Parasitic Capacitances

From the process parameters are given in Appendix B the precharge capacitance is calculated to be $47.1fF$. the parasitic capacitance for a source/drain node pair is $7.7fF$ for minimum sized devices and $10.6fF$ for the ϕ pulldown device. These capacitances are indicated in Figure 3.14. The precharge capacitance takes into account the area and perimeter of all polysilicon and diffusion regions. The capacitance of the metal layers was neglected. The precharge capacitance is made up of the drain capacitance of the top n-channel gate. the gate capacitance of the two devices that make up the output inverter. and the source capacitance of the p-channel clock device.

3.8.3. Charge Redistribution Tests

The waveforms for the charge redistribution tests were generated by three different pulse generators in sync with one another. The important input and output signals are shown in Figure 3.15. The time scale is microseconds. If charge redistribution occurs it will happen on the order of hundreds of nanoseconds. At the other extreme is the change of voltage on a node due to charge leakage. This usually becomes apparent in the millisecond time frame.

The worst-case charge redistribution problem is when the *top* input is run by ϕ and the *bottom* is run at half the ϕ frequency. In this example all signal polarities are those of the circuit: since the pad input signals are buffered they are just the inverse. The setup is as follows: Initially ϕ. *top*. and *bottom* are high and the output is high. The AND chain parasitic capacitances drain away through the DC ground path. *Top*. *bottom*. and ϕ go low. The precharge node has charged dumped in to it from V_{DD}. but the parasitics remain at ground potential because they are isolated from the precharge (prech) node. The output goes low. Now ϕ and *top* are brought high. *Bottom* remains low. Precharge ends. evaluate begins. the precharge node capacitance can drain into the parasitics (which were at ground potential) however no DC path to GND exists. The output should therefore remain low. If there is a charge redistribution problem. however. it will show up now and the output

will go high. The cycle completes by ϕ and *top* going low and *bottom* going high, and the output returns to (or remains at) ground. This cycle then repeats. This particular CR problem should be independent of frequency— as long as there is enough time for the precharge node to precharge and the parasitics to drain.

Charge redistribution is not the problem of low frequency operation. The low frequency problem is leakage from the precharge node through the substrate and occurs below 10KHz for the devices fabricated here. Leakage from the precharge node is almost independent of fan-in; larger fan-ins have a greater leakage problem, but the amount of leakage does not seem to be strongly related to fan-in. It was possible to cycle the test die at frequencies up to 3MHz, the usable range of the frequency generator employed. A 1KΩ off-chip pullup resistor was used. A smaller resistor could have been used safely and would have given better response.

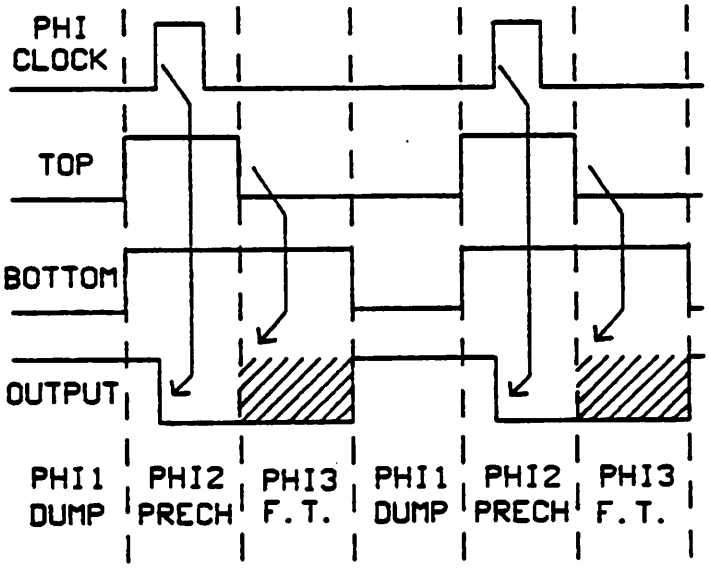


Figure 3.15: Waveforms for Charge Redistribution Test

3.8.3.1. Charge Redistribution Measurements

Figures 3.16, 3.17, and 3.18 are oscilloscope photos from the 2-, 7-, and 31-input AND gates, respectively. Figure (a) in each sequence shows the charge redistribution effect, if any. Figure (b) in the sequence is a control. In this test both the precharge and the parasitic capacitances were charged, then the *top* signal was toggled. Because the parasitic capacitances are at the same potential as the precharge node there is no CR effect. For the 2-input AND case Figure 3.16a shows no evidence of charge redistribution. This is because the ratio of parasitic to precharge capacitance is large, as described in Chapter 2.

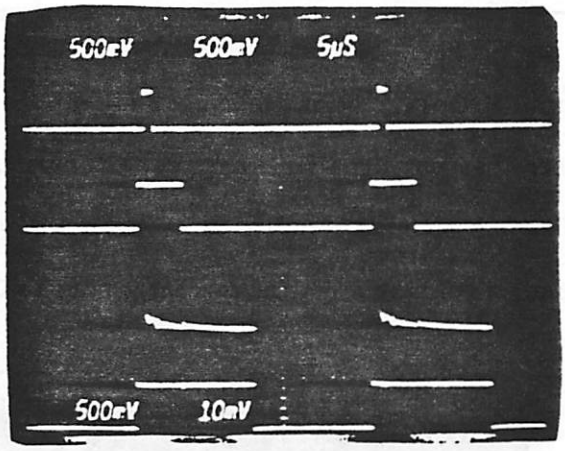


Figure 3.16a: 2-Input AND Gate Charge Redistribution Test

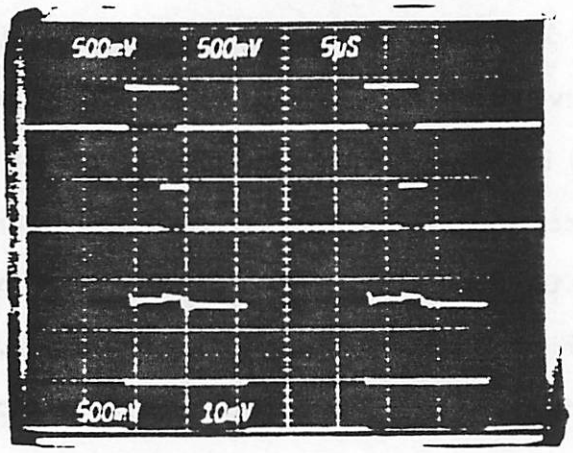


Figure 3.16b: 2-Input AND Gate Control Test

Figure 3.17, the 7-input AND gate, begins to show a CR effect. Coincident with the falling edge of *top* (remember that pin signals have opposite the polarity of the internal signal) the output begins to switch high. It is pulled to its maximum voltage when the *bottom* signal finally falls, opening a DC path from the precharge node to GND.

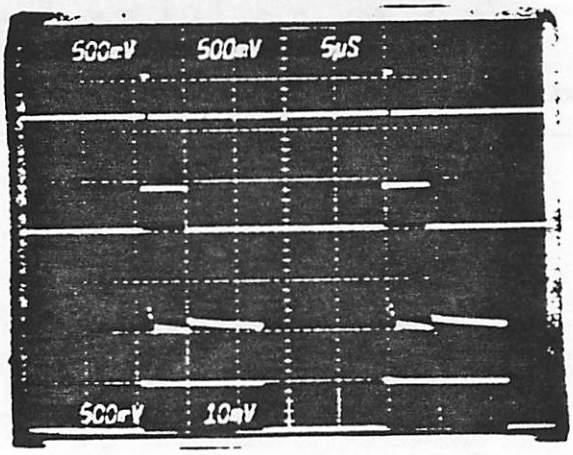


Figure 3.17: 7-Input AND Gate Charge Redistribution Test

Figure 3.18 was made from a 15-input AND gate. The third trace from the top is the output voltage of the AND inverter. As a result of charge redistribution it has risen about 0.2V; this signals the onset of redistribution.

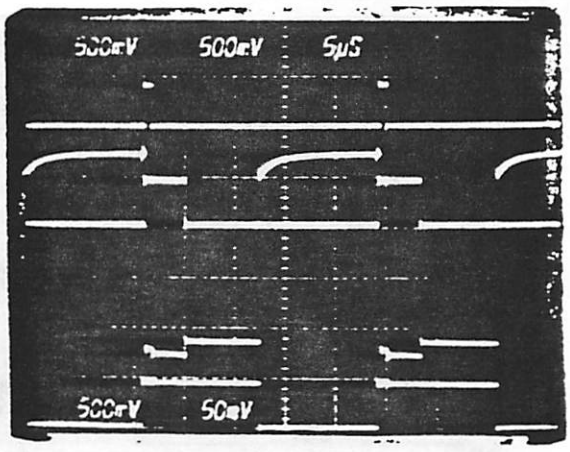


Figure 3.18: 15-Input AND Gate Charge Redistribution Test

Figure 3.19a was taken from the 31-input AND gate which shows a CR effect. As soon as the *top* signal falls the output of the inverter/buffer rises about 0.5V. The transition of the *bottom* signal causes the output signal to complete its transition.

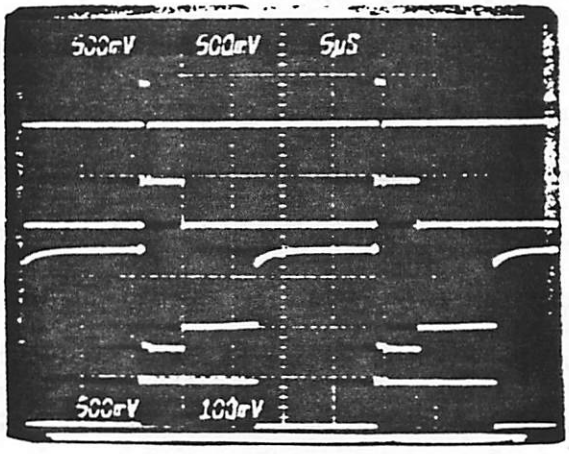


Figure 3.19a: 31-Input AND Gate Charge Redistribution Test

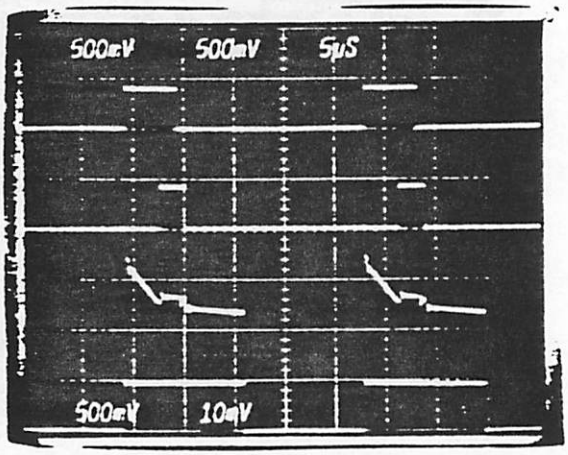


Figure 3.19b: 31-Input AND Gate Control Test

These measurements show that it would be unsafe to construct circuits using 15-input AND gates assuming minimum size devices and a fairly loose layout style that introduces some extra capacitance in interconnection routing. The inverter/buffer, which contributes almost all the precharge capacitance, was designed as a single cell for automated generation. It might be used by a layout tiler, for example the tiler described in Chapter 7 of this dissertation, in the generation of a regular logic structure. If the circuit designer used larger than minimum devices in the n-channel pulldown core it would aggravate the charge redistribution effect. On the other hand, by employing a cell with a greater precharge capacitance this effect would be lessened. In this case more devices could be placed in series but the gate delay would be longer for two reasons— first because of the greater chain length and second because the larger precharge capacitance must be discharged before the output can switch. In other words, to safely increase the number of series devices both the resistive path to GND and the capacitance which must be discharged to ground can be increased at the expense of greater circuit delay.

The p-channel V_t appears to be higher than reported in Appendix A. This high V_t has the consequence of delaying the CR effect; it can be compensated for by pumping less initial charge into the precharge node. the effects of this compensation for 7-input and

31-input AND gates are shown in Figures 3.20 and 3.21, respectively.

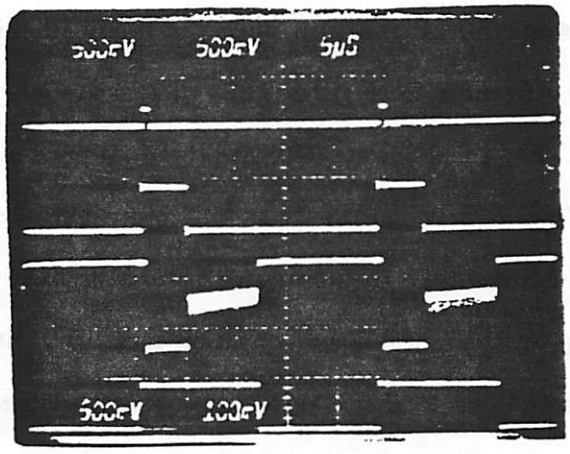


Figure 3.20: 7-Input AND Gate V_t Compensated

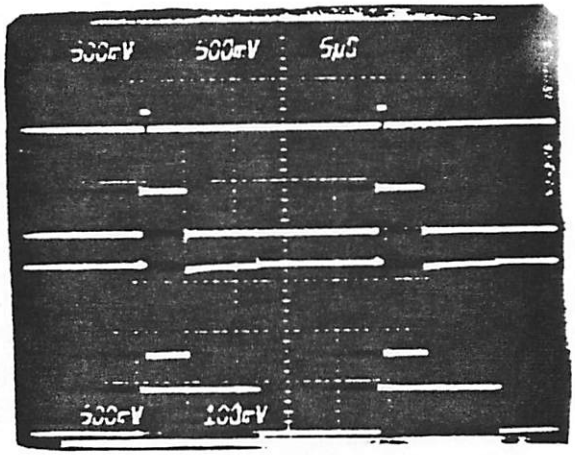


Figure 3.21: 31-Input AND Gate V_t Compensated

These results suggest that even a 7-input AND gate design would be unsafe. It is difficult to assess the actual amount of compensation needed to overcome the V_t disparity. It is reasonable to expect that the chain length limit lies between 7 and 15 inputs.

3.8.3.2. SPICE2 Simulations of Charge Redistribution Tests

SPICE2 simulations were performed on the 2-, 7-, 15-, and 31-input AND gates using the parameters in Appendix A and also with the modified, higher V_t for the p-channel devices examined in Appendix B. Figures 3.22— 3.25 show the simulation results in order of increasing fan-in. For the 2-input AND gate the difference in results due to the differing V_t's is negligible. The larger V_t in the 7- and 15-input gates more closely agrees with the measured results. In these cases the simulated results bracket the measured figure. These simulations also suggest that the V_t-compensation measurement given above may be larger than necessary. Simulations of the 31-input gate at both threshold levels indicate a very strong CR effect, giving a full swing at the output buffer, which does not correlate with the measured results. The observed result may be due to higher than expected capacitances at the precharge node.

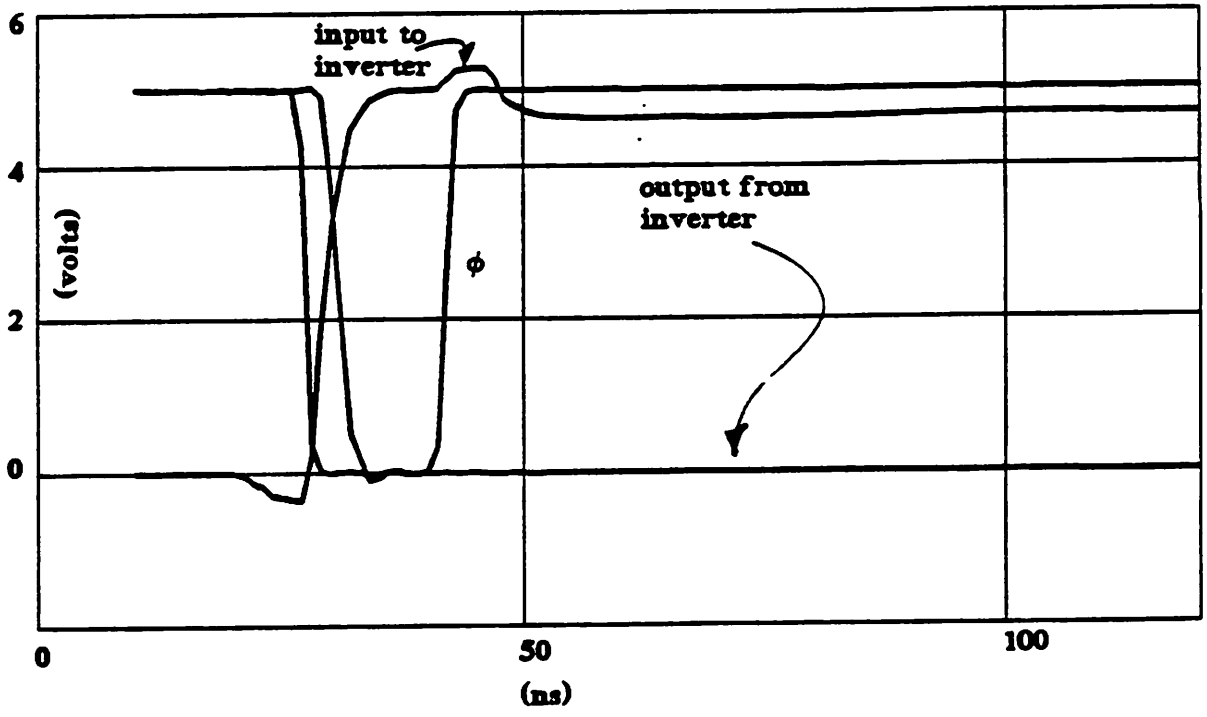


Figure 3.22a: 2-Input AND Gate, V_t = 0.9V

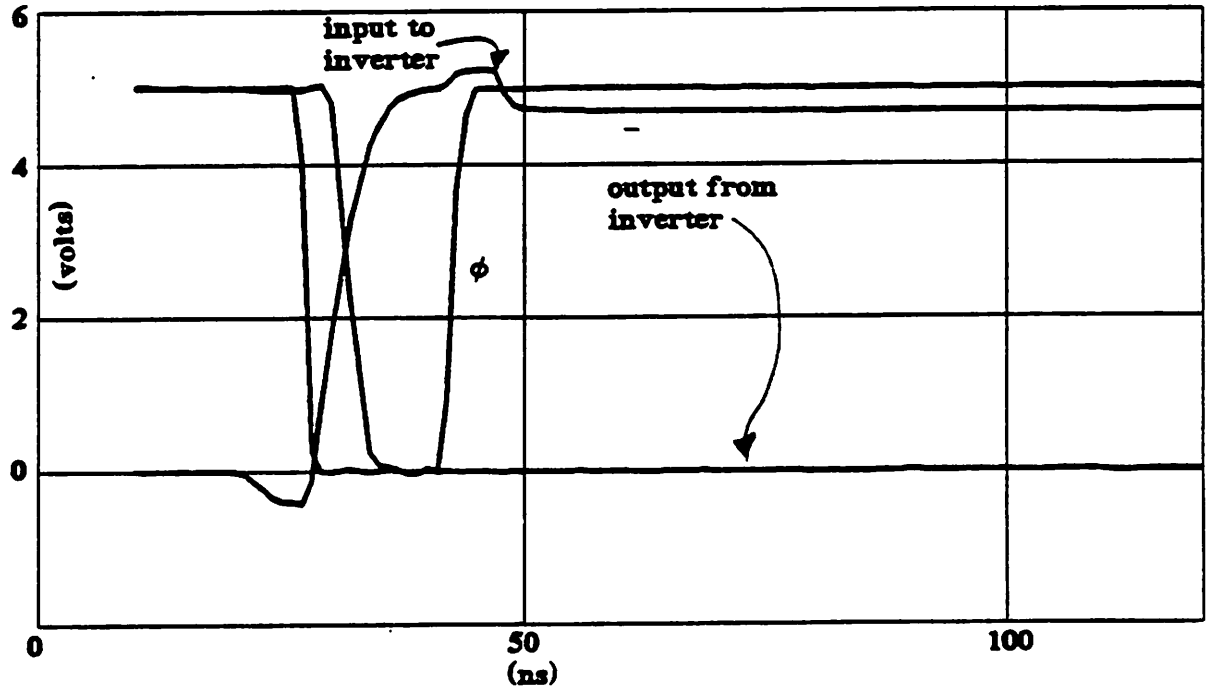


Figure 3.22b: 2-Input AND Gate, V_t = 2.2V

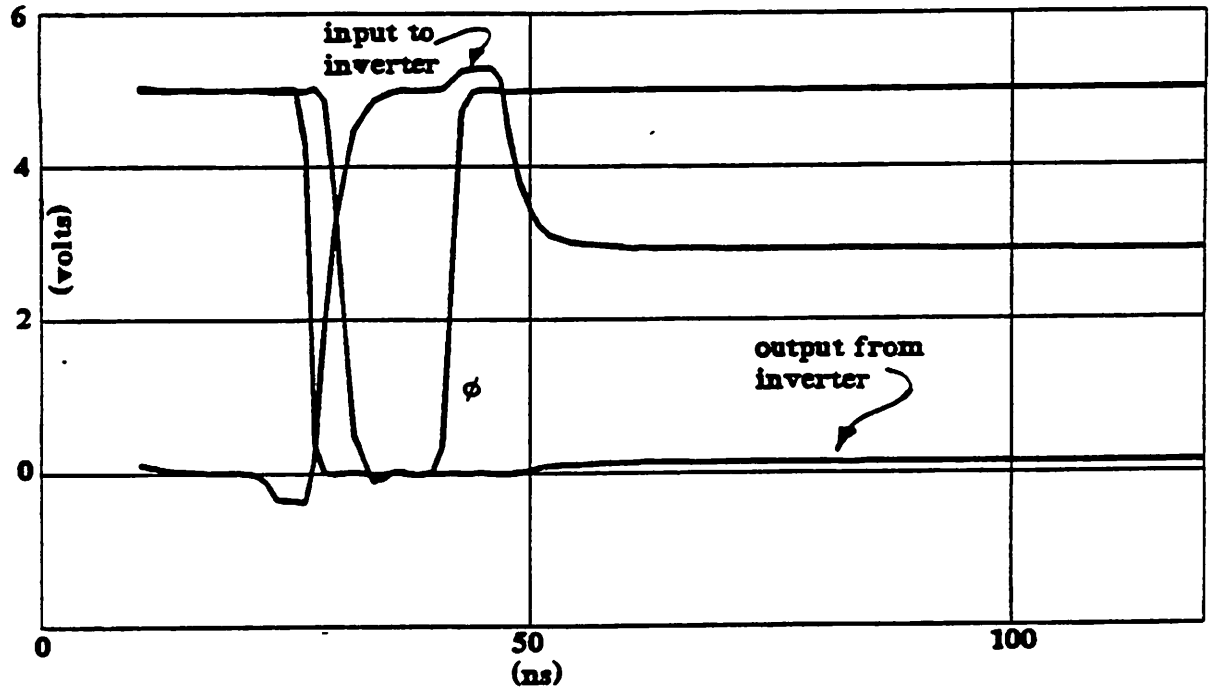


Figure 3.23a: 7-Input AND Gate, V_t = 0.9V

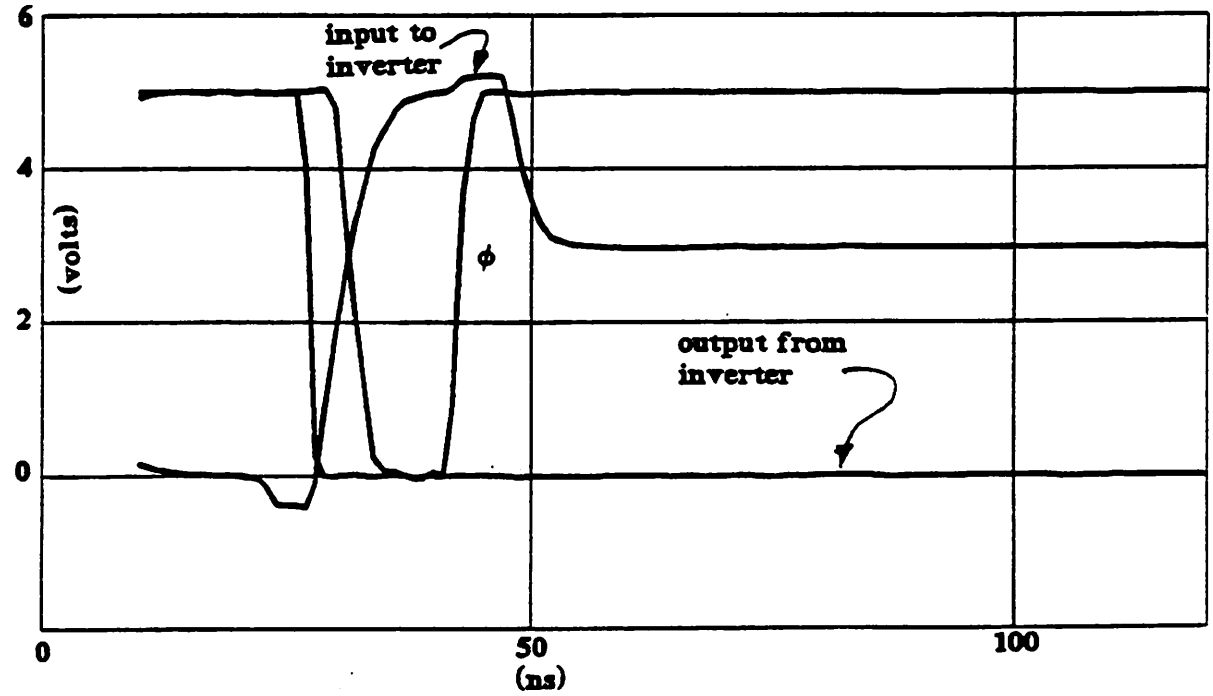


Figure 3.23b: 7-Input AND Gate, V_t = 2.2V

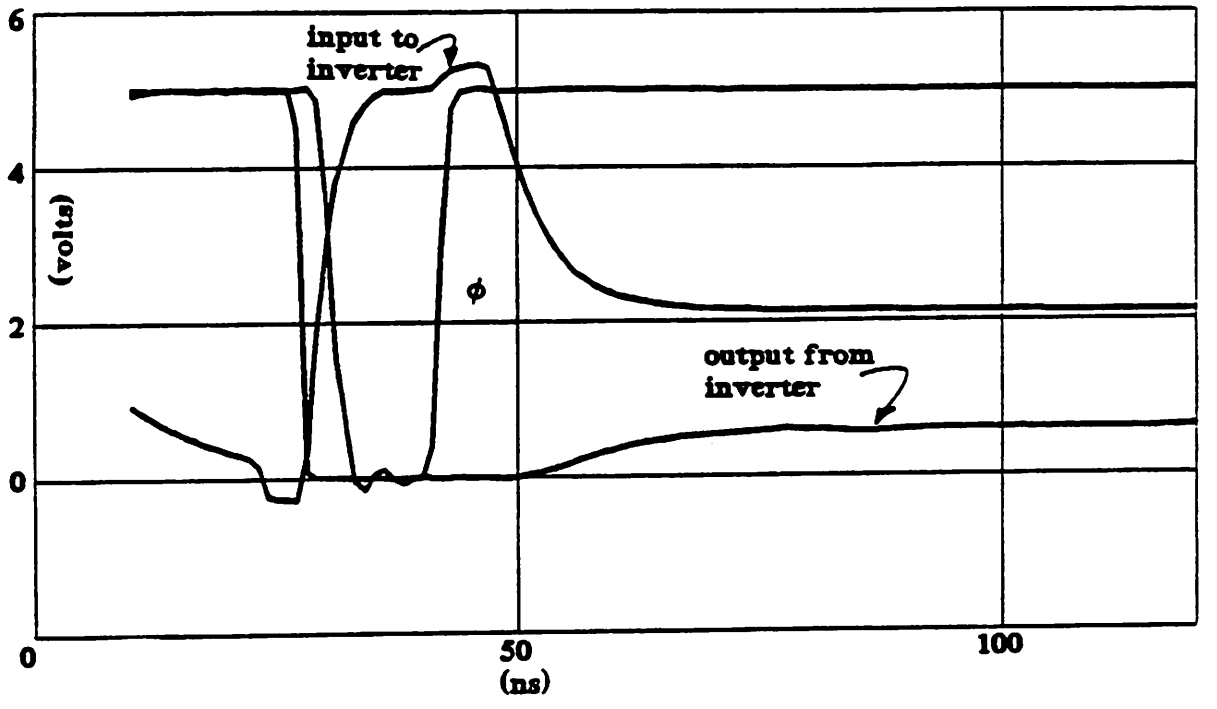


Figure 3.24a: 15-Input AND Gate, V_t = 0.9V

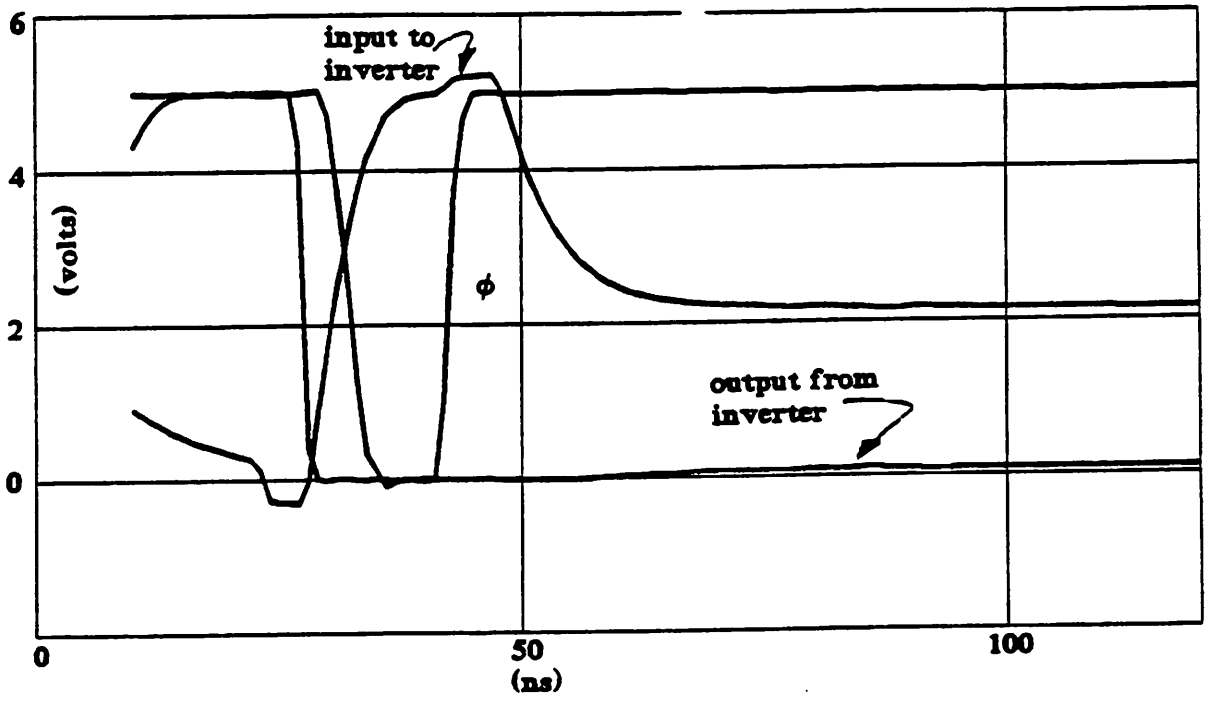


Figure 3.24b: 15-Input AND Gate, V_t = 2.2V

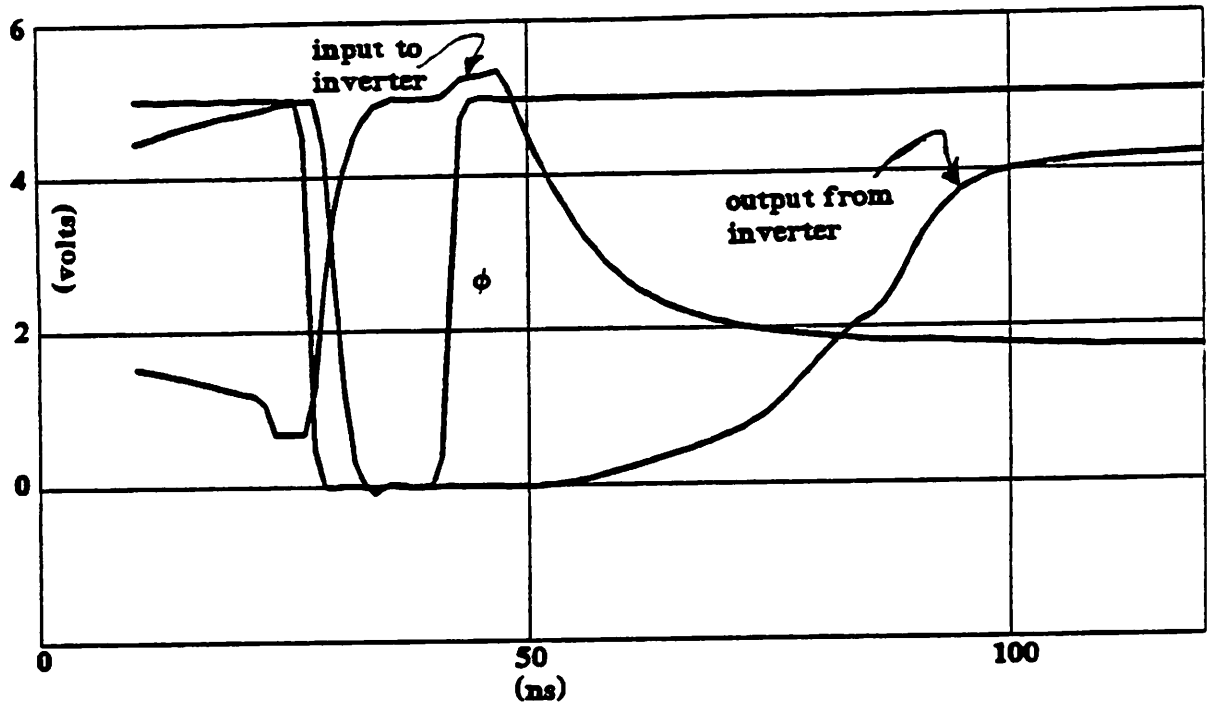


Figure 3.25a: 31-Input AND Gate, V_t = 0.9V

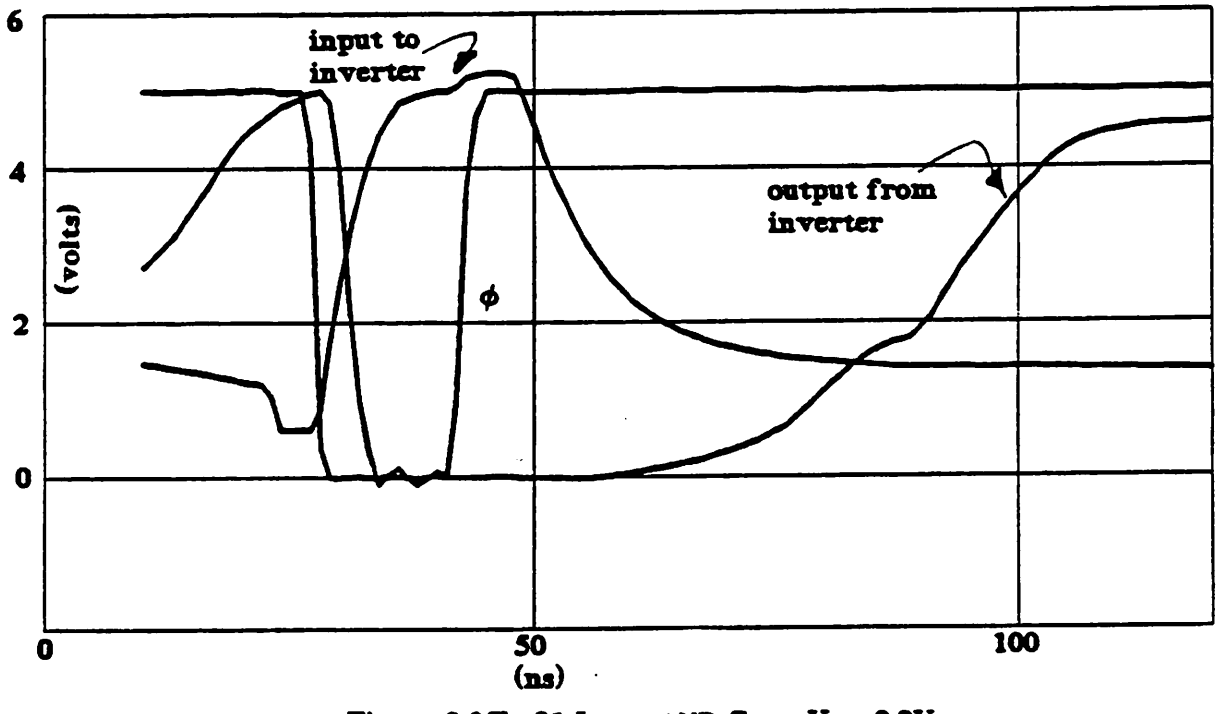


Figure 3.25b: 31-Input AND Gate, V_t = 2.2V

3.8.4. Dynamic AND Delay Tests

The worst-case (longest delay) input configuration in the AND gate occurs when both *top* and *bottom* are wired "on". The testing regime is to first precharge both the precharge node and all core parasitics (performed at the same time since all inputs are held high). Then when ϕ drops low the precharge phase ends shutting down the path from precharge node to V_{DD} and opening a path from the precharge node to GND. The delay of the circuit is measured from the 50% point of the falling ϕ signal against the 50% point of the rising output buffer signal.

3.8.4.1. AND Gate Delay Measurements

The AND gate delay measurements were performed at the package pins and therefore included the conditioning inverter delay as well as the delay of the output pad driver to charge up the oscilloscope probe capacitance. The delay test measurements were taken on 2-, 15-, and 31-input ANDs. Figures 3.26, 3.27, and 3.28 show the measured delays for the 2-, 15-, and 31-input circuits, respectively.

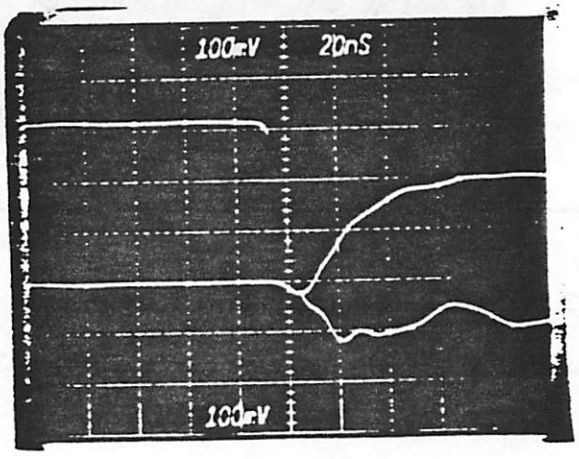


Figure 3.26: 2-Input AND Gate Delay Test

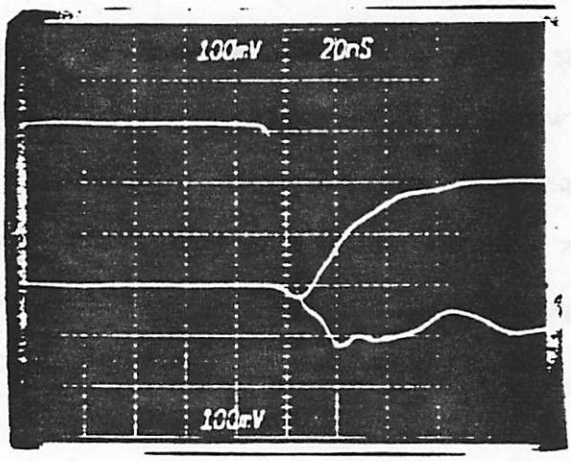


Figure 3.27: 15-Input AND Gate Delay Test

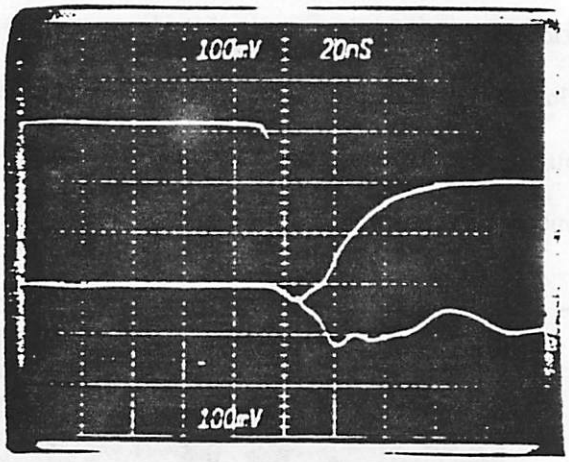


Figure 3.28: 31-Input AND Gate Delay Test

After subtracting 23.3*ns*, which is the calculated delay through a conditioning inverter and output pad plus routing, the actual gate delay can be derived. The results are summarized in Figure 3.29.

Fan-in	Pad-Pad Delay *(ns)*	Gate Delay *(ns)*
2	28ns	4.8ns
15	32ns	8.7ns
31	45ns	21.7ns

Figure 3.29: Derived AND Gate Delay

The second-order least-squares fit of this dataset is: $4.6 + 0.012f + 0.017f^2$ where f is *fan-in*. Fitting the data linearly one obtains: $2.27 + 0.591f$. However the sum-of-squares error is large and, therefore, the quadratic model is more appropriate, as one would expect. Increasing fan-in increases both R and C, if one considers the FET to be an RC device in a simple model. The large constant is due to the output inverter and to the precharge capacitance which are independent of fan-in.

3.8.4.2. SPICE2 Simulations of Delay Tests

SPICE2 simulations were performed on 2-, 7-, and 15-input AND gates. The results of these simulations give a second-order least-squares fit of $4.1 + 0.83f + 0.019f^2$. While the initial delay constant is about equal to the measured result this curve exhibits a stronger linear component than the measured data. Again, this may be due to higher diffusion capacitance per unit area on the test die than in the simulation models. The delay values are summarized in Figure 3.30.

Fan-in	Gate Delay *(ns)*
2	5.8
7	10.8
15	20.8

Figure 3.30: Simulated AND Gate Delay

3.9. Measurements of a 32-bit Dynamic Domino ALU

The design and measurement of a 32-bit Domino ALU is now presented. The techniques used in the design of this circuit helped validate the design style used in the implemented synthesis package. The design approach employed in the MAMBO automated synthesis package is identical to the approach used in the ALU, which was handcrafted.

The *SOAR* project served as a test bed for the dynamic CMOS circuit work. *SOAR* stands for *Smalltalk On A RISC* and is part of a larger architecture effort at UC Berkeley to develop a compact, fast, reduced instruction set Smalltalk workstation. The SOAR chip was implemented in both NMOS and CMOS technologies; the work described here was applied in the CMOS version of the chip. In the sections that follow the definition, architecture, layout, and performance of the dynamic SOAR 32-bit ALU are presented.

3.9.1. Design of a Dynamic 32-bit ALU — General Issues

The ALU of the CMOS SOAR processor is partitioned into three components, the **byte inserter/extractor** *(BIE)*, the **complementer/buffer** *(COM)*, and the **ALU** itself. The BIE is used to extract or shift bytes of data within the 32-bit datapath. Its operation is mutually exclusive from the rest of the ALU and is described further in [hofm83]. The COM receives the operands A and B and generates the buffered signals A, $\overline{A}$, B, and $\overline{B}$. The COM was designed principally to provide clean, buffered inverted and non-inverted signals to the ALU and is described in detail in [hofm83]. The remainder of this section covers the design of the ALU proper. The block diagram in Figure 3.31 shows the interconnection among the three major blocks. Data flow is left to right, from the A and B buses onto the EAbus (output of the ALU).

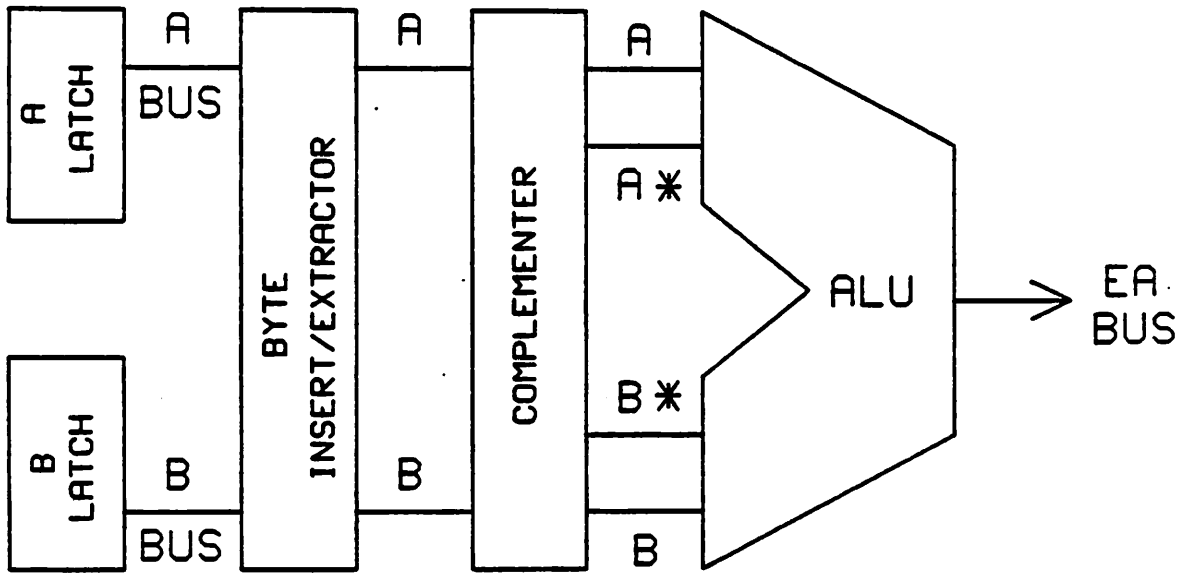


Figure 3.31: Block Diagram of SOAR ALU Section

CMOS SOAR was submitted through the MOSIS foundry system and the MOSIS micron-based design rules were used. Micron-based rules allow the designer greater freedom and give greater flexibility as compared to lambda-based rules. Lambda-based rules are more easily shared between different fabrication lines [gris82]. The 1982 MOSIS process technology was 3μ (drawn) p-well. It uses a single layer of polysilicon and a single layer of metal; buried contacts are not allowed.

The ALU section was designed as a bitslice. Early in the design process it was decided that GND would run along the bottom of the LSB cell and V_{DD} along the top. Every other cell is mirrored so that GND and V_{DD} buses are shared.

The critical pitch for the datapath is in the y—direction. perpendicular to signal flow. and is set by the ALU. The ALU is the most complex single cell in the datapath. The critical pitch was made loose to allow for later design changes. The final y—pitch of CMOS SOAR is 117μ. The BIE is 255μ in the x—direction. while the is COM 171μ and the ALU 513μ. Figure 3.32 is a die photo of the ALU test chip. The photograph shows static latches on either side of the ALU which are used to load operands and store results.

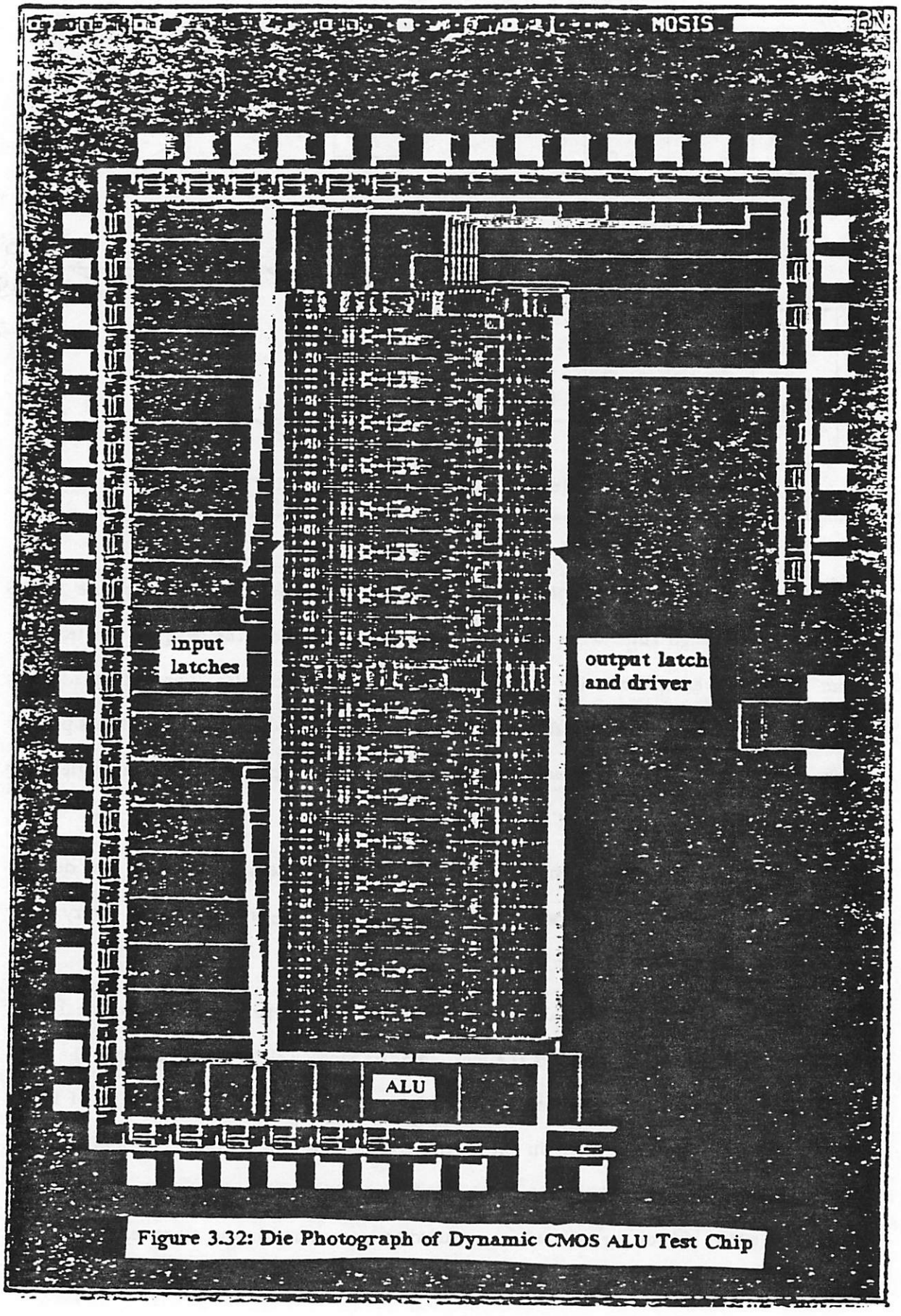


Figure 3.32: Die Photograph of Dynamic CMOS ALU Test Chip

3.9.2. ALU Design

Design of the ALU section was driven by design of the carry circuitry. The ALU computes during $\phi 3$ of a three phase, asymmetrical clock cycle. It was desired to perform an ALU operation in around $110ns$. In order to come close to this specification some type of carry acceleration is required: a full ripple-carry would take too much time. A carry bypass scheme was chosen which, while not as fast a full carry lookahead, speeds up the worst-case carry and is quite economical in layout area. In fact the scheme appears to represent a good tradeoff between layout area and circuit speed as shown in the table in Figure 3.33 from [whal84].

Design Metrics for CMOS Adders			
Adder	Device Count	Area	Speed
Brent-Kung			
Static	2846		
	1423n	2.975E06 μ^2	36.28ns
	1423p		
Nora	3123		
	1677n	3.373E06 μ^2	44.98ns
	1446p		
Manchester carry chain	1024		
	320n	8.696E05 μ^2	87.45ns
	704p		
Carry Select			
27/5	1594		
	619n	1.347E06 μ^2	83.06ns
	975p		
15/17	1342		
	487n	1.143E06 μ^2	59.47ns
	855p		
74181	1577		
	773n	2.049E06 μ^2	45.16ns
	804p		
Kuck	1979		
	907n	2.251E06 μ^2	37.96ns
	1072p		
Q Circuit	2108		
	1113n	2.456E06 μ^2	38.99ns
	995p		*25.22ns
Carry Bypass	1694		
	1291n	2.068E06 μ^2	**45-50ns
	403p		

Figure 3.33: Comparison of Adder Schemes

The speed figure given for the carry bypass circuit, the method implemented here, is inaccurate. The actual measured delay, as will be shown later, was 140*ns*. The number in the table represents the worst-case measured delay for the carry bypass chain alone. A full add with no carry requires about 95*ns* and to this the carry bypass delay must be added to give the full worst-case adder time. Also, the area given for the carry bypass scheme represents the area of the entire 7-function ALU portion (excluding the the BIE and COM) of the circuit; for the other adders the area and device-count values reflect only the portion required to perform an add operation.

3.9.3. The Dynamic Carry Chain

The carry bypass scheme employed by Siemens in their MIKERW-83 [pomp82]. was chosen for carry acceleration in SOAR. This scheme works by providing two separate carry chains. The 32-bit circuit is divided into blocks. The idea is to have the carry signal bypass a block entirely if there is a propagate signal in each bit of the particular block. Figure 3.34 shows a block diagram of an NMOS carry bypass scheme.

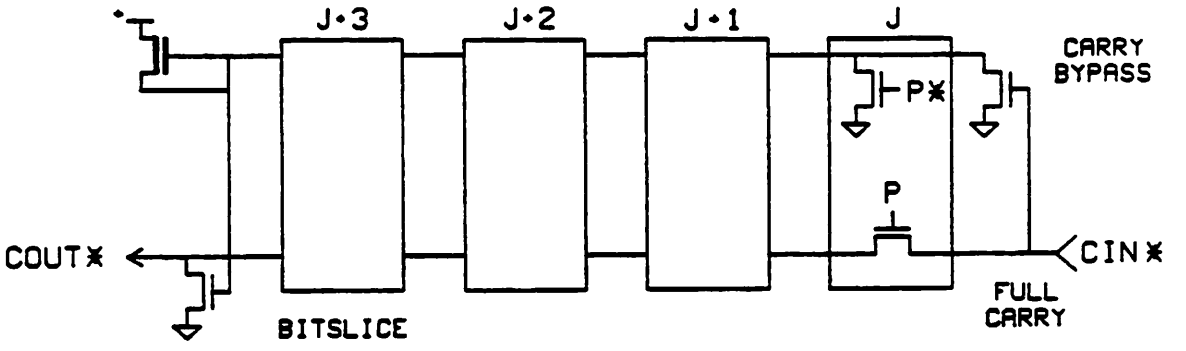


Figure 3.34: NMOS Carry Bypass Scheme

The signals P and $\bar{P}$ represent the presence and absence of a propagate condition for a given bit, respectively. The block size has been fixed at four bits. The upper signal run is in effect a fast carry chain and is used to accelerate the carry logic value with the potentially greater delay. For NMOS this is a logic *1*. This second chain can be thought of as "propagate-kill". In operation $\overline{C_{out}}$ is held to the slower logic state (logic *1*). If either $\overline{C_{in}}$ is high (no carry in) or any of the bit $\bar{P}$ signals are high (no propagate) then the propagate-kill line is activated. It is pulled low which shuts down the path between $\overline{C_{out}}$ and GND. Thus $\overline{C_{out}}$ remains high indicating no carry out. In the opposite case $\overline{C_{in}}$ is low and all of the $\bar{P}$ signals are low. This indicates a carry in which is not absorbed by any bit in the current block and is thus propagated across the block. In this case the propagate-kill lines remains high (*ie.* allow carry to propagate). This opens up the path between $\overline{C_{out}}$ and GND and thus $\overline{C_{out}}$ goes low indicating a carry out. The standard $G + PC$ logic in the lower carry chain operates independently of the fast bypass. If a

carry is generated in any of the intra-block bits it is then propagated via this slower ripple-carry chain to $\overline{C_{out}}$, which is grounded to indicate a carry out.

The choice of number of bits per block is governed by two considerations. One consideration is the amount of capacitance on the bypass line that must be discharged by the pulldown devices (note that it is precharged or held high). The greater the number of bits the higher the capacitance. On the other hand, the introduction of a non-inverting buffer introduces additional circuit delay. The $\overline{C_{out}}$ line is buffered after each block. The second consideration is the probability of bypassing a given number of bits versus the gain in speed by taking the bypass. The larger the number of bits bypassed the greater the bypass acceleration. In the limit, to bypass the greatest number of bits would mean a block size of one, which is equivalent to no bypass at all. Again there is the consideration of the additional device count versus speed gain. In [pomp82] a block size of four was used.

This NMOS carry bypass method relies heavily upon static, ratioed logic. In the NMOS scheme the $\overline{C_{out}}$ signal is held high by a weak depletion load device and the signal is activated (brought low) by a wide pulldown device. Thus the implementation relies on device ratioing and consumes static power. In keeping with the Domino design style, the bypass scheme has been modified to be ratioless and dynamic. A block diagram of the modified bypass circuit is shown in Figure 3.35. Again four bits were used per block.

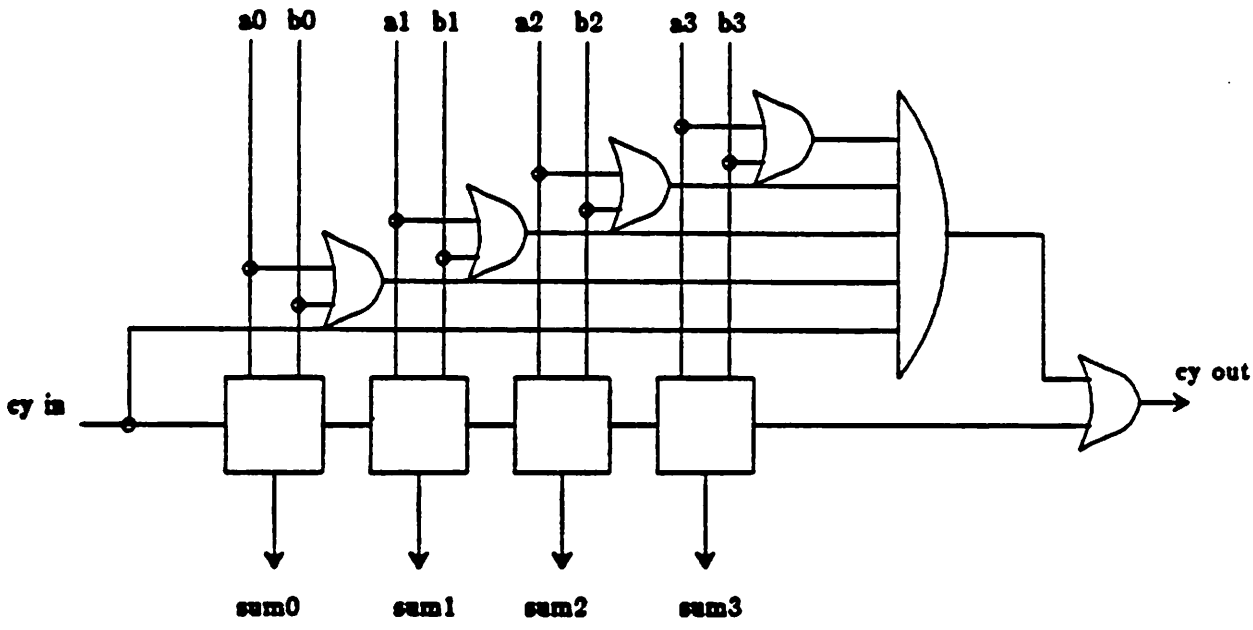


Figure 3.35: Dynamic CMOS Carry Bypass Scheme

The idea of a faster, alternative carry chain, the bypass, has been retained in the Domino implementation. However, this line is now "propagate" instead of "propagate-kill". This line is precharged high, rather than being held high, as in the static version. In other words the sense of the line has been inverted. This was necessary to keep within the Domino design rules that require all Domino gates run off the same clock to be precharged to the same state. Since inversions are not allowed within Domino logic, the signals P and $\bar{P}$ cannot both exist unless they were produced from other, more fundamental gates. While this duplication of logic was considered, the idea was discarded as being too costly in terms of device count.

In operation both the *propagate bypass* and *carry nibble* (full $G + PC$) lines are precharged high. In this case the precharged level of logic *1* indicates no carry. A transition to logic state *0* indicates the presence of a carry. A schematic diagram of a 4-bit block of the carry bypass is shown in Figure 3.36. The carry bypass is just the propagate function. If either or both of the operands (A or B in Figure 3.36) are asserted in each of the four bits of the block then the propagate line, which was precharged high, is brought low.

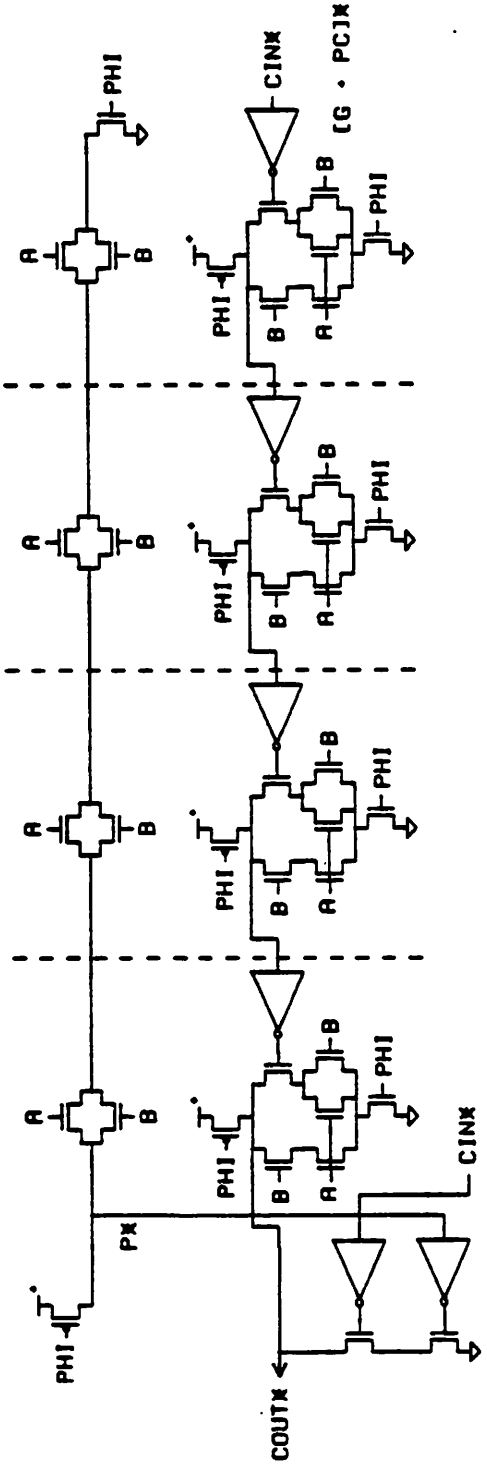


Figure 3.36: Schematic of Dynamic Carry Bypass Block

If, in addition, there is a carry into the block then the *carrynibble* line will go low, indicating a carry out. However, if the carry into the block is high (indicating no carry in) or there exists at least one intra-block bit that has neither *A* or *B* asserted then the propagate line remains at its precharged high level and *carrynibble* remains high indicating no carry out.

In this implementation of the carry bypass the important factor deciding block size is the number of passgates that can be chained together before a buffer is required. SPICE2 simulations of CMOS passgates by [whal84], indicate that best speed is obtained when buffers are inserted every three gates. There is only a slight degradation at four bits per buffer and since this figure is an integral multiple of two it fits in better with a regularly-structured bitslice approach.

Each bit of the ALU has a total of 51 devices: 39 of these devices are used in calculation of Boolean operations and add/subtract. There are 9 devices in the ripple-carry generation; an additional 3 devices per bit are required for the carry bypass in the dynamic CMOS implementation. This is the same number required in the static NMOS version. The total number of devices in the ALU is 1694. This is greater than 51×32 due to control signal buffering at the bit 15 — bit 16 boundary.

3.9.3.1. Speed of Worst-case Carry Bypass

Both carry bypass schemes work in the same way. Since it is assumed that all 32 bits of operands *A* and *B* are stable in parallel at the same time, if a decision about the outcome of the carry can be made on the basis of these bits only (and an initial carry signal from the LSB), carry generation can be accelerated. The carry bypass scheme achieves its greatest speedup on the worst-case carry propagate situation. It gives proportionally less speedup the better the original case. The worst-case carry occurs during an add instruction when a carry is generated in bit 0. This carry must ripple through the first four-bit block. It then bypasses six nibbles. It must ripple through the last nibble again to effect the MSB, thus it ripples through the top four bits. This is the worst case, because

any other carry bit generation must occur at a more significant bit (other than 0) and thus have a shorter distance to travel to the MSB. It is entirely possible that there are two or more carries using the different parts bypass at the same time. Such a situation exists when carries are generated in multiple bits. However in this case all generated carries will follow a shorter path and so take less time. These shorter paths will probably involve fewer block bypasses simply because there are fewer bits to jump around. Thus the bypass scheme speeds up the worst case (longest carry propagation path) most.

The 32-bit NMOS ALU fabricated in [pomp82] uses a single metal, polysilicide process. The polysilicide layer has a sheet resistance of about 3 Ω/□. A worst-case carry bypass ALU operation was measured (from latch to latch) at 66*ns*. Simulations were also performed assuming a standard polysilicon process with a sheet resistance of 30 Ω/□. In this case the authors found a fourfold increase in control signal delay. They estimate such a circuit would work at clock frequencies of less than 5MHz. Measured delay, summarized below, of the MOSIS device which did not use a polysilicide layer was 140*ns*. In contrast, [frie84] implemented a 16-bit ripple-carry adder in a 5μ CMOS polysilicon gate technology. They utilized a NORA style of implementation which they claim should be 30% faster than a corresponding Domino implementation. The authors report a total delay across the 16-bit adder of 180*ns* which corresponds to a ripple-carry delay per bit of 11.3*ns*. Since this is a ripple-carry circuit the 32-bit add time should be 360*ns*.

3.9.4. ALU Logic Functions

The ALU performs seven functions. They are AND, OR, XOR, ADD, SUB, SR, and PASS. The PASS function simply passes the operand *A* to the ALU output. The function SR shifts the *A* operand one bit to the right. The shift is arithmetic or logical depending on control signals provided by an external (off module) condition code PLA. The SUB function is *A* — *B* and is performed exactly like an add, except that a carry is injected at the LSB. The COM circuit detects that a subtract operation has been requested and inverts the *B* operand. The Boolean operations are all implemented in Domino logic. The XOR function

is the only part of the ALU which requires A , $\overline{A}$, B , and $\overline{B}$. Inverted signals are provided by the COM. XOR is implemented as $A\overline{B} + \overline{A}B$ and is the only one of the Boolean functions which breaks even in device count as a Domino function. The propagate/generate logic saves one gate over a static implementation. Recall that the overhead for a Domino gate is four devices. Thus when the fan-in is less than four, static CMOS implementations require fewer devices. Because only the XOR required inverted signals it was placed at one end of the ALU to avoid running these signals throughout the whole slice. N-core Domino logic was used in the ALU design. A schematic of the ALU without the carry logic is shown in Figure 3.37a. Figure 3.37b shows the mask layout of five contiguous bits.

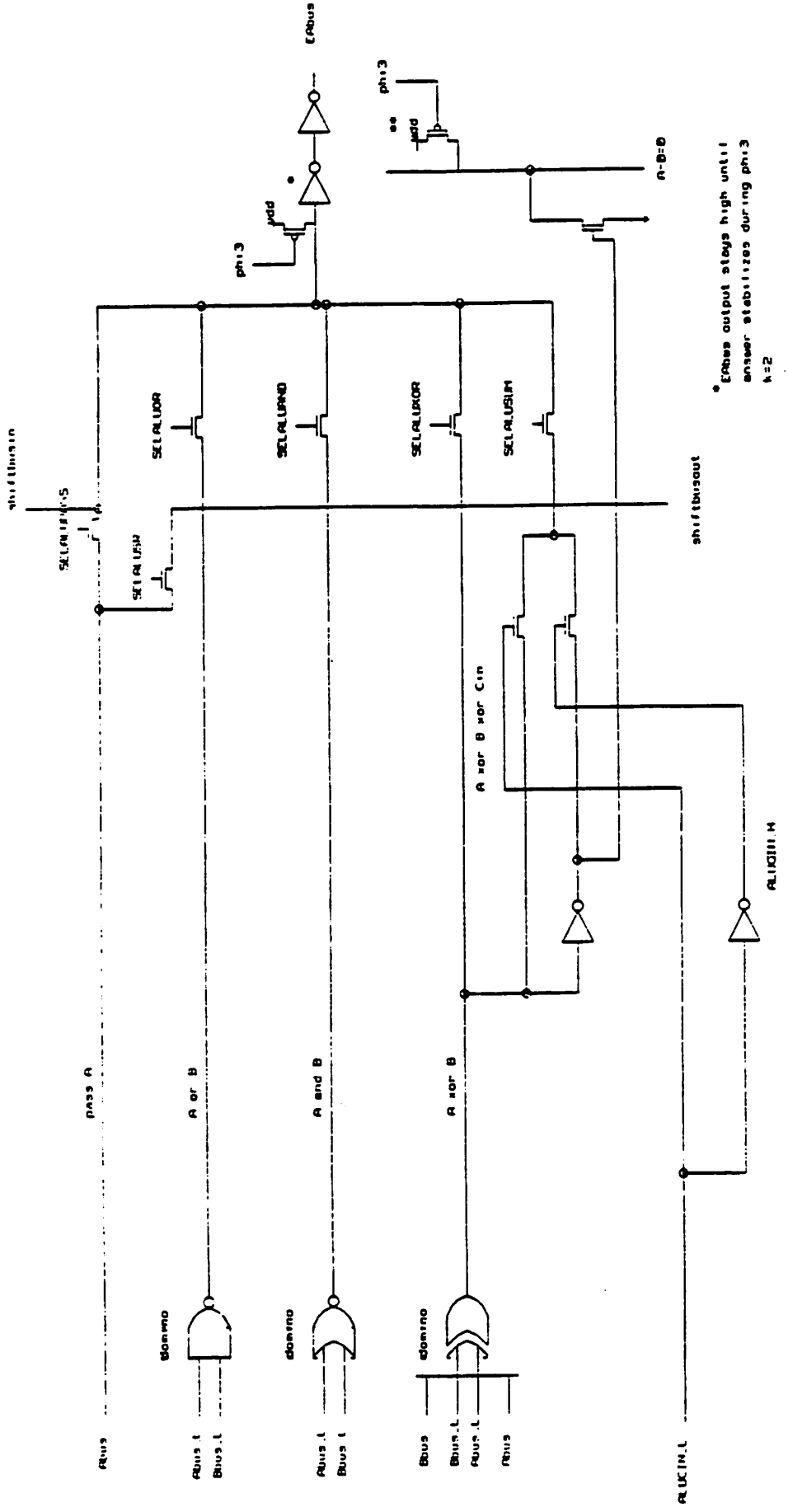


Figure 3.37a: Schematic of a Single Bit of the Dynamic ALU

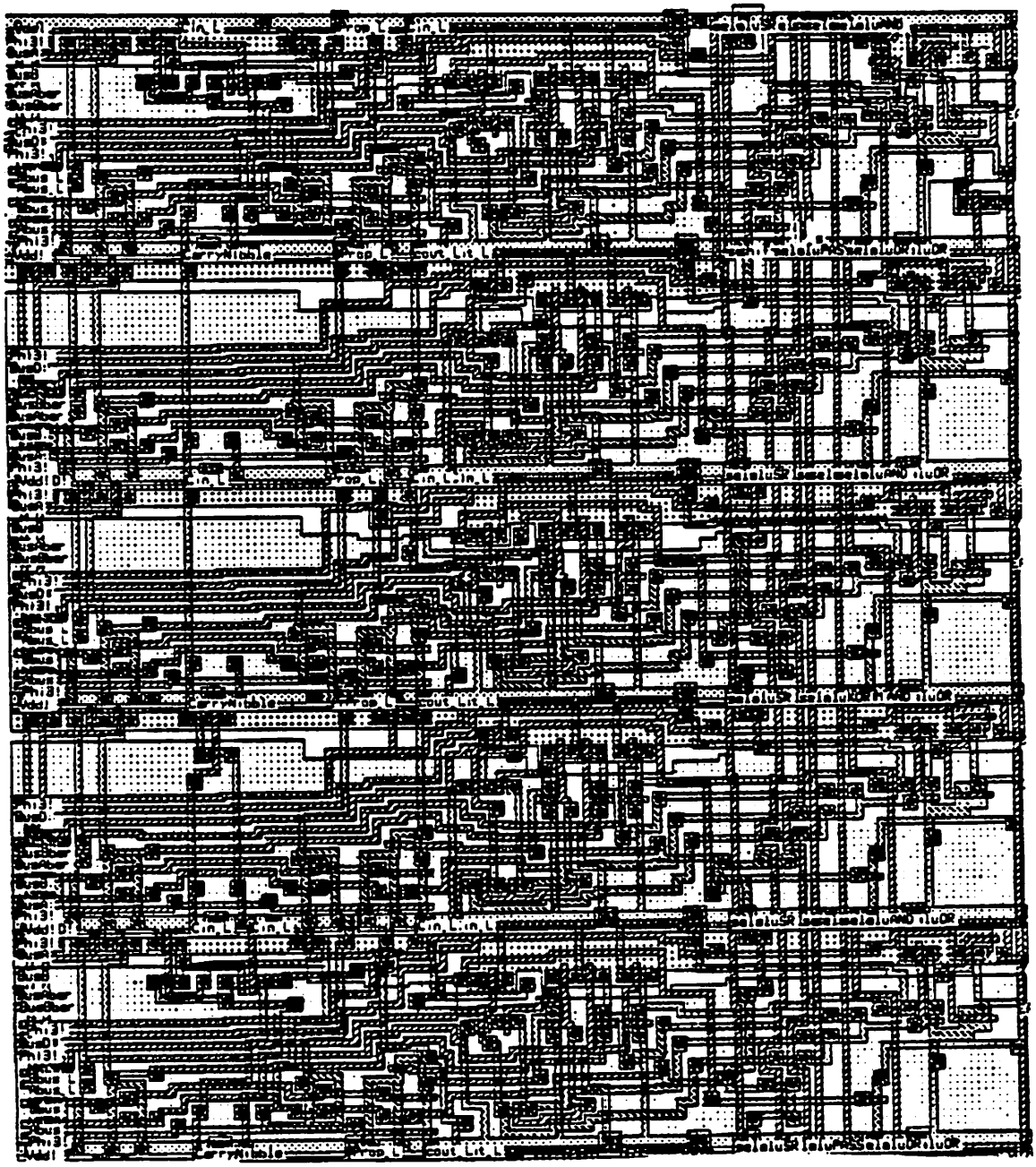


Figure 3.37b: Plot of Five Bits of the Dynamic ALU

The ADD function is not pure Domino in implementation: it works by employing *carry_in* and $\overline{carry\_in}$ to gate signals *XOR* and $\overline{XOR}$, respectively. Because the inverted signals are generated by simply adding an inverter to the original gate some caution must be exercised in dealing with signals controlled by these signals. In particular signals which

are logic *1* and are fed through an *n*-channel passgate may be degraded. Thus, while the succeeding buffer is not ratioed in the normal sense of a static inverter, still transistor sizing becomes important. By adjusting the pullup and pulldown sizes of the ALU output bus drivers it is possible to use an *n*-channel passgate multiplexer to select ALU functions.

3.9.5. Miscellaneous ALU Operations

Because the output buffer to the ALU output is precharged high only a logic *0* must be passed through the *n*-channel transmission gate. As part of its Smalltalk specialization the ALU operates in a 31-bit tagged mode as well as non-tagged 32-bit mode. The decoding scheme to control these modes is handled by a condition-code PLA. This preserves the bitslice regularity of the ALU. The ALU also provides a flag for the case when $A-B=0$. This is accomplished through a precharged line which is active high. It is connected as a 32-bit NOR and goes low when any bit or bits of the inverted *B*-operand does not match the corresponding *A*-operand bit. According to the SOAR architectural specification, this flag is only checked as a result of a subtract (SUB) operation. Therefore it may be driven directly by the $\overline{XOR}$ signal without requiring additional computation logic or time. Because it is implemented in this manner, however, the $A-B$ flag is only valid for the SUB instruction.

3.9.6. Comparison of Simulated and Measured ALU Delays

The bitslice *alufirst*, which is the first bit in every 4-bit block, was analyzed using CRYSTAL [oust83]. These results are compared with measured results from the a set of five chips which made up the first silicon batch. For simplicity only 1 bit, rather than all 32 bits, was analyzed. The capacitance and resistance values used by CRYSTAL are listed in Figure 3.38. These are relatively accurate figures for the MOSIS CMOS process. The capacitance figures for the *p*-channel devices in this *p*-well process are slightly pessimistic.

Parameter	Value	Units	Remarks
cperarea	0.0004	pF / μ^2	for nchan and pchan
cperwidth	0.00025	pF / μ	for nchan and pchan
metalcperarea	0.00003	pF / μ^2	first layer metal
metalresistance	0.03	$\Omega / \square$	
polycperarea	0.00004	pF / μ^2	inside or outside well
polyresistance	30.0	$\Omega / \square$	
diffcperarea	0.0001	pF / μ^2	inside or outside well
diffcperperim	0.0001	pF / μ	
diffresistance	10.0	$\Omega / \square$	

Figure 3.38: Process Parameters used in CRYSTAL Simulation

3.9.6.1. ALU Delay Simulations

For this analysis CRYSTAL's simple RC FET model (the default) was used. Results on bitslice *alufirst* for the three Boolean operations and ADD are given in Figure 3.39.

Function	CRYSTAL	Measured
ADD	41.70*ns*	45-50*ns*
XOR	23.69*ns*	40-45*ns*
AND	20.96*ns*	35-50*ns*
OR	17.48*ns*	30-40*ns*

Figure 3.39: Simulated versus Measured ALU Delays

It is expected that analysis of *alueven*, *aluodd*, and *alulast*, the remaining three bits in the four-bit block replicating unit, would show similar results. A version of *alufirst* laid out to contain metal signal runs in place of polysilicon runs in an attempt to reduce the RC time constant of these lines yielded the results shown in Figure 3.40.

Function	CRYSTAL
ADD	41.53*ns*
XOR	23.39*ns*
AND	20.91*ns*
OR	17.47*ns*
A—B=0	11.98*ns*

Figure 3.40: Simulated Results with Metal Signal Lines

The delay time differences are insignificant and the metal version was not fabricated. The delay times are similar probably because the on resistance of the minimum-sized FETs and the parasitic capacitances associated with their sources and drains are the overriding fac-

tors in determining RC delay. It is not completely clear, however, whether the timing simulator has correctly modelled the circuit delay.

The test equipment used was not able to measure the small single-bit delay times directly. Instead an indirect calculation was performed and hence some uncertainty is reflected in the tolerances in the measured values. The measured values also span the fastest and slowest of the five chips measured from the first lot. The test setup measured a no-carry ADD at around 95*ns*. A PASS or SR operation takes 45-50*ns* on chips where the ADD could also be measured. These latter operations do not depend on the ϕ clock signal to start. Their delay time is purely a measure of the amount of time it takes the latch control signal to travel across 32 bits. This time is roughly equivalent to the delay associated with the ϕ clock signal. Both must traverse 32 bits, and the same buffers are used in each case. Thus, by subtracting the clock delay time of 50-45*ns* from the no-carry ADD, the ADD time from clock pulse is seen to be 45-50*ns*. Similar calculations were performed for XOR, AND, and OR with the results shown in Figure 3.39 above.

The CRYSTAL simulations were based on output from the circuit extractor, MEXTRA [fitz82]. MEXTRA was run in a mode that reported the area and perimeter on all layers of all circuit nodes. From this information CRYSTAL can infer aspect ratios and make resistance estimates.

The CRYSTAL delay values are low in part because of the additional capacitance that each ALU bitslice output must drive. Only the ALU bitslice was simulated in CRYSTAL and no compensating capacitance to simulate a bus load was added. This capacitance should be small however. The ALU output travels about 100 microns in metal to the input passgate of the destination latch. The destination latch holds the ALU result. The ALU output bus (EAbus in SOAR) is driven by L=3μ W=20μ *p*-channel, L=3μ W=16μ *n*-channel buffers. CRYSTAL results do not reflect the additional delay time from the ALU output to the destination latch. The main delay-time component seems to be the slow speed of the actual arithmetic and Boolean logic itself. Minimum devices are used almost everywhere (exceptions are the *n*-channel pulldowns in inverters where a 7μ width, equal

to a contact cut, were easy to construct). Greater speed could be obtained by increasing the size of the logic gates. SPICE2 simulations, performed without extracted resistance values, did not show this speedup. This is probably because the line capacitance, which was modelled as a single lump, obscured the delay benefits of wider gates.

3.9.6.2. Measured ALU Speed

A detailed account of the test equipment and the testing procedures used to measure the ALU speed can be found in Appendix C. The best results from the functional chips tested are summarized in Figure 3.41. Not all chips were operational in all modes. Chip number 5 failed basic power-up tests and does not appear in this figure.

Function	Delay (*ns*)	Chips at Speed	Number Tested
ADD (wc)	140	1,3	3
SUB (wc)	125	2	3
Cp-ADD (wc)	100	1,2,3	3
Cp-SUB (wc)	90	1,2	3
No Cp (ADD/SUB)	95	1,2,3,4	4
XOR	90	2	4
AND	85	1,2	4
OR	80	2,3	4
SR	35	6	5
PASS	35	6	5

Figure 3.41: Summary of First Silicon Speed

The add and subtract times represent worst-case carry propagate figures. The table entries *Cp-ADD* and *Cp-SUB* are worst-case times for generation of C_{out} from bit 31. These times are faster than the add and subtract times because bit 31 falls on an integral block boundary. Therefore the *carry_out* signal only ripples through four bits and bypasses seven nibbles in the worst case. This is compared to a ripple of eight bits and a bypass of six nibbles for the add and subtract times. The difference in these figures indicates a ripple-carry time of 9-10*ns* per bit. The remaining table entries are self-explanatory. The more complex functions have longer delays. The simple pass and shift functions are limited by the time it takes the control signals (run in alternating poly and metal) to traverse 32 bits.

The control signals are buffered between bits 15 and 16. If one assumes an *RC* model for the control signals runs this effectively cuts delay time by a factor of four by halving both *R* and *C*. The control signal delay time could have been reduced even more if the signals had been run in polysilicide or second-layer metal. Second and third silicon chips have been fabricated but are not yet tested.

3.10. Summary

Simulations and comparisons of static and dynamic CMOS gates were presented in this chapter. The conclusion is that dynamic gates are faster in situation where the precharge phase can be hidden. They are more area compact as gate complexity increases and are easier to layout because typically each signal drives only one device per gate. Therefore, they are good candidates for an automated generation approach. In the second major portion of this chapter measurements and simulations of dynamic CMOS charge redistribution problems and gate delays were presented. Due to V_t variations of the test chips it is difficult to accurately predicate a series chain length limit, but it appears to be between 7 and 15 devices. The measured circuits were faster than the simulated versions. This can be attributed to higher device mobilities than those assumed by the circuit simulation models. In the final part of the chapter the design and operation of a 32-bit ALU laid out manually in the Domino style was examined. Measurements of the fabricated chip indicate speeds competitive with similar published designs. The circuit delay measurements were correlated with simulated results. Using the dynamic Domino scheme it is possible to construct a fast, area-efficient complex combinational circuit that consumes negligible static power. The Domino ALU serves to validate the dynamic approach used in the combinational synthesis framework described in the following chapters.

CHAPTER 4

The MAMBO Synthesis Package

This chapter serves both to introduce the MAMBO synthesis system and to explain the "front-end" tools in the package. MAMBO is a collection of tools organized as pipeline to help a designer realize a combinational circuit at the mask-level. The primary goal of the MAMBO package is to construct complex combinational functions which have been optimized for circuit delay and layout area. The designer specifies a function or set of functions by Boolean equations which are then mapped into combinational logic. The MAMBO package employs a context-based tiler to create the mask-level geometries. This realizes a second goal of the package which is to be relatively process independent. New tiles must be designed to reflect process changes but the tile assembly tool itself does not have to be altered.

4.1. Overview of the MAMBO Pipeline

The synthesis of a combinational logic function can be broken down into four broad areas. The areas are *logic minimization*, *electrical design*, *topological compaction*, and *physical layout*. The synthesis process is illustrated in Figure 4.1. The tools used in each phase of the pipeline are also mentioned.

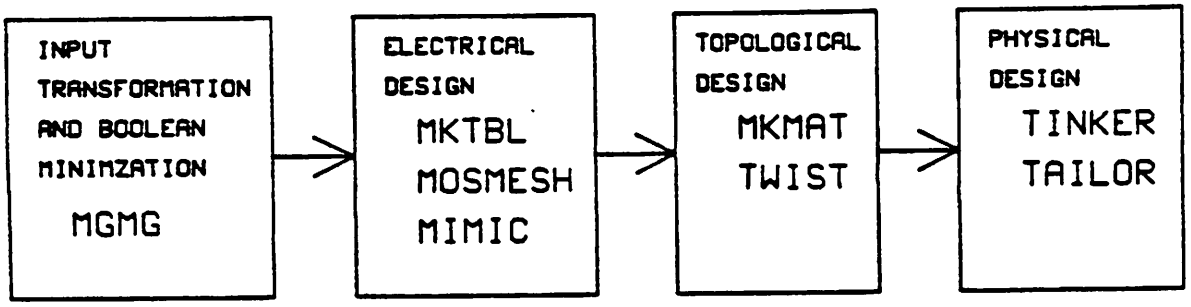


Figure 4.1: Stages in Synthesis Process

The input, output, and nature of each of the tools are examined more closely in the following sections.

4.1.1. Input Transformation and Logic Minimization

The MGMG program performs two services. First, it is the high-level interface to the designer. It is used to parse and translate Boolean expressions into a specified *target technology*. Second, MGMG will optionally perform two-level expansion of an n-level input expression. It is assumed that the input Boolean equations are already in a logic-optimized form. The equations may have been optimized by any of the methods mentioned in Chapter 1. MGMG, however, can apply simple logic minimization rules to reduce circuit complexity. Though MGMG can be used to produce a truth-table-like output, in the MAMBO system it produces a gate-level netlist. This netlist specifies the function of each gate and how it is connected with other gates. Each function is represented as a single, complex gate in the netlist. A LISP-like syntax is used. A complete description of target-technology transformations and of the types of logic minimization that MGMG provides is given in the latter part of this chapter and an alternate logic minimization pathway is also described.

4.1.2. Electrical Design

The goal of electrical optimization is to decrease the delay from input to output of the synthesized circuit. It is assumed that all signals are stable and valid at the beginning of the evaluate phase in Domino-style logic. The delay optimization program MOSMESH works by breaking up the large, complex gates generated by MGMG into smaller, manageable pieces. This process is termed *partitioning*. The partitioning depends both on electrical and physical factors. The electrical constraints are the charge redistribution effect seen in dynamic circuits and the direct effect of series chain length on gate speed. These effects were described in detail in previous chapters. For ease of routing and automated generation it is best to have a regular layout structure. Structure regularity imposes physical constraints on MOSMESH. If the designer wishes to use a more complex gate interconnection scheme the regularity restriction can be removed.

MOSMESH manipulates a gate netlist by first finding the electrical critical path through the circuit. The critical path is found by recursively tracing each function output back to its fundamental inputs. The tool MKTBL is used to produce a table of delays of various gate clusters under various conditions. MOSMESH refers to this table in its search for the longest delay path. MKTBL constructs a SPICE2 deck of a set of user-provided gate configurations. The program then determines the critical path through the particular gate under various conditions, performs transient analyses, and stores the results in a table.

Finally, the partitioned pieces of the circuit are examined to see if any are redundant. Duplicate gates may be eliminated to reduce layout area if they do not compromise circuit speed. The MIMIC tool performs this task by recursively checking for gates with identical inputs and functionality, but different outputs. Typical reductions in cluster count vary from 10 to 40%. The programs involved in electrical design are examined further in Chapter 5.

4.1.3. Topological Design

The MKMAT tool processes the netlist produced by MIMIC: it transforms the netlist into a connectivity matrix. It does this by replacing each gate with a set of symbols which declare whether a given input or output signal affects or is affected by a particular gate. The result is a matrix of characters quite similar to a PLA's personality matrix. To assemble this matrix MKMAT must internally build two constraint matrices. Part of the job of MOSMESH is to constrain signal ordering for least delay. This means that certain signals (those that change fastest) are generally assigned to transistors that are closer to output nodes than those signals which change more slowly. This ordering constraint must be observed. Also, it is often the case that a gate can be realized in a single column of the connectivity matrix. However, for more complex gates, especially those which are parallel in function at their top level, this is not the true. For such gates a column-constraint matrix expresses which columns must be contiguous. MKMAT builds the connectivity matrix based on these constraints. An example of a connectivity matrix appears in Figure 4.2 and in Chapter 6 where MKMAT is described in detail.

```
new 30 19

                    pssssssssssssp.ssss
18                  ..s.............o.
20                  ...s............o..
14                  ....s...........o...
24                  .......s...o........
26                  ........s.o.........
f0                  o...................
1                   po..................
2                   p.o.................
3                   p..o................
4                   p...o...............
6                   p....o..............
10                  p.....o.............
11                  p......o............
12                  p.......o...........
16                  .s..................o
c3*                 .ss.............s..
a0                  .s.s.....s.....
b0                  ..ssp......s.....
15                  .....s......o......
7                   .....s.......o.....
28                  ......s..o..........
c2                  ......sss........ss.
c3                  ......sss...s..s...
cin                 .........s..........
b0*                 .........ss...s.....
a0*                 .........s.s.s......
cin*                ..........ss........
c1                  ............s...s.s
c2*                 ............s..s..s
c1*                 ..............s.s.
```

Figure 4.2: Example Connectivity Matrix

Program TWIST reads the matrix structure and attempts to compact it topologically. The designer may run this program either interactively or as a segment in a pipeline. TWIST respects external constraints. External signals must be brought to the edge of the circuit for connection off-module and, in addition, the designer can specify on which edge (left, right, or both) each external signal must appear. TWIST performs simple column folding and multiple row folding. Simple column folding is used so buffers, which are

required on a per-gate basis, can be brought out at the top or bottom of the array. Multiple row folding (with internal signals in the middle) increases layout density. TWIST can perform either row-after-column or column-after-row folding or any mix in between. The circuit designer makes such a decision generally based on which aspect ratio (tall and thin or short and squat) is most favorable for layout. The general folding problem is NP-complete; TWIST employs a series of heuristics to accomplish area compaction. In Chapter 6 the theoretical aspects of the algorithms used by TWIST are presented.

4.1.4. Physical Design

The output from TWIST is a (possibly row- and column-folded) connectivity matrix. This is the input to TINKER which is a context-based electrical tiler. TINKER generates an electrical matrix based on the connectivity matrix and a separate, user-provided, *context* file. This file gives possible situations for each of the characters in the connectivity matrix. Each connectivity symbol represents a gate, an interconnection, or a cluster of gates (such as a buffer). Depending on the number of signals a gate must drive, or the number of devices in a series chain, a given connectivity symbol can be transformed into various characters in the electrical array. By this approach TWIST performs purely topological operations and is not required to deal with implementation considerations. On the other hand the last tool in the package, TAILOR, need make only mask-level decisions, and does not require knowledge about device sizing or drive capabilities. The output of TINKER is another matrix of the same aspect ratio as its input, but drawn from a richer set of characters.

TAILOR is the final program in the MAMBO pipeline. It interprets the character set in the matrix provided by TINKER. For each character it refers to a cell library. TAILOR simply performs a one-for-one substitution of each character for mask geometries from the cell library. The cell library can be described in any layout language. The current implementation stores the cells in CIF format, but TAILOR has no knowledge of this and is therefore fairly process independent. TAILOR gets information about tile extent from a separate *symbol* file. This file provides TAILOR with the designer's view of the cell. The

designer can have TAILOR generate abutting, overlapping, or completely enclosed cells by giving such information in the *symbol* file. The output structure is regular and TAILOR is careful to preserve constant row and column spacing on a per row and column basis. That is, all cells in a given row must be the same height, all cells in a given column must be the same width. The user may override this rule if he desires. Examples of both the electrical rules matrix and final mask-level layout are shown in Figure 4.3. The physical design process, used to construct these figures, is presented in detail in Chapter 7.

```
ssssssp.sssss
+r++l||br+++
+rr+++++r+++
+b+++++++++r
r+rr++++++++
r-++r+++r+++
+T++++d+++++
bR++++++++++
m++rr+++++++
T+r+++++++++r
++b++++++++r+
R+-++~+++~r+
++R|||~r++++
++L+++++++++
++L+++++d+++
+bL+++++++++
++L++++++c++
++L++++++++d+
++L+++++++++d
b+L||r+r++++
++K|||r++Rrr
R++d|||r++++
+++-e----~--
ssp
```

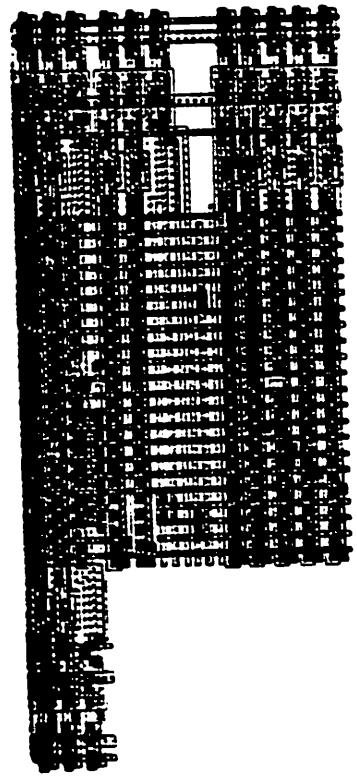


Figure 4.3: (a) Electrical Rules Matrix (b) Mask-Level Layout

4.2. Representation of Boolean Expressions— MGMG

The remainder of this chapter is devoted to the first of the four major stages in combinational logic synthesis, that of Boolean transformation and minimization. This stage is handled by the MGMG program, the first program in the seven segment MAMBO pipeline.

Figure 4.4 shows the input equations for a 2-bit parallel adder.

```
/*
 * 2-bit parallel adder example
 */
INORDER = cin a0 b0 a1 b1 ;
OUTORDER = sum0 sum1 cout1;

sum0  = (!a0&!b0&cin) | (!a0&b0&!cin) | (a0&!b0&!cin) | (a0&b0&cin) ;
sum1  = (!a1&!b1& ((b0&cin) | (a0&cin) | (a0&b0))) |
        (!a1&b1&! ((b0&cin) | (a0&cin) | (a0&b0))) |
        (a1&!b1&! ((b0&cin) | (a0&cin) | (a0&b0))) |
        (a1&b1& ((b0&cin) | (a0&cin) | (a0&b0))) ;
cout1 = (b1&cin) | (a1&((b0&cin) | (a0&cin) | (a0&b0))) | (a1&b1) ;
```

Figure 4.4: Input Format, 2-Bit Parallel Adder

The equations have been expressed using a standard set of Boolean operators. The operators and their operations are shown in the Figure 4.5 below.

Operator	Operation	Class
!	negation	monadic
&	disjunction	dyadic
|	conjunction	dyadic

Figure 4.5: Boolean Operators

The operations in this figure form a logically complete set. It is possible to define other operations, for example, *exclusive-or*, and add them to the set of legal input operations of MGMG for convenience of expression.

The program MGMG can be used to manipulate this input format in a variety of ways. Figure 4.6 shows the input example of Figure 4.4 transformed into two-level logic and expressed as the *personality matrix* of a PLA.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1

-0001  010
00001  010
-0010  010
00010  010
100--  100
00-01  010
001--  100
00-10  010
0-010  010
11-00  010
-1100  010
-1111  010
-111-  001
111--  100
0-001  010
11-11  010
11-1-  001
1-111  010
1-11-  001
1---1  001
---11  001
010--  100
1-100  010
```

Figure 4.6: Personality Matrix, 2-Bit Parallel Adder

Figure 4.7 explains each of the AND and OR plane characters.

Symbol	Plane	Interpretation
1	AND	variable affects product term
1	OR	product term affects output term
0	AND	negated variable affects product term
0	OR	product term does not affect output term
—	AND	variable does not affect product term

Figure 4.8: PLA Personality Matrix Symbols

By invoking MGMG with the *—expand* option it is possible to see more directly the effect of each input variable on the output variables. Figure 4.8 was produced by employing this option on the 2-bit adder example.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1

!a0 !b0 !a1 b1  sum1
!cin !a0 !b0 !a1 b1  sum1
!a0 !b0 a1 !b1  sum1
!cin !a0 !b0 a1 !b1  sum1
cin !a0 !b0  sum0
!cin !a0 !a1 b1  sum1
!cin !a0 b0  sum0
!cin !a0 a1 !b1  sum1
!cin !b0 a1 !b1  sum1
cin a0 !a1 !b1  sum1
a0 b0 !a1 !b1  sum1
a0 b0 a1 b1  sum1
a0 b0 a1  cout1
cin a0 b0  sum0
!cin !b0 !a1 b1  sum1
cin a0 a1 b1  sum1
cin a0 a1  cout1
cin b0 a1 b1  sum1
cin b0 a1  cout1
cin b1  cout1
a1 b1  cout1
!cin a0 !b0  sum0
cin b0 !a1 !b1  sum1
```

Figure 4.8: Expanded Format, 2-Bit Parallel Adder

When MGMG is invoked with either *personality matrix* or *expand* options an input expression or set of input expressions (possibly many levels deep) is transformed into two-level logic. In this process MGMG prefroms some simple Boolean minimization by reducing the number of product terms in the expansion. Standard cube covering algorithms are used [bray84c]. For example, if the strings:

100011--00011 0001

1000110100011 0001

represent the product terms (*pterms*) for a certain set of variables then the first pterm is said to *cover* the second in that the second pterm is a subset of the values of the input variables represented by the first pterm. Therefore the second term is unnecessary and

may be eliminated. Likewise, given the two pterms:

1000110000011 0001

1000111100011 0001

A new pterm can be generated with covers both:

100011--00011 0001

This is called a *distance one merge*. Lastly, given the pterms:

1000110100011 0001

1000110100011 0010

which have identical input values but which drive different outputs the new pterm:

1000110100011 0011

can be generated which covers both output variables. MGMG performs these simplifications by creating a unique signature for each pterm based on its input string. The signature is in the form of a hash function and is used to store the pterms in a binary tree. After each new pterm's hash function is computed it is either inserted into the binary tree if it is unique or discarded as redundant if its signature is equivalent to a pterm already in the tree. This type of minimization in product term cardinality is based purely on the 1/0/- signature of the pterm. MGMG has no notion of the Boolean relation of one variable to another, hence it cannot perform more sophisticated minimizations based on the rules of Boolean mathematics.

4.3. Transformation into Target Technology— MGMG

Though two-level logic expansions are often employed in the generation of combinational circuits, and MGMG is capable of producing minimized two-level expressions of Boolean functions, it is not always necessary or profitable to expand a function in this way. In fact, when the target technology is Domino CMOS, it is often necessary that gates with large fan-ins be broken into smaller multiple gates because of the problem of charge redistribution. Two possible ways of performing this fracturing of large gates are: 1) First expand n levels of hierarchy into two, minimize the functions, because this is a well

understood process for the two-level case, and then re-introduce hierarchy afterwards to reduce fan-in. 2) Retain the designer's original intent as much as possible by preserving the input hierarchy with the transformations necessary for the target technology. Partitioning will still be required but the end result will be closer to the original input. Minimization of common gates can be performed after the gates are broken up. Such post-partitioning minimization could also be performed for the first approach mentioned. The first approach has been explored in detail by Brayton *et. al.* and was reviewed in Chapter 1. This dissertation explores the second option for a number of reasons: The problem of realizing a correct, efficient, complex circuit on silicon through automated generation is multi-faceted. It is felt that the low level details of circuit construction, such as parasitic capacitance effects, required in-depth study. The second approach is simpler and thus more time can be spent identifying and studying problems at the electrical and layout levels. Also, since the second approach retains more of the designer's original intent it gives the designer more control over the final result. In the sections below options to the delay optimization stage of the pipeline are detailed. The options allow the designer to direct the circuit partitioning via specific constraints or according to a built-in clustering algorithm. It is anticipated that future versions of MAMBO will use more sophisticated multi-level logic synthesis techniques. Even in that case, however, detailed electrical and constraint management will still be required.

4.3.1. Input Parsing and Netlist Generation

The initial step in the transformation of Boolean expressions into a particular technology is parsing of the input format and generation of an netlist. The netlist specifies explicitly the type, fan-in, and fanout of all gates. In this case the target technology is Domino CMOS and the *gateset* is {*AND OR*}. The logic connectivity is also specified. For the 2-bit parallel example used above the input netlist for Domino-style design is shown in Figure 4.9.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1

# Input expression
NOT 14 : a0
NOT 16 : b0
AND 13 : 14 16
AND 12 : 13 cin
NOT 21 : a0
AND 20 : 21 b0
NOT 24 : cin
AND 19 : 20 24
OR 11 : 12 19
NOT 29 : b0
AND 27 : a0 29
NOT 31 : cin
AND 26 : 27 31
OR 10 : 11 26
AND 34 : a0 b0
AND 33 : 34 cin
OR sum0 : 10 33
# Input expression
NOT 42 : a1
NOT 44 : b1
AND 41 : 42 44
AND 48 : b0 cin
AND 51 : a0 cin
OR 47 : 48 51
AND 54 : a0 b0
OR 46 : 47 54
AND 40 : 41 46
NOT 59 : a1
AND 58 : 59 b1
AND 65 : b0 cin
AND 68 : a0 cin
OR 64 : 65 68
AND 71 : a0 b0
OR 63 : 64 71
OR 64 : 65 68
AND 71 : a0 b0
OR 63 : 64 71
NOT 62 : 63
AND 57 : 58 62
OR 39 : 40 57
NOT 77 : b1
AND 75 : a1 77
AND 82 : b0 cin
AND 85 : a0 cin
OR 81 : 82 85
AND 88 : a0 b0
OR 80 : 81 88
NOT 79 : 80
AND 74 : 75 79
OR 38 : 39 74
AND 92 : a1 b1
AND 97 : b0 cin
AND 100 : a0 cin
OR 96 : 97 100
AND 103 : a0 b0
OR 95 : 96 103
AND 91 : 92 95
OR sum1 : 38 91
# Input expression
AND 107 : b1 cin
AND 114 : b0 cin
AND 117 : a0 cin
OR 113 : 114 117
AND 120 : a0 b0
OR 112 : 113 120
AND 110 : a1 112
OR 106 : 107 110
AND 123 : a1 b1
OR cout1 : 106 123
```

Figure 4.9: Input Netlist, 2-Bit Parallel Adder

Each entry in the netlist is of the form:

gate_type *output* **:** *input0 input1 ... inputn*

Gate_type is a gate from the given target technology. In this case the gates AND and OR are allowed. The gate type NOT is always permitted because only logically complete gatesets are allowed. *Output* and *inputi* are a signal names. Signal names beginning with

alphabetic characters are user-given, all other signal names have been generated and represent intermediate values used in the computation of a particular function.

The netlist contains AND, OR, and NOT gates. Inverters are not permitted in Domino logic and hence the netlist as it stands cannot be implemented. This problem is overcome by "bubble pushing" that is, pushing the inversions to the bottom, or leaf-level, of the function using the theorems of Boolean algebra [nagl75]. As an inversion or "bubble" passes through a gate from output to input it may change the gate's function. Figure 4.10 below lists the transformation which occur due to bubble pushing on various gate types for inputs and outputs. The entry *TRUE* means the signal is invariant under the transformation while the entry *FALSE* indicates the signal is complemented under the transformation.

Transformations on Inputs	AND	OR	NAND	NOR
AND	TRUE	FALSE	TRUE	FALSE
OR	FALSE	TRUE	FALSE	TRUE
NAND	TRUE	FALSE	TRUE	FALSE
NOR	FALSE	TRUE	FALSE	TRUE

Transformations on Outputs	AND	OR	NAND	NOR
AND	TRUE	FALSE	FALSE	TRUE
OR	FALSE	TRUE	TRUE	FALSE
NAND	FALSE	TRUE	TRUE	FALSE
NOR	TRUE	FALSE	FALSE	TRUE

Figure 4.10: Transformations of Inputs and Outputs for Various Gate Types

It was found convenient to store all logic equations, regardless of their target technology, in a canonical form. The canonical form is composed of AND and OR gates with inversions pushed to the inputs. For the 2-bit adder example the canonical form is given in Figure 4.11.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1
# Re-canonicalized expression
NOT 11 : a0
NOT 13 : b0
AND 10 : 11 13 cin
NOT 17 : a0
NOT 20 : cin
AND 16 : 17 b0 20
NOT 24 : b0
NOT 26 : cin
AND 22 : a0 24 26
AND 28 : a0 b0 cin
OR sum0 : 10 16 22 28
# Re-canonicalized expression
NOT 33 : a1
NOT 35 : b1
AND 38 : b0 cin
AND 41 : a0 cin
AND 44 : a0 b0
OR 37 : 38 41 44
AND 32 : 33 35 37
NOT 48 : a1
NOT 52 : b0
NOT 54 : cin
OR 51 : 52 54
NOT 57 : a0
NOT 59 : cin
OR 56 : 57 59
NOT 62 : a0
NOT 64 : b0
OR 61 : 62 64
AND 47 : 48 b1 51 56 61
NOT 68 : b1
NOT 71 : b0
NOT 73 : cin
OR 70 : 71 73
NOT 76 : a0
NOT 78 : cin
OR 75 : 76 78
NOT 81 : a0
NOT 83 : b0
OR 80 : 81 83
AND 66 : a1 68 70 75 80
AND 89 : b0 cin
AND 92 : a0 cin
AND 95 : a0 b0
OR 88 : 89 92 95
AND 85 : a1 b1 88
OR sum1 : 32 47 66 85
# Re-canonicalized expression
AND 98 : b1 cin
AND 104 : b0 cin
AND 107 : a0 cin
AND 110 : a0 b0
OR 103 : 104 107 110
AND 101 : a1 103
AND 113 : a1 b1
OR cout1 : 98 101 113
```

Figure 4.11: Canonical Form, 2-Bit Parallel Adder

4.3.2. Transformations on Canonical Form

Finally, after translation into canonical form, the target-technology transformation is performed. For the Domino design style the target form is identical to the canonical form. The syntax is, however, modified somewhat to facilitate later parsing of the expression. A LISP-like syntax has been chosen. Again using the 2-bit parallel adder as an example, the final transformation is shown in Figure 4.12.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1

# Transformed expression
(o sum0
(p
 (s a0* b0* cin )
 (s a0* b0 cin* )
 (s a0 b0* cin* )
 (s a0 b0 cin ) ));
# Transformed expression
(o sum1
(p
 (s a1* b1*
  (p
   (s b0 cin )
   (s a0 cin )
   (s a0 b0 ) ) )
 (s a1* b1
  (p b0* cin* )
  (p a0* cin* )
  (p a0* b0* ) )
 (s a1 b1*
  (p b0* cin* )
  (p a0* cin* )
  (p a0* b0* ) )
 (s a1 b1
  (p
   (s b0 cin )
   (s a0 cin )
   (s a0 b0 ) ) ) ));
# Transformed expression
(o cout1
(p
 (s b1 cin )
 (s a1
  (p
   (s b0 cin )
   (s a0 cin )
   (s a0 b0 ) ) )
 (s a1 b1 ) ));
```

Figure 4.12: Domino Target Technology, 2-Bit Parallel Adder

For comparison, Figure 4.13 shows the same function expressed in NAND gates only. Such a transformation might be useful if the circuit were built using bipolar devices.

```
#INORDER = cin a0 b0 a1 b1
#OUTORDER = sum0 sum1 cout1

# Transformed expression
(o sum0
(s*
 (s* a0* b0* cin )
 (s* a0* b0 cin* )
 (s* a0 b0* cin* )
 (s* a0 b0 cin ) ));
# Transformed expression
(o sum1
(s*
 (s* a1* b1*
  (s*
   (s* b0 cin )
   (s* a0 cin )
   (s* a0 b0 ) ) )
 (s* a1* b1
  (s* b0 cin )
  (s* a0 cin )
  (s* a0 b0 ) )
 (s* a1 b1*
  (s* b0 cin )
  (s* a0 cin )
  (s* a0 b0 ) )
 (s* a1 b1
  (s*
   (s* b0 cin )
   (s* a0 cin )
   (s* a0 b0 ) ) ) ));
# Transformed expression
(o cout1
(s*
 (s* b1 cin )
 (s* a1
  (s*
   (s* b0 cin )
   (s* a0 cin )
   (s* a0 b0 ) ) )
 (s* a1 b1 ) ));
```

Figure 4.13: NAND Technology, 2-Bit Parallel Adder

The example chosen to illustrate the steps that MGMG goes through through is a simple one. For more complicated circuits MGMG also performs compression of "even-function" gates. The AND function is even: if inputs to such a function are asserted high the output is asserted high. In contrast, the opposite happens with a NAND gate. As a

consequence, one AND gate feeding another can be collapsed into a single, large AND gate. However, a hierarchy of NAND gates cannot be merged. This collapsing or *compression* of even gates is performed automatically by MGMG. The gates may be re-fragmented later to correct charge sharing problems, if they exist.

4.4. Alternate Transformation into Target Technology

Much of the gain in Boolean minimization of logic comes from the way in which the designer expresses his equations initially. The idea behind MGMG is to preserve the designer's intent. If the equations have been machine-generated, however, Boolean minimization at this stage may be beneficial. If the target gates are constrained to two-level logic, one can make use of well-known, fast minimization heuristics such as those employed in ESPRESSO [rude85]. The translation program EQNTOTT [cmel81] is first invoked to translate the multi-level logic into a two-level personality matrix. ESPRESSO is then run on the matrix and it attempts to minimize the functions by more general applications of the covering and merging operations mentioned above. Normally ESPRESSO attempts to reduce the number of cubes. However, in the MAMBO synthesis pipeline, literal count is sometimes a better optimization parameter since it maps directly into actual device count. ESPRESSO has an option to perform this type of optimization as well. The results from the two differing approaches are contrasted in Chapter 8. A third possibility, which MGMG implements, is to perform two-level minimization while attempting to preserve the original form of each function. This means that, even though a multi-level expression is expanded to two levels, the top level will remain constant, whether it is AND or OR. By contrast, since ESPRESSO is more tuned to PLA implementations, the top-level is always OR— which may not give a circuit configuration efficient in area or speed.

4.5. Summary

The MAMBO synthesis package was introduced in this chapter. The MAMBO package addresses four major areas. These are: 1) Input transformation of Boolean equations and

logic minimization; 2) Electrical design, in which delay optimization and circuit partitioning issues are addressed; 3) Topological design which attempts to reduce circuit area; and 4) Physical design which considers the mask-level construction of the circuit. The MGMG program, which transforms Boolean equations into a target technology, was described in detail in this chapter. It is assumed the equations have been logic-optimized. MGMG transforms the input into a canonical form. This canonical form can be translated into gatesets for implementation in NAND-only, NOR-only, or other technologies. In MAMBO the translation is into AND-OR form with inversions pushed to the leaf-level. This form is suitable for implementation in Domino CMOS.

CHAPTER 5

Delay Optimization and Partitioning of Dynamic Meshes

Algorithms for optimization of delay in multi-level combinational circuits are explored in this chapter. The electrical constraints of charge redistribution, series chain length, input signal type and output buffer size direct the partitioning of complex functions into a simpler set. Physical factors, such as those limiting the number of logic levels in any given gate, impose additional constraints on delay optimization. The optimization algorithms are implemented in the MOSMESH tool. A simple two-transistor model is derived which allows the modeling of complex meshes of charged and discharged devices. The model is an accurate predictor of transient delay in the Domino domain and does not require as much computational effort to evaluate as the full mesh it replaces.

The MIMIC tool is also described. MIMIC traverses the optimized circuit to determine whether any of the partitioned clusters are redundant. MIMIC removes duplicate gates from the network so long as they do not compromise circuit speed.

5.1. Partitioning of Transformed Gates— MOSMESH

The output from MGMG or ESPRESSO may be written to a temporary file or may be piped directly to MOSMESH. MOSMESH reads the LISP-like set of expressions and produces a partitioned set of gates in the same syntax. A partition or *cluster* of devices which make up a single, complex Domino circuit is considered a legal gate if it meets several criteria. First, the gate must not have a charge redistribution problem, even under worst-case conditions. Second, at the user's option, the number of devices allowed in series may be constrained further. Third, the depth of each partition or cluster may be limited at the user's request. Currently the maximum partition depth allowed is two. This limitation is placed by the automated layout section of the MAMBO pipeline which cannot handle more com-

plex clusters efficiently. However, MOSMESH itself places no restriction on gate complexity and if a different layout technique is chosen this optional constraint can be removed.

5.2. The Charge Sharing Criterion in MOSMESH

In Chapter 2 the problem of charging sharing and redistribution in dynamic circuits was examined. In the MOSMESH program the goal is to determine whether or not a given circuit configuration is prone to this charge sharing problem. The criterion for charge sharing was defined by Equation 2.6. MOSMESH makes reference to a technology file provided by the user to obtain process-dependent quantities in the calculation of charge redistribution. A sample technology file for a 3μ p-well CMOS process is shown in Figure 5.1. Capacitances are given in F/M or F/M^2, lengths in meters, and areas in square meters.

While it is possible for the program to automatically calculate the inverter switching threshold V_{TH} from more fundamental values, instead the threshold voltage has been made a command line option. By this method the user may elect how optimistic or conservative a specification he wishes to design to. As MOSMESH tries each new cluster arrangement it solves Equation 2.6. This calculation is computationally simple, the only values which must be freshly calculated are the number of devices contributing to the *head* and *middle* capacitances.

5.3. Data Structure for Gate Partitioning

After being placed in a LISP-like syntax by MGMG it is a simple task to build a tree data structure whose atomic elements are data-nodes and whose inter-nodal links reflect the electrical connectivity of the circuit. A *node* in the data structure in this case holds a logically grouped set of transistors. The node data structure is shown in Figure 5.2. For the purposes of the presentation here many of the fields can be neglected, others are self-explanatory. Briefly, the important fields are *value* which is the number of devices that this node covers; *top* which declares this node to be an output node, that is the top node in a newly-formed cluster; and *seen* which tells the program whether this node has been

```
nchan
Cgso          1.3e-10
Cgdo          1.3e-10
Cgbo          4.1e-10
Cj            6.0e-4
Cjsw          4.0e-10
Tox           5.5e-8

pchan
Cgso          1.3e-10
Cgdo          1.3e-10
Cgbo          4.1e-10
Cj            4.1e-4
Cjsw          2.5e-10
Tox           5.5e-8

inverter
Pwidth       16.0e-6
Plength       3.0e-6
Nwidth        8.0e-6
Nlength       3.0e-6

pullup
Width         7.0e-6
Area         21.0e-12
Perimeter    20.0e-6

core
Width         4.0e-6
Area         12.0e-12
Perimeter    14.0e-6
On_Resist     5.0e+3
:
```

Figure 5.1: Technology File with Process-Dependent Parameters

examined yet. *Pathhead*, *pathtail*, and *middle* represent the number of source/drain regions at the head, tail, and middle of a cluster, respectively. These values are only valid when *top* is TRUE. *Peer* and *kid* represent the next node on this level and one level down, respectively. The data structure is traversed using these pointers.

```
struct node {
  int id;              /* unique id for tracing                */
  char type;           /* series(s) or parallel(p)             */
  int value;           /* number of FETs this branch           */
  int top;             /* Boolean, T if node is head           */
  int crit;            /* Boolean, T if node on cpath          */
  int seen;            /* Boolean, T if node marked            */
  int partition;       /* # of levels to expand gate           */
  double delay;        /* valid delay to here if head          */
  int pathhead;        /* # edges incident wc pathhead         */
  int pathtail;        /* # edges incident wc pathtail         */
  int parallel;        /* # parallel children                  */
  int middle;          /* # S/D jcts - ph,pt wc path           */
  int depth;           /* depth in mesh from Root (==1)        */
  int lasthi;          /* node number closest to Vdd           */
  int lastlo;          /* node number closest to GND           */
  int next_output;     /* node # this sub-cluster drives       */
  char *output;        /* ptr to name this sub-c drives        */
  signal *ext;         /* ptr to external sig l-list           */
  node *peer;          /* next, this level                     */
  node *kid;           /* to lower level                       */
};
```

Figure 5.2: Node Data Structure

The input expression of a small series-parallel circuit is shown in Figure 5.3a. Some of the steps in the transformation of this single gate into its final form, which is a circuit made up of two clusters, are shown in Figure 5.3b.

```
(o out
  (s a b c d e
        (p f g h i )))
```

Figure 5.3a: Input Expression of a Simple Series-Parallel Circuit

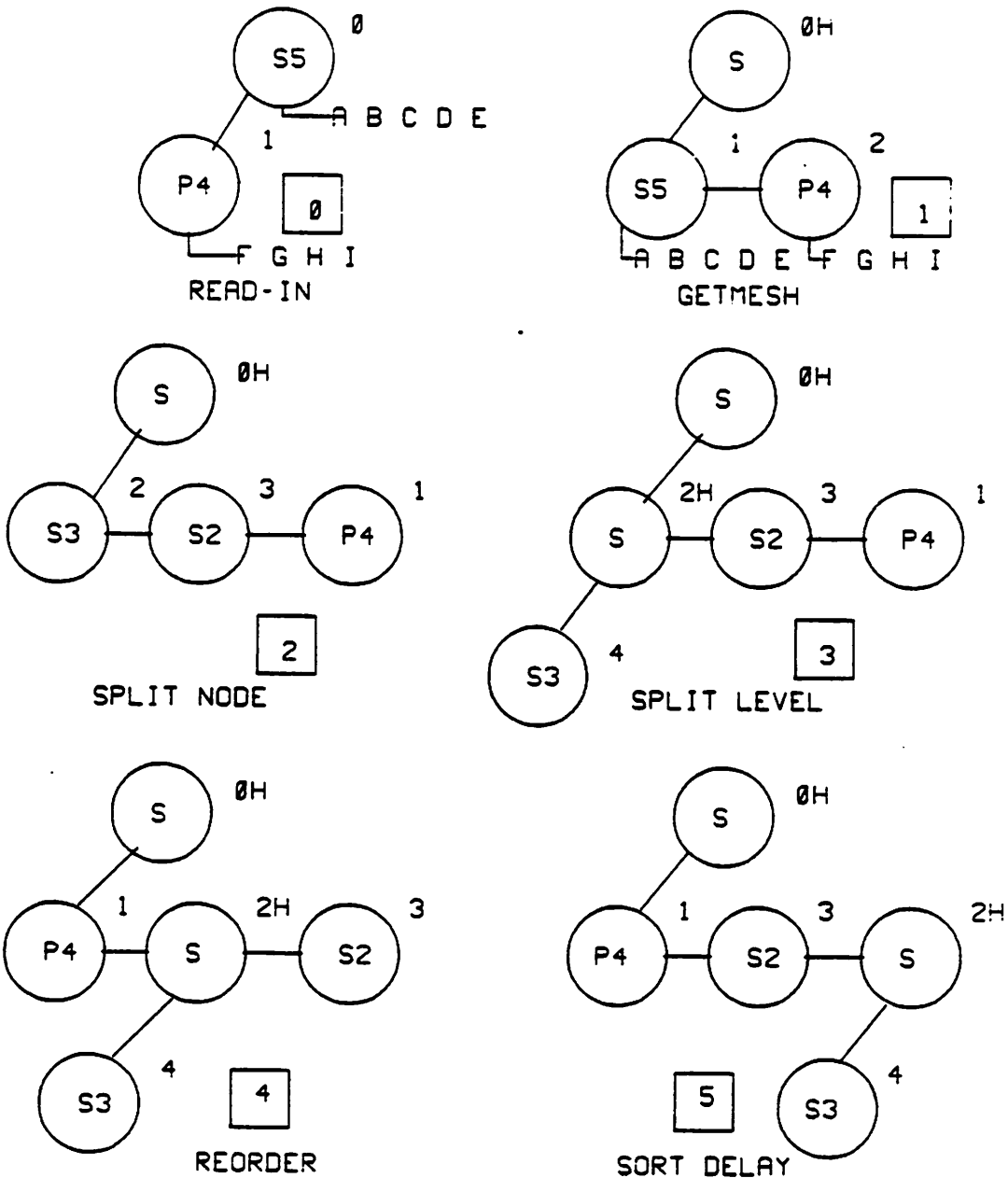


Figure 5.3b: Steps in Transformation of Example in 5.3a

The notation *s5* denotes a *series* node with 5 *external* signals; *p4* denotes a *parallel* node with 4 attached devices. An external signal, as defined in Chapter 2, is a signal which comes from outside the module and becomes stable during the precharge period. A *module* is a collection of Domino clusters. Internal signals are generated within the logic of the

module and stabilize during the evaluation period of the module. MOSMESH automatically fragments circuit nodes when it detects a charge sharing problem. In addition, in this case, the user has asked the program to limit the number of devices in a series chain to four. The user may limit partition depth, which is a measure of the layout complexity of the circuit. A simple series or parallel circuit has a partition depth of one, a series-parallel or parallel-series circuit has a depth of two, and so forth. In this example partition depth has not been limited, but it cannot exceed two since that is the depth of the original circuit. The transformation steps are now examined.

5.4. The Partitioning Algorithm

The top-level of the partitioning algorithm appears in Figure 5.4. After reading in the expression of Figure 5.3a the data structure contains two data-nodes as shown in Frame 0 of Figure 5.3b. The first step is to separate internal and external signals by placing them in separate data-nodes. Procedure *getmesh()* reads in the expression and performs internal/external signal separation. This is shown graphically in Frame 1 of Figure 5.3b. The presence of an internal signal is indicated by a non-nil *kid* pointer on a data-node. Also in Frame 1 of Figure 5.3b data-node *0* has been marked *H* meaning it is a *top* or *head* data-node. The root data-node is always a head data-node. Procedure *split_node()* fragments the *s5* data-node as shown in Frame 2 of Figure 5.3b. *Split_node()* breaks a single data-node into n pieces where n is $\left\lceil \frac{\#\ data-node\ devices}{max\ \#\ allowed} \right\rceil$. The last allocated data-node may get an odd remnant. Fragmenting data-nodes to meet the series restriction is not enough, however. From head data-node *0* it still appears that the series length is six. Procedure *split_level()* tallies the number of series devices per level and may create new levels if the series restriction has not been met. This is the case here, and the result is shown in Frame 3 of Figure 5.3b. Now no level has more than four devices in series. However, from data-node *0* the circuit connectivity has not changed, it has only been fragmented. By marking data-node *2* as top the topology of the circuit does change. A second cluster

```
mosmesh ()
/*
 * Top level of partitioning algorithm.
 */
{
   while (TRUE) {
      /*
       * read in and build data structure
       */
      if (getmesh()) break; /* TRUE if EOF  */

      /*
       * construct initial mesh subject to chain restrictions
       */
      split_node (chain);
      split_level (chain);
      clear mark field;

      /*
       * make the list of partitioned gates
       */
      makehead();

      if (critical path analysis requested) {
        sort_delay ();
      }
      /*
       * build the partitioned gates
       */
      make output mesh;
   }
}
```

Figure 5.4: Top-Level of Partitioning Algorithm

has been introduced. At this stage a search down the *kid* and *peer* pointers from data-node 0 terminating in either a leaf data-node (*ie.* a data-node whose *kid* pointer is nil) or a data-node with its *top* field TRUE shows that all clusters are found to satisfy the user's constraints. Pseudo-code for splitting procedures is listed in Figure 5.5.

```
split_node ()
/*
 * Case 1: When a given node exceeds the length limit it is split
 * into N. This is done by creating N-1 nodes at the current depth
 * and distributing the old elements over them as evenly as
 * possible. The last allocated node may get an odd remnant.
 */
{
  if (node is NIL) return;

  if (node is SERIES and length > chain) {
    if (chain is 0)
      error("Intractable charge redistribution problem");

    if (parent is PARALLEL or TOP)
      add in a level by creating a SERIES parent;

    compute number of elments to split node into;
    create new nodes and apportion signals;

    continue search at end of newly allocated nodes;
  }

  if (cur_node isnt TOP)              /* recursive, breadth first search
    split_node (nodes at this level);
  split_node (children of this node);
}
```

Figure 5.5a: Node-Splitting Procedure

```
split_level ()
/*
 * Case 2: The sum of SERIES node elements and the number of PARALLEL
 * nodes is greater than the length limit but the total number of nodes
 * is less than the length limit. By converting nodes with more than 1
 * element into parent nodes and pushing the elements down one level
 * the limit restriction is satisfied.
 * Case 3: As case 2 but the total number of nodes exceeds the length limit.
 * A new level is created by binary split. This level is inserted just
 * below the current level. Half the old nodes are attached to each of
 * the newly spawned nodes.
 */
{
  if (node is NIL or length is 0) return;

  check to see if all nodes are marked;
  if (not all marked)
    return;

  count the number of devices in the longest series chain;
  if (sum > chain) {
    if (nodes exist with length > 1) {                    /* case 2      */
      if (chain is 1)
        error("Intractable charge redistribution problem");

      for (all nodes) {
        if (node is SERIES and length > 1)
          create new node;
      }
    } else {                                              /* case 3      */
      divide elements among two kid nodes (binary split);
      increment node depth;
      split_level();          /* recursive call until case 3 satisfied
    }
  }

  if (node isnt TOP)                  /* recursive, breadth first search
    split_level (nodes at this level);
  return (split_level (children of this node));
}
```

Figure 5.5b: Level-Splitting Procedure

The cluster of Frame 3 in Figure 5.3b with the top data-node *0* is shown in Figure 5.6a. This configuration may have a charge sharing problem due to the large amount of parasitic capacitance represented by the parallel data-nodes.

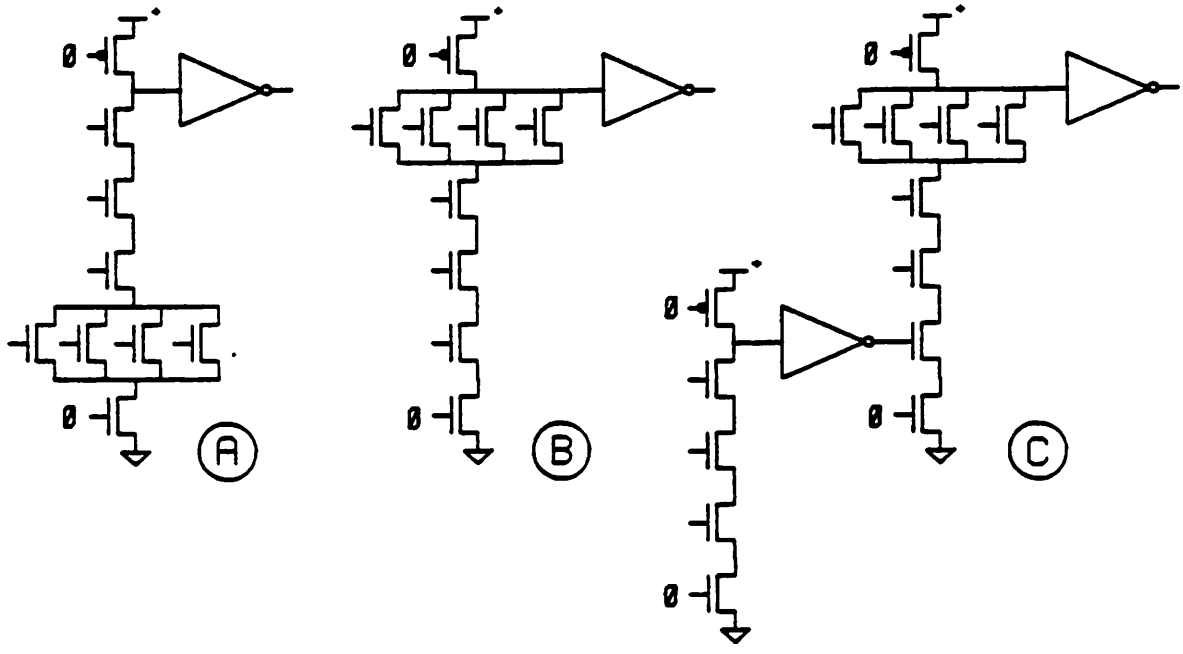


Figure 5.6: Schematic of Frame 3 of Figure 5.3b

If such a situation is detected, procedure *makehead()* tries to remedy the problem by calling procedure *reorder()*. *Makehead()* and associated procedures are shown in Figure 5.7. *Reorder()* attempts to increase the amount of precharge capacitance and/or decrease the amount of parasitic capacitance. If it is not possible to reorder the current cluster to get rid of a CR problem then the current cluster is pruned of data-nodes. This is accomplished by first calling the splitting procedures which mark likely candidates for deletion. The procedure *unmark_node()* performs the mesh alteration by removing the data-nodes furthest from the root data-node first. The result of reordering is shown in Frame 4 of Figure 5.3b. Figure 5.6b shows the circuit schematic.

```
makehead ()
/*
 * Traverses the node list and creates a list of anchor nodes corresponding to
 * the top node in each partition. A partition is a sub-cluster, a group of
 * gates sharing a single prech/buffer circuit and connecting to other
 * partitions in a hierarchical manner. Presently, a partition is limited by
 * depth or series length. A partition will be rejected if it potentially causes
 * a charge redistribution problem. It is possible to coalesce nodes or to
 * fragment them.
 */
{

  if (node is NIL) return;

  if (node is top) {
     clear mark field;
     if (chain limit unrestricted) {
        limit by partition level;
     } else {
        mark all nodes within chain limit;
     }
     if (no partition restrictions) {
        split_node();
        split_level();
     }
     chgshare();
  }

  makehead (nodes at this level);                    /* recursive, breadth first search */
  makehead (children of his node);
}
chgshare ()
/*
 * Checks for charge redistribution problem and attempts to rearrange
 * circuit if problem exists. If no difficulty calls markhead(), else
 * iterates until a solution is found or the problem becomes intractable.
 */
{
  chain_length = critpath();
  while (TRUE) {
    while (node fails ratiochk()) {
      if (can't reorder())
        break;
      find new critpath();
    }
    if (charge sharing problem) {
      if (no nodes to unmark) {
        if (chain_length is 1)
          error("Intractable charge redistribution problem");

        split_node (chain);
        split_level (chain);
      } else
        unmark_node();

      find new critpath();
    }
  }

  new head nodes marked by markhead();
}
```

Figure 5.7: Head and Charge Sharing Procedures

```
reorder ()
/*
 * Searches from the current node visiting all marked nodes. Each time a
 * parent node is SERIES its children may be rearranged if it is possible to
 * bring a more parallel piece to the pathhead.
 */
{
  if (node is NIL or not marked) return(FALSE);

  if (kid isnt NIL and node is SERIES) {
    for (all nodes) {
      if (first node in list) {
        keep track of element count in node and its children;
      } else {
        if (no elements in node but a kid exists
            or this node is PARALLEL and marked and has largest element count) {

          alter pointers to insert this node as first in chain;
          return (TRUE);
        }
      }
    }
  }

  if (reorder (nodes at this level)) return(TRUE);    /* recursive, breadth first search */
  return (reorder (children of this node));
}


ratiochk ()
/*
 * Checks the capacitance ratio between the
 * precharged node and the worstcase path to ground.
 */
{
  ratio = Prech Capacitance / (Prech Capacitance + Parasitic Capacitance);

  return (ratio >= Vth/Vdd);
}



markhead ()
/*
 * From the tree rooted at the current node proceeds by BFS to traverse tree.
 * Marks as head those marked nodes with unmarked kids.
 */
{
  if (node is NIL or not marked) return;

  if (node isnt TOP and kid isnt NIL and kid isnt marked) {
    node->top = TRUE; done = TRUE;
  }

  if (node isnt TOP) markhead (nodes at this level);    /* recursive, breadth first search */
  if (not done) markhead (children of this node);
}
```

Figure 5.7, continued: Reordering, Ratioing, and Marking Procedures

```
unmark_node ()
/*
 * Tries to find most likely candidate from set of nodes eligible to unmark.
 * The idea is to reduce charge redistribution. Nodes are not removed if they
 * are less than unmark_depth level of the tree.
 */
{
  get most distant children of current node;

  if (depth of parent < unmark_depth) return (FALSE);

  for (all children of selected parent node)
    set marked field to FALSE;

  return (TRUE);
}
```

Figure 5.7, continued: Unmarking Procedure

The final stage in partitioning is a call to procedure *sort_delay()* which looks up the transient delay times for signals to reach each of a cluster's inputs. Inputs are then sorted according to their delay times, with the fastest switching FETs being placed closest to the cluster's output. The justification for this is as follows: For the cluster's output buffer to switch the charge stored on the inverter input must be drained away. By placing the fastest signals closest to the buffer input this charge redistribution may be expedited. Note that the buffer will not switch until a path exists between precharge node and GND, assuming the circuit is properly designed. However, when slower changing inputs finally do switch the RC time constant of the shortened path will be less.

There is one added consideration in this process. Sorting input signals according to switching delay will only be performed if the output node will switch more rapidly. It is possible that the inputs which switch fastest are part of a parallel block, for example. By placing this parallel block at the top of the cluster the amount of precharge capacitance is increased. The increased precharge capacitance slows down the switching of the output inverter. *Sort_delay()* checks for this condition by calculating precharge and parasitic capacitances and by referring to the user-provided *technology* file to obtain process-dependent quantities. The RC time constants of different legal configurations are compared, and the circuit with the smallest time constant is selected. The final result for the example of Fig-

ure 5.3 is shown in Frame 5 of Figure 5.3b and schematically in Figure 5.6c. The pseudo-code in Figure 5.8 lists the *sort_delay( )* algorithm.

```
sort_delay ()
/*
 * Traverses the function tree is BFS fashion. All children of the current node
 * are first sorted and then their sub-delays are calculated. If the current
 * node is TOP and SERIES then the components that make up this cluster may
 * be rearranged to give best speed. The rearrangement alg. is N^2 where N is
 * the number of kids, typically less than 5. Sort order is with fastest gate
 * closest to output node and is only done if no CRP exists and additional
 * precharge capacitance does not slow the gate cluster down.
 */
{
  if (node is NIL) return;

  for (all kids of this node)
    sort_delay();

  if (node is TOP and node is SERIES) {
    begin bubble sort on this node's kids {
      if (sort vector[i].delay > sort vector[j].delay) {
        swap nodes;
        find new critpath();
        if (ratiochk() is okay) {
          if (TOP) {
            /*
             * precharge capacitance may have been modified.
             * make sure change speeds up gate overall
             */
            if (new_delay > old_delay) {
              /*
               * new configuration is slower, so swap back to original state
               */
              swap nodes;
              find old critpath();
            }
          }
        } else {
          /*
           * charge problem exists, set things as they were
           */
          swap nodes;
          find old critpath();
        }
      }
    }
  }
}
```

Figure 5.8: Delay Sorting Procedure

5.5. An Example of MOSMESH Partitioning

There are many ways to remedy the charge sharing problem and some of these were examined in Chapter 2. MOSMESH, as the previous example illustrates, tries to decrease the parasitic capacitance or it may increase the precharge capacitance- but only by moving around already existent capacitances. However, if the user wishes, he can add a larger precharge capacitance by specifying it in the *technology* file. The effect of a large capacitance at the precharge node will be to slow that particular circuit cluster down: however it may reduce the charge sharing problem and therefore allow more complex clusters to be built, which in turn will result in a shallower buffer hierarchy and possibly a faster circuit overall.

The sequence of Figures 5.10-5.12 shows the function of Figure 5.9 in various configurations.

```
(of0(p(s 1 2 3 4)
     (s(s 1)
       (p(s 1 2)
         (s
          (p(s 1)
            (s(p(s 1 2)
                 (s 1 2 3))
            (p(s 1 2)(s 1 2)(s 1))))
          (s 1)))
        (s 1 2))));
```

Figure 5.9: 8-Level Parallel/Serial Function

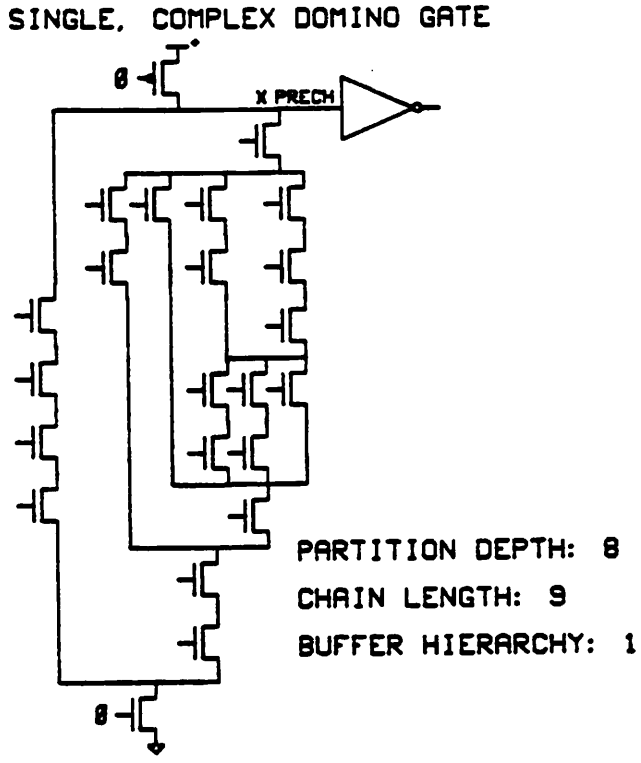


Figure 5.10a: Monolithic Domino Gate

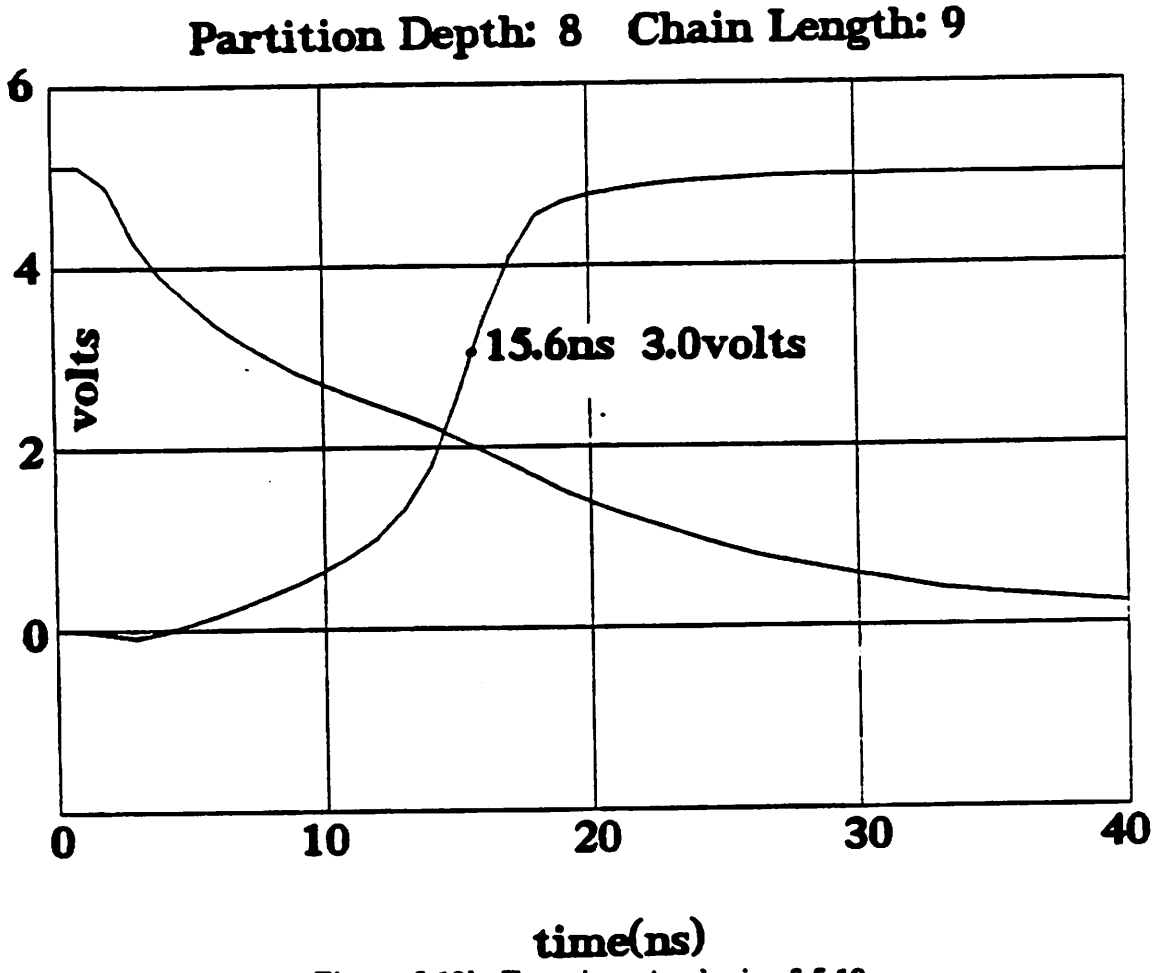


Figure 5.10b: Transient Analysis of 5.10a

Figure 5.10a shows the function realized as a single, monolithic cluster. This circuit has a buffer hierarchy of one but a worst-case series length of nine. In order to make the circuit work under worst-case conditions it was necessary to add extra capacitance to the precharge node. Figure 5.10b shows the result of a SPICE2 simulation of the circuit. At time $t = 0$ all external signals begin to switch. External signals were conditioned by driving them through nominal size buffers. The voltage on the precharge node falls, the output buffer switches, and the output signal rises. It was assumed that the output buffer drives three nominal size inputs. For this example a voltage of 3.0V was chosen as the logic threshold. Figure 5.10b shows that the voltage on the precharge node begins to fall early but, because the precharge capacitance is large, the output voltage does not reach 3.0V until 15.6*ns*. In contrast, Figure 5.11a shows the same function this time broken up into

clusters, each with a partition depth of one. The deepest buffer hierarchy is eight, the longest series string is four devices. Figure 5.11b shows SPICE2 results of transient simulation. The waveforms are clean and sharp but that the total delay is very close to the monolithic function block. In this case no extra capacitance was added, but the circuit switches slowly due to the eight-deep buffer hierarchy.

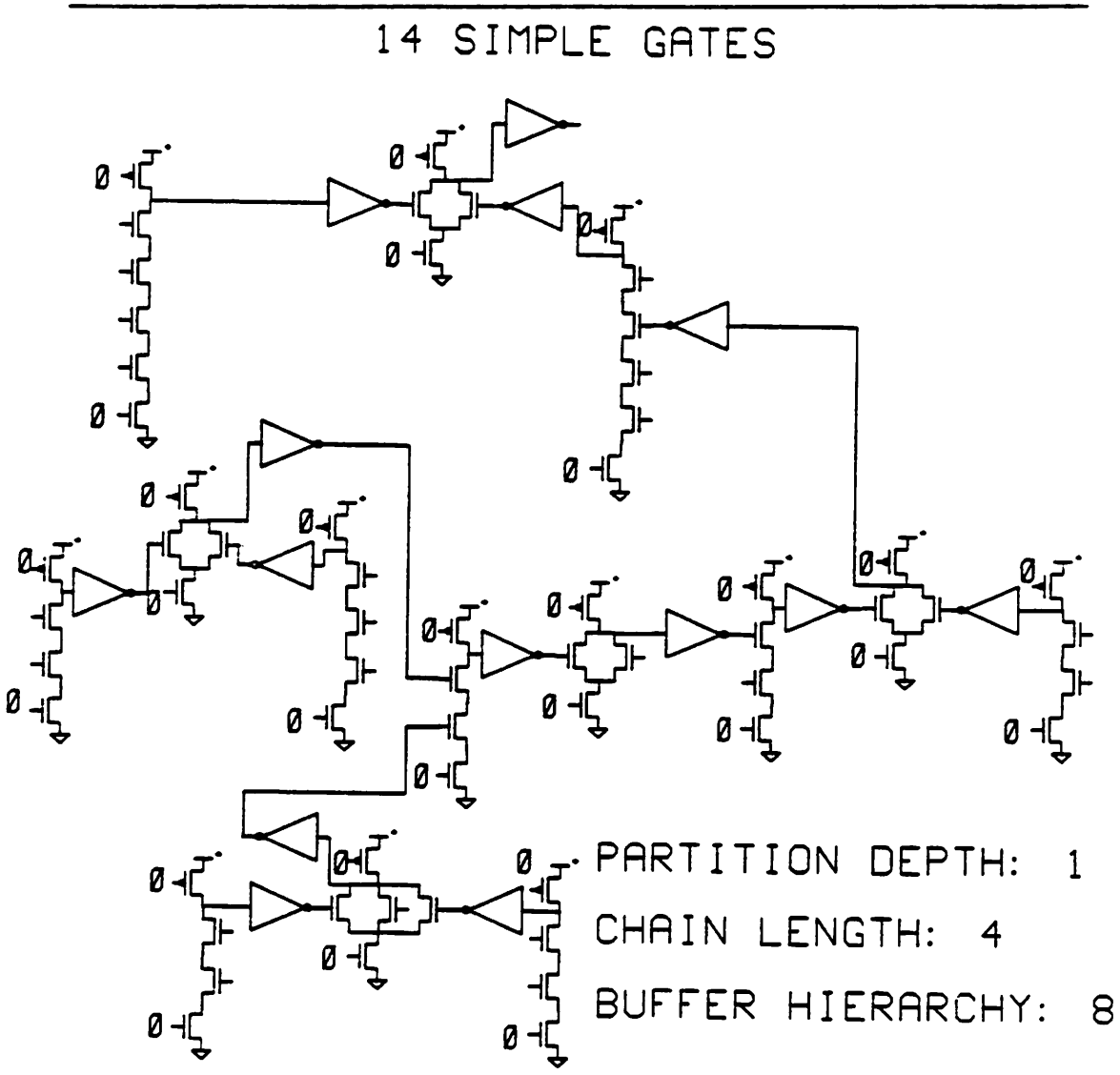


Figure 5.11a: Set of Single Partition Domino Gates

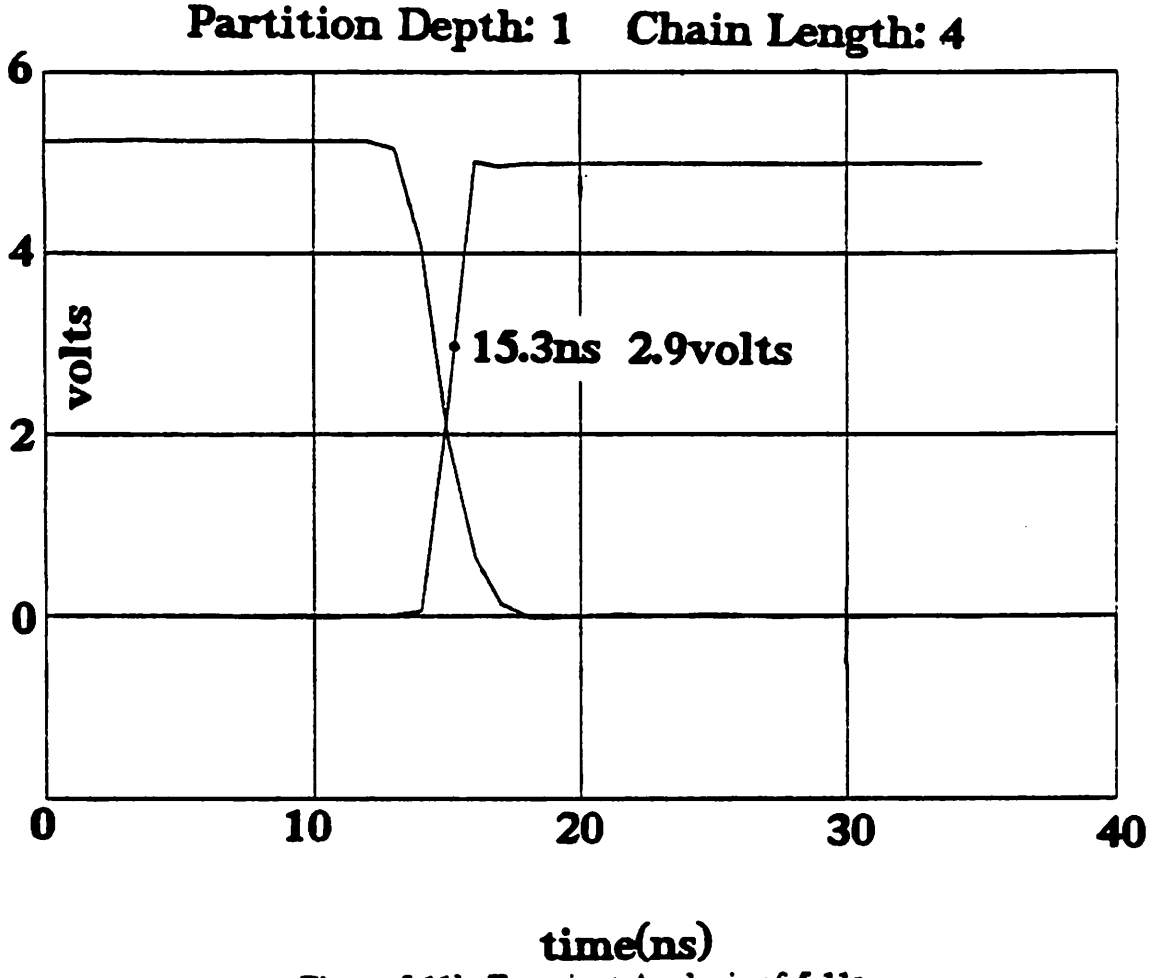


Figure 5.11b: Transient Analysis of 5.11a

Figure 5.12a represents a compromise between the previous two extremes. Here the complexity of each cluster is commensurate with the precharge capacitance. No extra capacitance was required. The associated simulation in Figure 5.12b shows that the function switches in 8.9*ns* a speedup of more than 40% over the previous solutions. This solution was obtained by MOMSESH.

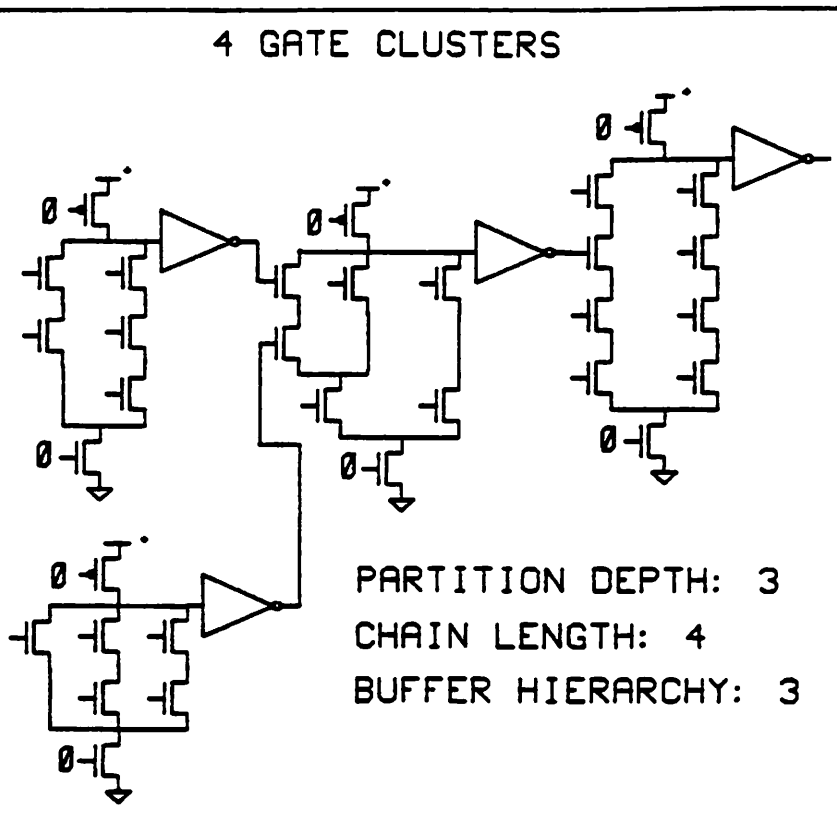


Figure 5.12a: 4-Cluster Domino Function

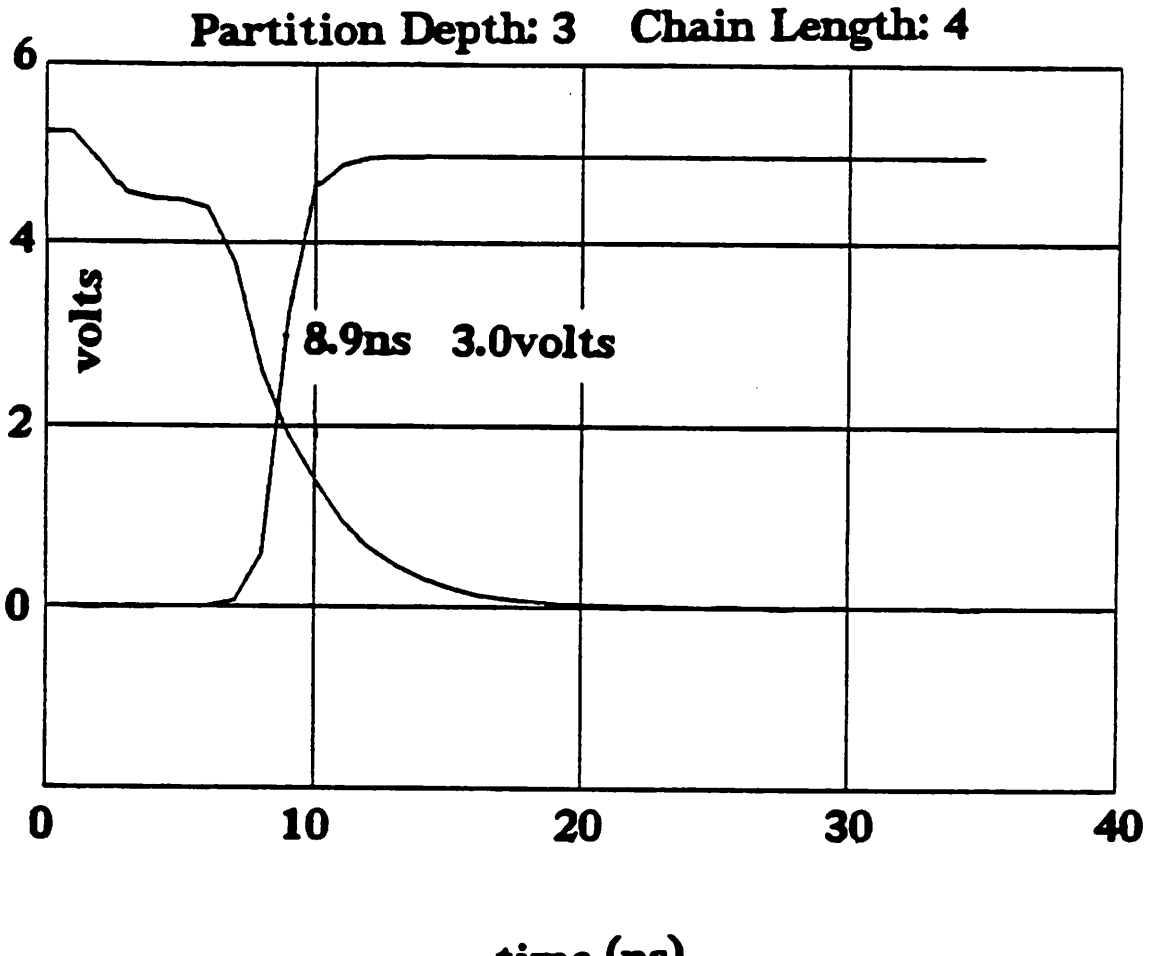


Figure 5.12b: Transient Analysis of 5.12a

While such large speed improvements will not be obtained in all cases, this response argues for matching parasitic and precharge capacitances, rather than adding "dummy" capacitance or over-partitioning a circuit. In Figure 5.12b there is an initial voltage drop at the output node. This is due to charge redistribution. The buffer does not switch, but when the other inputs become valid it does switch faster than other configurations. In fact it can be seen that as the parasitic capacitance is increased with respect to the precharge capacitance or, more exactly, as the *lhs* of Equation 2.6 approaches the *rhs* the gate will switch faster and faster until it finally functions improperly. The idea is to match capacitances as closely as possible without causing a CR problem. Another example from a RISC microprocessor [mari85], which produces a different result, is summarized in Figure 5.13.

Number of Clusters	Deepest Buffer Hierarchy	Worst-case Delay (ns)	Approx. Prech Capacitance (pF)	Critical Signal
247	16	41.0	0.02	CPIPE1load1
148	9	31.0	0.16	pALUtoMAL

Figure 5.13: Delay in Critical Path of Complex Circuit

In this case, because the functions being generated are complex, a charge sharing problem still exists, even after topological rearrangement. The first line of Figure 5.13 shows how breaking complex clusters into smaller ones yields a deep buffer hierarchy and a slow critical path. The second line of the figure shows the result when parasitic and precharge capacitances are balanced, not by subtracting parasitics but by adding to the precharge value. The result is individual clusters which switch more slowly but an overall critical path delay which is faster. Note that the critical signal in the two implementations of the circuit is different. produces the designer is able to explicitly specify both the switching threshold and the amount of capacitance on the precharge node. he can try several configurations and choose a high-speed or more conservative design.

The detailed partitioning steps for the function shown in Figure 5.9, with an inverter voltage switching threshold of 2.4V for a 3μ p-well process, are shown in Figures 5.14a-f. The set of three integers separated by slashes indicates the number of source or drain parasitics that contribute to the *head*, *middle*, and *tail* of each cluster, respectively.

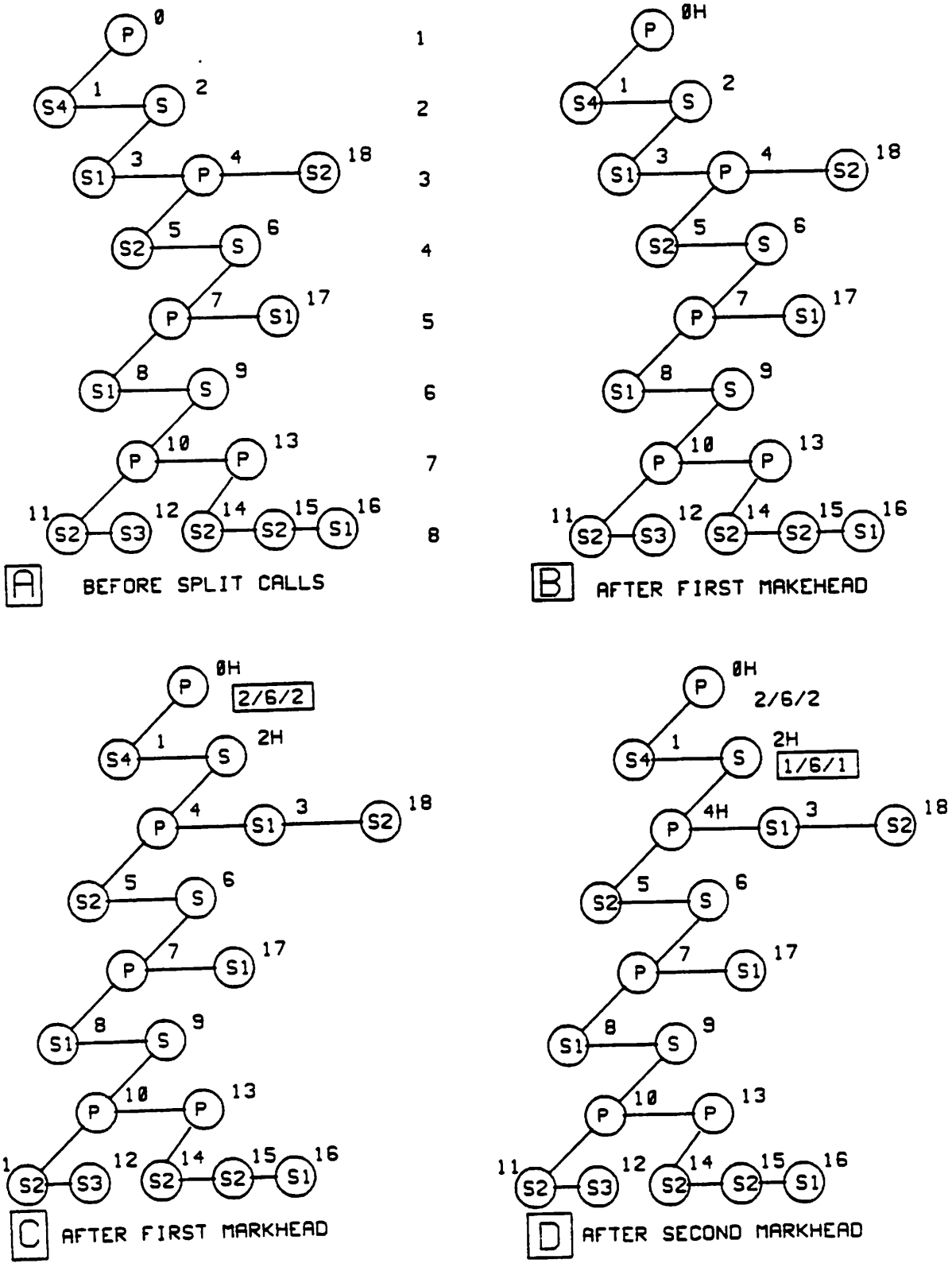


Figure 5.14: 8-Level Parallel/Serial Partitioning Sequence

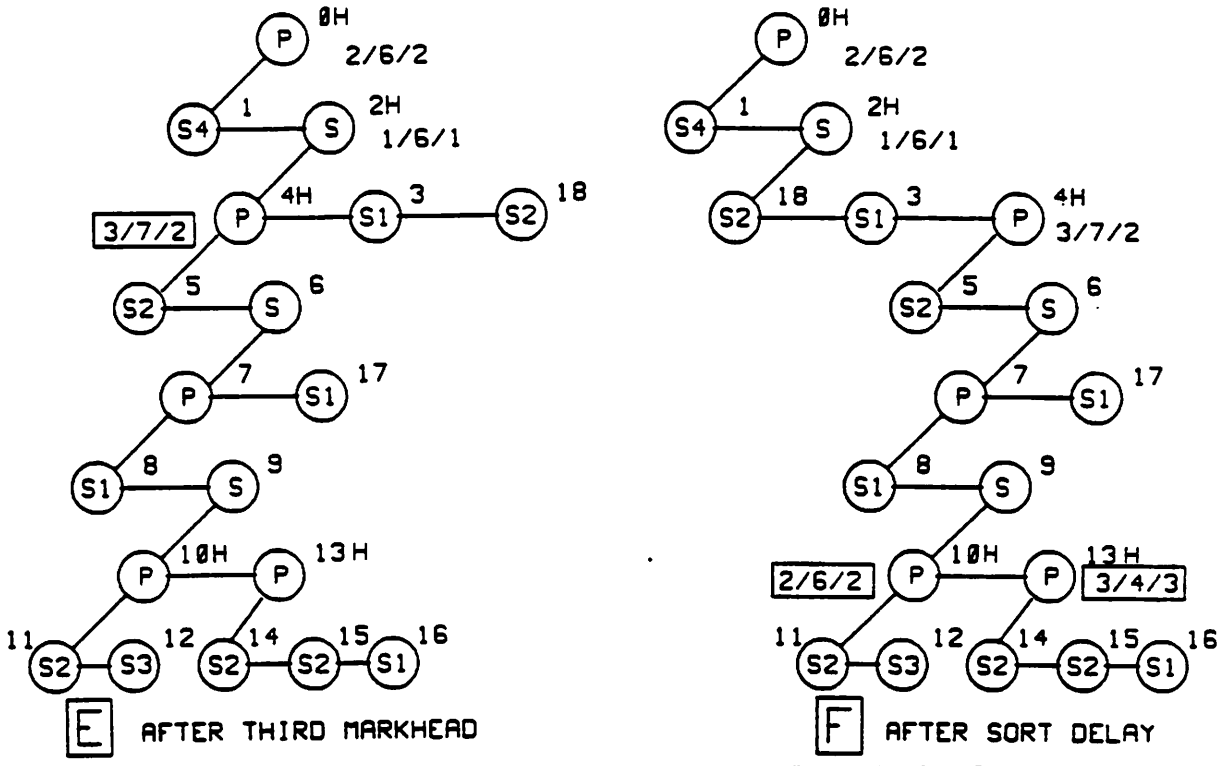


Figure 5.14, continued: 8-Level Parallel/Serial Partitioning Sequence

For the process conditions given and no user restrictions on chain length and partition depth the end result is the 5-cluster partition shown in Figure 5.15.

```
# charge tolerance ratio is 2.78374

(o f0
 (p
 (s d c b a ) 2 )):

(o 2
 (s a b a 4 )):

(o 4
 (p
 (s b a )
 (s
 (p
 (s a )
 (s 10 13 ) ) a ) )):

(o 10
 (p
 (s b a )
 (s c b a ) )):

(o 13
 (p
 (s b a )
 (s b a )
 (s a ) )):


# 5 cluster(s)
# Longest series string: 4 [target maximum unrestricted]
# Deepest Partition: 4 [target maximum unrestricted]
# Deepest Buffer Hierarchy: 4
# Worst Ratioing Problem: 0.505734 [target minimum: 0.48 (Vth 2.4)]
# Signal :f0: has worst delay of 8.21233ns
```

Figure 5.15: 8-Level Function; Chain Length and Partition Depth Unrestricted

5.6. Calculation of Signal Delay in a Partitioned Mesh— MKTBL

After MOSMESH has produced a legally-partitioned circuit the final step is the use of procedure *sort_delay()* which may alter the order of signals for greater speed. *Sort_delay()* does not calculate the delay of a circuit cluster each time it performs an analysis. Instead, it refers to a pre-calculated table of delays for a given cluster. This table is produced by

another tool, called MKTBL. MKTBL reads a simple file, called a *pattern* file, and produces a series of SPICE2 decks for each pattern (cluster). The tool then invokes SPICE2 automatically, after first identifying the critical path in the cluster. A transient simulation is performed and MKTBL reads the SPICE2 result and computes the delay from input to output. It summarizes this information and for each cluster, for different sets of input conditions. MKTBL writes a single line to the output file.

Example input patterns and their corresponding circuit schematics are shown in Figure 5.16.

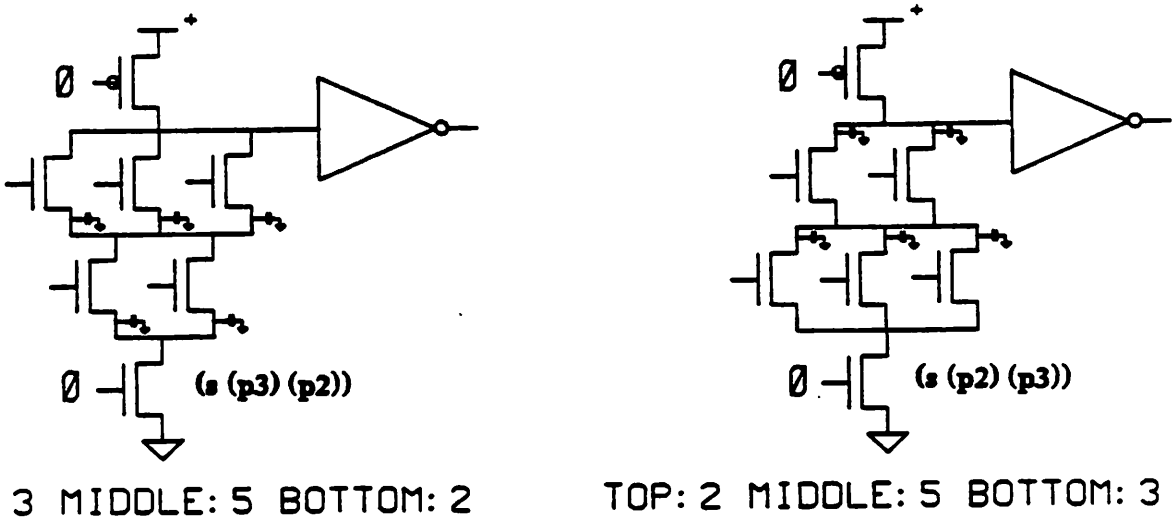


Figure 5.16: MKTBL Patterns and Corresponding Circuit

Each line in the transient delay table is of the form:

Morph: (*head middle*) **Buf:** (*width*$_p$ *width*$_n$ *width*$_{core}$) ...
...**Ext:** *number external* **Cr:** *slowest signal* **Time:** *delay (ns)*

Morph stands for *morphology*: the two integer quantities represent the number of source and drain nodes which contribute to the precharge and the parasitic capacitance, respectively. **Buf** gives information about the size of the output buffer as well as the pullup clock device and the core devices. The user may provide different sizes of output buffers or core devices for transient evaluation. **Ext** is the number of inputs which are driven

externally. External inputs switch at time $t=0$ and are assumed to be closest to the output buffer. Of the remaining, internally driven signals Cr marks the slowest, or critical one. Finally, Time is the delay time in nanoseconds for the particular circuit configuration. To run MKTBL the user must provide two files in addition to the pattern file and the technology file. First, the user must supply a *model* file. This file contains the SPICE2 models to be used to evaluate n- and p-channel devices. The models may be any valid SPICE2 model. All transient analyses presented here were run with the *level*=2 model. Second, the user constructs a *subckts* file which holds both a single transistor core device subcircuit and the inverter/pullup encapsulated as a subcircuit. The user may provide several sizes of both core subcircuit and buffer subcircuit. If multiple subcircuits are found then MKTBL will run each pattern in the pattern file for all possible combinations of subcircuits. Figure 5.17 lists the transient delay table for the two patterns given in Figure 5.16.

```
Morph:(3 8) Buf:(8 4 4) Ext:2 Cr:0 time:3.08844e-09
Morph:(3 8) Buf:(8 4 4) Ext:1 Cr:1 time:2.71408e-09
Morph:(3 8) Buf:(8 4 4) Ext:0 Cr:2 time:1.87178e-09
Morph:(3 8) Buf:(8 4 4) Ext:0 Cr:1 time:1.42817e-09
Morph:(3 8) Buf:(16 8 4) Ext:2 Cr:0 time:3.23581e-09
Morph:(3 8) Buf:(16 8 4) Ext:1 Cr:1 time:2.88215e-09
Morph:(3 8) Buf:(16 8 4) Ext:0 Cr:2 time:2.0516e-09
Morph:(3 8) Buf:(16 8 4) Ext:0 Cr:1 time:1.67881e-09

Morph:(2 7) Buf:(8 4 4) Ext:2 Cr:0 time:3.12969e-09
Morph:(2 7) Buf:(8 4 4) Ext:1 Cr:1 time:2.58566e-09
Morph:(2 7) Buf:(8 4 4) Ext:0 Cr:2 time:1.7723e-09
Morph:(2 7) Buf:(8 4 4) Ext:0 Cr:1 time:1.32752e-09
Morph:(2 7) Buf:(16 8 4) Ext:2 Cr:0 time:3.12352e-09
Morph:(2 7) Buf:(16 8 4) Ext:1 Cr:1 time:2.7229e-09
Morph:(2 7) Buf:(16 8 4) Ext:0 Cr:2 time:1.92684e-09
Morph:(2 7) Buf:(16 8 4) Ext:0 Cr:1 time:1.55663e-09
```

Figure 5.17: Transient Delay Table Created by MKTBL

5.6.1. Flexibility and Accuracy of MKTBL Transient Model

Since each entry in the transient delay table represents a separate transient analysis of SPICE2 it can be seen that for many different patterns in different configurations such a table is computationally expensive to generate. However the transient analyses are performed on relatively simple circuits and it is anticipated that several designers will share a common transient delay table if they are designing in the same process. The table needs to be regenerated only when there are major changes in the process. In everyday use a designer or synthesis tool will occasionally need to append a new pattern to the delay table as a less common cluster type is encountered. The current pattern file contains around 70 different morphologies and has been found to cover a wide variety of combinational circuits. Note also that MKTBL handles construction and interpretation of the SPICE2 simulation automatically so that the designer may start up the program and let it run as a background job. Anomalies encountered during execution of any SPICE2 run are written to a logfile for later reference.

The transient delay time that MKTBL computes is based on conditioned inputs and an output with a fanout of three. The delay time is taken to be the difference between the 50% level of the rising input signal and the 50% level of the rising output signal (Domino functions are even). The computation of worst-case path delay in MOSMESH is obtained by searching recursively from the function output to the inputs and summing the cluster delays. This superposition approach is valid because each cluster is buffered and because the threshold switching voltage at the output is taken to be equal to the threshold switching voltage of the input.

The accuracy of the transient analysis of each cluster must also be justified. The pattern which describes each circuit cluster type is reduced to a more general morphology. In general, it is possible to construct multiple circuits which have the same morphology. It is clear that different circuits with identical morphologies will have different transient responses, but Domino circuits represent a restricted class of dynamic circuits. N-channel Domino circuits are always precharged high; if they make a transition it must therefore be

from a logic high to low. In fact, a simple RC model can be substituted for each FET. If such a model were found to be accurate, then the aggregation of parasitic capacitances discharging through a MOSFET, which is the MKTBL model, should be still more accurate. In practice a simple RC model was not found to give sufficiently accurate results. However, a model which aggregated capacitances and collapsed all MOSFETs into two series devices was found to give reasonable results. Therefore, the MKTBL model, which does not alter the number of MOSFETs, is accurate. The next section presents the derivation of a simple model for representation of a Domino cluster.

5.7. A Simple MOS Model for an Arbitrary Mesh of Mixed Precharged and Discharged MOS Devices

Delay calculations are process- and design-style-dependent. The transient delay model needed here must cover Domino logic circuits. Domino logic is built entirely out of n- or p-core devices and uses a single clock. This *core* of the gate may comprise an arbitrarily complex AND-OR logic function. The devices may or may not be precharged, on a device-by-device basis. The problem, therefore, is to construct a model which is flexible enough to handle these logic conditions. The model should be simple and its accuracy comparable to the other process-dependent procedures in the package. In the following presentation a MOSFET *mesh* is defined to be an arbitrary configuration of a graphical representation of FETs which may be *cyclic*. Specifically, there may be more than one path to ground. In this way a mesh differs from a *tree* which has no cycles: from any given node there is only a single ground path. This definition of *mesh* is consistent with that given in [horo84].

5.7.1. Approach to Solution

Two possible approaches were considered. The first approach is to simulate, via SPICE2, those extreme cases of MOS meshes which bracket the useful Domino configurations. For a particular layout method, termed the *zero-deep* model, it is possible

to construct a set of equations which can accurately predict Domino circuit delay. The zero-deep model does not allow for arbitrary Domino mesh circuits. Instead it constrains the circuit such that an inverting buffer is placed between each NAND or NOR function. This constraint strongly limits the set of circuits that can be constructed because complex Domino functions, for example an AND/OR gate, cannot be built. By simulating a range of AND and OR gates, and by varying the ratio of external to internal gates, it is possible to completely characterize this model. A set of four equations is used: one equation each to predict delay versus fan-in for AND and OR gates and two more equations to predict the effect the ratio of internal to external signals has on gate delay. Quadratic equations, created by the method of least-squares, give a good fit to the data from the simulations. The fit from linear formulae was not accurate enough for critical path prediction. In this approach the nature of the driving gates and the loading of the driven gates may be ignored. This is because each AND or OR gate is buffered. The buffer is of a fixed size and its gate capacitance is considered in the circuit simulations. All inputs are assumed to be driven by these buffers. Figure 5.18 lists results of this approach, not compensated, however, for different ratios of internal to external inputs.

Comparison of Simulated Results with Zero—Deep Model						
Circuit	Model	SPICE2	Delta		Critpath Stages	Notes
			(%)	(ns)		
1.n	13.20	15.0	-12.0%	1.8	2	
2.n	18.90	20.5	- 7.8%	1.6	3	
timing.n	28.87	28.2	+ 2.3%	0.7	4	
ao.n	35.64	27.5	+20.6%	7.1	4	(23.0 reordered)
tpla.n	67.34	55.5	+19.1%	12.8	6	(from SOAR)
cla.n	91.18	93.5	- 2.5%	2.3	8	(8-bit ripple-carry)

Figure 5.18: Comparison of Simulated Results versus Zero—Deep Model

While this approach is feasible for the zero-deep model, it is too difficult to apply to the more general arbitrary mesh problem. In general, the number of bracketing cases to be considered becomes too large, and the method proves unworkable. The general problem is to simulate an arbitrary mesh of devices under various input loads and output drive requirements. However, for the simulation of regular structures, one can assume inputs

and outputs are well buffered. Therefore one can consider the more restricted case of modeling meshes with conditioned inputs and outputs.

This formulation still proves too complex to solve directly. Instead a simplifying step is used. For the arbitrarily complex logic between two inverting buffers a two-transistor ($2-T$) equivalent MOS circuit is substituted. It is then possible to simulate the usable range of equivalent gates and produce an equation which accurately predicts delay of the original complex mesh without having to do any additional simulation. The crux of this method lies in the construction of the $2-T$ circuit. The derivation of the model is now examined.

5.7.2. Derivation of 2—T Model from an Arbitrary Mesh

5.7.2.1. Difference between RC and MOS models of MOSFETs

Recently there have been a number of papers published on the modeling of RC-networks [horo84] [rube83] [wyat83] [toku83] [lin84]. The goal of this research has been to accurately characterize gate delays and signal waveforms so that complex sets of gates could be evaluated quickly. Typically the goal is to produce results comparable to direct simulation programs like SPICE2 but with at least an order of magnitude reduction in computation time.

For the most part these papers confine themselves to modeling RC trees and not MOS devices. The paper by Tokuda *et.al.* does consider the modeling of MOS inverters and transmission gates but not more complex circuits. If it were possible to model a MOS device by an equivalent RC network then the results of the RC modeling papers could be exploited. Unfortunately due to the inherently non-linear nature of a transistor, FETs cannot be simply translated into RC equivalents. Figure 5.19 shows two curves.

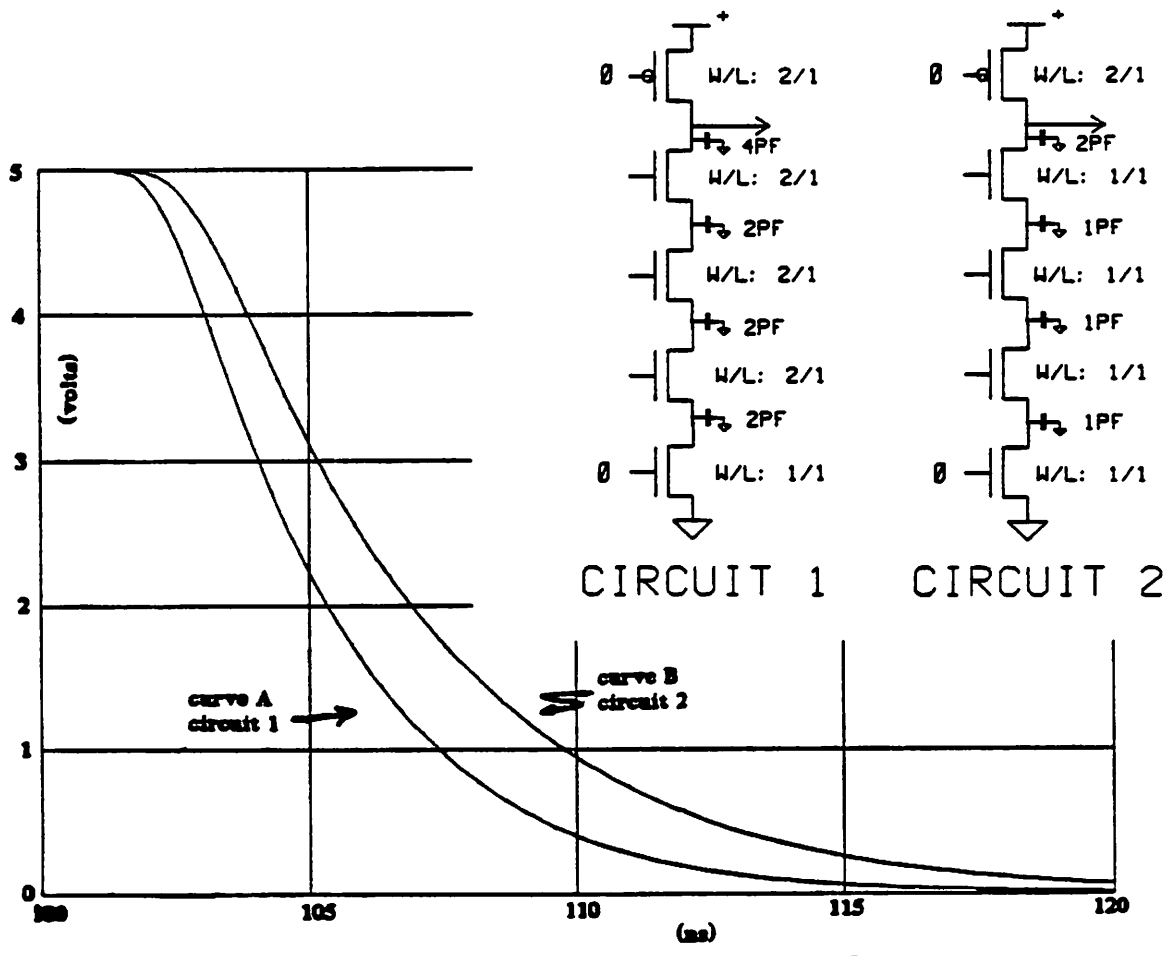


Figure 5.19: Comparison of Two MOSFET Chains

Curve A is the waveform produced by circuit 1. Curve B is the waveform produced by circuit 2. It can be seen that the shape of the two curves does not match. Circuit 1 was simulated by SPICE2 with no intrinsic source/drain capacitance specified. A separate capacitance element was explicitly added to all source and drain nodes to make the comparison more relevant. Circuit 2 was modeled in a similar fashion. The difference between the two circuits is that the source/drain capacitance of circuit 1 is twice that of circuit 2 and the MOSFETs of circuit 1 are twice as wide as those of circuit 2. If the MOS device behaved approximately linearly as V_{GS} varied, then one would expect the two circuits to exhibit similar waveforms since their RC time constants are identical. The fact that this is not the case shows that a simple RC model cannot supplant a MOS device for the purpose of this

work.

For reference, Figure 5.20 shows two curves from RC Circuits 3 and 4.

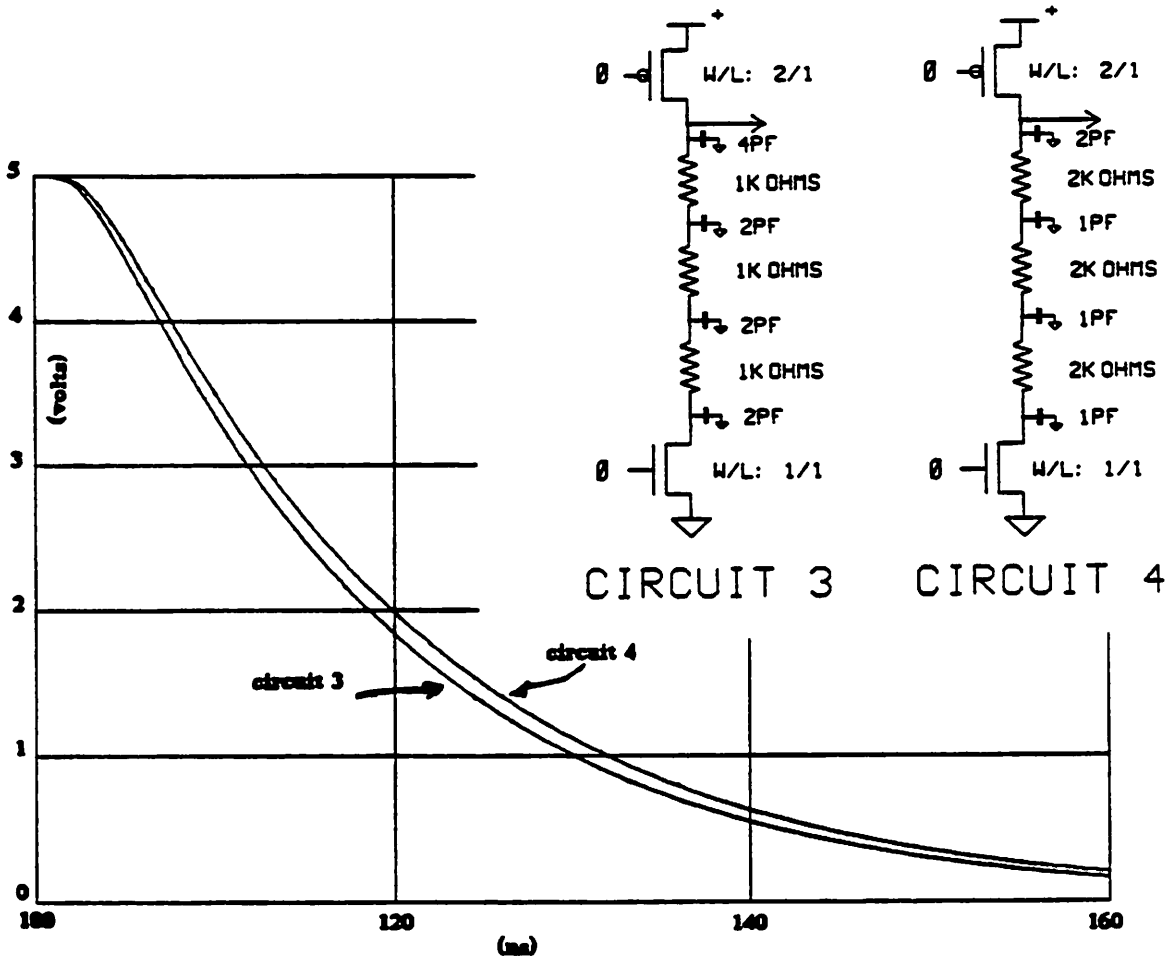


Figure 5.20: Comparison of Two RC Chains

Circuit 4 is related to Circuit 3 in the way Circuit 2 is related to Circuit 1. It can be seen in this case that the shapes of the two curves match quite closely.

Instead of approximating the behavior of a MOS device by a linear or nonlinear RC network, a MOSFET model is used. An arbitrary mesh is collapsed into two equivalent MOS devices. This equivalent circuit can then be simulated, for example by SPICE2. The MOS model may be as simple or detailed as the designer wishes. Simulation time is greater than for an RC model, however accuracy is better. It is also possible to store results of

various equivalent devices and interpolate between them. In such cases no simulation is required.

5.7.2.2. Determination of an Equivalent Circuit

An example of a mesh of devices representing a complex Domino gate is shown in Figure 5.21. This is the n-channel core of the 8-level parallel/serial function of Figure 5.9.

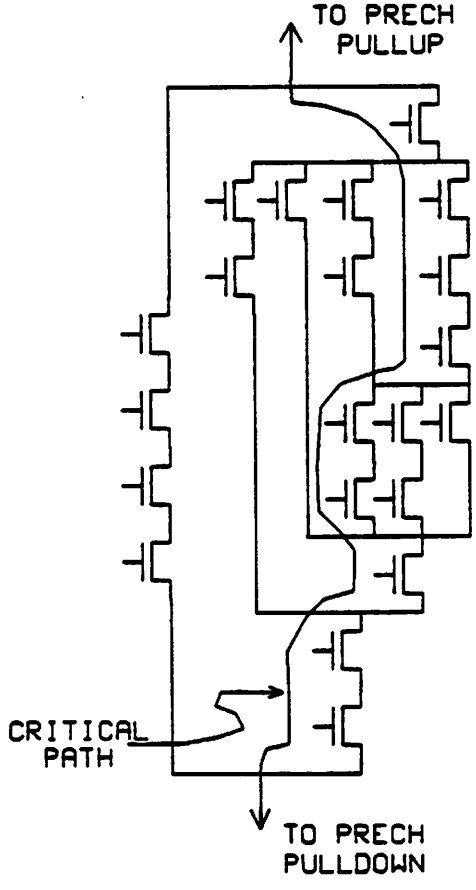


Figure 5.21: Complex Mesh in Domino CMOS

Even though this gate contains 21 devices, only 9 directly participate in the calculation of an equivalent circuit. These devices are connected by the heavy black line. This line indicates the worst-case path through the Domino gate. In a worst-case model all but a single

gate in a parallel OR configuration is on and the longest series AND path is taken so that from output to ground the maximum number of devices is traversed. It is assumed that all devices are of the same length and width. The equivalent circuit devices have the standard width. The sum of the length of the 2–*T* devices is proportional to the sum of device lengths along the worst-case path. Source and drain parasitics are calculated by summing the area and perimeter of nodes along the critical path. Wide deviations in the number of FETs tied to a given node will cause worse agreement between the equivalent and actual circuit; conversely if the distribution of FETs along the critical path is uniform, the equivalent model will be in good agreement with the actual circuit. Figure 5.22 compares simulations of the actual and equivalent circuit of Figure 5.21.

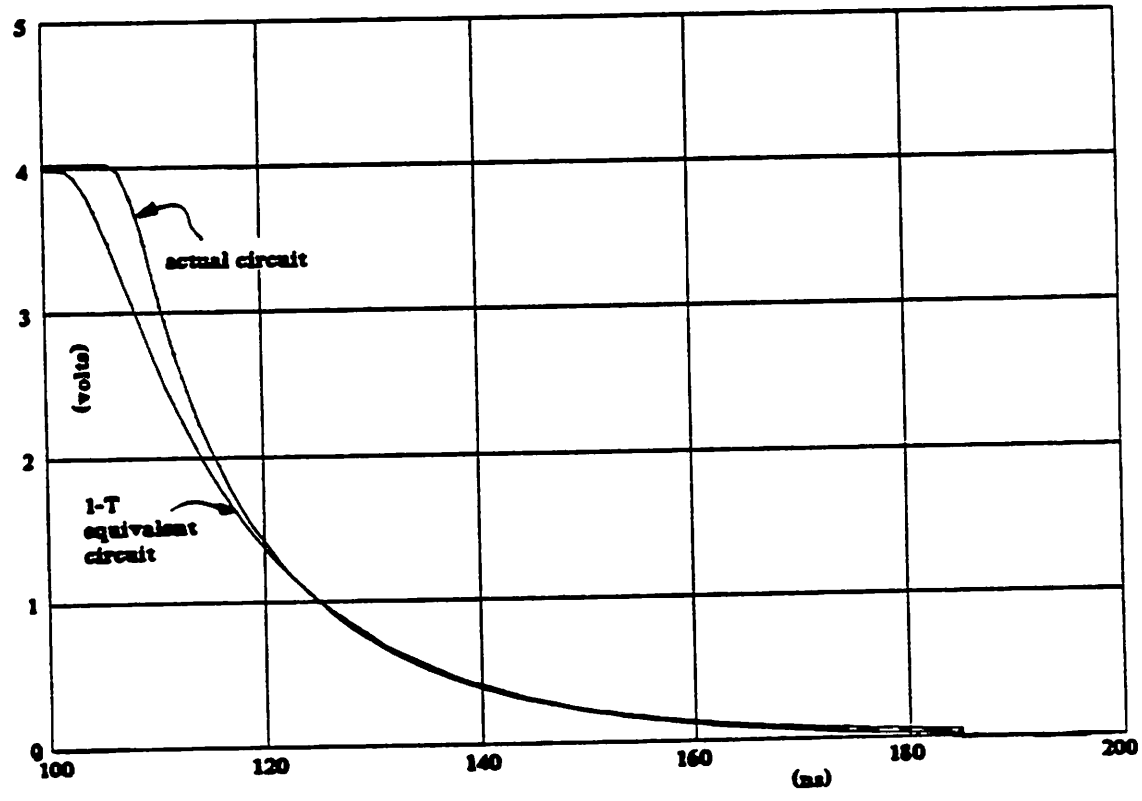


Figure 5.22: Comparison of Actual- and Equivalent-Circuit Simulations

In this simulation all nodes were precharged; this corresponds to the external input case for Domino devices.

In general, a Domino gate will have some of its gates charged and some discharged. Whether or not a gate's source/drain nodes are charged depends on the gate's driving signal. Internal signals— signals which come from other Domino circuits— must hold their gates off during precharge. Therefore all source/drain nodes below this gate will not be charged. Gates driven by external, or non-Domino signals, may be either on or off during precharge. Externally driven gates just below the clocked precharge gate will therefore be charged high, if the external signal is high. For reasons of speed, and to reduce charge sharing problems, it is advantageous to locate external inputs close to the Domino circuit output and place the internally driven devices below these inputs.

The mix of precharged and discharged devices is modeled in the 2–*T* case by lumping precharged devices into the *proximal* FET (the transistor closer to the Domino output node) and the discharged devices into the *distal* FET (further from the Domino output). The precharge/discharge ratio affects the length and source/drain parasitic calculation of the equivalent devices. Consider the circuit in Figure 5.23.

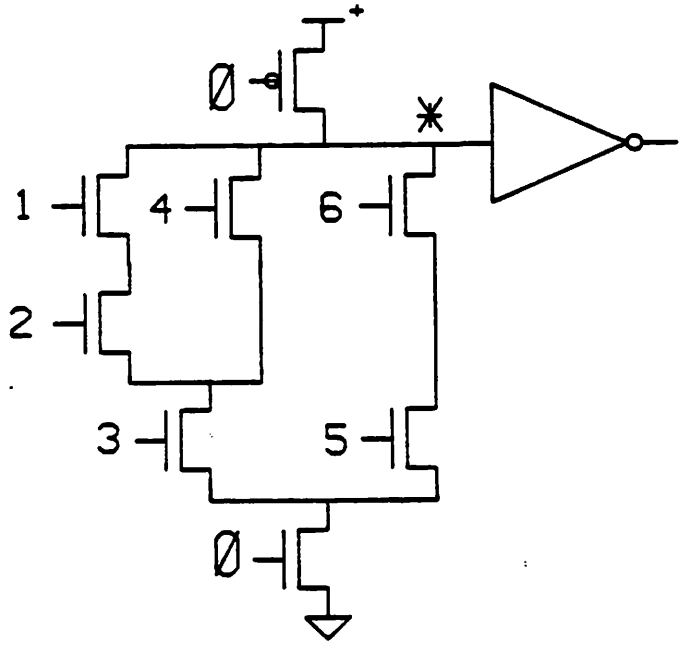


Figure 5.23: Complex Domino Gate

Assume all inputs are internal. In this case the precharged node (starred node) consists of the parasitic capacitances contributed by the three n-core devices 1, 4, 6 (as well as the p-channel pullup and the output buffer). Discharged parasitic capacitances are contributed by the source nodes of FETs 1, 2, 3, 4, and 5 and by the drain nodes of devices 2 and 3. The 2—T model for this circuit has a proximal FET of length equal to the length of FET 1. The drain capacitance is three times a single FET drain, the source capacitance is that of a single FET. The distal FET length is given by:

$$L_{distal} = 0.9 \times \sum_{i} L_i \qquad (5.1)$$

where $i = 1, \ldots, N_{\text{wc internal devices}} - 1$. The multiplication factor of 0.9 was determined empirically by simulation. For the example shown the worst-case length is three so $L_{distal} = 1.8 \times L_{FET}$. The discharged parasitics are distributed equally between source and drain. Thus $C_{source} = C_{drain} = C_{discharged} / 2$. In the example the parasitic is three times the single FET parasitic.

The length and parasitic components of the 2—T devices are varied to correspond with the mixture of precharged and discharged nodes. It is advantageous to place all external devices in a cluster closer to the Domino output node than the internal devices. The 2—T model, however, does not require this assumption for accurate modeling. If internal and external devices are intermixed, the proximal device models devices and capacitances from the Domino output until the first internal node is reached. The distal FET models the remainder of the devices. Thus the proximal device models at most one internal device (if all devices are internal, as in the Figure 5.23 case) while the distal FET may model any mix of internal and external devices.

5.7.3. Limitations to the 2—T Model

Dynamic circuits are prone to the problem of charge redistribution described in Chapter 2. The 2—T model assumes that no charge redistribution problem exists. For reasons of automated layout and compaction it is often more convenient to deal with circuits

which have no CR problem rather than to attempt to correct the problem. Therefore the $2-T$ model is useful once it has been ascertained that the circuit is CR-free. This can be determined by summing up capacitances on the Domino output node and on the parasitic nodes and comparing the two values as explained in Chapter 2.

A larger example, which does exhibit charge redistribution effects but which functions properly, is now examined. The circuit of Figure 5.21 has a CR problem. The circuit of Figure 5.24 has no CR problem: it represents the optimal charge sharing partition chosen by MOSMESH and was shown previously in Figure 5.12a.

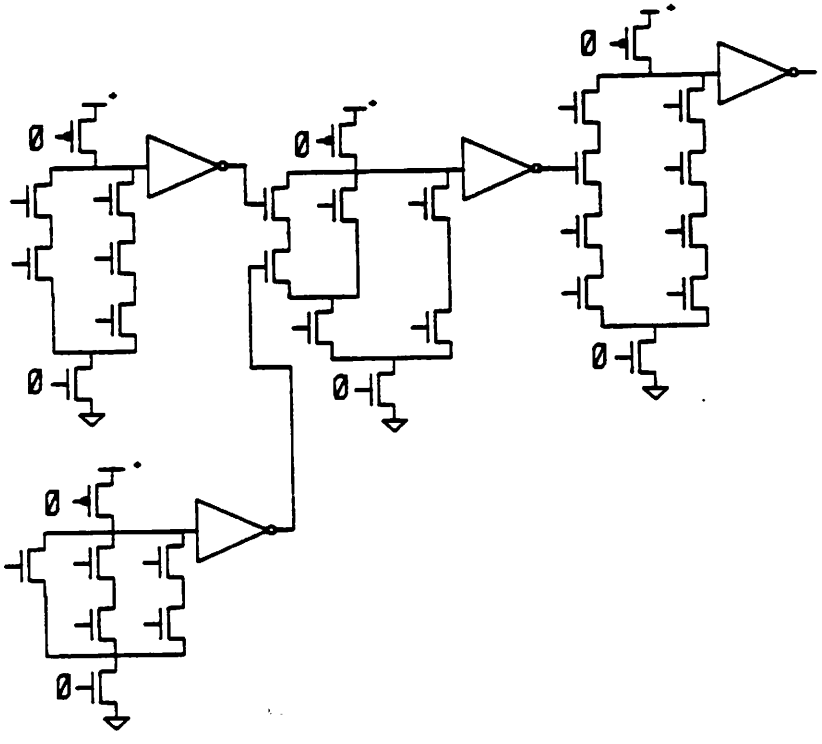


Figure 5.24: Partitioned Circuit of Figure 5.21

Figures 5.25a-d show simulation pairs of each of the four sub-functions.

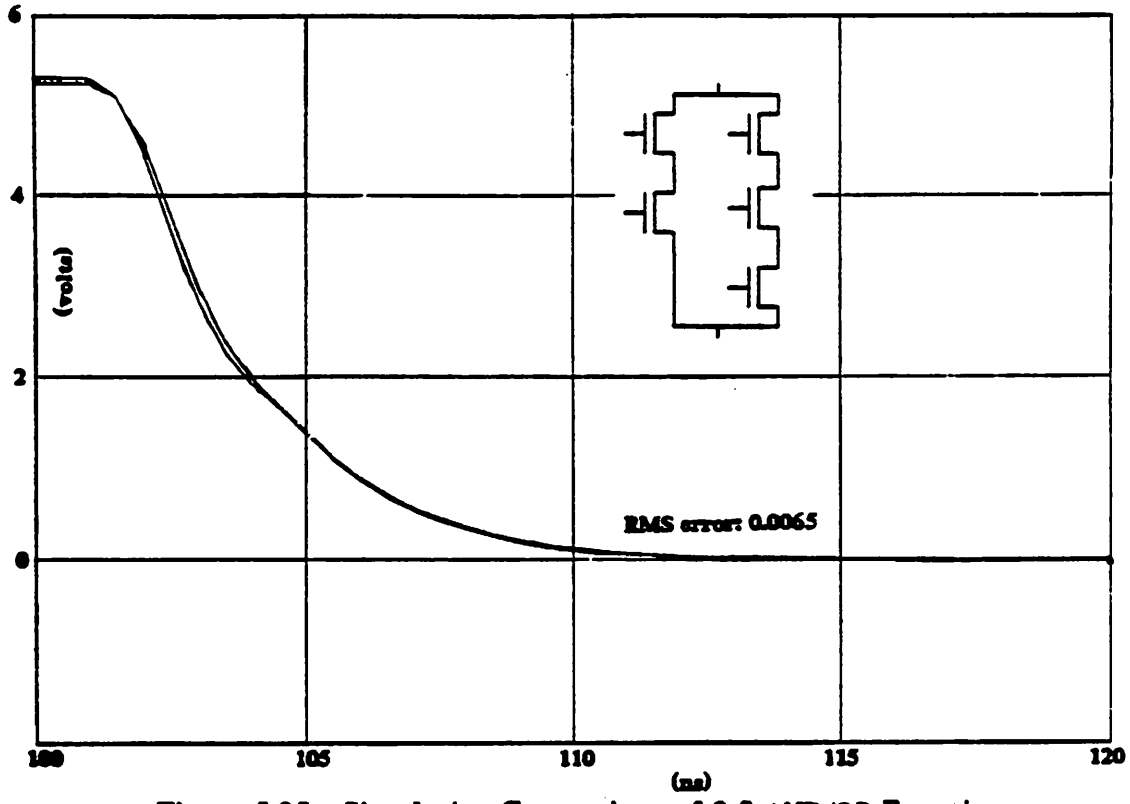


Figure 5.25a: Simulation Comparison of 2-3 AND/OR Function

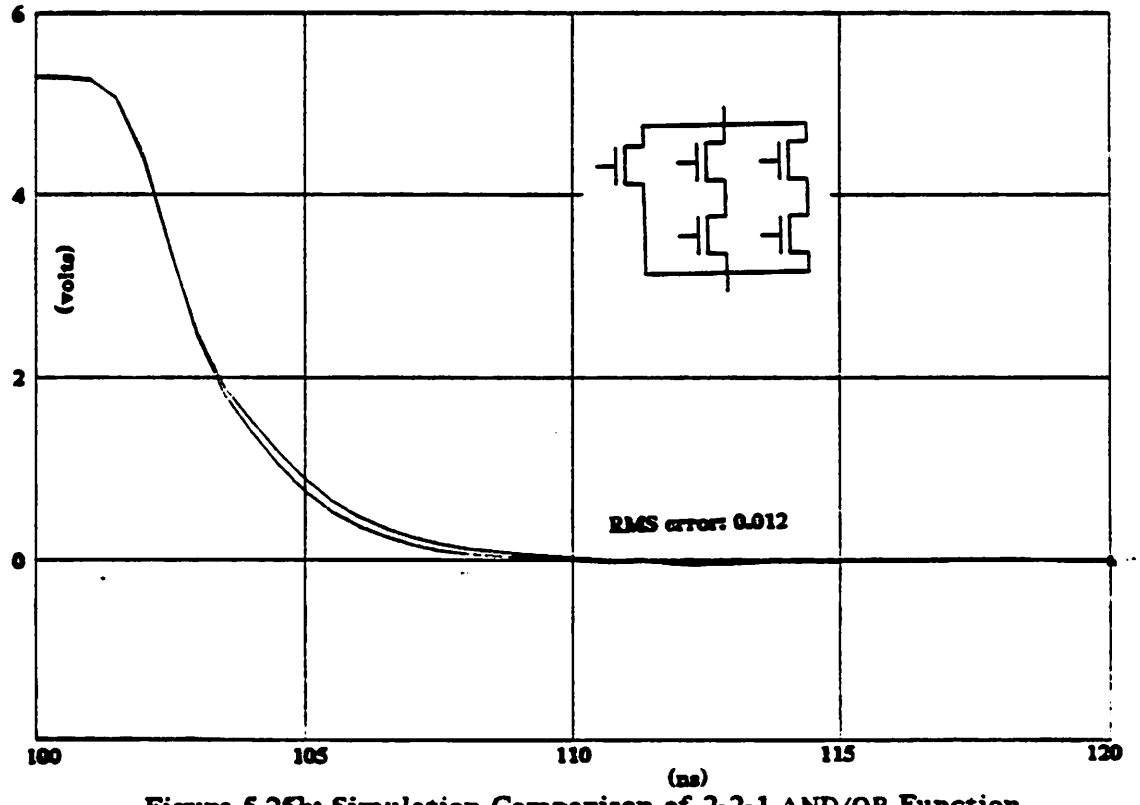


Figure 5.25b: Simulation Comparison of 2-2-1 AND/OR Function

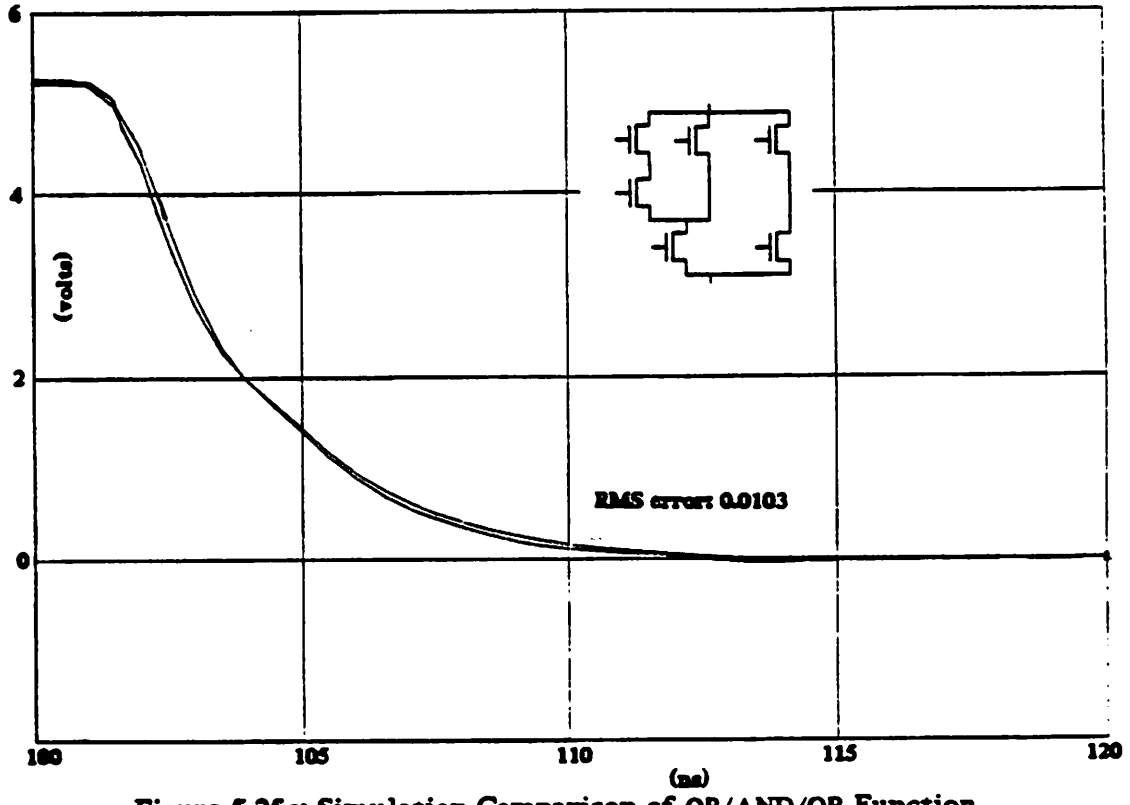


Figure 5.25c: Simulation Comparison of OR/AND/OR Function

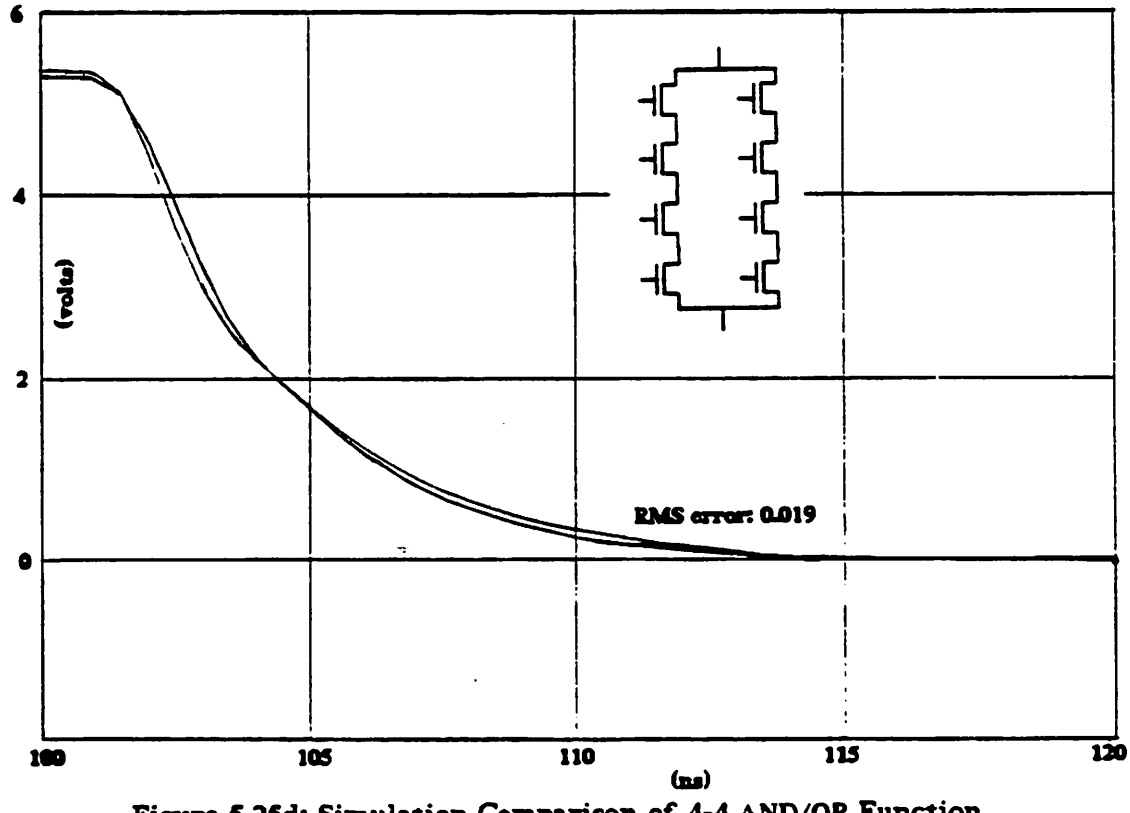


Figure 5.25d: Simulation Comparison of 4-4 AND/OR Function

One curve of each pair represents a worst-case SPICE2 simulation of full circuit, while the

other curve is the result of the $2-T$ model simulation. For each set the product of the RMS error in x and y between the two results is given (in *volt-ns*). When the RMS error is less than 0.1*volt-ns* the simulation results are seen to be nearly identical. Figures 5.26a-b show the result of a SPICE2 simulation on the full circuit versus simulation with $2-T$ models.

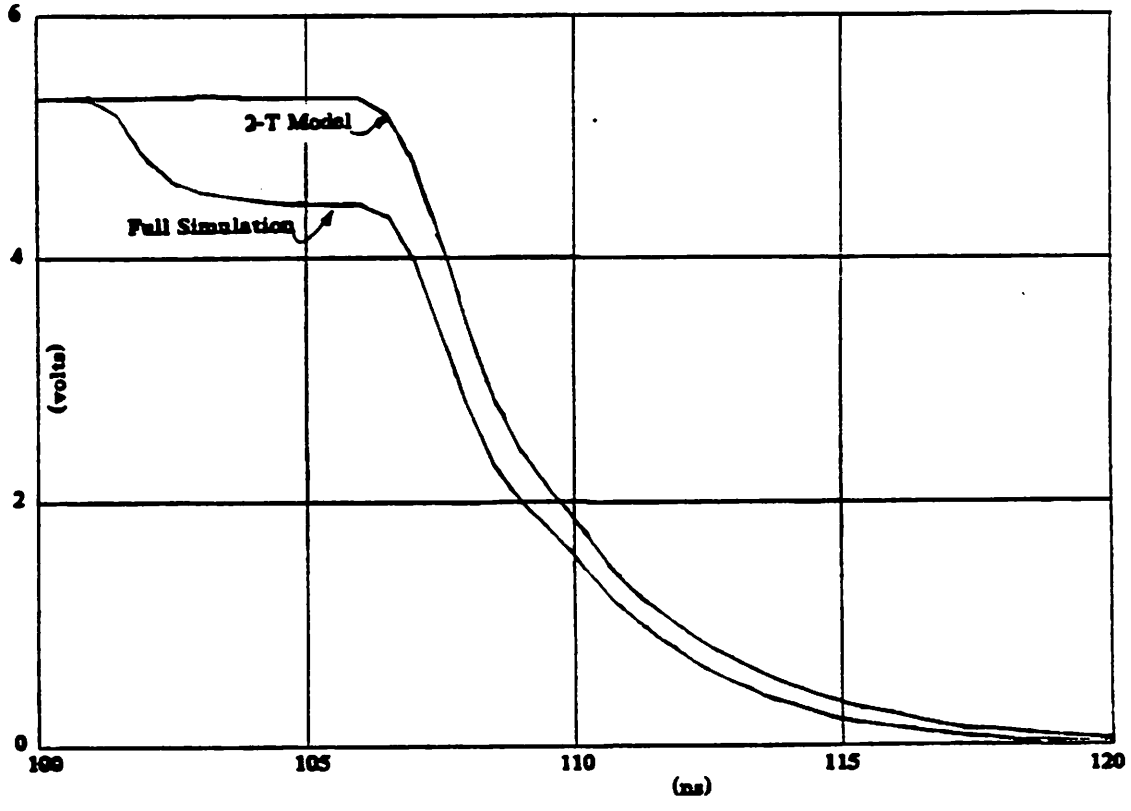


Figure 5.26a: Simulation Comparison of Full Circuit: Input to Last Stage

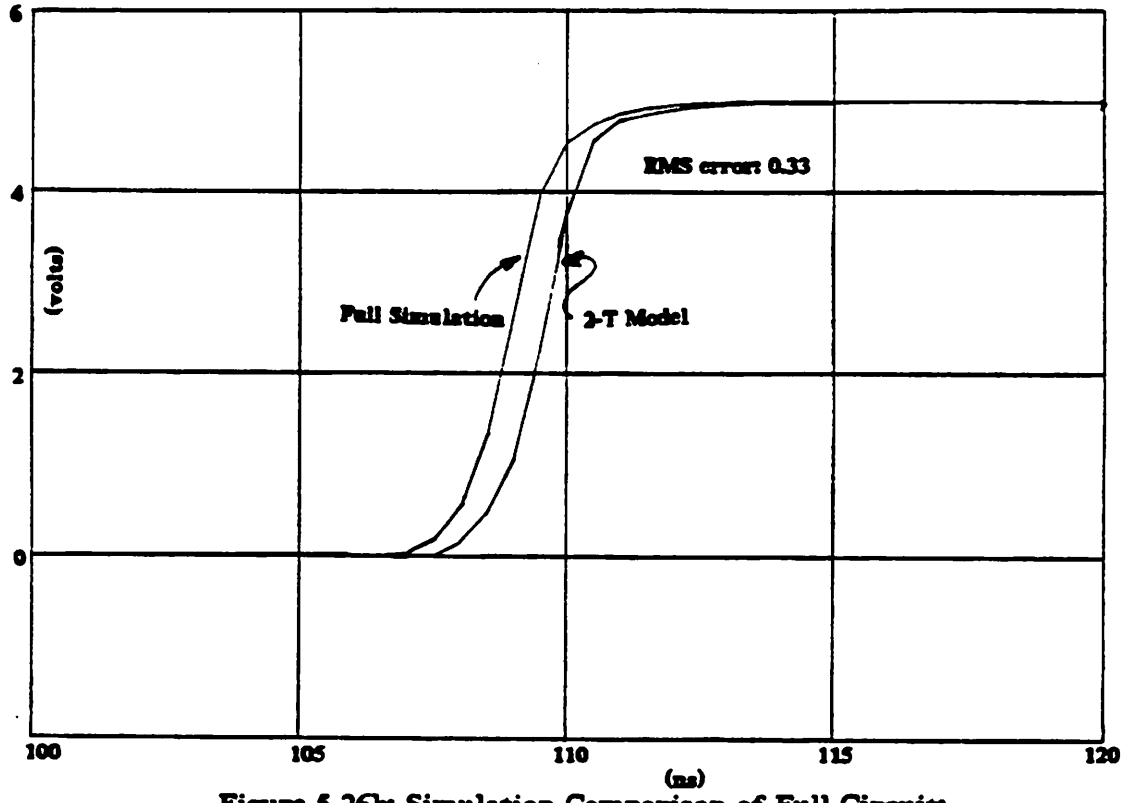


Figure 5.26b: Simulation Comparison of Full Circuit: Output from Last Stage

The actual circuit exhibits a dip at the precharged node before switching. This is due to charge redistribution. The problem is not serious enough to affect proper circuit operation: in fact, the circuit switches faster because of charge redistribution. The $2-T$ model, which assumes no CR problem, switches more slowly. This example represents an extreme. A slightly greater charge redistribution effect would have caused improper operation: a circuit which exhibits a smaller effect will agree better with the $2-T$ model. The $2-T$ model can be adjusted to account for the charge redistribution effect by precharging "on" devices to one V_t below V_{DD}. In this condition any charge redistribution between precharged and parasitic capacitances will cause an immediate switching of the output buffer. Therefore this represents the most sensitive charge sharing case. The simulation results shown in Figure 5.27 confirm the match between the actual circuit and the model. The precharge waveforms are in closer agreement and the RMS error at the output node has decreased by more than an order of magnitude.

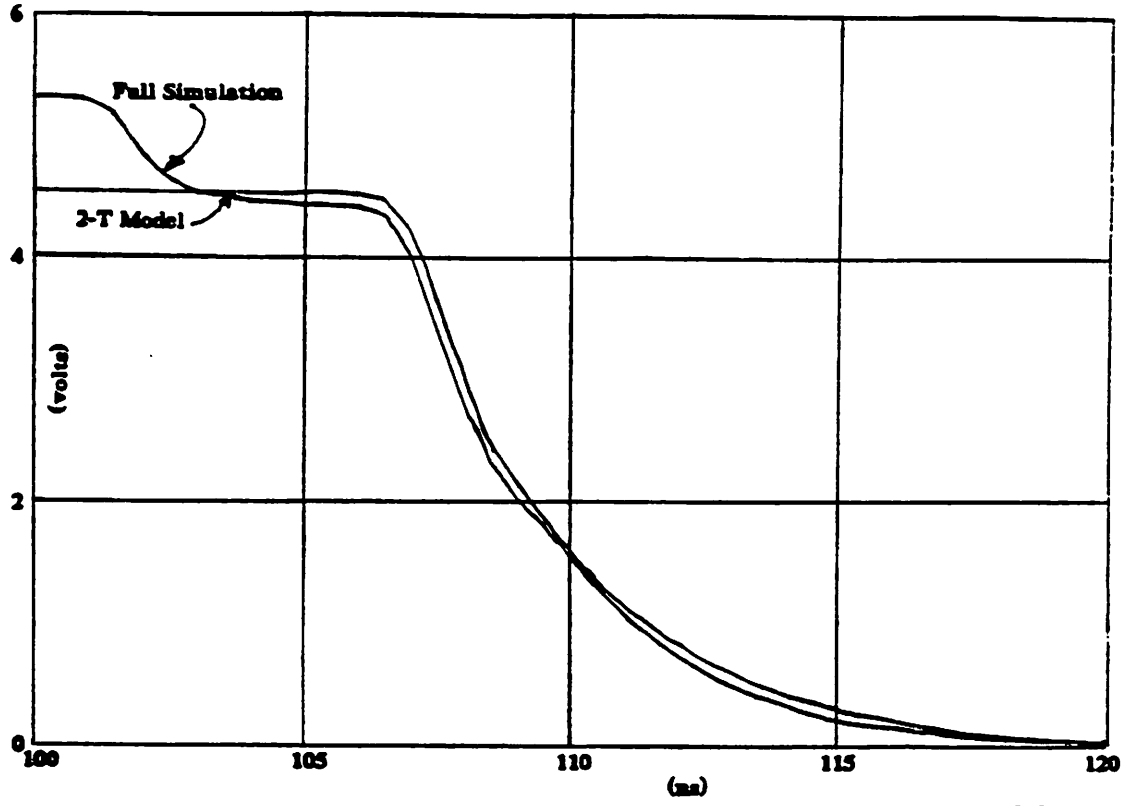


Figure 5.27a: Simulation Comparison of Charge-Compensated Model: Input to Last Stage

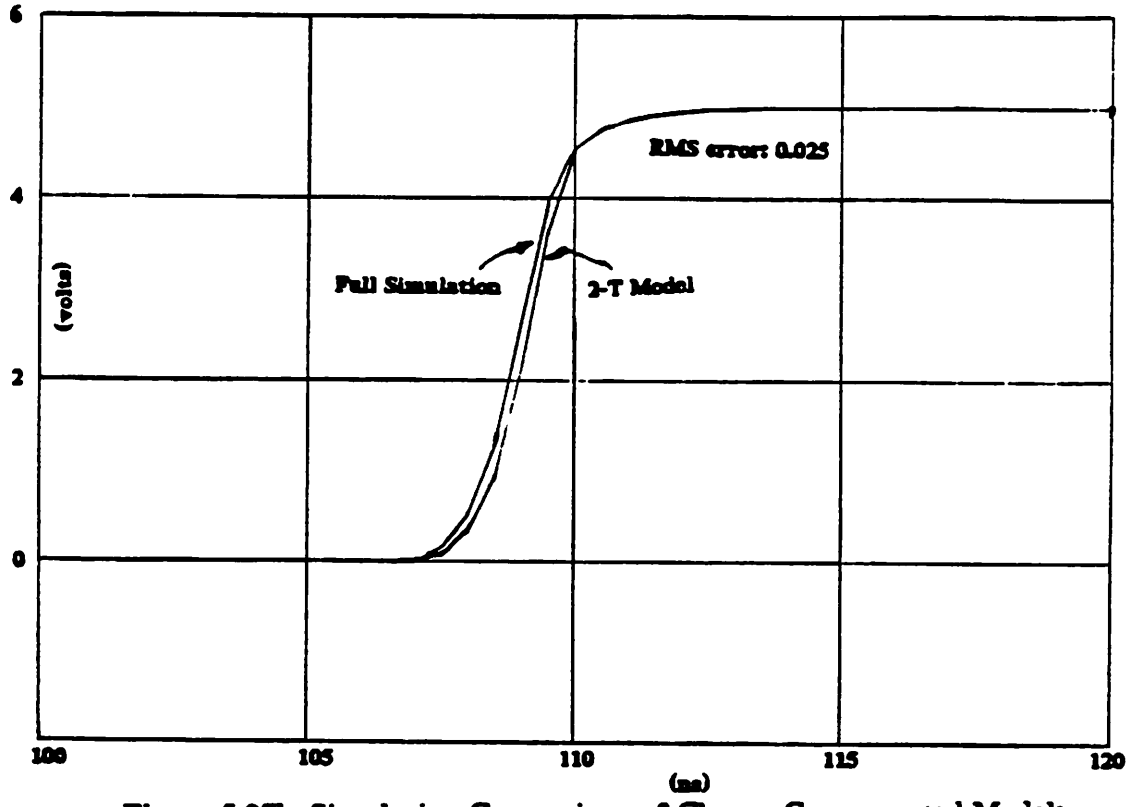


Figure 5.27b: Simulation Comparison of Charge-Compensated Model: Output from Last Stage

5.8. Elimination of Redundant Clusters— MIMIC

After a collection of Boolean expressions has been processed by the delay optimization stage of MAMBO, the MIMIC tool is used to reduce the number of gate clusters. MIMIC reads in the LISP-like netlist and performs a recursive matching of all inputs of each partitioned cluster with every other cluster. MIMIC thus requires $O(n^2)$ operations where n is the number of partitioned clusters.

When substituting one gate for another MIMIC is careful to preserve both input signal order and primary outputs. It is assumed that the series inputs to each cluster have been sorted for optimum delay. Therefore, even if two gates have the same inputs in a series string, MIMIC does *not* assume they are interchangeable. If two equivalent clusters are found (*ie.* with like inputs in the same sequence) then one gate may be deleted. The choice of gate to delete is based on the output signal. Clusters which produce external signals take precedence over internally used signals. External signals are part of the designer's

specification. If the matching signals are both external or both internal then either signal may be deleted. The existence of two identical external signals implies the designer has specified two separate Boolean equations which actually perform the same function.

A circuit from CMOS SOAR before and after processing by MIMIC is shown in Figure 5.28.

```
# charge tolerance ratio is 9.71276

(o readRFaccessA1
 (s
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) 3 )):

(o 3
 (p pbusDtoINA DSTvalid* SRC1equalDST2*
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) )):

(o readRFaccessB1
 (s
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) 11 )):

(o 11
 (p pbusDtoINA DSTvalid* SRC2equalDST2* SRC2equal16
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) )):

(o Alzero1
 (p
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) 18 )):

(o 18
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o Alzeroforce
 (p
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) )):

(o busDtobusAa
 (p 26 29 32 36 )):

(o 26
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o 29
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o 32
 (s pbusDtoINA* DSTvalid opc2load* SRC1equalDST2
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):
```

```
(o 36
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o DSTtobusDa2
 (p pbusDtoINA 44 48 )):

(o 44
 (s pbusDtoINA* DSTvalid opc2load* SRC1equalDST2
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o 48
 (s pbusDtoINA* DSTvalid opc2load* SRC2equalDST2 SRC2equal16*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o preadTBtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o preadSWPtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o pForwardtoINB
 (s pbusDtoINA* DSTvalid opc2load* SRC2equalDST2 SRC2equal16*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

(o preadPCtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) )):

# 19 cluster(s)
# Longest series string: 8 [target maximum unrestricted]
# Deepest Partition: 2 [target maximum 2]
# Deepest Buffer Hierarchy: 2
# Worst Ratioing Problem: 0.526874 [target minimum: 0.48 (Vth 2.4)]
```

Figure 5.28a: Delay Optimized Circuit Before MIMIC

```
# Deleted gates
# (o 48
#  (s pbusDtoINA* DSTvalid opc2load* SRC2equalDST2 SRC2equal16*
#  (p CPIPE1s<7> CPIPE1s<5>* )
#  (p CPIPE1s<7> CPIPE1s<5> ) )):

# (o 36
#  (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>
#  (p CPIPE1s<7> CPIPE1s<5>* )
```

```
# (p CPIPE1s<7> CPIPE1s<5> ) ));

#(o 44
# (s pbusDtoINA* DSTvalid opc2load* SRC1equalDST2
# (p CPIPE1s<7> CPIPE1s<5>* )
# (p CPIPE1s<7> CPIPE1s<5> ) ));

#(o 29
# (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>
# (p CPIPE1s<7> CPIPE1s<5>* )
# (p CPIPE1s<7> CPIPE1s<5> ) ));

#(o 26
# (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>*
# (p CPIPE1s<7> CPIPE1s<5>* )
# (p CPIPE1s<7> CPIPE1s<5> ) ));

# Irredundant gates
(o readRFaccessA1
 (s
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) 3 ));

(o 3
 (p pbusDtoINA DSTvalid* SRC1equalDST2*
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) ));

(o readRFaccessB1
 (s
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) 11 ));

(o 11
 (p pbusDtoINA DSTvalid* SRC2equalDST2* SRC2equal16
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) ));

(o Alzero1
 (p
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) 18 ));

(o 18
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

(o Alzeroforce
 (p
 (s CPIPE1s<7>* CPIPE1s<5> )
 (s CPIPE1s<7>* CPIPE1s<5>* ) ));

(o busDtobusAa
 (p preadSWPtoA preadTBtoA 32 preadPCtoA ));
```

```
(o preadSWPtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

(o preadTBtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2> SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

(o 32
 (s pbusDtoINA* DSTvalid opc2load* SRC1equalDST2
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

(o preadPCtoA
 (s pbusDtoINA* SRC1s<4> SRC1s<3>* SRC1s<2>* SRC1s<1>* SRC1s<0>
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

(o DSTtobusDa2
 (p pbusDtoINA 32 pForwardtoINB ));

(o pForwardtoINB
 (s pbusDtoINA* DSTvalid opc2load* SRC2equalDST2 SRC2equal16*
 (p CPIPE1s<7> CPIPE1s<5>* )
 (p CPIPE1s<7> CPIPE1s<5> ) ));

# 5 gates deleted: 14 gates in irredundant set
```

Figure 5.28b: Delay Optimized Circuit After MIMIC

The results of MIMIC on two delay optimized circuits are summarized in Figure 5.29.

Circuit	Initial Clusters	Deleted Clusters	Final Clusters
apla	19	5	14
cpla1	148	60	88

Figure 5.29: Results of MIMIC Processing

After redundant gates have been eliminated the remaining gates drive a higher fanout. The user should bear this in mind in the design of buffers. The tool does not yet automatically increase buffer size. As a result of gate elimination that critical path will not be altered. The goal of gate elimination is to reduce device count without adversely affecting speed.

5.9. Summary

In this chapter the delay optimization algorithms for electrical circuit design were examined. The MOSMESH program partitions complex combinational gates according to charge redistribution and series chain length constraints. The MKTBL program performs SPICE2 transient analyses on the partitioned gates. MOSMESH then uses this information to optimize the critical path of the combinational circuit. A simple two-transistor model has been developed to evaluate arbitrary meshes of precharged and discharged devices. The model assumes no charge redistribution problem exists and gives excellent results when simulating circuits with minimal charge sharing effects. Circuits which exhibit charge redistribution effects can be properly simulated by the $2-T$ model if the proximal FET is precharged to a V_t below the supply rail.

After the circuit has been partitioned into gate clusters, the MIMIC program recursively removes duplicate gates. A gate is considered redundant if its inputs and function are logically equivalent to those of another gate cluster. MIMIC does not remove clusters if doing so would decrease circuit speed.

CHAPTER 6

Compaction and Layout of Domino Matrix Structures

The delay optimization methods presented in the previous chapter were used on partitioned meshes or gate clusters. The Boolean minimization step was performed by first translating each group of clusters that realizes a particular function into a group of two-level personality matrices. For the topological compaction and succeeding stages in the MAMBO package it is necessary to work with a representation that more closely resembles the finished layout. The tool MKMAT is used to construct the initial matrix structure from the list of gate clusters. The three remaining programs to be described, TWIST, TINKER, and TAILOR, produce matrix-like structures with increasing detail. At the topological compaction level only connectivity information is needed, while TAILOR, the automated layout tiler, deals with mask-level geometries. This chapter examines circuits at their topological level.

6.1. Conversion of Partitioned Circuit to Matrix Structure— MKMAT

TWIST accepts as input an uncompacted *connectivity* matrix and produces a compacted connectivity matrix. The tool which constructs the initial connectivity matrix is MKMAT. MKMAT tries to create an efficient representation of the partitioned circuit. A fragment of an adder circuit, which will serve as an illustration, is shown in Figure 6.1.

```
(o f0
 (p 1 2 3 4 6 10 11 12 )):

(o 1
 (s 16 c3* a0 )):

(o 2
 (s 18 c3* b0 )):

(o 3(o 15
 (s 20 b0 a0 )):

(o 4
 (s 14
 (p b0 a0 ) )):

(o 6
 (s 15 7 )):

(o 10
 (s 28 c2 c3 )):

(o 11
 (s 24 c2 c3 )):

(o 12
 (s 26 c2 c3 )):

(o 28
(s cin b0* a0* )):

(o 26
(s cin* b0* a0 )):

(o 24
(s cin* b0 a0* )):

(s c1 c2* c3 )):

(o 7
(p
(s b0 a0* )
(s b0* a0 ) )):

(o 14
(s c1* c2* c3 )):

(o 20
(s c1 c2 c3* )):

(o 18
(s c1* c2 )):

(o 16
(s c1 c2* )):
```

Figure 6.1: Adder Fragment, Input to MKMAT

The fragment contains 18 gate clusters and 30 distinct signals (inputs, outputs, and internal, numbered signals). As a result of running MKMAT the output shown in Figure 6.2 is constructed.

new 30 19

```
                psssssssssssp.ssss
18              ..s.............o.
20              ...s............o..
14              ....s...........o...
24              .......s...o........
26              ........s.o.........
f0              o...................
1               po..................
2               p.o.................
3               p..o................
4               p...o...............
6               p....o..............
10              p.....o.............
11              p......o............
12              p.......o...........
16              .s..................o
c3*             .ss.............._.s..
a0              .s.s_.....s..._.....
b0              ..ssp......s._......
15              .....s......o.......
7               .....s.......o......
28              ......s..o..........
c2              ......sss........ss.
c3              ......sss...s..s....
cin             .........s..........
b0*             .........ss...s.....
a0*             .........s.s.s......
cin*            ..........ss........
c1              ............s...s.s
c2*             ............s..s..s
c1*             ..............s.s.
```

Figure 6.2: Adder Fragment, Output from MKMAT

In this figure external and internal inputs and outputs run horizontally, forming rows of the connectivity matrix. The columns of the matrix are gate clusters. The row at the top of the matrix holds header information. The character s declares that the gate in this column is *series* or *AND* in nature at its top level. Similarly the character p declares that the cluster in this column is *parallel* or *OR* in nature at its top level. The character o indicates a gate-column output connection to a signal-row: the . character indicates that signals

are bussed through this tile without connection to the current gate. In this small example all but two clusters are single-level in nature and therefore can be adequately described by a single column. For the two clusters:

```
(o 4
   (s 14
      (p b0 a0 ) ));

(o 7
   (p
      (s b0 a0* )
      (s b0* a0 ) ));
```

of Figure 6.1 it would be necessary to allocate two columns each to realize their functions following the rule that each column is either AND (*s*) or OR (*p*) in nature. However two-level expressions are permitted in each gate or column. Higher level expressions are currently forbidden. The addition of the character ~, interpreted to mean *toggle*, allows two-level logic to be expressed symbolically. For example, referring to Figure 6.2, the two-level gate with output *4* and inputs *14*, *b0*, and *a0* is shown in the fifth column from the left. The column realizing the gate is series at the top-level, indicated by the header character. After a series transistor placement at signal *14* the gate toggles to the parallel type at signal *a0*. The transistor at *b0* is also parallel so the gate type remains unchanged.

Note that being able to generate two-level clusters is not a guarantee that all such clusters can be placed in a single column. Again referring to Figure 6.2 it can be seen that the gate with an output of *7* takes up two columns. Two columns are used because the two series groups or legs of the gate made up of {*a0 b0**} and {*b0 a0**}, respectively, intersect one another. It is not possible to interconnect all the devices in each of the series groups without intersecting devices in another group. It is sometimes possible to work around this problem by reordering the signal rows. Figure 6.3 shows the result of running MKMAT on the example of Figure 6.1 with additional ordering constraints.

```
new 30 18

                  psssssssssssspssss
20                ...s...........o..
14                ....s.........o...
18                ..s.............o.
b0                ..ss.....s.....
24                .......s...o......
26                ........s.o.......
a0*               .........s.s.s....
f0                o.................
1                 po................
2                 p.o...............
3                 p..o..............
4                 p...o.............
6                 p....o............
10                p.....o...........
11                p......o..........
12                p.......o.........
16                .s...............o
c3*               .ss............s..
a0                .s.sp.....s.....
15                .....s......o.....
7                 .....s.......o....
28                ......s..o........
c2                ......sss......ss.
c3                ......sss...s.s...
cin               .........s........
b0*               .........ss..s....
cin*              ..........ss......
c1                ............s..s.s
c2*               ............s.s..s
c1*               .............s.s.
```

Figure 6.3: Output from MKMAT with Additional Constraints

6.1.1. Constraints Placed on MKMAT by MOSMESH

It is not possible to sort signals in general so that the number of columns per gate is minimum. As a result of delay optimization performed by MOSMESH the ordering of signals in series gates or series portions of two-level gates is constrained. MOSMESH orders signals so that the fastest changing ones are closest to the gate cluster output. Even though a gate may not be on the critical path of a module it is currently considered

constrained. However, external signals which by convention begin with a non-numeric character, are all assumed to change (if they change at all) at time $t=0$. This rule is enforced by design in Domino logic. In the adder fragment example the delay optimization was turned off to show the effect of reordering rows. When delay optimization is in effect the placement of internal signals is considered immutable. The ordering of gates is not constrained by MOSMESH.

The designer is not allowed to specify the ordering of external signals. It was felt that the optimization for delay more than compensates for this. The circuit designer may, however, specify on which side of the array he wishes the signals to be accessible. This specification will be described in detail later in this chapter.

There is one more condition which forces MKMAT to use multiple columns for gate realization and that is multiple instances of an input signal. Assume one has the following gate:

```
(o output
   (p
      (s a b ) (s a c ) (s a d ) ));
```

Though this is a two-level structure it could be realized as a single column if all input signals were distinct. For this gate, however, signal name *a* is repeated. Rather than duplicate a signal line, which introduces routing problems to the module, an extra column is added. The rule is that all signals realized in a single column are distinct. The gate shown in the example would thus require three columns.

The above example factors easily. Expressed in LISP-like notation the expression may be rewritten:

```
(o output
   (s a
      (p b c d ) ));
```

The factored expression not only needs fewer devices but it remains two-level. In general, pulling out one or more factors of a function will increase the depth of the expression. In MAMBO the rule is that factoring can be done when the resulting expression's depth does not exceed two.

6.1.2. Valid MKMAT Structures

While the output of MKMAT is limited to clusters two levels deep, it can construct some three level partitions. For simplicity, because not all three-level partitions can be formed, none are permitted. There are four primary structures and concatenations thereof that MKMAT may generate. Examples of the four primary structures in schematic representation with their corresponding connectivity matrix shorthand are shown in Figure 6.4.

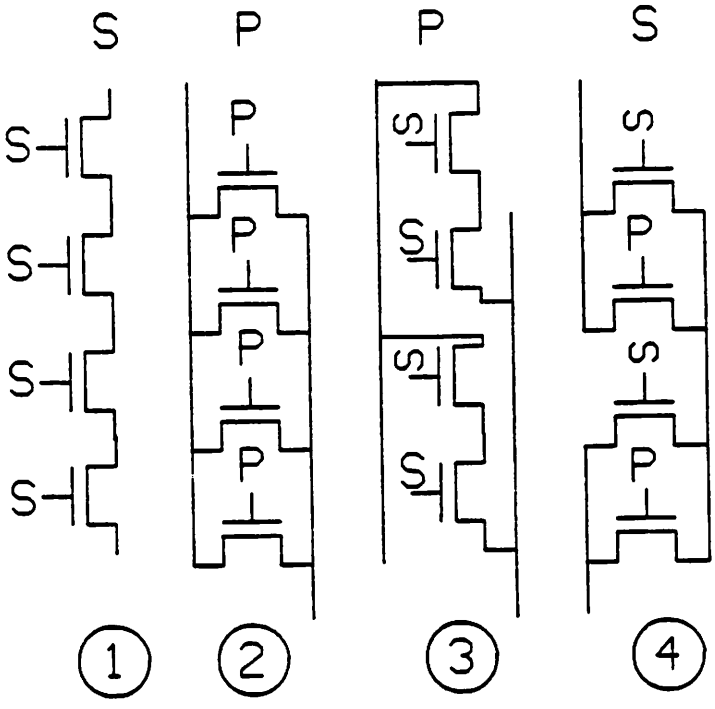


Figure 6.4: Primary MKMAT Structures

In order to create a regular structure which may be densely packed by folding algorithms only these four basic structures are permitted. For reference Figure 6.5 shows two three-level structures as a designer might envision them and as they would look in the matrix layout style. While the structure in Figure 6.5a can be built with the current system, the layout shown in Figure 6.5b is not amenable to structured layout.

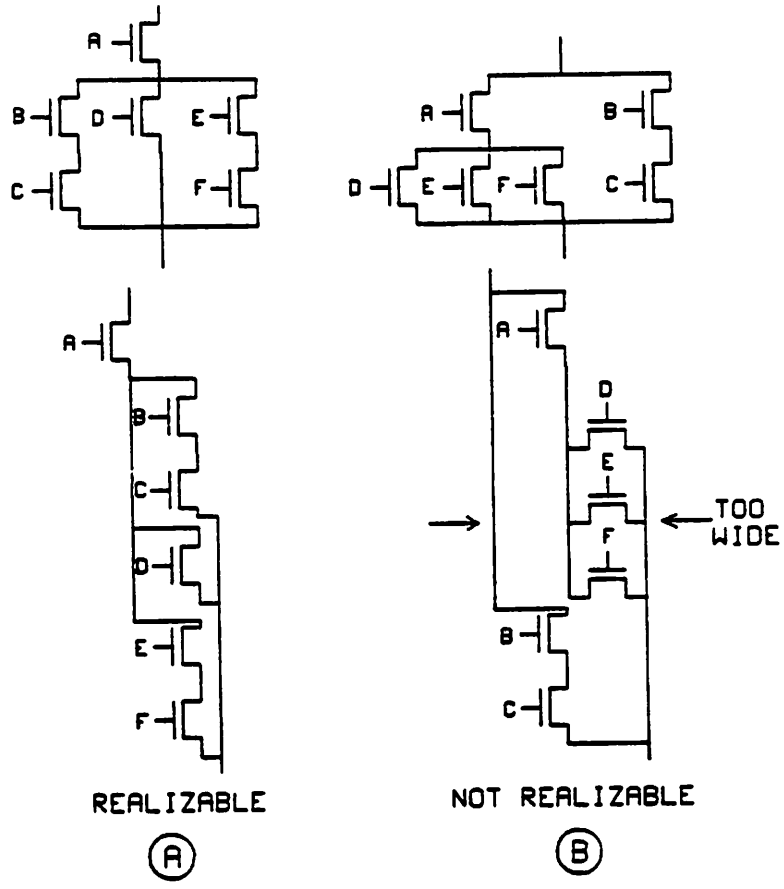


Figure 6.5: Three-level Domino Structures

If special wide cells were built to accommodate three-level layouts topological compaction would be complicated by the different size cells. On the other hand, if cell sizes were standardized they would have to accommodate the most complex structures, which would mean that those cells housing simple structures, usually in the majority, would waste area. It is also pointed out that the notation system of {*s p o* ~ .} does not extend beyond two levels.

6.1.3. How MKMAT Works

MKMAT has three major sections. After parsing the input netlist and storing it in a tree data structure, the tool builds a list of distinct signal names. This includes not only

external inputs and outputs but also internal, machine-generated signal names that feed from one cluster to another. Each signal name will be represented as exactly one row in the output matrix.

The second section of MKMAT builds a row constraint matrix. The constraints in this matrix are of two types: *hard* constraints which are inviolable and which are imposed by prior delay optimization and optional *soft* constraints which, if obeyed, may result in a more compact matrix, with fewer columns. The expression:

```
(o o1
    (s 1 2 3 4 ));
```

produces the constraint matrix of Figure 6.6.

```
4 3 2 1 o1
- - - - - 4
x - - - - 3
x x - - - 2
x x x - - 1
- - - - - o1
```

Figure 6.6: Cycle-Free Constraint Matrix

The *x*'s indicate hard constraints imposed by the series ordering of internal signals. The matrix indicates that rows *4* and *o1* may be placed without constraint. Signal *3* can only be placed after *4* has been placed. Likewise signal *2* can only be placed after signals *3* and *4* have be positioned. It can be seen that eventually all rows will be placed. In contrast, the two equations:

```
(o o1
    (s 1 2 3 4 ));

(o o2
    (s 2 1 3 4 ));
```

will generate the constraint matrix of Figure 6.7.

```
o24 3 2 1 o1
- - - - - - o2
- - - - - - 4
- x - - - - 3
- x x - x - 2
- x x x - - 1
- - - - - - o1
```

Figure 6.7: Cyclic Constraint Matrix

This matrix contains a cycle. The hard constraints at positions *(2,1)* and *(1,2)* of the matrix indicate that row *1* must be placed after row *2* but that row *2* must be placed after row *1*. The delay optimization tool ensures that such a case cannot occur. If MKMAT detects cycles a fatal error is generated and the program exits. In this case the message is:

sort_signals: fatal error: Signal 2 is involved in cyclic constraint.

Finally, MKMAT can also generate soft constraints. The input:

```
(o o1
    (p
        (s 1 2 ) (s 3 4 ) ));
```

will create the constraint matrix shown in Figure 6.8.

```
2 1 4 3 o1
- - - - - 2
x - - - - 1
o o - - - 4
o o x - - 3
- - - - - o1
```

Figure 6.8: Matrix with Hard and Soft Constraints

The *o*'s indicate soft constraints. If these constraints are followed signals *1* and *2* will be placed before signals *3* and *4*. This guarantees that the gate can be realized in a single column. However this is not the only configuration that gives a single column result. Since the general problem of optimum placement of groups for best packing is $O(n!)$ where n represents the number of signal groups, and all that is really necessary is that signals {*1 2*} and signals {*3 4*} form contiguous, disjoint groups, by finding a single solution through the use of constraint matrices, MKMAT reduces computation time.

The last step in MKMAT is to determine the depth of each individual cluster. Clusters with a depth of one can be directly translated into a single column. For gates with a depth of two MKMAT determines the number of columns the gate will require. It does this by finding the upper and lower extremes of each group or subcluster of signals. A subcluster is an atomic expression— either $(s\ sig_1 \ldots sig_n)$ or $(p\ sig_1 \ldots sig_n)$. Once the extent of each subcluster is calculated the well-known "left-edge" algorithm [hash71] can be used to find the optimum packing. After the gates have been placed in an output structure a print procedure creates the actual character personality matrix.

6.2. Algorithms for Topological Compaction— TWIST

TWIST reads the array generated by MKMAT and produces the potentially compacted array in the same format. To indicate where rows and columns have been broken, extra information, in addition to the matrix personality, is produced. TWIST has two modes of operation. It may be used as part of the MAMBO pipeline like the rest of the tools men-

tioned here or it may be used interactively. In interactive mode the designer may enter a personality matrix directly to try out a configuration or input may be read from a file. In either case, when TWIST is used interactively the designer may manually fold rows and columns to create the compact structure he wants. The remainder of this chapter will examine TWIST running as part of a pipeline. In this mode TWIST automatically folds as many rows and columns as possible, though the user still controls the sequence of folding— row- or column-first. The designer may also specify on which side external signals must be brought out. Signals may be made available on two sides of the matrix as bus-through connections.

TWIST implements simple column folding and multiple row folding. The buffering and precharge devices are contained in a single cell and may be placed either on the top or bottom of the multi-level array. The left and right boundaries are reserved for input/output access to/from the module by external signals. Since the buffer cell can only be placed at the top or bottom of the module, only simple column folding is allowed. The term *simple folding* means that only two terms may be folded into one. Rows may be multiply folded. While external signals can only be brought out on the left and right sides of the module, internal signals joining one cluster to another do not need to touch the module boundaries. Thus it is possible to fold an arbitrary number of row signals into a single physical row.

The ordering of signals is significant. Signals are ordered relative to the location of the output buffer. When columns are folded the column on the bottom of the array is inverted. The buffer is placed along the bottom edge of the array and the ordering of signals that contact the flipped gate must also be inverted. The ordering of these signals is recorded in a row ordering matrix or *rom*. The necessity to invert or "flip" constraints when a column is folded differentiates the folding of these arrays from other structured arrays, like PLAs or static gate matrices. The consequences of column inversion are examined more closely in the column folding heuristic presented below.

The methods used to produce simple column and multiple row folding are now explored. The core of TWIST consists of three basic steps. These are: 1) selection of trial folding candidates: 2) detection of folding cycles: and 3) construction of the folded matrix. These topics are the subject of the next three sections.

6.2.1. Selection of Trial Folding Candidates

As mentioned above. TWIST may fold either rows or columns or both. In pipeline mode. TWIST either creates a complete set of row folds and then attempts column folding. or tries column folding first and row folding second. In interactive mode the user has finer control: he may elect to fold several rows. then fold several columns. and then try row folding again. The heuristics employed are order sensitive and row-after-column folding will in general produce a finished module of different aspect ratio than column-after-row folding. The designer may prefer a tall. thin module to a short. squat one. TWIST uses a "straight through" algorithm for row folding and a slightly more complex approach for column folding.

6.2.1.1. Row Folding Heuristic

Because multiple row folds are allowed. a relatively fast heuristic was needed to eliminate the many folding possibilities in the often-sparse matrix. Figure 6.9 lists the row folding heuristic in pidgin-C.

```
if (Cycle.cfm or Cycle.rfm or Cycle.rom) {
   Cycle.stop = TRUE;
   return (stop);
} else {
   new_try = get_right();
   if (new_try is NOT_SET) {
      Cycle.stop = TRUE;
      return (stop);
   }
   store(new_try);
   new_try = get_left();
   while (new_try is NOT_SET) {
      reset (row vector);
      reset (external constraints);
      new_try = get_right();          /* Get a new right element for pair */
      store (new_try);
      if (new_try is NOT_SET) {
         Cycle.stop = TRUE;
         return (stop);
      }
      new_try = get_left();
   }
   store (new_try);                   /* install left element of latest row fold */
   return (new);
}
```

Figure 6.9: Row Folding Heuristic

This code fragment chooses the next pair of row folding candidates. Immediately upon entering the procedure cycle checking is performed. There are three possible types of cycles; each of them indicates that further folding is not possible. *Cycle.cfm* indicates the presence of a cycle in the column folding (or intersection) matrix. Likewise, the *rfm* flag indicates a cycle in the row folding (intersection) matrix. The final flag, *rom*, checks the row ordering matrix. If any of these flags are TRUE the row folding procedure terminates after setting the stop flag. When the calling procedure becomes active again it detects that the stop flag is TRUE. The last consistent state of the matrix (*ie*. with no cycles), saved previously, is restored and row folding is declared done.

If none of the cycle flags are set the "best" next right folding element of a row pair is selected by procedure *get_right()*. The "best" right element is defined as that element with the rightmost leftmost non-DOT character. This is the row which has the shortest extent from the right edge module boundary to the left. The idea is to attempt a fold while disturbing the current matrix structure as little as possible. Thus the assumption is that the

initial ordering is reasonable. In fact this is the case, since, as a result of previous delay optimization, signals which participate in the same function will be grouped together. In the case of a tie, two or more rows with identical left extents, that row is chosen which has the fewest non-DOT elements in columns not already used by previous folds. The notion here is to introduce as few new constraints on the remaining unfolded rows and columns as possible. Elements in columns used in previous folds are already constrained and further constraining these columns restricts future folding possibilities less than constraining unfolded elements. The *get_right()* procedure respects the designer's external signal constraints. The chosen row may have already been folded previously. Signals which must be brought out on either side of the module are marked here as terminating on a module boundary; the procedure ensures that signals which must terminate on the right side are not internally buried in the array. A signal which must be brought out on the left side *may* be brought out on the right side as well. Such a signal might be used as a bus through. If all right side signals have been used, *get_right()* returns with *NOT_SET*, the matrix is taken to be fully folded, the stop flag is set, and the procedure terminates. If a right row element is chosen successfully it is stored and procedure *get_left()* is called. *Get_left()* chooses a row which not only is disjoint from the right element but also does not overlap it. The choice of left row is guided by the position of the right row. The left row closest to the right row which does not overlap it is chosen. The procedure *get_left()* searches in widening oscillations about the right row position. *Get_left()* also checks boundary conditions in the same manner as *get_right()*.

If *get_left()* can find no row to match the current right choice then the right element is unfolded. *get_right()* is called again, and the procedure for finding a left element is repeated. Eventually, either a {left, right} pair is found or all rows have been examined as choices for right rows. In the latter case the stop flag is set and the folding algorithm terminates. If the selection of a new folding pair is successful the coordinates of the new pair are entered into the state vector and the folder returns to its calling parent. Row folding is straight through; it proceeds from an initial matrix to a final placement without

exploring alternate folding paths. If a right or left row choice is unacceptable other possibilities will be tried, but once a cycle is introduced the procedure terminates. By contrast an exhaustive approach would push back cycles to the initial unfolded matrix and keep track of the largest number of folds obtained by doing a depth first search on all valid folding combinations.

6.2.1.2. Column Folding Heuristic

Typically, fewer columns than rows can be folded. Also, the number of rows will generally exceed the number of columns since each column has an output which runs on a row either to the module boundary or to the input of another gate. Thus the number of rows is equal to the number of columns at minimum, and in addition there must be at least one external input, so row count exceeds column count. However, some gates take more than one column to realize. There may be a single function (and therefore a single output) which spans an arbitrary number of columns. Currently the column folding algorithm leaves multiple-column gates alone. For these reasons and because only simple folding is allowed a more detailed column folding algorithm is employed. The column folding algorithm attempts to break cycles and continue folding until all possibilities from a given initial choice are exhausted. Figure 6.10 lists the column folding heuristic in Pidgin-C.

```
if (Cycle.cfm or Cycle.rfm) {
   new_try = get_top();
   if (new_try is NOT_SET) {
      Cycle.stop = TRUE;
      return (deleted);
   } else {
      store (new_try);          /* Substitute one fold for another, update state vector */
      return (stop);
   }
} elif (Cycle.rom) {
   new_try = break_intra();
   if (new_try is NOT_SET) {
      if (break_inter()) {
         new_try  = get_top();
         if (new_try is NOT_SET) {
            Cycle.stop = TRUE;          /* Delete an old fold */
            return (deleted);
         } else {
            reset (ccv);
            store (new_try);            /* Substitute one fold for another, update state vector */
            return (stop);
         }
      } else {
         Cycle.stop = TRUE;             /* Delete an old fold */
         return (deleted);
      }
   } else {
      pflist->element = new_try;        /* Flip a single column, enter it into flip list */
      return (stop);
   }
} else {
   if (flip_list isnt NIL) {
      store (pflist->element);
      new_try = get_top();
      if (new_try isnt NOT_SET) {
         pflist->element = NOT_SET;
         store (new_try);
         return (new);
      }
   }
   new_try = get_bottom();
   if (new_try is NOT_SET) {
      Cycle.stop = TRUE;        /* No more folding possibilities so give up */
      return (stop);
   }
   store (new_try);
   new_try = get_top();
   if (new_try is NOT_SET) {
      Cycle.stop = TRUE;        /* No more folding possibilities so give up */
      return (stop);
   }
   store (new_try);             /* Add a new folding pair */
   return (new);
}
```

Figure 6.10: Column Folding Heuristic

Immediately upon entering this code fragment *cfm* and *rfm* cycles are checked for. If at

least one of these cycles exists then the top gate in the last attempted fold is deleted and perhaps replaced. This gate must participate in the fold since the column procedure always starts with a consistent state. The procedure *get_top( )* looks for columns which are disjoint from the bottom folding partner. From this set of columns a column is chosen with the highest lowest non-DOT element. This is analogous to the row folding procedure, where the idea is to cause the least disturbance to the current matrix. The only difference here is that the column folding pairs may overlap, that is the top element of the bottom column may extend past the bottom element of the top column. Again, if there is a tie for the shortest downward extent, the chosen column will be the one with the fewest non-DOT elements that do not already participate in previous folds.

If *get_top( )* returns *NOT_SET* then no more top folding candidates exist; the last top fold is deleted, the stop flag is set and folding terminates. If a new top element was found it replaces the last top element.

If no *cfm* or *rfm* cycles exist but a *rom* cycle does exist then the algorithm attempts to break the cycle. Cycles are caused by two different types of constraints, intra- and inter-column, and different methods of attack are used to break each. An intra-column constraint is caused by a signal ordering conflict between top and bottom elements within a folding pair. For example, the top element of a pair may demand that signals be ordered {foo, bar} while the bottom element requires {bar, foo}. All bottom elements have had their constraints inverted at this time. Intra-column constraints, if they exist, are fixed before inter-column constraints. Procedure *break_intra( )* attempts to find a vertex in the current cycle which is caused by a single bottom element. A vertex in the cycle graph corresponds to an x in the *rom*. It is possible that more than one column is responsible for a vertex in the cycle. If there are several vertices with a constraint multiplicity of one then the column that has its elements most compressed toward the bottom of the matrix is removed. The chosen column will be *added* to the *flip_list*. This is a list of columns which have been inverted but which currently have no matching top element. The column with most elements compressed toward the bottom causes least rearrangement of the

matrix since this column will now have its output buffer placed along the bottom of the matrix. If no vertex is found which has a multiplicity of one, *break_intra()* returns *NOT_SET*.

Instead of removing a column from the folded pairs an attempt is made here to maintain those columns already folded by flipping additional columns. Flipping columns does not increase the number of folds but it may allow previous folds to be retained.

If *break_intra()* was not successful in finding a column to flip the procedure *break_inter()* is called. *Break_inter()* ascertains whether the last top folded element is involved in the current cycle. If it is not then the cycle must be caused by another, earlier, fold. Rather than unfolding columns to locate the cause of the cycle, the stop flag is set and the column folder terminates. However, if the last folded top element does participate in the current cycle it is deleted and *get_top()* is called to replace it with another column. If *get_top()* finds no eligible columns the stop flag is set and the folder terminates, otherwise the new folded column is substituted for the old, and the state vector is updated.

If no cycles exist when the code fragment of Figure 6.10 is called then a new fold can be created. First an attempt is made to fold a top column with currently unfolded bottom elements on the *flip_list*. If the list is not empty and a top fold match is returned by *get_top()* then the state vector is updated and the algorithm returns to the calling program. If there are no items in the *flip_list* or no suitable top columns can be found then the procedure *get_bottom()* is called. *Get_bottom()* searches through the set of unfolded columns (recall that only simple folding is permitted) for a column with the lowest highest non-DOT element, in other words, the column most compressed toward the bottom of the matrix. If there is a tie, then the column with the smallest weight is chosen. The "weight" of a column is computed as follows: DOT elements count 0, PARALLEL and OUTPUT elements count 1, and SERIES elements count 2. These values reflect the number of constraints each element causes. SERIES elements not only cause constraints between columns, they also dictate signal ordering within a column. Since the bottom element of a

folding pair will have its constraints inverted the smaller-weight algorithm attempts to minimize the number of new constraints introduced.

If *get_bottom()* finds no unfolded columns among gates that span a single column it returns *NOT_SET*. **stop** is set, and the folder terminates. If a bottom element is found then *get_top()* is called to try to find a match. Again, if no match is found, **stop** is set and the folder terminates. If a new folding pair is identified the state vector is updated and the column folder returns to the calling program.

The cycle-checking algorithm is run in concert with the folding selection algorithms. The next section describes the cycle detection algorithm and give bounds on its complexity.

6.2.2. Matrix Representation of Gate Matrix Folding Problem

This section considers theoretical aspects of the topological compaction of the multi-level structures produced by MKMAT. The algorithms presented here have been implemented in TWIST. De Micheli [demi84] presented a graph theoretic interpretation of the general multiple folding problem for PLAs. The work detailed below describes a similar approach based on that of De Micheli but in matrix form and tailored to particular aspects of multi-level matrix (*MLM*) structures.

6.2.2.1. Problem Statement

Given a connectivity matrix, like that produced by MKMAT, construct column and row intersection matrices which indicate which columns and rows, respectively, can be merged. From the intersection matrices determine what folds are implementable to give a minimum cardinality (area) MLM. The connectivity matrix may be translated into a more abstract form. This abstract form closely resembles the AND plane personality matrix of a PLA. Figure 6.11 presents an example personality matrix. A device placement, shown as a *1*, indicates a connection between the *i*th row and *j*th column. A *0* indicates no connection.

```
1 0 1 1 0 0 1
2 0 1 0 1 0 0
3 1 0 0 0 0 1
4 1 0 0 0 1 0
5 1 0 0 0 0 0
6 0 0 0 0 0 1
  -----------
  1 2 3 4 5 6
```

Figure 6.11: Personality Matrix

A column-intersection matrix indicates which columns have intersecting row contact sets. A row-intersection matrix indicates which rows have intersecting column contact sets. Hachtel [hach80] presented the same information in graphical form. Figure 6.12 is the column-intersection matrix for the example in Figure 6.11.

```
1 x         x x
2   x x x     x
3   x x       x
4   x   x
5 x           x
6 x x x       x
  -----------
  1 2 3 4 5 6
```

Figure 6.12: Column Intersection Matrix

The *x*'s indicate which columns intersect. The matrix is symmetrical about the *(0,0)—(n,n)* diagonal.

6.2.2.2. Folding Algorithm

Folding candidates are chosen subject to external constraints imposed by the prior delay optimization stage and by the designer. In addition, folding candidates must be disjoint. This rule is enforced by the intersection graphs. Another requirement is that the folding candidates must not create a ζ—cycle. This requirement is examined below.

The condition that two unfolded candidates be disjoint is enforced by the x's in the intersection matrix. A folded candidate may be folded with another folded or unfolded candidate to create a multiply folded result. In this case all folding candidates must be disjoint from all other folding candidates.

There are three different cases of ζ—cycles. Cycles are potentially introduced when folding candidates are entered into the column folding matrix (*cfm*). The x—axis of the *cfm* is defined as the *from*-axis and the y—axis as the *to*-axis. If column p is to be folded on top of column q the from-axis is p, the to-axis is q. A potential fold is entered into the *cfm* at (p,q). Since the fold has a "polarity", once the column intersection matrix has folding candidates it is no longer symmetric and it becomes a column folding matrix.

The first cycle case is indicated by the matrix:

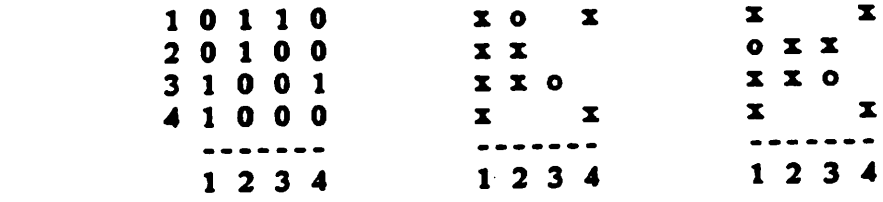


Figure 6.13: Case-1 (Simple) ζ—Cycle

The symbol *o* indicates a requested fold. Since it does not fall on an x it is necessarily disjoint. The *cfm* is no longer diagonally symmetric.

Remark:

(simple case)
Given folds at $CFM(i,j)$ and $CFM(m,n)$ if $CFM(m,j)$ and $CFM(i,n)$ are x then a simple case-1 ζ—cycle exists and the MLM cannot be folded.

(general case)
Or, in general, given folds at $CFM(i,j)$ and $CFM(m,n)$ for (i,j) and (m,n) chosen from the set of disjoint columns then a case-1 ζ—cycle exists if vertices $CFM(m,j)$ or $CFM(i,n)$ chosen from the set of intersecting columns induce a cycle with the vertices alternating *disjoint*, *intersecting*, *disjoint* and so forth.

An example of a complex case-1 cycle is shown if Figure 6.14.

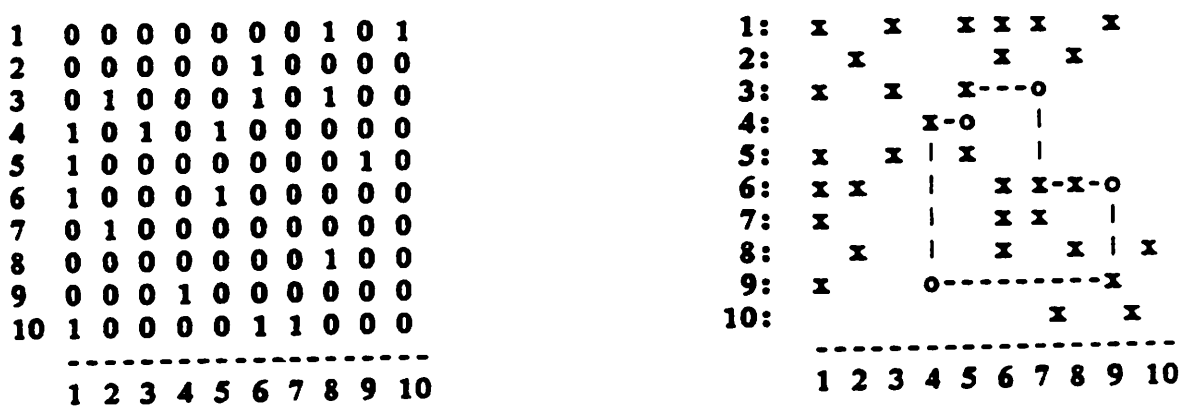


Figure 6.14: Case-1 (Complex) ζ—Cycle

Case 1 corresponds to the *χ—cycle* case of De Micheli [demi84]. In Figure 6.13c the top-bottom ordering of the requested folds of Figure 6.13b has been altered. Now no cycle exists and the fold is said to be *implementable* [egan82].

The case-2 ζ—cycle is shown in Figure 6.15:

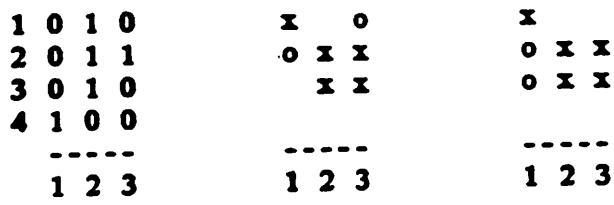


Figure 6.15: Cases 2 and 3 ζ—Cycle

Remark:

Given folds at *CFM(i,j)* and *CFM(m,n)* if $i = n$ and *CFM(m,j)* is *x* or $j = m$ and *CFM(i,n)* is *x* then a case-2 ζ—cycle exists and the folds cannot be implemented.

A case-2 cycle is a multiple folding request (*ie.* more than two logical columns are folded into a physical column). A case-2 cycle is detected in tandem with case-1 ζ—cycles. This type of cycle has one or zero undirected edges. Such a cycle cannot be broken by

reordering the fold, or by breaking up the multiple fold into several simpler folds. In fact, breaking up this fold into its components yields a case-3 ζ—cycle.

Figure 6.15c shows a case-3 ζ—cycle obtained by fracturing the fold *2—1—3* into its two parts: *2—1* and *3—1*.

Remark:

> Given folds at $CFM(i,j)$ and $CFM(m,n)$ if $i = m$ and $CFM(j,n)$ is x or $j = n$ and $CFM(i,m)$ is x then a case-3 ζ—cycle exists and the folds cannot be implemented.

Note that case-3 folds are parallel to a matrix axis. The presence of a case-3 cycle indicates that multiple folds emanate from or terminate in a particular folding candidate. Because case-3 cycles are *in-line* a special consideration must be made to detect them. Either a cycle-{*1 2*} or a cycle-*3* check is performed, but not both, so the order of the cycle-check algorithm is not increased.

The only way to break the case-2 and case-3 cycles is by removing a requested fold from the matrix. If the *2—1* fold were removed in the examples above both case-2 and case-3 cycles would disappear.

6.2.3. Bounds on CFM Construction and Cycle Checking Algorithm

The construction of the column intersection matrix requires $O(c^2 \times r)$ operations where c represents the number of columns and r represents the number of rows in the MLM. This is an upper bound. The algorithm is as follows: Compare each column C_i with every other column to check for intersection. The intersection check is performed by *anding* the row entries of the columns under test. If the sum of the bits of the row-wise *and* is non-zero then the columns in question intersect. The *anding* requires r operations. Of course, if it is performed serially, as soon as an intersection is found the process may be halted. Since the column compare procedure is commutative (checking column C_i against C_{i+1} is the same as checking C_{i+1} against C_i) $\frac{c^2}{2}$ tests must be performed. Thus the total

number of steps is $O(c^2 \times r)$ in the worst case.

The cycle-checking algorithm will now be shown to be $O(f^2)$ where f represents the number of trial folding candidates. The argument is as follows: Assume a set of folding candidates F. To detect cycles one must ascertain for each f_i in F whether it forms any cycles with any other element or elements in F. To do this an empty *link* list is created and some first element is inserted; the element is marked as *used*. Each remaining, unused element is then compared against the list. If the candidate adds two links to the elements in the list (corresponding to two x's between o's in the CFM) then a cycle exists and the procedure terminates. If the folding element adds no new cycles it is temporarily discarded. If the element adds one link it is added to the *new* list and marked. This process continues until all elements are marked or until all unmarked elements are compared against the link list. Each time a complete pass of the folding candidates finishes the link list becomes the new list and the new list is set to nil.

Since remaining unmarked elements must be checked against a list each time the list is updated. $O(f^2)$ operations are needed. At the end of each pass some number of candidates will potentially remain unfolded. These are the elements which were discarded earlier. For these unmarked elements the same process is required to check for cycles. It thus appears that the algorithm is $O(f^3)$ overall. This is, however, not the case, as is now shown. Suppose there are k partitions of folding candidates. A *partition* is a singly-linked set of folding candidates. Suppose, also, that a particular partition requires p passes or separate *new* lists to generate. Each new list represents a subcluster of linked candidates. Then the number of operations per partition is equal to

$$\sum_{i=1}^{i=p} n_i \times (n - \sum_{j=1}^{j=i} n_j) \qquad (6.1)$$

Where n is equal to f, the total number of folding candidates and n_i represents the number of candidates in link list i. The Σn_j term represents those candidates which have already been chosen and are therefore used on previous link lists. The expression can be rearranged:

$$n\sum_{i=1}^{i=p} n_i - \sum_{i=1}^{i=p}\sum_{j=1}^{j=i} n_i n_j \tag{6.2}$$

Clearly,

$$n^2 - \sum_{i=1}^{i=p}\sum_{j=1}^{j=i} n_i n_j \; < n^2 \tag{6.3}$$

because $\Sigma n_i \leq n$ by definition. The total number of operations is the sum over k partitions: $\Sigma n_i^{\,2}$. By the "triangle theorem":

$$\sum_{i=1}^{i=l} n_i^{\,2} \; < \left(\sum_{i=1}^{i=l} n_i\right)^2 \tag{6.4}$$

Thus an upper bound on the complexity of the cycle-checking algorithm is $O(f^2)$. It remains only to demonstrate the goodness of this bound. There are two extreme cases: either all candidates can be folded and there are no cycles or all candidates combine to form a single, complex cycle. In the former case there are f folding lists but only f checks are required of each list, since all candidates are distinct. Thus this case is $O(f^2)$. In the latter case the construction of the single folding list requires $O(f^2)$ operations but at the end of the process all elements have been marked, so the process terminates. In addition, a cycle has been found. For this latter case,

$$\sum_{i=1}^{i=f} (f - i) \tag{6.5}$$

operations are required, which is $O(f^2)$. Therefore both extreme cases exactly fit the bound and hence $O(f^2)$ represents a tight upper bound on the complexity of the cycle-checking algorithm.

6.2.4. Construction of the Folded Matrix

The final stage in TWIST is the construction of the output matrix. In the cycle-checking phase it was determined whether or not a solution existed: at this point an acceptable solution must be found.

A solution (in general there will be more than one) which satisfies the folding requests is constructed by building constraint matrices. One matrix is built for column constraints, another for row constraints. These constraint matrices are exactly analogous

to the constraint matrix used by MKMAT to construct an initial signal (row) ordering. For the folded matrix a slight elaboration of the constraint matrix is needed. Row folding imposes constraints on column ordering, column folding imposes constraints on row ordering. In addition in the column constraint matrix, for example, where two columns are to be folded into one, the constraints on both (unfolded) columns must be satisfied in order to properly place the single (folded) column. Similarly all the constraints on each of the rows that are to be folded together must be met simultaneously for the folded row to be properly placed. Folded rows and columns are placed by the constraint matrices as a group. The output construction procedure creates the folded matrix by referring to the constraint matrices. Where there is latitude in column or row placement, topmost and leftmost slots are chosen, respectively. This results in the now empty extra rows and columns being pushed to the edges of the matrix. In the final printing of the matrix these elements are trimmed away.

It was shown in the previous section that the cycle detection algorithm is $O(f^2)$ for f the number of trial candidates in the *cfm* and *rfm* or folding constraints in the *rom*. The constraint checking procedure is $O(n^2)$ where n is the number of rows or columns in the constraint matrix. Thus it can be seen that if the input matrix has many folds the constraint matrix yields a faster check on the consistency of the matrix. For the *cfm* and *rfm*, f will always be less than the number of columns or rows, by definition. In the *rom*, however, where f is the number of constraints it is quite possible that f exceeds the row count, since a single column may introduce many *rom* constraints. Therefore the constraint matrix algorithm is used as a pre-check for cycles in the *rom*. The constraint matrix procedure only asserts that a cycle must exist, it does not provide any information about the cycle or cycles. Therefore, once it is known that a cycle exists, recursive cycle-checking procedures are called to find the cycle's vertices. Such information is necessary in order to attempt to break the cycle.

6.3. Examples of Row and Column Folding

The folded matrix is built from the same set of characters as the input, unfolded matrix. For clarity in display, the uppercase characters {*S P O*} are employed to represent SERIES, PARALLEL, and OUTPUT devices, respectively, in the bottom element of a folded column. The fold locations are indicated by additional information displayed on the edge of the matrix. Figure 6.16 shows an unfolded, unconstrained multi-level matrix, part of an adder circuit, in both personality matrix and mask-level form. Mask-level generation is described in Chapter 7.

```
new 27 15

        pssssssssp.ssss
c1      ........s...s.s
c1*     ...........s.s.
18      ..s..........o.
20      ...s........o..
28      ......so.......
c2      ......s.....ss.
14      ....s......o...
16      .s............o
c3*     .ss.........s..
b0      ..ss....-.....-
cin     .......s..-....
b0*     .......s..-....
c2*     ........s..s..s
f0      o..............
1       po.............
2       p.o............
3       p..o...........
4       p...o..........
6       p....o.........
10      p.....o........
11      p..............
12      p..............
a0      .s.sp.....s....
15      .....s..o......
7       .....s...o.....
c3      ......s.s..s...
a0*     .......s.s.....
```

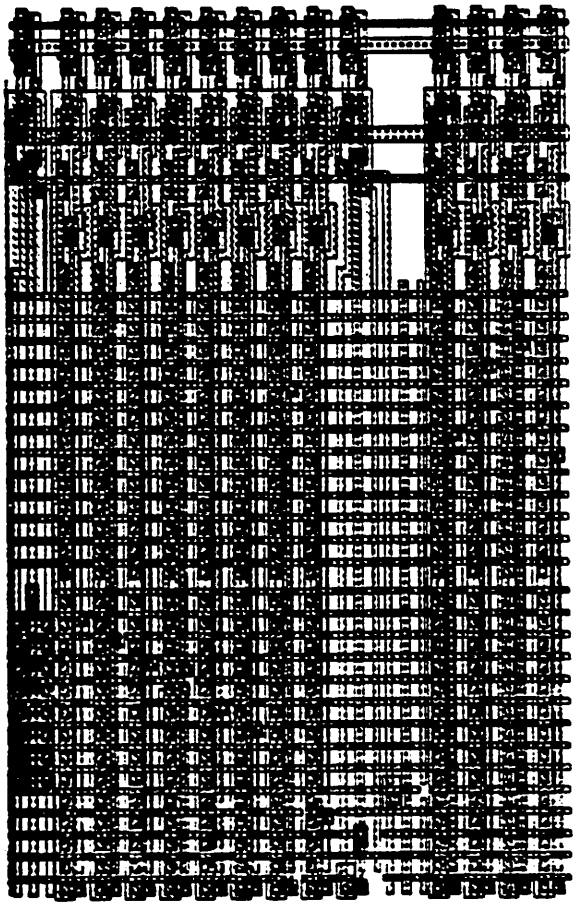


Figure 6.16: Unfolded Unconstrained Matrix
(a) Personality Matrix (b) Mask-Level Layout

This MLM has 27 rows and 15 columns. The term "unconstrained" means that the designer has not imposed additional constraints on the routing of external signal lines. One would expect a better compaction with an unconstrained matrix compared to the same example, constrained: this is illustrated in the examples that follow. Figure 6.17 shows column folding and then row folding for the unconstrained input case. This example was produced by letting TWIST first compact all columns and then attempt row compaction.

```
folded 23 11

                 sssssp.ssss
c1*              .ss....s...      cin(8)
c1               s..ss..os..      28(8)
c2*              s.s.s.s...       b0*(6)
14               ..o......s.
c2               .s.s....s..
18               .o........s
c3               s.s.....s..
7                .S....o....
15               oS.........
a0               S.S..s...p.
c3*              S..s......s
b0               ..S.....s
20               ..So.......
12               ...P..ss...      a0*(7)
11               ...P.......
10               ...P....o..
6                .O.P.......
4                ...P.....o.
3                ..OP.......
2                ...P.......o
1                O..P.......
f0               ...O.......
16               S...o......
                 sssp
                 (10) (8) (10) (14)

Current column folds/flips:
  (14,6) (13,1) (9,2) (12,4)
Current row folds:
  (c1*,cin) (c1,28) (12,a0*) (c2*,b0*)
Signals brought out to the right:
   cin b0* a0*
Signals brought out to the left:
   c1 c1* c2*
```

Figure 6.17: Column — Row Folded Unconstrained Matrix

The example shows four column folds and four row folds. As a result of column folding buffers appear along the bottom edge of the matrix. The signals to the right of the matrix indicate those brought out on the righthand side. The numbers in parentheses indicate on what column the folded row signal begins or on what row the folded column sig-

nal begins for rows and columns, respectively. The upper lefthand corner is taken to be location *(1,1)*. State information is printed out and the folded column indices and folded signal name pairs are identified. In addition, signals which are available only on the right or left side of the module are listed. External signals not appearing on the list are available on both module boundaries, for example signal *f0*.

Figure 6.18 shows the matrix of Figure 6.16 this time with rows folded first and then columns.

```
folded 17 14

                ssssssp.sssssssp
c1*             sso.....s.....          28(3)
c1              ...ss..so....p          10(9)
16              ....o.......s.
c2              s..s....s.....
18              o..........s..
c3*             ...s.......ss.
20              ...o......s...
cin             ..sO.........p          6(4)
c2*             .s..s..s.o...p          4(10)
b0*             ..s..~....o..p          3(11)
b0              ......~..~ss..
a0*             ..s...s....o.p          2(12)
c3              .s.....ss...op          1(13)
7               ...S..o......p          11(14)
15              ...S...o......
a0              .....s...ps.sp          12(14)
14              .o.......s...o          f0(14)
                    s
                (8)

Current column folds/flips:
  (13,6)
Current row folds:
  (a0,12) (14,f0) (7,11) (c3,1) (a0*,2)
  (b0*,3) (c2*,4) (cin,6) (c1,10) (c1*,28)
Signals brought out to the right:
  f0
Signals brought out to the left:
  c1 c1* cin b0* c2* a0 c3 a0*
```

Figure 6.18: Row — Column Folded Unconstrained Matrix

This result shows a single column fold and ten row folds. In this case all inputs are available on the left side of the array while *f0*, the output, is available on the right side. The area of this module is slightly less than its column/row folded counterpart and the aspect ratio is closer to 1:1 which might mean that this module fits better in the context of an overall chip design. The designer can, of course, adjust the aspect ratio by manually issuing row and column folding commands in the interactive mode of TWIST.

The next series of three examples shows the same matrix, but this time with constraints placed on several of the external signals. Note that signal *c3* is being used as a bus-through connection.

```
new 27 15

                          pssssssssp.ssss
c1                        ........s...s.s
c1*                       ...........s.s.
18                        ..s...........o.
20                        ...s.........o..
28                        .....so.........
c2                        ......s.....ss.
14                        ....s......o....
16                        .s.............o
c3*                       .ss..........s..
b0                        ..ss.....~.....
cin                       .......s..~.....
b0*                       .......s..~.....
c2*                       .........s..s..s
f0                        o...............
1                         po..............
2                         p.o.............
3                         p..o............
4                         p...o...........
6                         p....o..........
10                        p.....o.........
11                        p...............
12                        p...............
a0                        .s.sp.....s.....
15                        .....s..o.......
7                         .....s...o......
c3                        ......s.s..s....
a0*                       .......s.s......

right c1 c1* c3* c3 cin ;
left f0 b0 b0* c3 ;
```

Figure 6.19: Unfolded Constrained Matrix

The external folding constraints have been entered by the designer and included in the input file to TWIST.

Figure 6.20 shows the constrained matrix after column-then-row folding.

```
folded 26 11

                ssssssp.ssss
28              ........os..
c1*             s..s........
c1              .ss.s.......
c2*             sss...s...      cin(8)
b0*             ......_.s...
14              o.........s.
c2              ...ss...s..
18              ...o......s
c3              ss.......s..
7               ...S..o.....
15              .o.S........
a0              SS...s...p.
c3*             .S..s_____s
b0              S....._..._s
20              S...o.......
a0*             ......ss....
12              ....P.......
11              ....P.......
10              ....P...o...
6               ...OP.......
4               ....P....o.
3               O...P.......
2               ....P.....o
1               .O..P.......
f0              ....O.......
16              .So.........
                ss sp
                (12) (12) (10) (17)

Current column folds/flips:
  (14,6) (13,1) (9,2) (12,4)
Current row folds:
  (c2*,cin)
Signals brought out to the right:
   c1 c1* c3* cin c3
Signals brought out to the left:
   b0 b0* c2* f0 c3
```

Figure 6.20: Column — Row Folded Constrained Matrix

In this case it was still possible to achieve four column folds, however only a single row fold was possible. The signal list at the end of the matrix structure recapitulates the

designer's request and adds any additional signals that are available on only one edge of the matrix. Signal *c3* appears on both the left and right edge, as the designer requested.

Figure 6.21 shows the constrained example of Figure 6.19 this time with row folding first and then column folding.

folded 20 14

```
                ssssp.sssssssp
28              .......os.....
c1*             .s.s..........
c1              s.s...s.......
c2*             s..s..s..o...p      4(10)
16              o...........s.
c2              Oss.....s....p      6(1)
18              .o.........s..
cin             .......s......
c3*             ..s........ss.
20              ..o.......s...
b0*             ....~..s..o..p      3(11)
b0              .....~...~ss..
a0*             .....s.s...o.p      2(12)
c3              ...s..s.s.....
7               S....o.......p      11(14)
15              S.....o.......
a0              ....s....ps.sp      12(14)
10              ........o....p
f0              ............o
14              ...o.....s..op      1(13)
                s
                (6)
```

Current column folds/flips:
(15,6)
Current row folds:
(a0,12) (7,11) (14,1) (a0*,2) (b0*,3)
(c2*,4) (c2,6)
Signals brought out to the right:
c1 c1* c3* cin c3
Signals brought out to the left:
c2 b0 b0* c2* f0 a0 c3 a0*

Figure 6.21: Row — Column Folded Constrained Matrix

In this case only a single column fold was possible after seven row folds were made. This

case may be compared to the unconstrained row-then-column folded example. For the unconstrained case four column folds and ten row folds were found; as a result of constraints there were three fewer column and three fewer row folds.

6.4. Summary

In this chapter an algorithm for topological compaction of a multi-level matrix was described. In the first part of the chapter a technique for translating a netlist into a connectivity matrix was examined. The MKMAT program accomplishes this translation by constructing a set of constraint matrices which indicate in what order rows and columns should be placed in the MLM. The TWIST program, examined in the latter part of the chapter, is employed to fold the MLM. TWIST can function as part of the MAMBO package or interactively. It employs heuristics to perform simple column and multiple row folding. TWIST also uses constraint matrices and cycle-checking algorithms to guarantee a folding selection is implementable. The cycle-checking algorithm is $O(f^2)$, where f is the number of folding candidates. The compaction program allows both constrained and unconstrained folding.

In the next chapter methods for the automated generation of the topologically compacted matrices are described. The physical design considerations involved in mask layout generation are implemented in two programs which form the final stages of the MAMBO system.

CHAPTER 7

Physical Design: Comparison of Layout Tiling Methods

In this chapter the physical design aspects of MAMBO are described. These design considerations are implemented in two programs used by MAMBO. The layout method employed by the present synthesis package is contrasted with methods currently used in the semi-automatic and automatic generation of asynchronous and synchronous circuits. Three well-known styles of layout: Weinberger Arrays, Gate Matrix, and Storage/Logic Arrays, are examined. These styles cover the range of present tiled techniques. The context-based tiled approach for structured layout of dynamic circuits such as Domino and NORA, which is used in MAMBO, is then presented. The context-based translator is implemented in tool TINKER, while the tiling into mask-level geometries is the domain of TAILOR.

7.1. Distinction Between Routed and Tiled Methods

There are two distinct methods of automated circuit generation. The first method keeps the active circuit area separate from the interconnection area. This style is typified by the gate array and standard cell [souk81] approaches. In the gate array approach compact, general purpose, uncommitted logic blocks of fixed pitch are placed in a regular array on chip. A programming step, involving the modification of a small number of masks, commits the function of each circuit block. In the standard cell scheme cells of fixed pitch, with predetermined inputs and outputs, are chosen from a library of previously designed modules. In both these approaches a channel router is required to connect the proper inputs and outputs between the logic blocks. Hence, these methods are termed *routed*.

The advantage of this approach lies in its generality. A single gate array can be made to perform a myriad of different functions- depending only on mask steps to program the

blocks and channel route between them. The disadvantage of such approaches lies chiefly in their inefficient use of area due to the need for preassigned channels and to the "uncommittedness" of circuit function. These approaches find their use principally in full gate array or standard cell chips: they are only infrequently mixed with custom logic on the same chip.

The second set of methods presented here falls under the heading of *tiled* methods. These methods generate circuits by gluing together cells from a cell library. *Cells are routed by abutment*. Therefore there is no need for a routing tool. Because no routing is required, it is not necessary to allocate space for routing channels. This fact, coupled with a higher degree of customization (all mask layers are generated), results in a more area-efficient layout. Within the tiled category there are methods which admit of greater optimization and methods which are better suited to particular technologies. One drawback of tiled approaches is that the cells that compose the layout may, themselves, be sparse. This is because the routing is now internal to the cell; if no optimization algorithm is employed, tiling may also be area-inefficient.

The distinction between the two methods is important. Automated generation tools which deal which synthesis of active area separately from routing can lose in routing area whatever gain they may make from a tight cell layout. The need to handle interconnectivity together with active area is especially important in the creation of multi-level logic. In two-level logic, for example as realized by a PLA, large matrices of active devices or placement sites are created. A single matrix corresponds to one level of logic. Hence, for the two-level PLA, the interconnection problem is almost trivial. It involves the connection of two regular arrays plus connections to input and output buffers. In multi-level logic, on the other hand, the number of levels is greater and the amount of active area contained in each level is less. Therefore the percentage of area taken up by routing increases, so a compact interconnection scheme is a requirement.

7.2. Tiled Methods

7.2.1. Weinberger Arrays

Weinberger arrays were first proposed in 1967 [wein67]. At that time they were used to fabricate PMOS gates in metal gate technology. A Weinberger array is one dimensional. Figure 7.1 shows a layout of a simple Weinberger array.

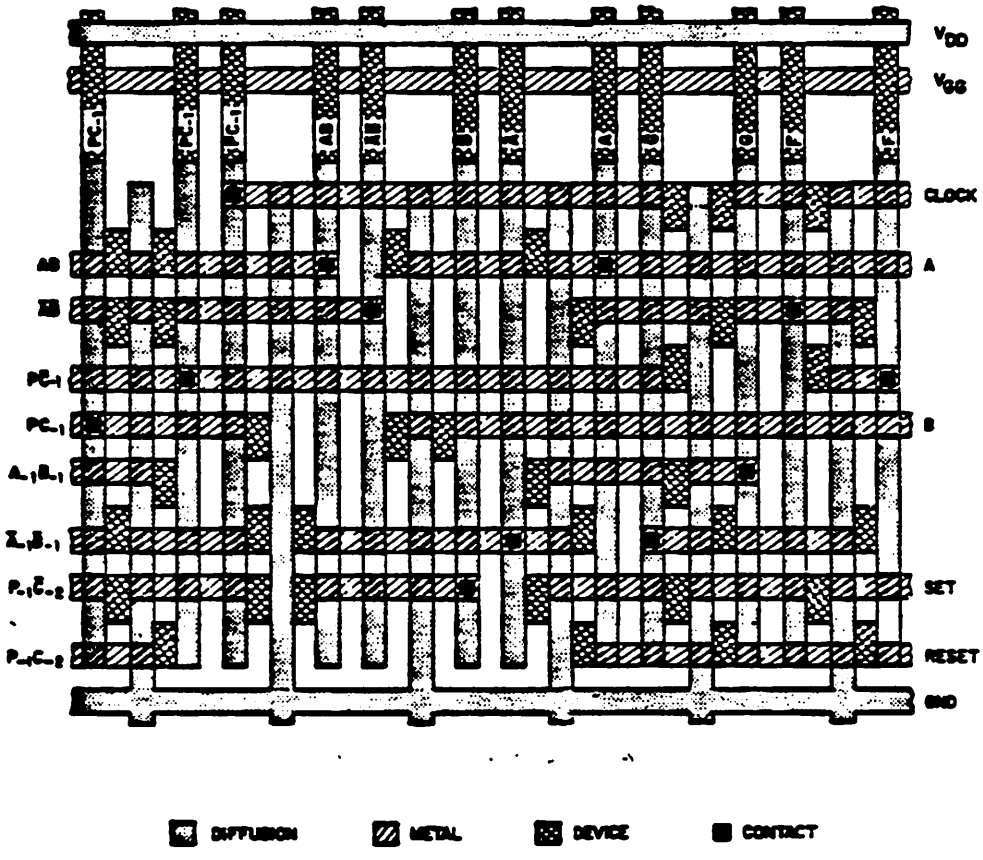


Figure 7.1: Weinberger Array Layout

Since PMOS gates were used in the original realization, the tile of choice was the NAND gate. To first order, switching speed is constant with increased fan-in for a PMOS NAND gate. Thus it is possible to build large fan-in gates. Increased fan-in seriously degrades the switching time of PMOS NOR gates. NAND-only circuits are logically complete. Since high fan-in gates can be constructed, synthesis programs need not be concerned with parti-

tioning large gates into smaller ones and the consequent speed tradeoffs between a single, large gate and a series of smaller gates. Thus module generators targeted to this style are straightforward. The MACPITTS/LBS [sisk82] system in use at MIT Lincoln Laboratories is an example of a datapath and control generation program using this approach.

The disadvantage of such a scheme is that internally the array may be quite sparse. To see why, note that in the Weinberger style, where power and ground busses are at fixed pitch, one dimension of the array is constrained by that gate with the greatest combination of fan-in and route-through. For Weinberger arrays composed of hundreds of NANDs it is likely that there will be a wide variation in gate fan-in.

It is possible to optimize the layout of Weinberger arrays and so compact them somewhat. Optimization can be accomplished on a purely topological level by rearranging the ordering of the NAND gates. Since both input and output signal lines must traverse the array and can be intermixed, it is possible through gate reordering to have two or more distinct signals occupy a single physical track. This method of rearrangement is similar to the permutation of product terms in a PLA so that multiple inputs and outputs may share a single physical input or output term.

Optimization algorithms to obtain the best packing on a one-dimensional interval have been intensively investigated [ohts79] [asan82]. In conjunction with such optimization programs Weinberger layout tiles can produce area-efficient results for single MOS (*eg.* NMOS, PMOS) technologies. Such technologies are favored because their simple tile abutment allows routing in the array. By contrast, CMOS technology presents difficulties in Weinberger layout routing since each signal must drive both a PMOS and an NMOS device. Optimization in this case also is more involved. One method suitable for tile layout of CMOS logic is gate matrix.

7.2.2. Gate Matrix

The gate matrix technique was first proposed in 1981 [lope81]. This approach came about as an outgrowth of a system to symbolically represent the topology of matrix circuits. It was used in the design of the BELLMAC-32 series microprocessors. Gate matrices have been used to implement both static and dynamic CMOS circuits. While this technique can also be used with single MOS technologies, it is clearly well-suited for CMOS.

Static CMOS gate matrix favors the implementation of low fan-in NAND gates. Figure 7.2 contains a schematic representation of a 2-input NAND gate and a 2-input NOR gate. These two gates represent the most commonly used CMOS gates.

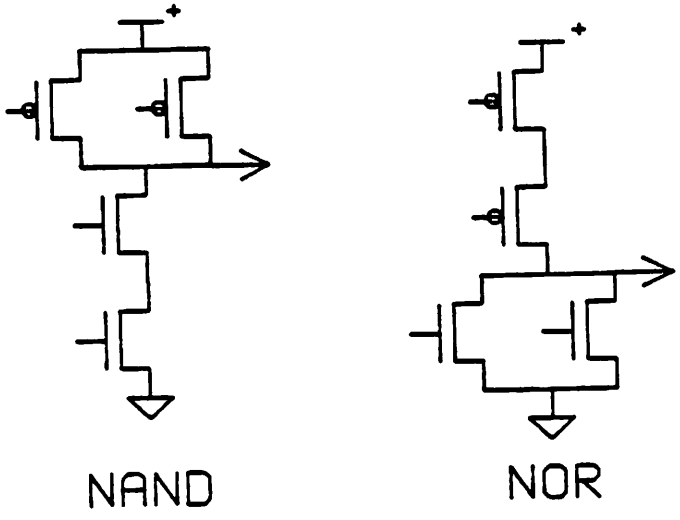


Figure 7.2: Static CMOS NAND and NOR Functions

Because of mobility differences in n- and p-type devices, the p-channel device must be two to three times wider than its n-channel counterpart. In the construction of a static NAND gate the effect of ratioing due to mobility tends to be canceled by the requirement of ratioing to compensate for the series arrangement of the n-channel devices. If one assumes a mobility difference of two to one between n- and p-channel devices, then for a two-input NAND gate the two effects exactly cancel one another and all devices may have minimum width. Not only does this reduce circuit area, it also reduces the capacitance on

the gate output. This means that the NAND gate exhibits a smaller gate delay. On the other hand, for a CMOS NOR gate the two effects are multiplicative. Thus, if one draws a minimum width n-channel device, the p-channel device must be twice as wide as the n-device for mobility and twice as wide again because the p-devices are now in series. The additional capacitance of the p-channel source node that is shared by the output of the gate slows down the circuit. Figure 7.3 shows a portion of a BELLMAC-32 gate matrix which is entirely NAND in structure.

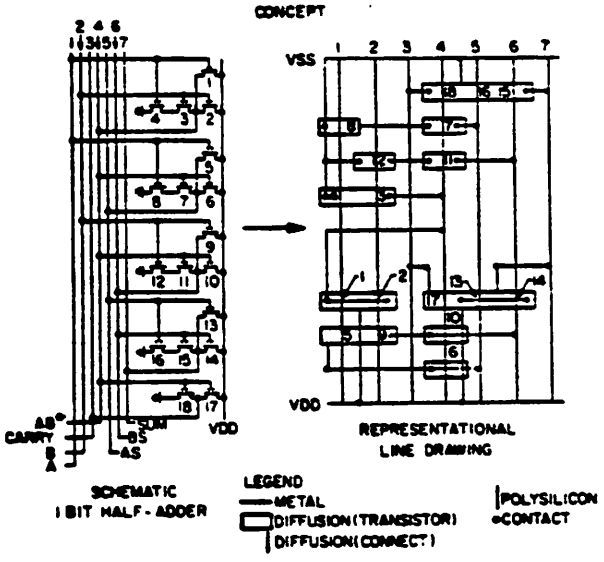


Figure 7.3: Gate matrix from BELLMAC-32

Topological optimization of these circuits is more difficult due to the branching structure of the individual gates. Reordering techniques are not usually applied to such circuits. The automation of gate matrix layout is a topic of current research [kang83b] and there are a number researchers developing automated tools which use heuristic one-dimensional methods to achieve compact layouts [wing85] [ishi83]. These results indicate that optimization algorithms can greatly reduce matrix area. The results also show that hand optimization is still better than machine compaction so that more work remains to be done in this field. The optimization algorithm described in [ishi83] solves the NP-complete problem heuristically in $O(n^3)$ steps, where n corresponds to the number of distinct gates

in the matrix.

While optimization is more difficult, gate matrix is nonetheless better suited to the CMOS technology because it has a smaller "granularity". That is, instead of constructing a tile which forms a whole gate, the gate matrix approach places separate rectangles of polysilicon, diffusion, and metal. Thus, gate matrix is more flexible. This can lead to more area-efficient circuits but is also responsible, in part, for the difficulty in circuit compaction.

7.2.3. Storage/Logic Arrays (SLAs)

SLAs were first described in 1975 [pati75] and later extended by Patil and Welch [pati79]. They are regular structures derived from PLAs. Their chief advantage over PLAs is their ability to embed storage functions within the combinational logic array.

Typically a designer will construct an SLA from a tile set designed for a specific technology. SLAs have been built in I^2L, NMOS and CMOS. The CMOS SLAs appear most promising. CMOS SLA elements include a single inverter, a double inverter, a simple NAND gate latch, and a pass transistor [smit82]. Each of these cells is represented by a single character to the designer. In much the same way as the personality matrix of a PLA can be manipulated symbolically, the characters representing the SLA elements can be arranged. Figure 7.4 shows the symbolic layout of an adder/subtractor in CMOS.

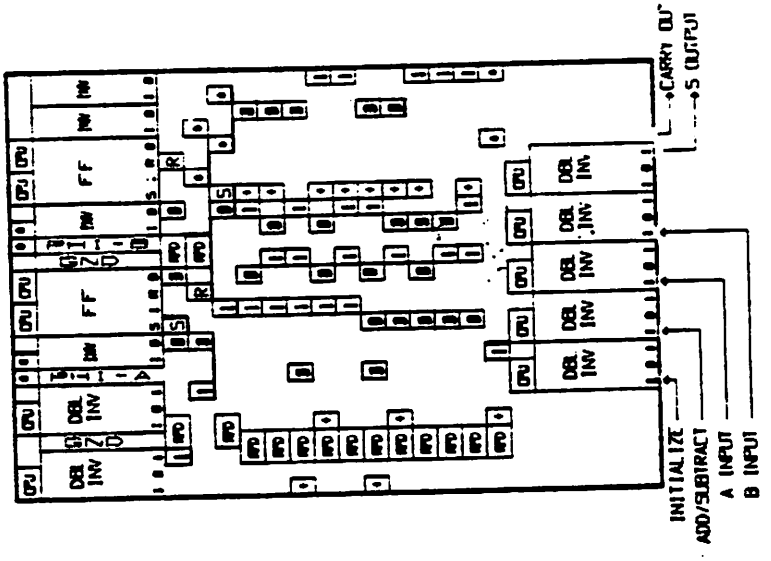


Figure 7.4: Symbolic Representation of USCM Adder/Subtractor

Because the layout is very similar to a PLA, it is possible to employ similar folding and splitting algorithms to optimize SLA area. Again because the SLA, like the PLA, is regular, such structures are easy to machine generate.

While SLAs have proven useful in static NMOS and CMOS circuits [smit82], they are more difficult to construct for dynamic circuits. Because of the embedded logic concept used in SLAs an additional clock would be required for each level of inverter logic. The extra space needed to route these signals would override any gain in other parts of the circuit. Another drawback of SLAs is that the minimum cell pitch is constrained by the largest cell in the tile set. Since some tiles are quite complex (*ie.* a D flip-flop) area utilization can suffer. An alternative approach is to allow cells which are multiples of some fundamental dimension. This was done in the CMOS tile case. However, when this is done, special "blank" cells and "bender" cells must be created to route signals. The result is that SLAs in CMOS are best suited to static applications and to designs where the majority of logic is combinational. Notice that the USCM example shown contains a core of compact combinational cells (represented by *1* and *0* symbols) surrounded by a periphery of latches

and flip-flops. The "embeddedness" of the SLA has been lost. The structure begins to resemble a PLA with external synchronizing logic.

A further disadvantage of the CMOS SLA approach is the necessity for a process that supports Schottky diodes. The AND and OR planes of the SLA are compact because they utilize these diodes to create a "folded-plane" structure. Diagrams of AND and OR plane construction are shown in Figure 7.5.

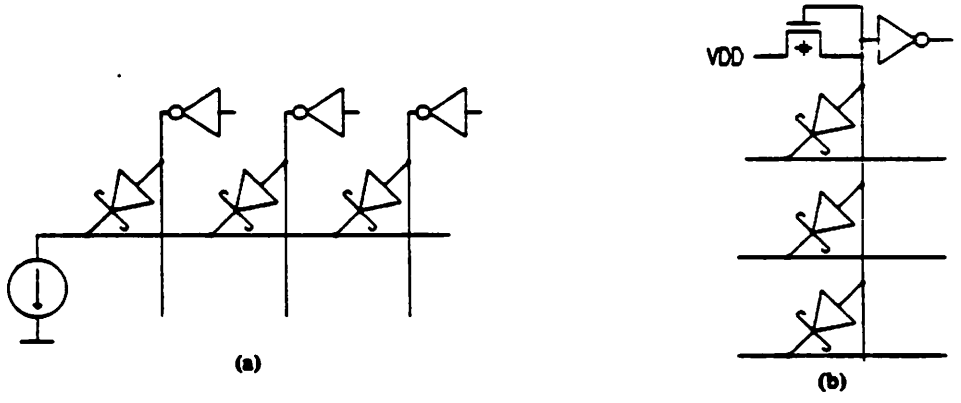


Figure 7.5: CMOS AND Plane(a) and OR Plane(b)

The conclusion is that SLAs offer a structured design approach quite similar to PLAs. SLAs can implement multi-level and dynamic logic, but best success appears to have been obtained with single AND and OR plane implementations in static designs. CMOS SLA designs do not appear particularly area-efficient. While designs based on the SLA have been submitted for fabrication, no test results have yet been published in the literature.

7.3. A Structure for the Layout of Complex Domino Cells

It has been shown that the Weinberger array layout technique is a good match for single MOS technologies and that the gate matrix technique is suitable for layout tiling of static and dynamic CMOS circuits, because of its greater flexibility. SLAs offer a means of combining combinational functions with storage but appear to have drawbacks in fabrication and area efficiency.

One advantage the Domino configuration has over static CMOS approaches is a smaller gate count and hence, a more compact layout. To take advantage of Domino-style circuitry individual gate clusters should be complex. In other words, since the overhead of clocking gates and output inverter is constant, greatest benefit is realized when complex AND/OR functions are packed into a single gate. None of the layout tiled structures reviewed here is ideally suited for Domino-style CMOS logic layout and a new structure is needed. This structure should have the following characteristics:

- Ability to take advantage of complex AND/OR gates
- Suitable for dynamic logic
- Able to handle multi-level logic without an extra routing penalty
- Allow mixing of static and dynamic functions and static latches so that the module may be used as part of a finite-state machine
- Can be generated and optimized by a synthesis tool

A hybrid structure having characteristics of both gate matrix and Weinberger arrays fits these requirements. The ALU portion of the BELLMAC-32 was designed employing gate matrix and using dynamic Domino CMOS logic. A section of the ALU block appears in Figure 7.6 (from [law83]).

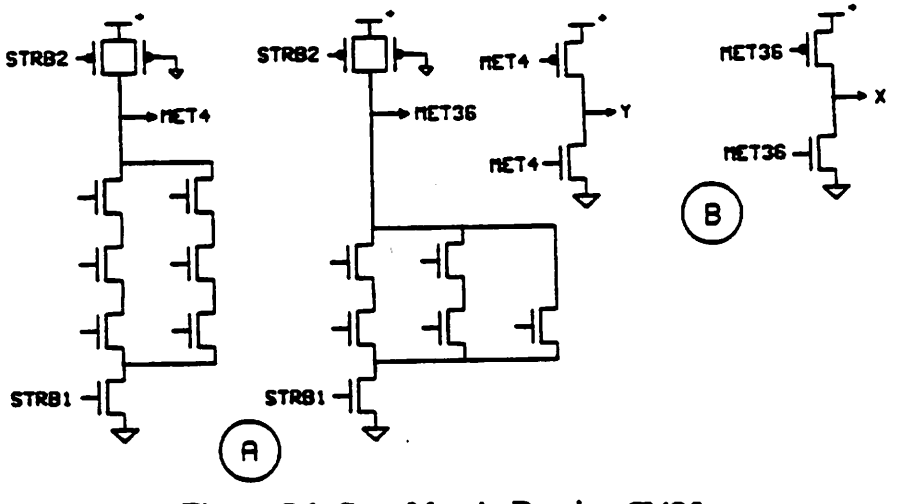


Figure 7.6: Gate Matrix Domino CMOS

The Domino gates in Figure 7.6a are complex: they combine two or more Boolean functions

but have a single output. The dual p-channel devices are employed to correct for the charge redistribution problem in dynamic circuits. The gates shown in Figure 7.6b are static inverters. It is clear that gate matrix allows intermixing of static and dynamic styles. The advantage of Weinberger arrays is that they are easy to layout, since there are only two basic cell types (NAND and NOR). They are also easy to optimize since they have a simple structure. While Weinberger arrays are routinely machine generated, gate matrix layout has traditionally been performed by a layout engineer at a text terminal. The designer manipulates symbols which correspond to layout geometry. The current gate matrix cell set contains about 20 symbols.

The new layout structure resembles gate matrix but extends it in an important way. Instead of hand-entry of symbols which map one-to-one into layout, the layout is generated from the higher-level Boolean description. This extension is important, not only because it frees the designer of hand-entry, but also because computer-aided algorithms can be employed to optimize the circuit layout. In particular, it is possible to generate more complex individual gates and increase area efficiency and circuit speed through the methods used by MOSMESH and TWIST and mentioned in Chapters 5 and 6. The new tiled approach is examined in detail in the next section.

7.4. Context-Based Tiling— TINKER

The tool TINKER fits in the MAMBO pipeline between the topological compaction of a logic array and its mask-level generation. It generates an intermediate representation of the logic array. This intermediate form contains more information than the personality matrix but not as much as the actual mask-level layout. It may reflect electrical design considerations and can contain information needed in the layout of folded rows and columns. While TWIST is technology independent since it deals only with connectivity, TINKER is certainly technology dependent and may be process dependent as well if it is used to resize devices based on electrical characteristics.

The personality matrix needed for topological operations is quite abstract. The input matrix to TWIST is composed of characters from the set {*s p o . ~*}. The output matrix characters include, in addition, the characters {*S P O*} to denote gate columns which have been flipped. The output matrix from TINKER, by comparison, contains characters from a much richer set of symbols. Currently there are about 60 different symbols defined in the TINKER/TAILOR library. The larger cell set is used to define the personality matrix cells in different contexts.

Two reasons to define a multiplicity of layout cells to handle a single cell are: 1) electrical resizing of a device in context and 2) need to truncate or stretch a cell as a result of a local anomaly (such as a break or fold) in the structured array. Electrical resizing of devices is generally most important in ratioed logic, such as static logic where device size relates directly to speed. For the dynamic circuits used in MAMBO cells with differing device lengths and widths are not currently employed. However, stretched and truncated cells are used.

7.4.1. Definition of the CONTEXT File

The notion of context-based tiling allows the construction of two separate tools: one which considers only topological aspects of an array and another which can do a simple one-for-one translation of a character matrix into mask geometries. The translation from a personality matrix to a more detailed, electrical matrix is accomplished by a user-supplied *context* file. A fragment of such a file is reproduced in Figure 7.7.

```
((symbol s) (shift s) (boundary )
   ((left sSpPoO~ ) (below sSpPoO~ .) (right sSpPoO~ ) (above sSpPoO~ .) r )
   ((left sSpPoO~ ) (below ;) (right .;) (above ;) m )
   ((left sSpPoO~ ) (below sSpPoO~ .) (right .;) (above ;) l )
   ((left .;) (below ;) (right sSpPoO~ ) (above ;) t )
   --
)

((symbol o) (shift s)
   ((left sSpPoO~ ) (below sSpPoO~ .) (right sSpPoO~ ) (above sSpPoO~ .) u )
   ((left sSpPoO~ ) (below sSpPoO~ .) (right .;) (above sSpPoO~ .) d )
   ((left .;) (below sSpPoO~ .) (right sSpPoO~ ) (above ) b )
   ((left .;) (below sSpPoO~ .) (right sSpPoO~ ) (above sSpPoO~ .) b )
   ((left .;) (below ;) (right sSpPoO~ ) (above ;) e )
   --
)

((symbol o) (shift p) (adjacent )
   ((left sSpPoO~ ) (below sSpPoO~ .) (right sSpPoO~ ) (above sSpPoO~ .) U )
   ((left .;) (below sSpPoO~ .) (right sSpPoO~ ) (above .) b )
   ((left sSpPoO~ ) (below .;) (right .;) (above .;) c )
   ((left .;) (below sSpPoO~ .) (right .;) (above sSpPoO~ .) K )
   --
)

((symbol .) (shift s) (boundary )
   ((left sSpPoO~ ) (below sSpPoO~ .) (right sSpPoO~ ) (above sSpPoO~ .) + )
   ((left .;) (below sSpPoO~ .) (right .;) (above sSpPoO~ .) | )
   ((left .;) (below ;) (right .;) (above sSpPoO~ .) - )
   ((left .) (below ) (right .) (above ) . )
   --
)
```

Figure 7.7: Fragment of TINKER Context File

Each cell is assumed to be rectangular and thus has up to four neighbors. (It may have fewer if it is on the edge of the tile matrix.) The four edges of the cell are named *left*, *right*, *above*, and *below* as shown in Figure 7.8.

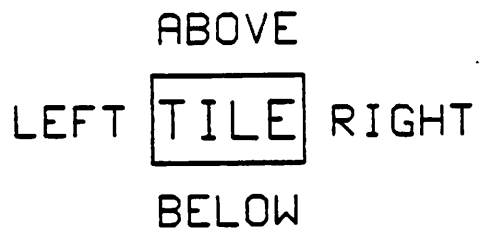


Figure 7.8: The Four Borders of a Tile

For each symbol appearing in the personality matrix there is an associated set of rules which specify which electrical tile implements that particular personality tile. Each symbol and a shift character are defined in the context file. Thus there should be a line of the form:

((symbol *personality_character*) (shift *column_header*) [(boundary) (adjacent)]

Each personality character is defined for each possible gate type. At present the two possible *column_headers* are *s* and *p*. In addition there may be two optional arguments *boundary* and *adjacent*. *Boundary* applies in the horizontal direction (the direction of signal flow) and specifies that signal lines are to be brought out to the module boundary. This option is important in the routing of external signals. Normally mask geometries carrying signal information are suppressed when a "blank" cell, a cell which has no active devices, is encountered. External signals, however, must bus through such cells and be available at the module boundary for interconnection. The *adjacent* option applies only in the vertical direction (the gate orientation). Normally TINKER searches the entire extent of a logical gate (a gate may be only a fraction of an entire column if that column has been folded). With the *adjacent* argument TINKER will only look at the cells immediately above and below the current tile to determine if a particular rule is satisfied. These options apply on a per *personality_character* basis. In the future, the context file format may be generalized to support the *adjacent* flag on a tile character basis. Also, at present, the search depth is a binary flag. Future versions might allow a variable search depth limit.

For each personality character, for each column header type, there is a rule set. The rule set has the form:

((left *char_set*) (below *char_set*) (right *char_set*) (above *char_set*) *tile_char*)

This is taken to mean:

> *Relative to the current matrix character*, **if** *a character on the left is in the left set and a character on the right is in the right set and a character above is in the above set and a character below is in the below set* **then** *select the target character*.

There is a special character for "edge" and if any set is null the predicate will never be selected. The edge character is ";" and must not be one of the usual personality matrix characters. When the edge character is a member of a border set it is taken to mean:

> **If** *the border matrix character is a member of the specified set or if there is no border character in this direction because the current matrix character is on a logical signal or gate edge* **then** *declare the border character to match the conditions*.

It is often possible to come up with a set of orthogonal rules. If this is the case the order of rule evaluation will not matter. However, it is easier to express rules in the form of:

> **If** *expression* **then** *expression* **else if** *expression* **then** *expression* ...

Rules are evaluated in the order they are entered into the context file by the designer. As soon as a match fails on any component of a rule evaluation the entire rule fails.

If a context arises which the designer has not anticipated then no rule will match. In this case the tiler will return "?" for the tile character. By turning on the *—debug* flag to TINKER the designer can trace the execution of each rule for the offending context.

The simplest rule set consists of a single rule which matches everything. In this case all edges contain the full personality character set. Such a rule might be used if a cell is completely context-independent. On the other end of the spectrum there may be a very large number of rules to match special cases. In practice about 10 to 15 rules per

personality matrix cell is found to be about average.

Figure 7.9 shows a personality matrix after compaction by the TWIST topological folding tool.

```
                sssssp.sssss
c1*             .s..s..os...    28(8)
c2              .ss.....s...
18              .o.........s
c1              s.ss........
c3              s...s...s...
7               .S....o.....
15              oS..........
c2*             s..ss.......
c3*             S.s........s
20              ..o.......s.
a0              S....~...~s.
12              ..P...~s....    a0*(7)
11              ..P.........
10              ..P.....o...
6               .OP.........
4               ..P......o..
3               ..P.......o.
2               ..P........o
1               O.P..s.s....    b0*(6)
f0              ..O...s..pss    b0(7)
16              S..o...s.~...   cin(8)
14              ....o....~..
                ssp
                (9) (6) (12)
```

Figure 7.9: Personality Matrix after folding by TWIST

After translation by TINKER the electrical matrix shown in Figure 7.10 is obtained.

```
ssssssp.sssss
+r++l||br+++
+rr+++++r+++
+b+++++++++r
r+rr++++++++
r-++r+++r+++
+T++++d+++++
bR++++++++++
m++rr+++++++
T+r++++++++r
++b+++++++r+
R+-++¯+++¯r+
++R|||¯r++++
++L+++++++++
++L+++++d+++
+bL+++++++++
++L++++++c++
++L++++++d+
++L+++++++d
b+L||r+r++++
++K|||r++Rrr
R++d|||r++++
+++-e----¯--
ssp
```

Figure 7.10: Electrical Matrix of Figure 7.9 after TINKER

The supplementary information about where row and column folds occur is now contained within the tile matrix. The "+" indicates placement sites occupied by interconnection area only. This file forms the input for the final tool in the MAMBO pipeline. This is the layout generation tool and is presented in the next section.

7.5. Mask-Level Layout Generation— TAILOR

The last stage in the MAMBO pipeline involves the translation of the electrical matrix produced by TINKER into a mask-level representation. While TINKER may need to be process dependent to express electrical considerations it is possible to make the back-end tiler completely technology and process independent. TAILOR need have no knowledge of the process that the array logic is to be fabricated in. The designer simply constructs a tile library in the mask-level language of his choice. The language used for the examples

presented here is CIF2.0 [mead80]. The circuit designer must also provide a *symbol* file which contains bounding box information about each tile. Also contained in this file is the binding between the tile's library name and its one character representation in the electrical matrix. All that TAILOR need do is refer to the information in the *symbol* file, look up each tile in the tile library, and "stitch" it together with the other tiles, to form the final matrix. In fact, TAILOR might be made independent of the geometric layout language, except that in the process of stitching tiles together it must produce a few constructs in the layout syntax. The language-dependent parts of TAILOR are restricted to a few procedures.

Since the designer specifies tile bounding box data it is possible to have cells abut, overlap, or have one tile wholly contained within another. For most structured layout applications, however, it is often best to enforce strict tile abutment and disallow overlaps. This rule may be enforced by giving the proper options to TAILOR. Two options, *—constant_width* and *—constant_height* tell TAILOR that all cells in each row and column are expected to have the same dimension. TAILOR thus allows enough space for the largest tiles and assumes that all tiles abut. TAILOR does not understand design rules and does not check for design rule violations. It is the tile designer's responsibility to enter correct tiles into the tile library. Tiles are design rule checked upon entry into the library. This is more economical than checking them each time a matrix is assembled. Of course, the tiles must not violate design rules at their boundaries with neighboring tiles.

TAILOR works by first building up an internal representation of the matrix. Depending on the options selected TAILOR makes several passes over the matrix, for example to determine row and column spacing. Signal names are produced on an optional label layer which the designer may specify. TAILOR traverses the input matrix along its y—axis, thus along the gates of the circuit structure. It first must ascertain the type of each gate by looking at the column header information. In the case of folded or flipped columns this information appears along the bottom edge of the array. TAILOR then places each tile, translating symbol for symbol from the tile matrix. Tiles are placed assuming their lower

left hand corner is the index origin. If the constant spacing options are invoked tiles will be spaced at regular intervals: without the constant spacing options tile are aligned so their bounding boxes, as specified by the user, abut. Finally, TAILOR calculates the physical bounding box in the user's units.

7.5.1. Tile Construction and Layout of Tiled Structures

This section examines the tiles required to build the one- and two-level structures produced by MKMAT and compacted by TWIST. The four basic gate structures were previously presented in Chapter 6. These basic gates are reproduced here in more detail and showing potential column folds.

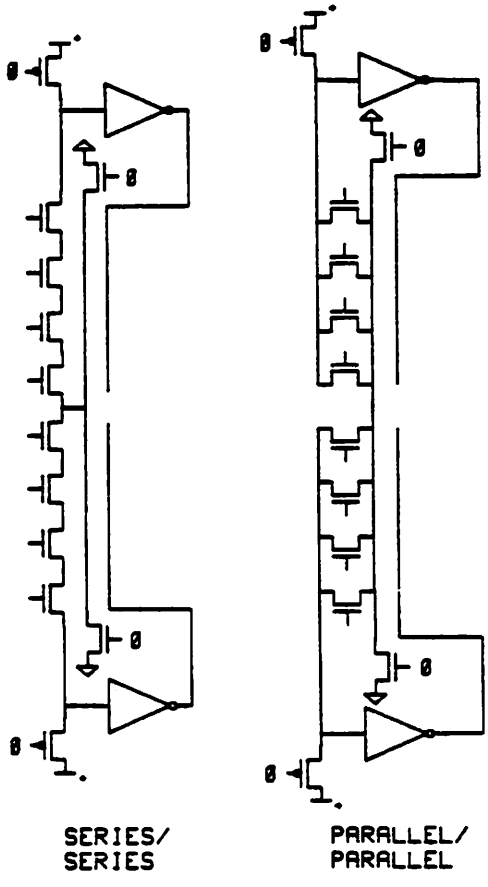


Figure 7.11: Four Basic Gates

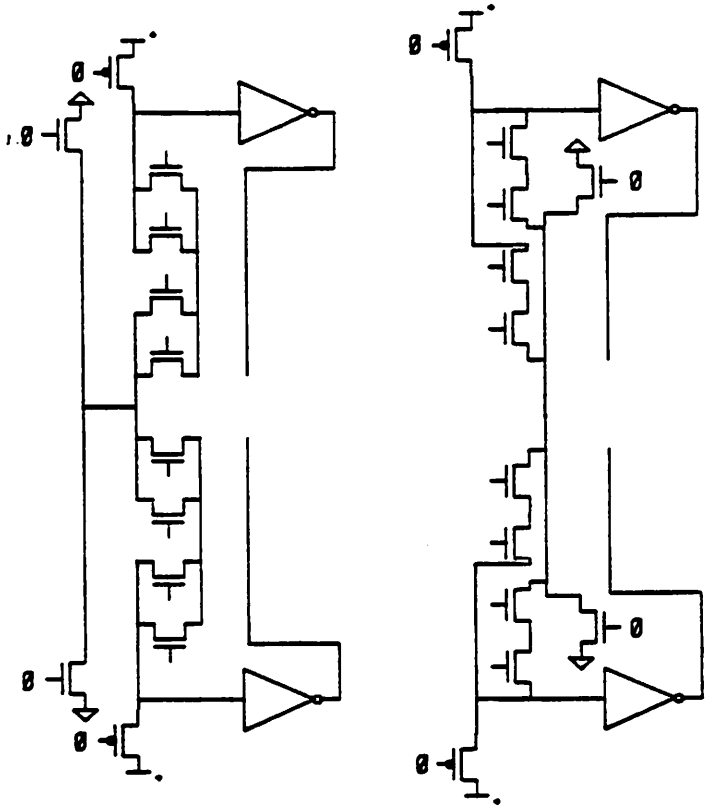


Figure 7.11, continued: Four Basic Gates

These structures can be broken down, in turn, into more fundamental cells which are the atomic tiles. There are two basic active element tiles, the SERIES and the PARALLEL tile. Cifplots showing their mask-level appearance in a current CMOS process are shown in Figure 7.12

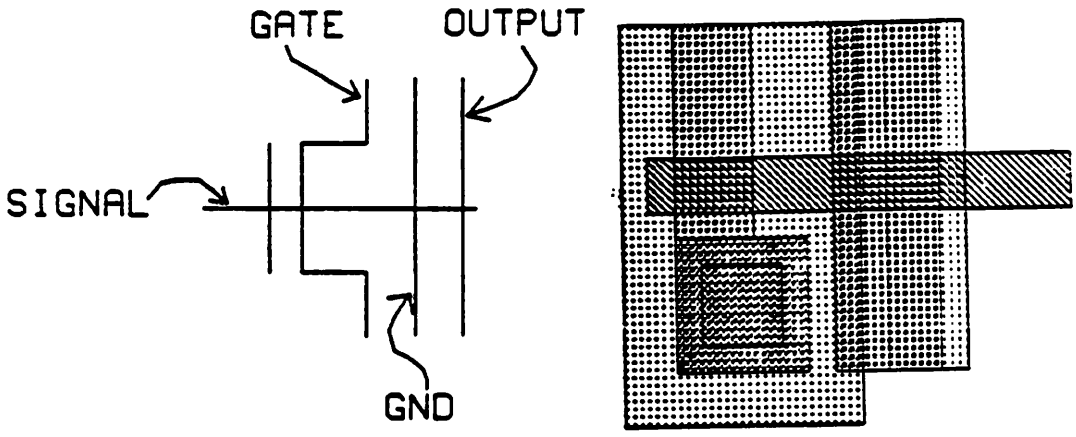


Figure 7.12a: Basic SERIES Tile

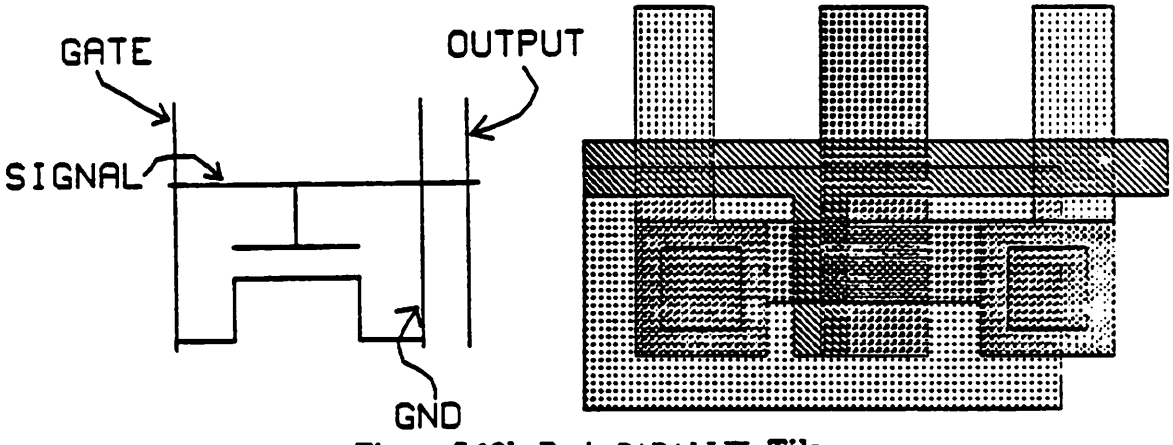


Figure 7.12b: Basic PARALLEL Tile

In addition to these tiles is an OUTPUT tile, used to connect the output from a gate, which runs vertically, to a signal term which runs horizontally. An OUTPUT tile is shown in Figure 7.13.

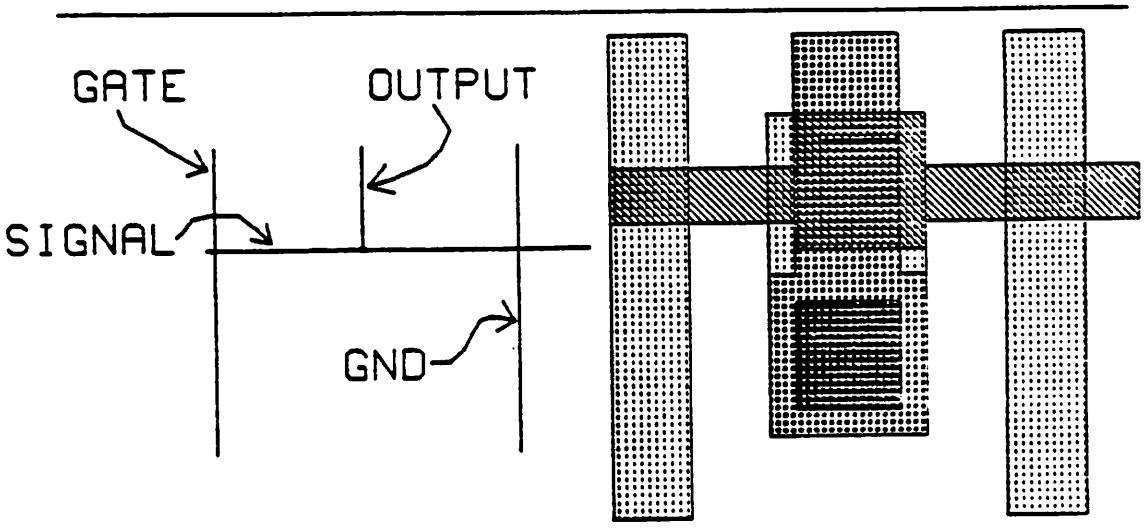


Figure 7.13: Basic OUTPUT Tile

The final core tile is the INTERCONNECT tile. This has two major variations depending on whether it spans a parallel or a series gate. This tile carries a signal line horizontally. If it is a parallel INTERCONNECT tile it carries power, ground, and gate output in the vertical direction. If it is a series INTERCONNECT tile it carries the gate signal, ground return, and output in the vertical direction. Examples of the interconnect tile are shown in Figure 7.14.

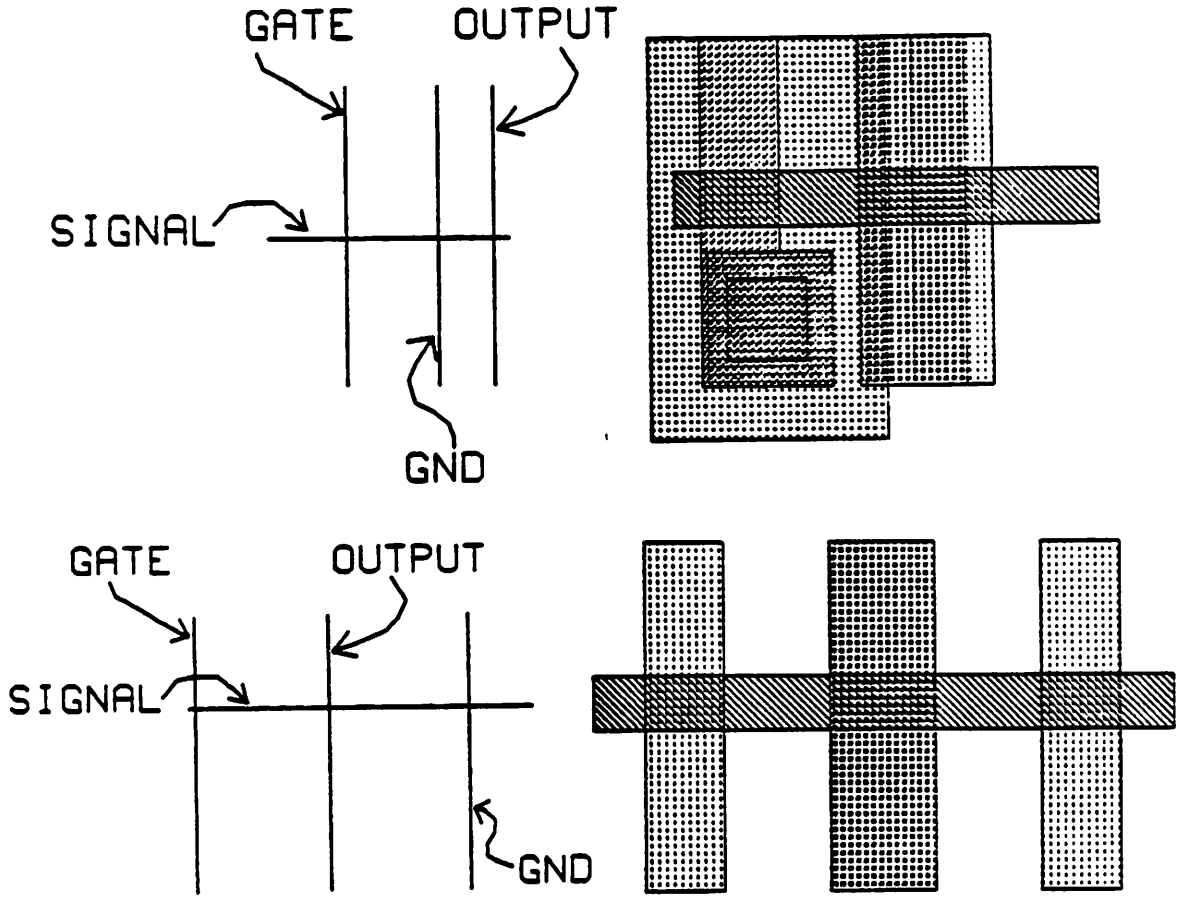


Figure 7.14: Two Examples of the INTERCONNECT Tile

These tiles or variations of them, selected by TINKER, are placed according to bounding box information to make up the gates shown schematically in Figure 7.11. The final mask layout of the electrical matrix of Figure 7.10 appears in Figure 7.15.

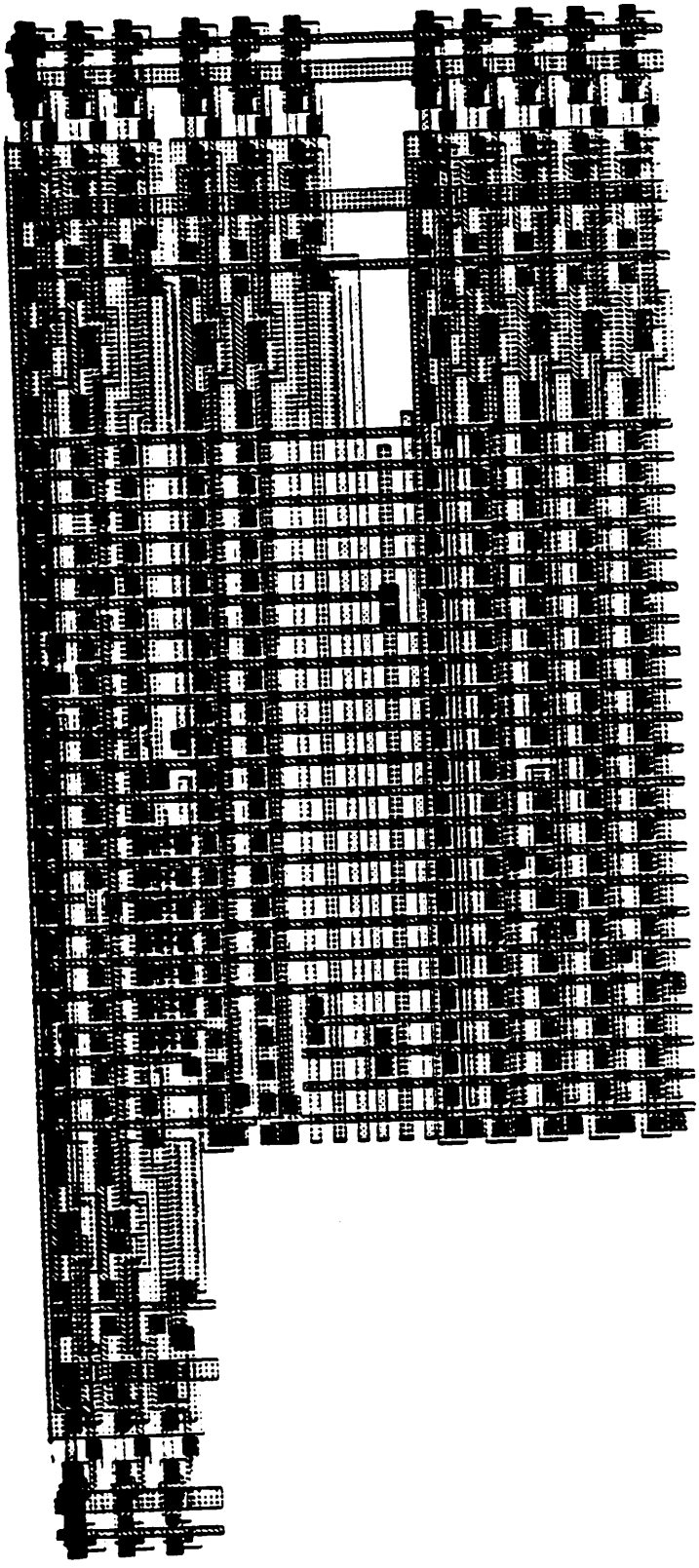


Figure 7.15: Mask Layout of Figure 7.10 after TAILOR

7.5.2. Modification of Tiles to Reflect Process Changes

The tiles shown in this chapter are based on a double metal, single poly process. It would be possible, though less area-efficient, to design an analogous set of tiles for a single metal, single poly technology. Alternatively, if a polysilicide layer were added it would be possible to design an electrically superior tile. The horizontal signal lines could be run in polysilicide. The inclusion of buried contacts and relaxation of design rules between vias connecting different layers would reduce tile area. When the process technology changes the tile sets can be redesigned to take advantage of the latest modifications. If the changes are small, for example a new set of design rules but no new layers, probably only the tileset at the TAILOR level need be updated. If, however, the process changes to the extent that the designer wishes to introduce new tile variations, then TINKER's *context* file would need modification. Note that it is only necessary to modify the tileset and context file but not the synthesis tool itself in order to reflect process changes. After such modification old designs can be regenerated in the new process. If greater modifications to the layout scheme were required, such as the addition of multiple column folding, then it would be necessary to extend MAMBO.

7.6. Summary

Three common layout methods, which cover the range of *tiled* techniques, have been been presented. Layout methods which require the use of routers were not examined; the focus of this research is on an automated approach where routing is not required. In order to take advantage of the property of dynamic CMOS circuits that allows construction of complex gates, as exemplified by the Domino style, a hybrid, context-based approach has been developed. This scheme has the flexibility of gate matrix and is efficient in the layout of dynamic, non-ratioed logic structures. Automated layout generation tools TINKER and TAILOR combined with the topological optimizer TWIST provide efficient, machine-generated layout much like the automation of Weinberger array construction.

CHAPTER 8

Comparison of Synthesis Methods

In this chapter several of the larger pieces of combinational logic used in the control portion of the CMOS SOAR chip are used as basis of comparison between synthesis methods. Much of the combinational logic in SOAR was implemented in 14 PLAs of varying sizes. Circuits designed by MAMBO are contrasted in delay and area with PLA implementations of the same logic functions.

8.1. Comparison Criteria for Multi-Level Matrices and PLAs

In the tables below the standard PLA implementations of the SOAR control logic are compared to the multi-level matrix implementation produced by MAMBO and presented in the previous chapters.

There are two basic metrics for comparison of logic implementations: critical path speed and circuit area. Often the optimization of one of these quantities is to the detriment of the other. This is the case in the MAMBO system. However, one can also use gates of differing capacitances to produce a range of circuits from fast, large designs to slower, more area-efficient ones. This area—speed tradeoff is one advantage the multi-level synthesis system has over a straight PLA tool which performs only area compaction and provides no convenient method for delay optimization.

The table in Figure 8.1 compares PLA and MAMBO implementations of five SOAR PLAs. In this figure the fastest multi-level implementation was chosen. Some explanation is necessary to understand this figure. First, the worst-case speed is easily determined in the MAMBO case where a recursive routine can trace back from all outputs to fundamental inputs and keep track of the longest path. This is not the case for the PLA-based implementation. To determine the worst-case delay through a PLA one must try all possible

combinations of input vectors and observe all outputs. Instead of doing such prohibitively expensive exhaustive tests, each input line was toggled individually and the longest delay at the outputs, indicating highest capacitance on the pterm lines, was taken as a "worst average" case. Both circuits were measured by transient analyses using SPICE2 and both circuits had outputs driving a nominal fanout of three. Circuit areas are given in tile units. The actual size of tile unit will vary according to implementation. A typical PLA tile might be $8\lambda \times 8\lambda$. A typical MAMBO tile in a current double-metal, single-poly process is larger, about $8\lambda \times 15\lambda$. It is assumed that the PLAs are implemented using an n-channel core and p-channel pullups with their gates grounded. Thus the PLA consumes static power. The MAMBO design consumes zero static power. If the dual of the n-channel PLA core were built then PLA power consumption would be reduced but device count, and area, would almost double. Area values for the folded PLAs were created using the simple folding option in PLEASURE [demi82]. For the MAMBO circuits simple column folding with multiple row folding was allowed since the more complex tiles can accommodate this without need for special folding cases.

Circuit	Speed *(ns)*		Area *(tiles)*		
	PLA	MAMBO	PLA		MAMBO
			Unfolded	Folded	
xcpla1	20.6	8.1	1763	1075	2948
cpla1	18.1	8.8	2346	1587	3696
apla	9.0	8.7	578	408	1591
tpla	16.0	8.3	805	483	2772
condpla	20.5	9.5	816	612	4026

Figure 8.1: Two-level Versus Multi-level Implementations

From the figure it is apparent that decreased circuit delay is paid for by increased circuit area. The largest MAMBO circuit is *condpla*, a circuit which is not particularly large as a PLA. *Condpla* has only two outputs but a very high dependency of input signals on these outputs. MAMBO achieves its speedup in part because it builds smaller, faster gates. For the *condpla* case a large number of these gates is needed. The result is a large, very sparse matrix. The *condpla* PLA matrix is relatively dense. If multiple column folding were possible the area could be significantly reduced.

8.2. Area Versus Speed Tradeoff in MAMBO

By constructing a high capacitance cell it is sometimes possible to make a good tradeoff between reduced circuit area and decreased speed. The table in Figure 8.2 shows runs using the same set of test cases but with a high capacitance tile.

Circuit	Speed	Area
xcpla1	18.7	1479
cpla1	21.5	1242
apla	8.1	841

Figure 8.2: Area/Speed Tradeoff in MAMBO

The control circuit *xcpla1* is both smaller and faster implemented as a multi-level matrix by MAMBO than the PLA implementation. Figure 8.3 shows the MAMBO mask layout in a 2μ industrial process. For comparison, the PLA implementation is shown in Figure 8.4 in the 3μ process employed to fabricate the SOAR chip. The plots are at the same scale.

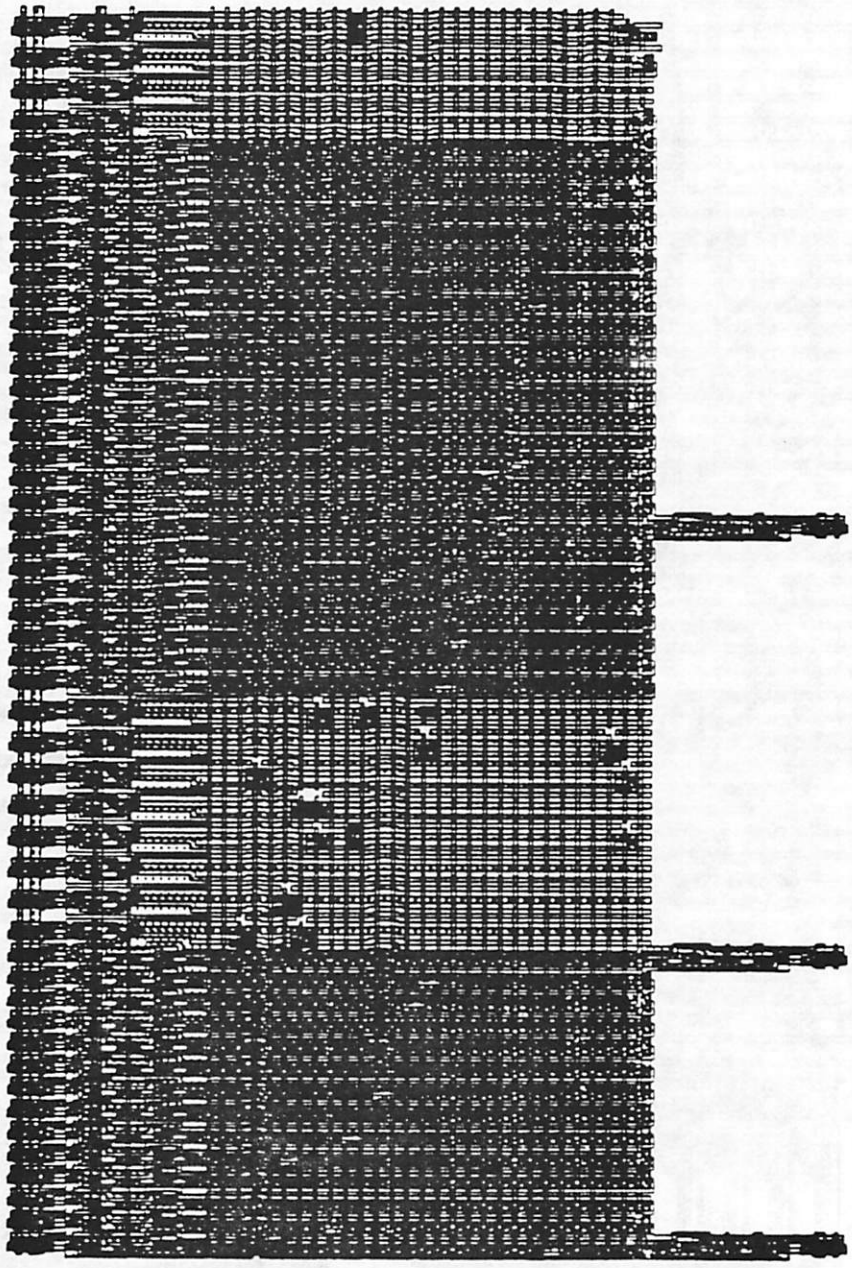


Figure 8.3: Control Circuit xcpla1 Synthesized by MAMBO

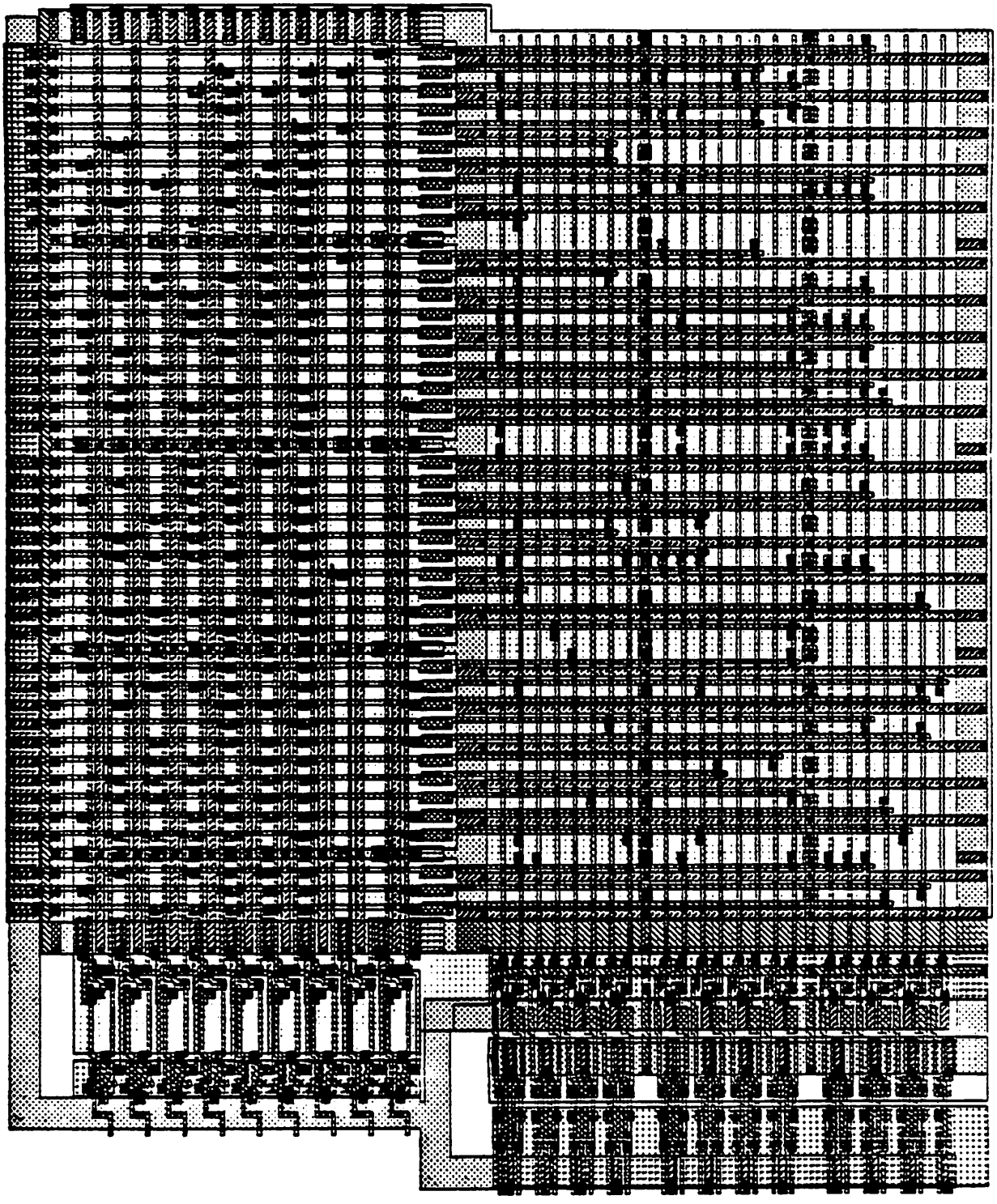


Figure 8.4: Control Circuit xcpla1 Implemented as a PLA

While *cpla1* is now slower than the PLA implementation it is also more compact. Since the

designer can easily modify a tile to increase capacitance he has the opportunity to create a cell which meets a variety of specifications.

8.3. Effect of Series Chain Length on Circuit Speed

By making use of the *—chain* option in MOSMESH it is possible to study the effect of limiting chain length on circuit size and speed. The table in Figure 8.5 shows that limiting chain length often increases circuit speed. In those cases where it does not it is because the chain restriction has forced the introduction of more stages into the critical path and thus caused a slower circuit overall.

Circuit	Series Length	Buffer Depth	Clusters		Speed *(ns)*
			before	after	
xcpla1	8	2	32	30	26.0
	5	3	66	54	18.9
tpla	6	4	68	52	13.1
	5	4	66	52	10.0
	4	3	70	55	8.3
cpla1	7	3	25	24	21.5
	5	3	29	28	22.3

Figure 8.5: Series Chain Length Versus Circuit Complexity and Speed

8.4. Effect of ON-set Versus Literal Count Minimization

If the designer chooses to construct circuits which have individual gate clusters with a depth limited to two levels, then the ESPRESSO program can be used to perform Boolean minimization. ESPRESSO can be invoked in two ways. In its normal usage within the MAMBO pipeline it performs single output circuit minimization by merging cubes and checking for containment of one cube within another cube or set of cubes. It produces the resulting minimum ON-set of cubes. If the Boolean equations are entered by a circuit designer with knowledge of the function he wishs to construct, this mode will frequently yield the best results. However, if the Boolean equations have been generated by a higher-level tool and are themselves an intermediate step, then minimization of equations by literal count is often helpful. When this form of minimization is used the resulting

circuit will often be of a mixed ON-set, OFF-set form. That set which has the fewest literals will be selected to implement the function. In ESPRESSO the natural ON-set is always in AND-OR form, thus the OFF-set will be in OR-AND form. By complementing inputs and outputs ESPRESSO can be made to produce an ON-set which is OR-AND and an OFF-set which is AND-OR. In uncomplemented form the top-level of a two-level OFF-set gate will be series (AND) in nature. Unless a long series gate can be tolerated (for example by a high capacitance cell) the OFF-set optimized cell will generally be slower than its ON-set dual. However, the circuit which has been literal-minimized and built with a high capacitance cell may not only be faster than its dual, it may be more compact as well. Figure 8.6 below summarizes results for various test cases with and without literal optimization.

Circuit	Optimization	Series Length	Clusters		Speed *(ns)*
			Before	After	
xcpla1	ON-set	8	32	30	26.0
	Literal	8	30	28	18.7
	ON-set	5	66	54	18.9
	Literal	5	41	35	15.2
cpla1	ON-set	8	30	28	25.0
	Literal	7	25	24	21.5
	ON-set	5	58	52	21.7
	Literal	5	29	28	22.3
condpla	ON-set	9	25	25	30.9
	Literal	5	14	14	69.5

Figure 8.6: ON-set Versus Literal Boolean Minimization

For *xcpla1* the literal minimization gives smaller, faster circuits; for *cpla1* the circuits are smaller but not always faster; for *condpla* a very compact, but very slow circuit is generated with literal count minimization.

The literal count/ON-set minimization switch gives the designer another way to trade off the parameters of circuit speed and layout area. Therefore the designer can choose the speed-area ratio he needs by constraining the number of devices in a series chain, by adding in precharge capacitance, and by selecting different Boolean minimization schemes.

8.5. Summary

The multi-level synthesis technique implemented in the MAMBO pipeline has been compared with the two-level approach of PLAs on combinational circuits. It was shown that the parameters of circuit area and speed trade off against each other. For all the examples tested it was possible to construct a multi-level circuit which was faster than the PLA implementation. In a particular case the multi-level array was both faster and more compact than the PLA version of the same circuit. By constructing varying capacitance tiles or constraining the number of devices in series the designer can choose an implementation that meets specification. The MAMBO system may also be run on two-level logic functions. In this case powerful Boolean minimization algorithms can be used to reduce the number of devices by literal count minimization.

CHAPTER 9

Conclusions and Further Work

The purpose of the research presented in this dissertation was to create a system for the synthesis of combinational logic. The system was constructed as a sequence of smaller programs called the MAMBO synthesis package. Each program in the sequence or pipeline reads from and writes to a text file; and thus new tools may be inserted into the pipeline easily. As a result, the system can be used as a framework for the development of combinational logic synthesis algorithms. An example of this framework nature is the addition of the ESPRESSO minimization program. After many of the other tools had been developed, the need for a two-level Boolean minimization tool became apparent. Because the framework is modular, the ESPRESSO program was inserted into the pipeline within a few days.

To focus on a number of specific synthesis issues the problem of constructing a structured multi-level logic circuit in a dynamic technology was chosen. A tiled layout approach was used because such methods combine active circuit area with routing area. This obviates the need for an additional routing tool. Multi-level logic was selected because it provides a degree of flexibility in circuit construction not found in methods like the PLA, which expand all logic expressions to two levels. This flexibility allows the designer to trade off layout area and circuit speed. Finally, a dynamic CMOS design style was selected as the target technology. The CMOS technology was chosen over NMOS because of the former's lower power consumption. By using a dynamic style device count may be significantly reduced over static CMOS designs. Further, dynamic designs pose fewer problems for automatic layout transistors with each input, thus easing the problem of signal routing.

In the first chapter the problem of combinational circuit synthesis was introduced and previous approaches were reviewed. Chapters 2 and 3 contain a comparison of different implementation technologies and present the results of two fabricated test chips. Chapters 4, 5, 6, and 7 are devoted to explaining the minimization, delay optimization, area compaction, and physical design algorithms employed in MAMBO. In Chapter 8 results of the MAMBO package and comparisons with PLAs used to implement the same combinational logic functions from a 32-bit microprocessor were presented.

The major topics of this dissertation are threefold. First, the design and implementation of a novel multi-level combinational logic synthesis system with a regular matrix layout structure was presented. The synthesis package creates a mask-level layout from Boolean expressions and the circuits generated are efficient in both speed and layout area. In addition the circuit has been electrically optimized to be free of charge redistribution problems. The second major topic is the development of a delay model to evaluate an arbitrary mesh of charged and discharged MOS devices. This model allows one to quickly and accurately determine the transient delay of a cluster of devices, for example a complex Domino-style gate, rather than requiring a detailed circuit simulation. The third aspect of this work is the context-based layout tiler. Layout tilers have been constructed in the past, but by adding the notion of context to a basic tile, one of a series of variant tiles may be selected depending on the tile's neighbors. This feature is useful in resizing active devices, for example, in a series chain, or for substituting compacted tiles in a region where a column or row of the layout matrix is folded. In addition to these topics a 32-bit dynamic Domino CMOS ALU was designed and fabricated to gain insight into Domino design techniques. The ALU uses a novel modification of the carry bypass scheme to provide carry acceleration while maintaining zero static power consumption and a ratioless style.

The current MAMBO pipeline consists of the seven stages described in this report plus an auxiliary program to generate transient delay times for circuit clusters. In addition, the tools EQNTOTT and ESPRESSO may be inserted into the pipeline. Figure 9.1 gives a breakdown of the amount of 'C' code (including comments) in each of the various pipeline

stages.

Tool	Lines of Code
MGMG	2665
MKTBL	1538
MOSMESH	2882
MIMIC	843
MKMAT	1653
TWIST	4112
TINKER	1048
TAILOR	1244
Total	15985

Figure 9.1: Breakdown of MAMBO Pipeline

As a result of studying synthesis problems with the MAMBO framework a number of conclusions can be drawn. First, it has been shown that the problem of charge redistribution must be taken into account in dealing with dynamic circuits. Because of the charge redistribution effect it is often necessary to partition large, complex dynamic networks into a tree of smaller clusters. The partitioning of circuit networks has a direct result on the speed of the logic circuit and circuits may be partitioned into many low capacitance nodes for less delay, or a smaller number of higher capacitance nodes for greater area compaction. Second, it is possible to topologically rearrange circuits to reduce layout area in much the same manner as topological folding of PLAs. In the current framework algorithms have been developed which allow simple column and multiple row folding. Signals may be placed according to external constraints. Third, context-based tiling is a convenient method for implementing circuits which have been modified geometrically or electrically depending on area compaction or electrical optimization, respectively. Finally, by performing both delay optimization and topological compaction the circuit designer can build the module which best fits into a larger scheme. MAMBO results show that circuit configurations can be constructed which are more than twice as fast as similar PLA implementations and further speed improvements may be possible with better multi-level optimization techniques. At the other extreme are configurations slightly faster than PLAs which are comparable or more compact in size.

Several aspects of circuit design described in this dissertation involve the analysis of "special cases" and it does not seem likely that a single algorithmic approach will ever handle these cases uniformly. A number of the MAMBO tools, for example MOSMESH, use heuristic "rules" to process these special cases. It would be interesting to re-implement these tools using a rule-based system. However, at various stages of circuit optimization, powerful algorithmic procedures are used (*eg.* the cycle detection algorithms in TWIST) and therefore a framework which supports a mixed rule-based and algorithmic approach seems best.

There are many open questions in multi-level logic design. As yet there are no good methods for efficient logic optimization of multi-level networks. Presently MAMBO can take advantage of Boolean minimization by using the standard two-level techniques employed by MGMG and ESPRESSO. It would be interesting to insert a multi-level optimization tool into the MAMBO pipeline.

The present synthesis tool is designed for a matrix layout structure. In this structure cells are routed by abutment. One of the drawbacks with this approach is that although Domino logic allows complex gates (in fact complex gates are more area-efficient than simple gates) the matrix scheme only accepts two-level clusters. If a layout method were created which supported multi-level circuits, such as standard cells, then it would be possible to compare the area efficiency of the two approaches. Standard cells have the drawback that channel area is required for cell interconnection.

At present the precharge and buffer devices of each Domino gate are constrained to lie along the top and bottom edges of the matrix layout and input and output signals are available from the left and right sides. This approach was chosen to simplify bussing of V_{DD}, GND, and clock signals and to aid in folding the matrix. It may be worthwhile to relax these constraints and create matrices in which all four sides of the array are treated uniformly. By doing this routing problems between the generated matrix and other cells of a floorplan may be eased.

Tuning a circuit for greater speed often directly conflicts with reducing circuit area. In current minimization schemes one often attempts to minimize the number of literals in a single function, as is done here, or across several functions, as for example in [bray82]. While these methods directly reduce device count and therefore result in more compact circuits they may also lengthen the electrical critical path. Delay optimization and area compaction are coupled problems. It might be beneficial to try different minimization algorithms along the electrical critical path of a circuit as opposed to non-critical paths. It would then be necessary to iteratively apply the delay and compaction steps until the critical path does not change.

APPENDIX A

SPICE2 MOS Models

MOSIS MOS Models

```
* as from MOSIS, note low vto values
* device models


.model nmo nmos level=2 vto=0.5 lambda=0.01
+ uo=475 phi=0.5
+ cgso=1.3e-10 cgdo=1.3e-10 cgbo=4.10e-10 rsh=40 cj=6e-4
+ cjsw=4e-10 js=3.5e-5 tox=5.5e-8 nsub=5e15 nss=-2.8e11 nfs=1e10
+ xj=6e-7 ld=4e-7 ucrit=8e4 uexp=0.25 utra=0.25 vmax=5e4
+ mj = 0.5 mjsw = 0.5 pb = 0.7 neff = 2.5



.model pmo pmos level=2 vto=-0.5 lambda=0.01
+ uo=190 phi=0.492
+ cgso=1.3e-10 cgdo=1.3e-10 cgbo=4.10e-10 rsh=40 cj=4.1e-4
+ cjsw=2.5e-10 js=3.5e-5 tox=5.5e-8 nsub=3e14 nss=11.7e11 nfs=1e10
+ xj=6e-7 ld=4e-7 ucrit=8e4 uexp=0.25 utra=0.25 vmax=5e4
+ mj = 0.5 mjsw = 0.5 pb = 0.7 neff = 2.5
```

SPICE2 MOS Models

Industrial MOS Models

```
* nmos model
 .model cmosn nmos level=2 ld=0.23u tox=300.00e-10 nsub=1.0e+16
+  vto=.9 kp=5.9e-5 gamma=.44 phi=.6 uo=512.6
+  xj=.3u lambda=2.7e-2 tpg=1 rsh=35.0 uexp=6.0e-2
+  vmax=5.7e4 cgso=2.7e-10 cgdo=2.7e-10 cj=2.8e-4
+  cjsw=1.5e-10 mj=.5 mjsw=.3333
```

```
* pmos model
 .model cmosp pmos level=2 ld=.38u tox=300e-10 nsub=1e16
+  vto=-.9 kp=2e-5 gamma=.492 phi=.6 uo=173.8
+  xj=.5u lambda=4e-2 tpg=1 rsh=120 uexp=.44
+  vmax=9e4 cgso=4.5e-10 cgdo=4.5e-10 cj=3.1e-4
+  cjsw=3e-10 mj=.5 mjsw=.3333
```

APPENDIX B

Measurement of the Dynamic CMOS Test Chip

In this appendix the equipment and testing procedures for a chip to test the properties of dynamic CMOS circuits are described. Two bonded test dies were received in late 1984 and testing was begun in December, 1984. The chips were fabricated in a double metal 2μ (drawn) n-well process at an industrial site. Specifically, the first chip (referred to as "chip #1") has a buried contact layer, a single level of polysilicon, and two layers of metal. The pre-implant oxidation was wet: there was no post-implant oxidation. The die was not passivated. The second chip ("chip #2") does not have a buried contact layer, but does have a polysilicide layer and two layers of metal. There was a 50:1 gate oxide etch for the polysilicide. The pre-implant and post-implant oxidations were dry. The die was not passivated. Neither chip made use of the buried contacts. Most of the results presented here were obtained from chip #1, with the polysilicon layer.

The test setup consisted of a super-strip mounted on a metal jig which was grounded to the power supplies. There were separate power supplies for V_{DD} and V_{sub}. For most of the measurements reported in Chapter 3 V_{sub} was 0.0V. The chip was driven by three signal generators. The first, an HP 8011A, provided synchronization trigger pulses for the other two generators. The HP 8011A provided the *bottom* input signal mentioned in Chapter 3. An HP 3312A provided the clock signal, ϕ. An Exact 126 provided the input signal *top*. Waveforms were displayed on a Tektronix 7094 series oscilloscope.

Figure B.1 shows the bondmap for the chip and Figure B.2 is a mask-level plot.

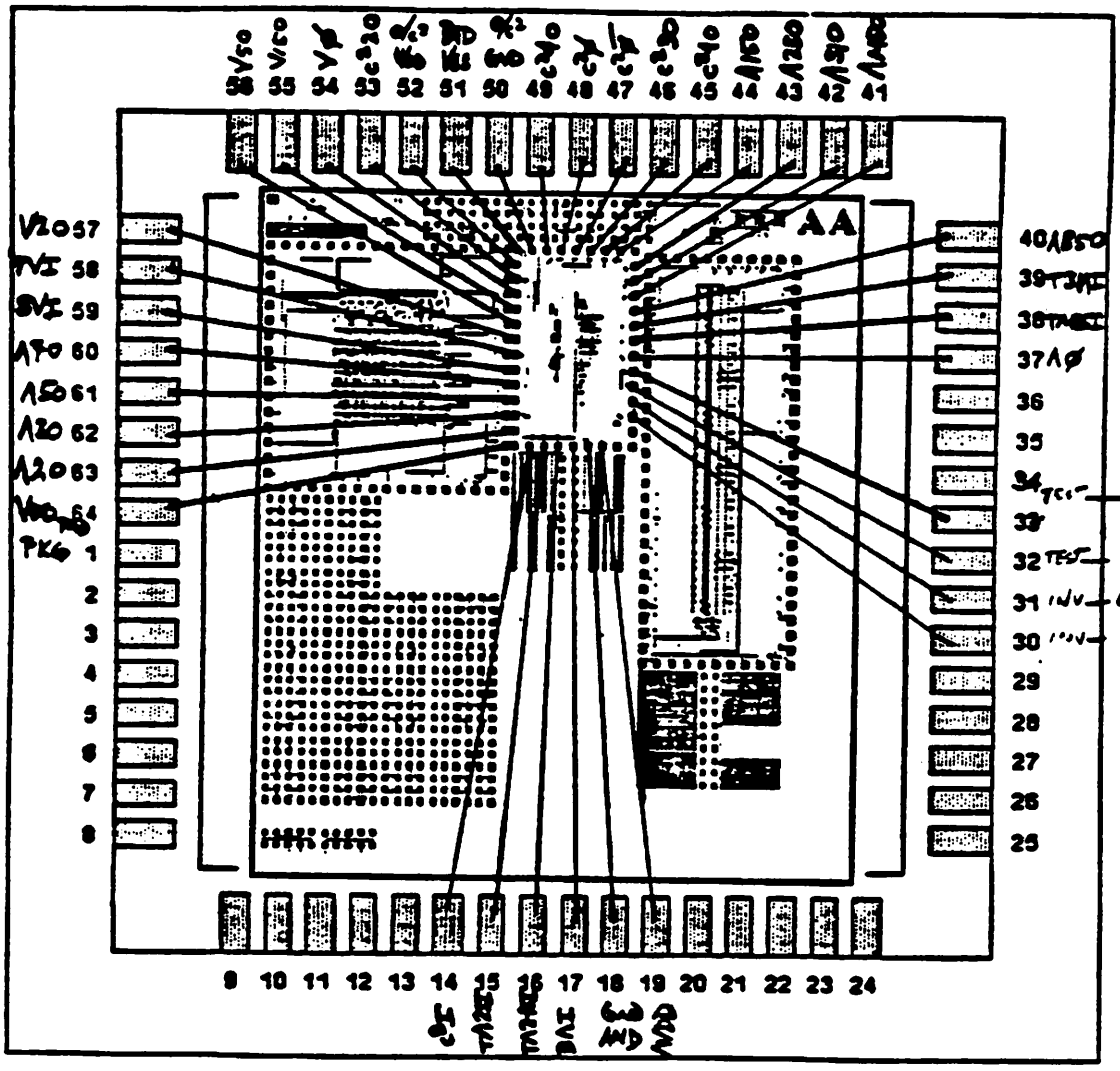


Figure B.1: Bondmap for Test chip

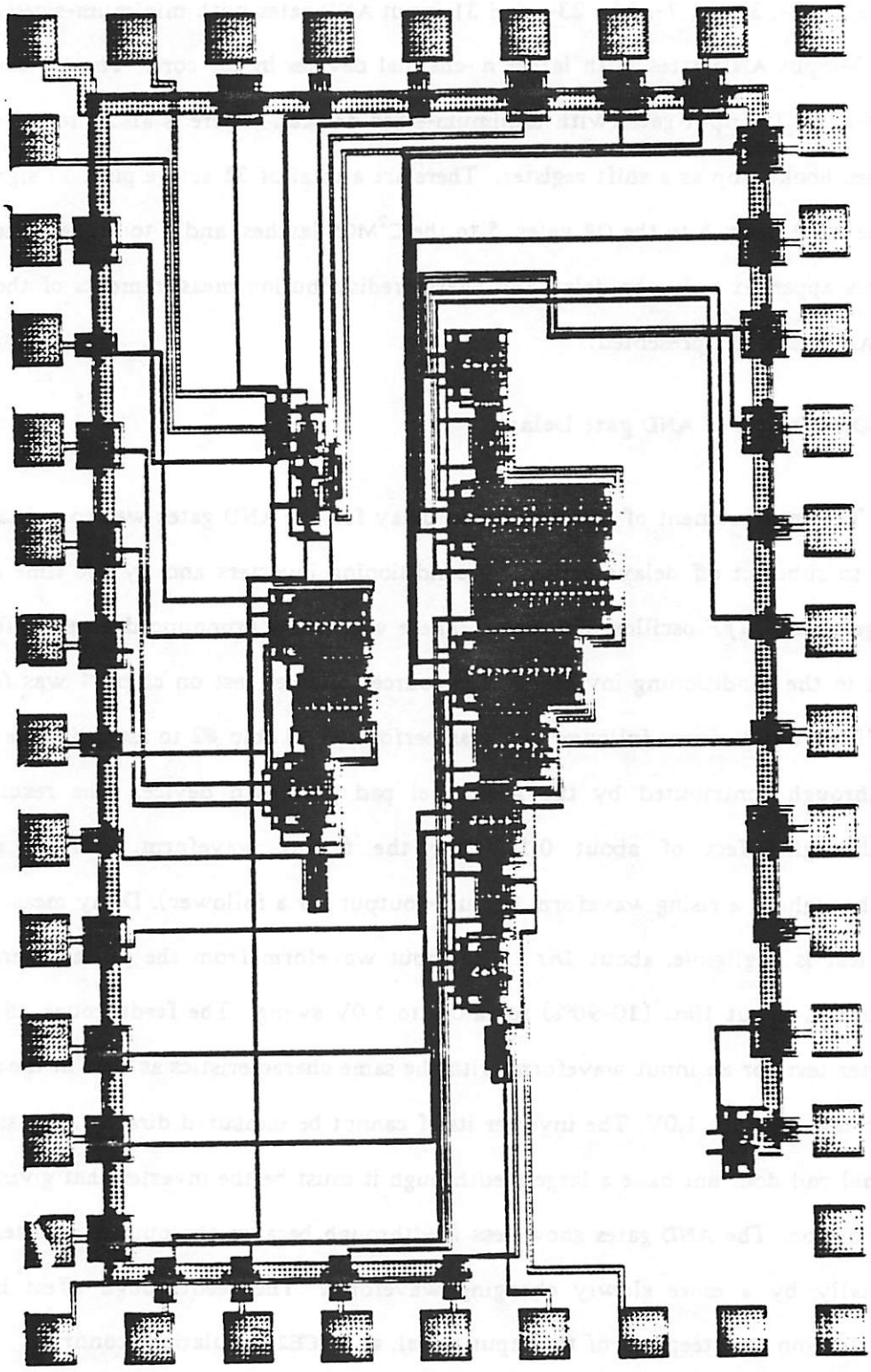


Figure B.2: Cifplot of Test Chip

The chip was laid out in three main blocks. The largest block is a bank of AND gates. There are 2-, 3-, 5-, 7-, 15-, 23-, and 31-input AND gates with minimum-sized devices and two 5-input AND gates with larger n-channel devices in the core. The OR bank contains 2-, 5-, and 15-input gates with minimum-sized devices. There is also a set of four C^2MOS latches hooked up as a shift register. There are a total of 38 active pins, 17 signals pertain to the AND gates, 8 to the OR gates, 5 to the C^2MOS latches, and 8 to miscellaneous signals. In this appendix only the delay and charge redistribution measurements of the minimum size AND gates are presented.

Derivation of AND gate Delay

The measurement of the worst-case delay for the AND gates was complicated by the need to subtract off delays caused by conditioning inverters and by the time required to charge up an $11pF$ oscilloscope probe. There was also a pronounced inverter feedthrough effect in the conditioning inverters. The source follower test on chip #1 was found to be unreliable but a source follower test was performed on chip #2 to ascertain the amount of feedthrough contributed by the p-channel pad pulldown device. The results show a feedthrough effect of about 0.12V for the falling waveform, and no appreciable feedthrough on a rising waveform (input = output for a follower). Delay measured by the 50% test is negligible, about $1ns$. The input waveform from the pulse generator had a risetime of about $10ns$ (10-90%) for a 0.0 to 5.0V swing. The feedthrough effect for the inverter test for an input waveform with the same characteristics as that in the source follower test is about 1.0V. The inverter itself cannot be measured directly, but since the p-channel pad does not have a large feedthrough it must be the inverter that gives the major contribution. The AND gates show less feedthrough because the output inverter is driven internally by a more slowly changing waveform. The feedthrough effect is strongly dependent on the steepness of the input signal, as SPICE2 simulations confirm.

The inverter delay test was run twice, once with a large swing on the input and once with a swing of about 1.0V. The reduced swing lessened the feedthrough effect. The

falling input signal shows about 0.3V feedthrough at the output. The rising input signal shows about 0.27V feedthrough at the output. Test results for this test and the previous inverter test are summarized in Figure B.3.

Falling Input Feedthrough *(V)*	Delay *(ns)*	Rising Input Feedthrough *(V)*	Delay *(ns)*
0.3	28	0.27	15
1.0	39	1.2	21

Figure B.3: Feedthrough Effect on Inverter Delay

All the AND gates show a negligible feedthrough effect. Applying a linear fit to the values in the figure implies a risetime of about 23.3*ns* with 0.0V feedthrough and a falltime of about 13.3*ns*. This figure is the calculated delay through a conditioning inverter plus routing and an output pad driving the oscilloscope probe.

Charge Redistribution Measurements

Probably due to careless handling of the unpassivated chips, AND gates 3, 5, 15, and 23 were damaged. Charge redistribution *(CR)* tests were performed with the 2-, 7-, and 31-input AND gates. Chip #2 did not have functional AND gates, as mentioned above. Tests were set up using the 3 signal generators, 1 each for signals ϕ, *top*, and *bottom*. The scheme was to first discharge the AND chain, then charge up the precharge *(prech)* node, and then allow the charge on the prech node to redistribute along the AND chain. The result is that the potential on the prech node decreases and, thus, the inverter which is driven by this node may react by making a false $0 \rightarrow 1$ transition. Therefore there are 3 parts to the CR test cycle: 1) dump chain charge; 2) charge up precharge node; and 3) look for a false transition ($0 \rightarrow 1$) at the output of the inverter. The 3 signal generators were triggered by the generator that provided the *bottom* signal. *Bottom* was a square wave; it was low for about 12μs and then high for 12μs. Signal *top*, triggered from *bottom*, was a pulse. The rising edge of the pulse was approximately coincident (slightly delayed) with the rising edge of *bottom*. It had a duration of about 4μs. Signal ϕ was delayed slightly with respect to both *bottom* and *top*. It was a pulse beginning about 1μs after *top* and of

less than 1μs duration. Special care was taken to ensure that *top*—gated devices were turned *off* before the p-channel ϕ—gated device (prech device) was turned *on*. Also *top*—gated devices were turned *on* only after the ϕ—gated p-channel prech device was turned *off*. Depending on device thresholds it might have been possible to drive the top of the AND chain and the clock devices from one signal. This latter arrangement, however, would not have guaranteed that there was no time at which both prech and chain devices were on. If such a condition existed then the prech current would charge up not only the prech node but all devices in the AND chain gated by *top*. Thus the chain would not be discharged to ground. Phase 1 occurs when all signals (*top*, *bottom*, ϕ) are low- thus it is bounded by the 12μs *bottom* signal. Remember that these signals are input to the chip and that all signals are fed through conditioning inverters before affecting the test circuitry. Therefore when all signals are low the clock p-channel device is *off*, the *top*— and *bottom*—gated n-channel devices and n-channel clock device is *on*. Charge is cleared from the AND chain. Phase 2 begins when ϕ goes high. Thus there is a short dead time when *bottom* and *top* are high and ϕ is low. In this period all devices except the n-channel clock device are off. In phase 2, when ϕ goes high (all signals are now high), the p-channel clock device is turned on, the n-channel FET is turned off. Charge is pumped into the prech node. The precharged area includes only the gates of the n- and p-channel output inverter, the p-channel pullup (prech device) and the drain of the topmost n-channel AND FET; none of the AND chain parasitics are charged since *top* and *bottom* are off. Still in phase 2, ϕ is lowered. Now ϕ is low, but *top* and *bottom* are high just like the beginning of phase 2. In this last part of phase 2 charge leaks slowly from the prech node. Tests show, however, that leakage takes milliseconds to occur while this stage has a duration of about 3μs. Phase 3 begins when *top* is lowered. Now ϕ and *top* are low, but *bottom* is still high. At this point the p-channel clock device remains off so there is no path from V_{DD} to the prech node. *Top* is low now so the top $n-1$ AND n-channel devices are turned on and CR takes place from the prech node to AND n-channel source/drain parasitics. This lowers the voltage on the prech node. The new voltage is: $V_{DD} \times C_{sub} \frac{prech}{C_{prech} + C_{parasitic}}$ — in words it is

proportional to the ratio of the prech capacitance to the prech capacitance plus the parasitic capacitance. If $C_{parasitic}$ is large enough, compared to C_{prech} then the voltage at the prech node falls to the point that the p-channel device in the inverter switches on and the output of the inverter goes high, making an incorrect $0 \rightarrow 1$ transition. This is called "false trigger". There is no DC path to ground (*bottom* is still high), so the n-channel AND gate (the last in the chain) is still off. Therefore, the output should remain off (low). Phase 3 ends when *bottom* goes low. At this point all signals are once again low and Phase 1 begins. This test was used to produce the measurements detailed in Chapter 3.

As a control experiment to show that the effect observed was due to CR, tests were conducted in which both the prech node and all the *top*—gated $(n-1)$ AND devices were charged up. Then the *top* input was toggled to cause a transient. The oscilloscope photos show a very slight opposite CR effect: that is charge flowing from the parasitics into the prech node. The prech node voltage decreases slightly. The exact same voltage decrease is seen in the 2-input AND as the 31-input AND case. It is independent of gate fan-in. This test was conducted by modifying the prech phase (2) of the test described above. In phase 1 the charge is still dumped from the parasitics because all devices except the n-channel clock FET are off. However, in phase 2 ϕ goes high while *top* is still low, thus *top* and ϕ are on at the same time. *Bottom* is high, or off. Now the p-channel clock device charges both the prech node and the parasitics. Then *top* goes high, turning off the AND chain parasitic devices. Then ϕ goes low, so the p-channel clock device is off and the n-channel clock device is on. At this point prech is off, *top* is off, *bottom* is still off, and the n-channel clock is on. Thus charge remains on the prech node and the (isolated) *top* nodes. Now *top* goes low (on) allowing CR between prech node and parasitics. However, there is little redistribution because both nodes are at virtually the same potential, having been charged by the same p-channel device about $1\mu s$ before. This concludes the CR measurements.

Voltage Threshold Measurements

The process parameters for circuit simulation were provided by the industrial facility. The voltage threshold of a p-channel device can be measured however, since the output driver is a p-channel device. The measurement was made as a check on the given parameters. Figure B.4 shows the circuit schematic of the output pad.

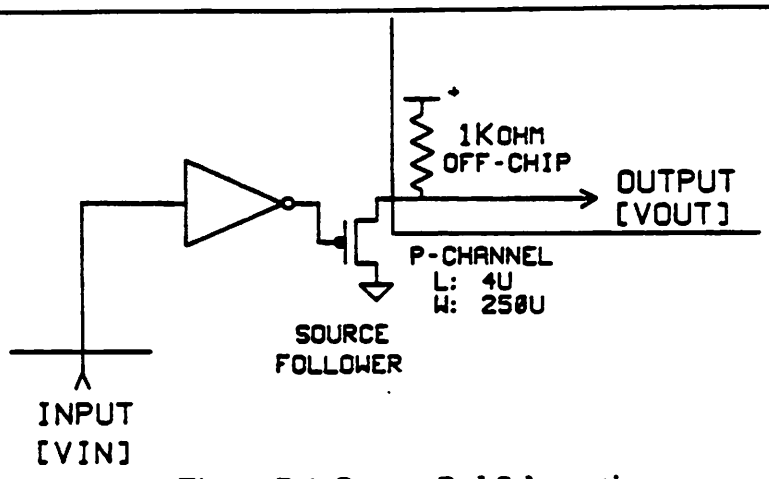


Figure B.4: Output Pad Schematic

The table in Figure B.5 shows results of V_{in} versus V_{out} tests on chip #2.

PadGnd = GND = V_{sub} = -0.098V V_{DD} = 5.15V; Resistor: 1K Ω off-chip	
Input *(V)* Pad30	Output *(V)* Pad31
0.00	5.13
0.50	5.13
1.00	5.13
1.50	5.13
2.00	5.13
2.50	5.13
3.00	4.09
3.50	3.77
4.00	3.70
4.50	3.69
5.00	3.69

Figure B.5: V_{in} versus V_{out} for Chip #2

Figure B.6 presents these results in graphical form from an oscilloscope photo. The horizontal and vertical axes are 0.5V per division. The lower, lefthand corner represents 0.0V in x and y.

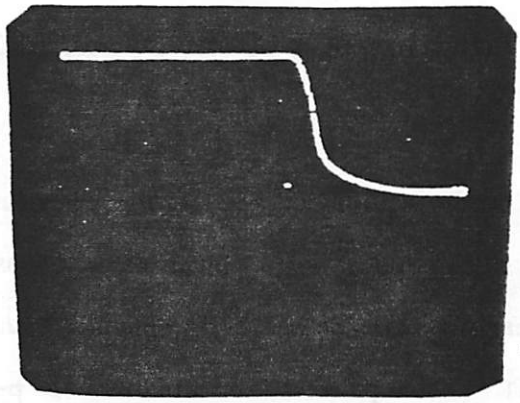


Figure B.6: V_{in} versus V_{out} for Chip #2

The V_t calculations are based on a first-order Shichman-Hodges model which assumes a quadratic dependence of voltage on current and ignores short-channel effects. When V_{in} is V_{DD} (5.0V) then V_{p-gate} is 0.0V. Thus, $|V_{GS}|$ is 5.0V and the p-channel device is assumed to be in saturation because $|V_{GS}| - |V_t| < |V_{DS}|$. In saturation the current flowing in the source node is $\frac{K_p}{2} \times (V_{in} - V_t)^2 = \frac{V_{out}}{R}$ where R is the value of the resistor in the source leg. Therefore,

$$V_t = V_{DD} - V_{out} - \left[\frac{2 \times V_{out}}{R \times K_p}\right]^{1/2} \tag{B.1}$$

because $V_{in} - V_t = V_{DD} - V_{out} - V_t$. $K_p = \frac{W_p}{L_p} \mu_p C_{ox}$. Using the values from Appendix A, $\mu_0 = 174 cm^2/volt-sec$. $T_{ox} = 300 \dot{a}$. and $W = 250\mu$. $L = 4\mu$. $R = 1000\ \Omega$ and $V_{out} = 3.69$V then $K_p = 1.25 \times 10^{-3}$ (factoring out W and L, $K_p = 2.02 \times 10^{-5}$) and $V_t = 2.24$V. Thus, for the p-channel device, $V_t = -2.24$V. This value seems high. According to [pugh85] the measured V_t for 2μ p-channel devices was about 1.0V. Applying a scale-up factor provided by the industrial site and 4μ device should exhibit a V_t of about 1.15V. One possible explanation for the disparity is the lack of a passivation layer on the test die. The chips were not carefully handled initially and some impurities may have affected the thresholds. The AND gates, however, all seemed to operate within a more reasonable V_t range.

APPENDIX C

Evaluation of a 32-bit Dynamic CMOS ALU

In this appendix the equipment and testing procedures for the dynamic ALU are described. The first silicon run of the 32-bit dynamic CMOS ALU was fabricated in March 1984 and tested in early June. The MOSIS P-name is *domalu*, the P-id was 16362, and the Fab-id was M41VHA1. The chips were fabricated in a 3μ, p-well, CMOS process at Telmos. Masks were made at Burroughs. There were six first silicon chips but the sixth chip, numbered 5, failed basic equality/carry chain tests and was not tested further. A test jig was built consisting of a piece of vectorboard and a 64-pin ZIF socket. A Tektronix Digital Analysis System DAS 9100 was used both to provide test vectors and to acquire and store the result vectors. An HP 8011A pulse generator was occasionally used outboard to provide slightly higher resolution timing pulses.

Figure C.1 shows the bondmap for the first silicon ALU. Forty-seven pins were bonded, 32 of which are hooked to tri-state buffers and used for I/O; 10 pins are used as control signals to load latches and select ALU operations; the ϕ clock is a single pin, power and ground take two pins and the signals $A-B=0$ and *Cout_H* are also brought out.

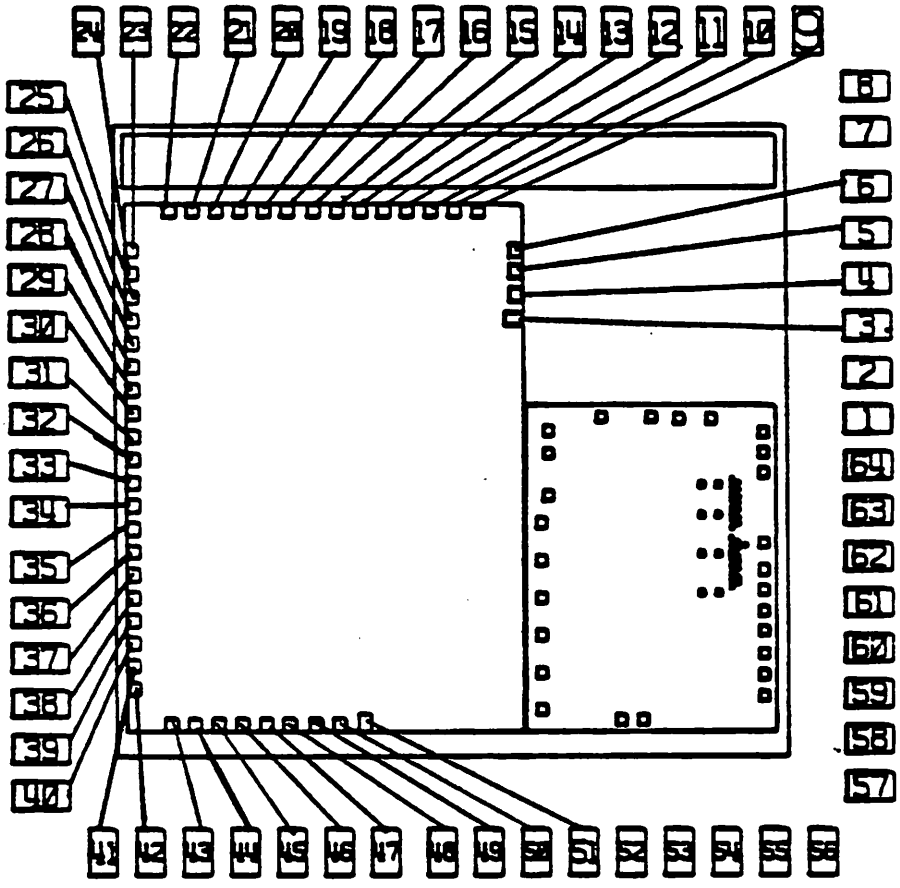
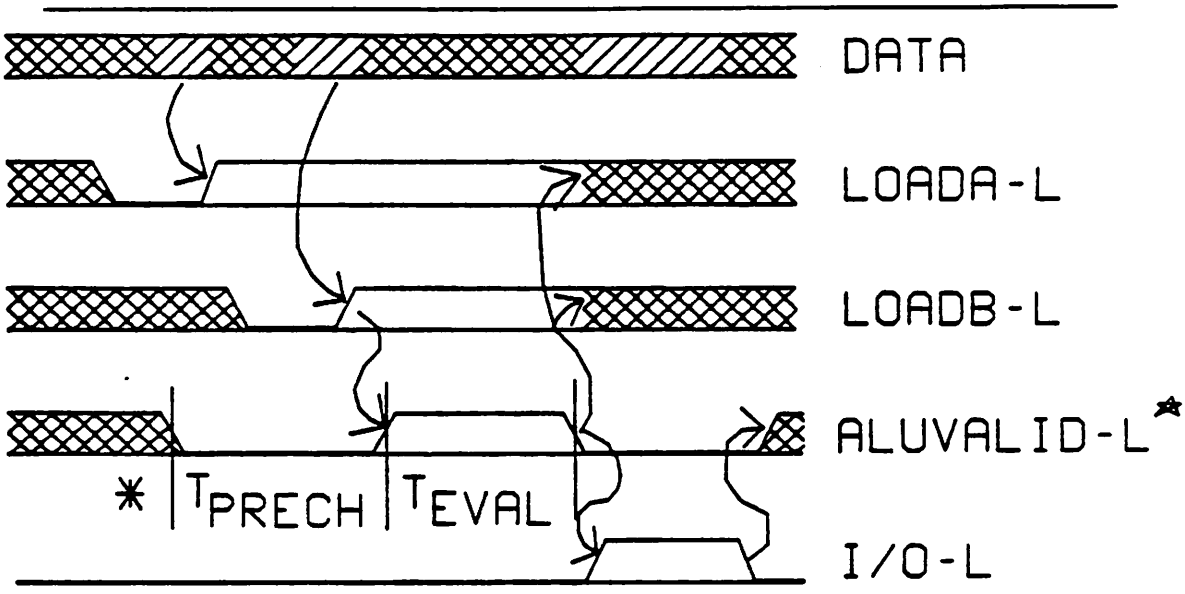


Figure C.1: Bondmap for ALU chip

Figure C.2 shows the basic ALU timing diagram.



DATA IN MUST BE STABLE OR DATA OUT IS VALID

SIGNAL MAY CHANGE

WAVEFORM AT ARROW HEAD DEPENDS ON WAVEFORM AT ARROW TAIL

* PRECH AND EVAL TIMES DETERMINED BY EXPERIMENTATION
EVAL CANNOT BEGIN UNTIL A AND B ARE LOADED
AND PRECH IS COMPLETE

★ ALUVALID-L = ALUEVAL-H = LOADD-L

Figure C.2: Basic ALU Timing Diagram

First the *A* operand is placed on the I/O pins and the *A* latch is loaded, then the *B* operand is placed on the same set of pins and the *B* latch is loaded. During this load time the ALU is in precharge. The precharge time was never measured, but simulations suggest it is not greater than 10*ns*, therefore the time to load the latches was the constraining factor in this part of the clock cycle. The ALU computes during the evaluation phase of ϕ, when signal *ALUVALID_L* is high. Bringing *ALUVALID_L* low stops ALU evaluation and simultane-

ously loads the ALU result into the destination latch. The *I/O_L* signal may then be brought high allowing the destination latch information to be read out through the I/O pins.

The Tektronix DAS 9100 was programmed to generate test vectors and loop on the data acquired from the output latch. Instructions were fed to the chip at a 200*ns* rate by the DAS. The template test program was:

```
ABC    ----   ----   0001
       ----   ----   0000
       ----   ----   0002
       ----   ----   0000   2345
       XXXX   XXXX   0000   repeat 20
       XXXX   XXXX   0000   goto ABC
```

The symbol '-' represents data to be filled in. The first 2 dashed lines should agree, as should the second pair of dashed lines. The symbol 'X' represents a "don't care" condition.

The 200*ns* cycle time is not critical. During each pair of instructions the A and B registers, respectively, are loaded. The *2345* field activates four time-critical strobes which control the evaluation time of the Domino circuit. The Domino circuit is precharged during the loading of the A and B registers (The time required to do this should be less than 20*ns*, although it has not been measured.). The *repeat 20* instruction is a wait loop to allow the chip's output buffers to drive the off-chip probe capacitances. Finally the *goto ABC* instruction makes the program an infinite loop. The test case repeats until stopped by an external interrupt.

The time-critical strobes were generated externally by the HP 8011A Pulse Generator. The trigger signal for the pulses was a 40*ns* pulse provided by the DAS strobe lines. The pulse generator provides a continuously variable pulse. The pulse height is also variable. Rise/Falltimes are about 20*ns*. All pulse widths were measured between the 2.5V points.

In the paragraphs that follow the test vectors for each of the various timing tests are described. The accompanying photograph was taken from the CRT of the DAS9100 and

contains the values of the input and output latches in HEX format.

ADD test:

```
FFFF   FFFF   0001
FFFF   FFFF   0000
0000   0001   0002
0000   0001   0000
```

This is a worst-case add. F's in reg A, 0...01 in reg B. The last group of four digits *0001* and *0002* is a command to load latches A and B, respectively. A carry is thus generated in the least significant bit and must propagate the full length of the carry chain. It goes through the first nibble slowly, the bypass for the next six nibbles, and finally through the last nibble in order to affect bit 31, which is the slowest settling bit. Since the carry_out signal does not have to go through this last nibble (it goes through seven bypass nibbles instead) it is faster. Figure C.3 shows the bytes (from MSB to LSB) *D, C, B, A* of operands A and B and then the result latch in order. Only the last two bits of field *E* are used. The LSB is the signal *A—B=0_L*, the second bit is the signal *Cout_H*. The final result at line 21 shows an output value of 0 with a carry out and A not equal to B.

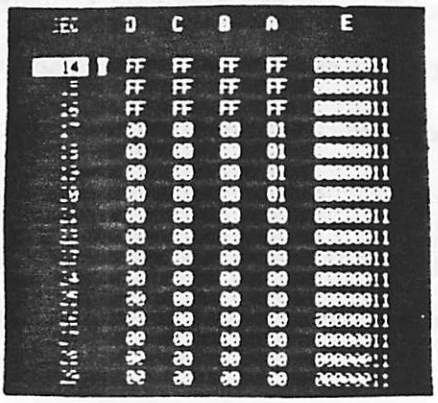


Figure C.3: DAS9100 ADD Test

SUB test:

```
FFFF FFFF 0001
FFFF FFFF 0000
FFFF FFFF 0002
FFFF FFFF 0000
```

alternate:

```
FFFF FFFE 0001
FFFF FFFE 0000
FFFF FFFD 0002
FFFF FFFD 0000
```

This is a worst-case subtract. F's in both A and B regs. Since the B reg. is complemented, this means that B really looks like: 0...0. In a SUB the carry_in bit is set to 1 and this injected carry forces carry propagation the length of the carry chain. In this case the carry is injected, not generated, so the first slow nibble is bypassed. Thus the worst case is seven bypass nibbles and one slow nibble. This SUB should be faster than the worst-case ADD. The final result at line 21 shows an output value of 0 with a carry out and A equal to B.

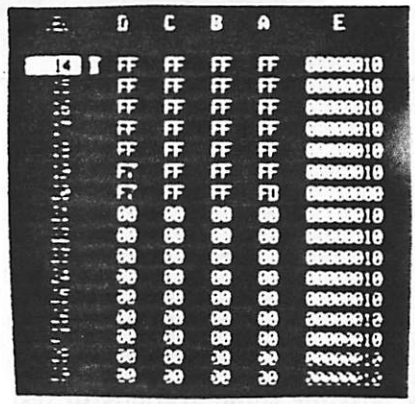


Figure C.4: DAS9100 SUB Test

The alternate SUB stops the carry_in propagation and instead forces a generate at bit 1. This is contrasted with the forced generate at bit 0 for the worst ADD case. Thus this situation should also be faster than the worst-case ADD. It provides an alternate worst-case path for the SUB operation.

XOR test:

9393 3939 0001
9393 3939 0000
3939 9393 0002
3939 9393 0000

This is one of many possible configurations to test this logic operation. The resultant pattern should be alternate 1's and 0's- in this case: AAAA AAAA. The 1/0 pattern of the test vector should also be complemented to make sure that all bits can be driven both high and low. The final result at line 22 shows an output value of AAAA AAAA with no

carry out and A not equal to B.

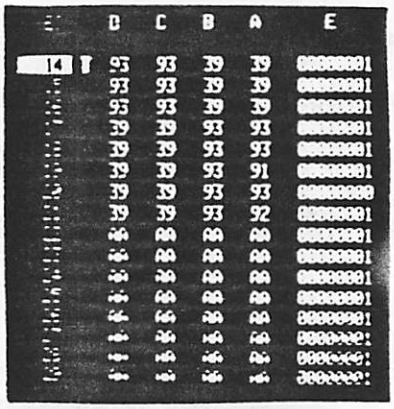


Figure C.5: DAS9100 XOR Test

AND test:

9C9C C9C9 0001
9C9C C9C9 0000
C9C9 9C9C 0002
C9C9 9C9C 0000

This is one of many possible configurations to test this logic operation. The resultant pattern should be a one 1 and three 0's, in this case: 8888 8888. The 1/0 pattern of the test vector should also be rotated through its four possible configurations. This particular pattern gives a 1 in the high-order which is the slowest to change since the AND control signal must travel through 32 bit slices to reach this bit. The final result at line 21 shows an output value of 8888 8888 with a carry out and A not equal to B.

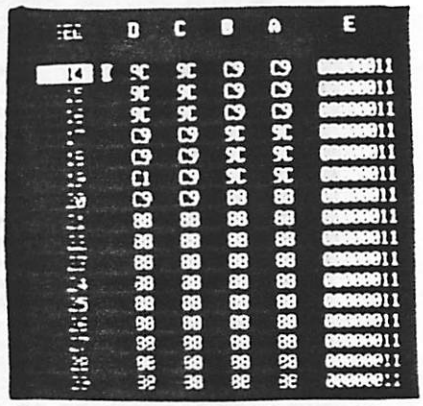


Figure C.6: DAS9100 AND Test

OR test:

3636 6363 0001
3636 6363 0000
6363 3636 0002
6363 3636 0000

This is one of many possible configurations to test this logic operation. The resultant pattern should be a one 0 and 3 1's, in this case: 7777 7777. The 1/0 pattern of the test vector should also be rotated through its four possible configurations. This particular pattern complements the above AND test pattern, so these two patterns ensure that all 32 bits can be driven both high and low. The final result at line 21 shows an output value of 7777 7777 with no carry out and A not equal to B.

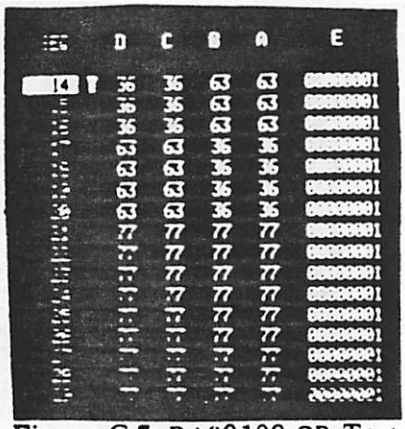


Figure C.7: DAS9100 OR Test

SR test:

AAAA	AAAA	0001
AAAA	AAAA	0000
7654	3210	0002
7654	3210	0000

This is one of many possible configurations to test the shift right logic operation. The resultant pattern should be an alternating 0/1 pattern, shifted right by one bit position, in this case: 5555 5555. The input vector is also a 1/0 pattern, to test that all bits toggle. The complemented vector should also be used to test that the bits can be toggled in the opposite direction. The argument in the B reg. is simply a foil. It must not affect the result. The final result at line 20 shows an output value of 5555 5555 with no carry out and the *A—B=0_L* line in its precharged state (A equal to B). This signal is not valid on SR and PASS.

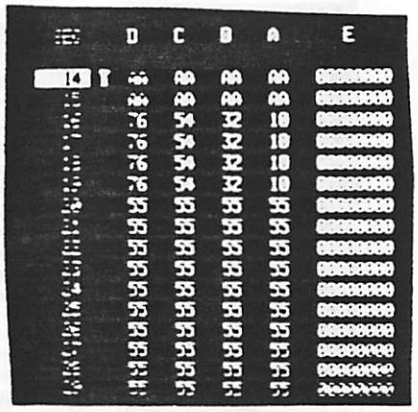


Figure C.8: DAS9100 SR Test

PASS test:

AAAA	AAAA	0001
AAAA	AAAA	0000
5555	5555	0002
5555	5555	0000

This is one of many possible configurations to test this logic operation. The resultant pattern should be the contents of reg A., in this case the alternating 0/1 pattern: AAAA AAAA. The complemented pattern should also be tested to ensure that all bits can be driven both high and low. The contents of reg. B are a foil, and must not affect the result. The final result at line 22 shows an output value of AAAA AAAA with no carry out and the *A—B=0_L* line in its precharged state (A equal to B). This signal is not valid on SR and PASS.

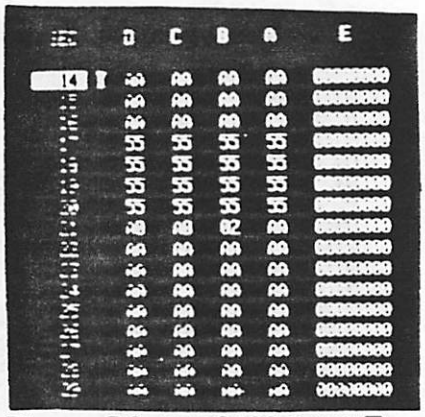


Figure C.9: DAS9100 PASS Test

Carry Propagate test:

ADD:			SUB:		
FFFF	FFFF	0001	FFFF	FFFF	0001
FFFF	FFFF	0000	FFFF	FFFF	0000
0000	0000	0002	FFFF	FFFF	0002
0000	0000	0000	FFFF	FFFF	0000

The *Cout_H* signal becomes valid before bit 31 is valid in ADD and SUB operations. This is because separate circuitry, the fast carry bypass, generates this signal. These two vectors test the worst *Cout_H* cases. The SUB case should be faster because it begins with carry injection. In the ADD case, by contrast, the carry must be generated at bit 0. The difference in the delays is the difference in the delay through a 4-bit bypass and full 4-bit ripple-carry circuitry. Figure C.10 shows the carry propagate test for the SUB instruction. The final result at line 20 shows an unsettled output value of FEC8 0000 with a carry out and A equal to B.

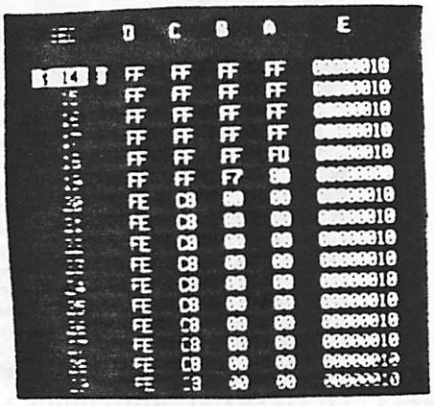


Figure C.10: DAS9100 Cp-SUB Test

Minimum Carry Arithmetic test:

ADD:			SUB:		
0000	0000	0001	0000	0000	0001
0000	0000	0000	0000	0000	0000
0000	0000	0002	FFFF	FFFF	0002
0000	0000	0000	FFFF	FFFF	0000

These tests represent the best-case ADD and SUB operations respectively. Many other vectors would also give this result. In the ADD test no carries are generated. in the SUB test there is a carry into the least significant bit (bit 0) as a result of carry injection. Outputs are precharged to 1 as a result of the Domino logic. thus the tests shown here are the most sensitive no-carry tests— the ADD result should be: 0000 0000; the SUB result: 0000 0001. Figure C.11 shows the minimum carry test for the SUB instruction. The final result at line 21 shows an output value of 0000 0001 with no carry out and A not equal to B.

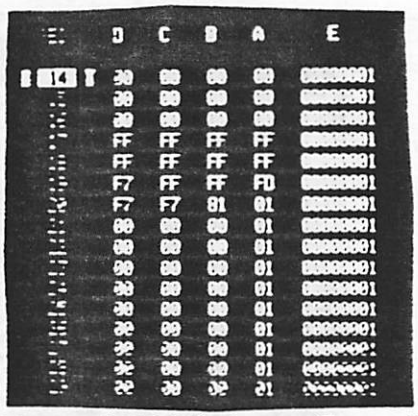


Figure C.11: DAS9100 Minimum Carry SUB Test

Figure C.12 shows a pulse generated from by the HP 8011A pulse generator. The x—scale is $20ns$ per division. the y—scale is 1.0V per division. The pulse shown has a duration of $100ns$ measured from the transition between the 2.5V rising edge to the 2.5V falling edge.

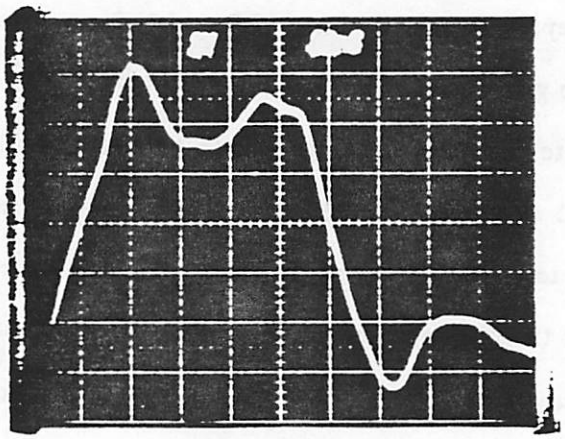


Figure C.12: HP8011A $100ns$ Pulse

Results of ALU Timing Tests

The best results from the table in Figure C.13 appear in Chapter 3. The full timing tests are presented below. If, in a given test, a particular chip does not appear it is an indi-

cation that it was completely non-functional under that particular test vector.

Function	Chip	Delay (*ns*)	Remarks
ADD	1	140	inter-carry bit 4 bad
	3	140	
	4	150	
SUB	1	130	inter-carry bit 4 bad, both SUB tests
	3	135	inter-carry bit 4 bad, both SUB tests
	4	125	
XOR	1	95	
	2	90	
	3	95	
	4	95	
AND	1	85	
	2	85	bit 11 stuck at 0
	3	95	
	4	95	
OR	1	85	
	2	80	low 16 bits not available
	3	80	bit 8 stuck at 0
	4	85	bit 8 stuck at 0
SR	1	50	
	2	50	
	3	45	complementary tests, bit 8 stuck at 0
	4	50	AAAA vector, bit 8 stuck at 0
		45	5555 vector
	6	35	complementary tests
PASS	1	45	
	2	50	
	3	50	
	4	50	
	6	35	
Cout_H	1	100	ADD test
		90	SUB test
	2	90	SUB test
	3	100	ADD/SUB test
	4	100	ADD test
Min Carry	1	95	ADD/SUB test
	2	95	ADD/SUB test
	3	95	ADD/SUB test
	4	95	ADD test

Figure C.13: Measured ALU Delays

APPENDIX D

MAMBO Source Listing

This appendix contains the C language source code for all the programs in the MAMBO package. To obtain this program contact Deborah Dunster at the following address:

EECS Industrial Liaison Program
457 Cory Hall
University of California
Berkeley, CA 94720

References

[asan82] T. Asano, "An Optimum Gate Placement Algorithm for MOS One-Dimensional Arrays", *Journal of Digital Systems*, vol. VI, no. 1, 1982, pp. 1-26, Computer Science Press, Inc..

[ayre79] R. Ayres, "Silicon Compilation- A Hierarchical Use of PLAs", *Proc. 16th Design Automation Conference*, June 1979, pp. 314-326.

[bray82] R. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions", *Proc. International Symposium on Circuits and Systems*, May 1982, pp. 49-54.

[bray84a] R. Brayton, C. Chen, C. McMullen, R. Otten and Y. Yamour, "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", *Proc. 1984 IEEE Custom Integrated Circuits Conference*, May 1984, pp. 346-355.

[bray84b] R. Brayton and C. McMullen, "Synthesis and Optimization of Multistage Logic", *Proc. 1984 International Conference on Computer Design*, October 1984, pp. 23-30.

[bray84c] R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984. ISBN 0-89838-164-9.

[carr72] W. Carr and J. Mize, *MOS/LSI Design and Application*, McGraw-Hill, 1972. ISBN 0-07-010081-0.

[cmel81] R. Cmelik, "Program EQNTOTT", Internal Memorandum, Department of EECS, University of California, Berkeley, 1981.

[demi82] G. DeMicheli and A. Sangiovanni-Vincentelli, "PLEASURE: A Computer Program for Simple/Multiple Constrained/Unconstrained Folding of Programmable Logic Arrays", UCB/ERL M82/57, Electronics Research Laboratory, University of California, Berkeley, August 1982.

[demi83] G. DeMicheli and M. Santomauro, "SMILE: A Computer Program for Partitioning of Programmed Logic Arrays", *Computer-Aided Design*, March 1983, pp. 89-97.

[demi84] G. DeMicheli, "Computer-Based Synthesis of PLA-Based Systems", UCB/ERL M84/31, Electronics Research Laboratory, University of California, Berkeley, April 1984.

[dunl83] A. Dunlop, "Automatic Layout of Gate Arrays", *Proc. 1983 International Symposium on Circuits and Systems*, vol. 3, May 1983, pp. 1245-1248.

[egan82] J. Egan and C. Liu, "Optimal Bipartite Folding of PLAs", *Proc. 19th Design Automation Conference*, June 1982, pp. 141-146.

[fitz82] D. Fitzpatrick, "Mextra: A Manhattan Circuit Extractor", UCB/ERL M82/42, Electronics Research Laboratory, University of California, Berkeley, May

1982.

[flei75] H. Fleisher and L. Maissel, "An Introduction to Array Logic", *IBM Journal of Research and Development*, vol. 19, March 1975, pp. 98-109.

[frie84] V. Friedman and S. Liu, "Dynamic Logic CMOS Circuits", *IEEE Journal of Solid-State Circuits*, vol. SC-19, no. 2, April 1984, pp. 263-266.

[glas80] L. Glasser and P. Penfield, "An Interactive PLA Generator as an Archtype for a New VLSI Design Methodology", *Proc. IEEE International Conference on Circuits and Computers*, October, 1980, pp. 608-611.

[gonc83] N. Goncalves and H. DeMan, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures", *IEEE Journal of Solid-State Circuits*, vol. SC-18, no. 3, June 1983, pp. 261-266.

[gris82] T. Griswold, "Portable Design Rules for Bulk CMOS", *VLSI Design*, September/October 1982, pp. 62-67.

[hach80] G. Hachtel, A. Sangiovanni-Vincentelli and R. Newton, "Some Results in Optimal PLA Folding", *Proc. IEEE International Conference on Circuits and Computers*, October, 1980, pp. 1023-1027.

[hach82] G. Hachtel, R. Newton and A. Sangiovanni-Vincentelli, "An Algorithm for Optimal PLA Folding", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. CAD-1, no. 2, April 1982, pp. 63-77.

[hash71] A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *Proc. of the 8th Design Automation Conference*, June 1971, pp. 155-169.

[hell84] L. Heller, W. Griffin, J. Davis and N. Thoma, "Cascode Voltage Switch Logic—A Differential Logic Family", *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, 1984, pp. 16-17.

[hodg83] D. Hodges and H. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw-Hill, 1983. ISBN 0-07-029153-5.

[hofm80] M. Hofmann, "A Method for Topological Compaction of Programmed Logic Arrays", Master's Report, Department of EECS, University of California, Berkeley, December 1980.

[hofm83] M. Hofmann, "Aspects of Design and Layout of a CMOS ALU for SOAR", Proceedings of CS292X, Department of EECS, University of California, Berkeley, December 1983.

[horo84] M. Horowitz, "Timing Models for MOS Circuits", PhD Dissertation, Integrated Circuits Laboratory, Stanford University, Stanford, January 1984.

[ishi83] M. Ishii, "GMMG: A System for Generating CMOS Gate Matrices", Internal Memorandum, Department of EECS, University of California, Berkeley, August 1983.

[kang83a] S. Kang, R. Krambeck, H. Law and A. Lopez, "Gate Matrix Layout of Random Control Logic in a 32-bit CMOS CPU Chip Adaptable to Evolving Logic Design", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol.

CAD-2, no. 1, January 1983, pp. 18-29.

[kang83b] S. Kang and H. Law, "Automation of VLSI Gate-Matrix Layout", *Proc. IEEE International Symposium on Circuits and Systems*, vol. 3, May 1983, pp. 1017.

[kram82] R. Krambeck, C. Lee and H. Law, "High-Speed Compact Circuits with CMOS", *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 3, June 1982, pp. 614-619.

[land82] H. Landman, "Automatic Layout of Optimized PLA Structures", UCB/ERL M82/64, Electronics Research Laboratory, University of California, Berkeley, September 1982.

[latt81] W. Lattin, J. Bayliss, D. Budde, S. Colley, G. Cox, A. Goodman, J. Rattner, W. Richardson and R. Swanson, "A 32-bit VLSI Micromainframe Computer System", *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, New York, NY, February, 1981, pp. 110-111.

[law83] H. Law, "Gate Matrix: A Practical, Stylized Approach to Symbolic Layout", *VLSI Design*, September 1983, pp. 49-59.

[lin84] T. Lin and C. Mead, "Signal Delay in General RC Networks", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. CAD-3, no. 4, October 1984, pp. 331-349.

[lope81] A. Lopez and H. Law, "A Dense Gate Matrix Layout Method for MOS VLSI", *IEEE Journal of Solid-State Circuits*, vol. SC-15, no. 4, August 1981, pp. 736-740.

[luby82] M. Luby, U. Vazirani, V. Vazirani and A. Sangiovanni-Vincentelli, "Some Theoretical Results on the Optimal PLA Folding Problem", *Proc. IEEE International Conference on Circuits and Computers*, New York, NY, October, 1982, pp. 165-170.

[mah84] G. Mah, "PANDA: A PLA Generator for Multiply-Folded PLAs", UCB/ERL M84/95, Electronics Research Laboratory, University of California, Berkeley, April 1984.

[mari85] C. Marino, "Smalltalk on a RISC - CMOS Implementation", Master's Report, Department of EECS, University of California, Berkeley, March, 1985.

[mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Inc., 1980. ISBN 0-201-04358-0.

[mosi82] Mosis, "New Standard CMOS/Bulk 3-Micron CMOS Design Rules", The MOSIS Project, University of Southern California, Information Sciences Institute, August 1982.

[nage75] W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", UCB/ERL M75/520, Electronics Research Laboratory, University of California, Berkeley, May 1975.

[nagl75] H. Nagle, B. Carroll and J. Irwin, *An Introduction to Computer Logic*, Prentice-Hall, Inc., 1975. ISBN 0-13-480012-5.

[newt81] R. Newton, D. Pederson, A. Sangiovanni-Vincentelli and C. Sequin, "Design Aids for VLSI: The Berkeley Perspective", *IEEE Transactions on Circuits and Systems*, vol. CAS-28, no. 7, July 1981, pp. 666-680.

[newt83] R. Newton, Private Communication., March, 1983.

[ohts79] T. Ohtsuki, H. Mori, E. Kuh, T. Kashiwabara and T. Fujisawa, "One-Dimensional Logic Gate Assignment and Interval Graphs", *IEEE Transactions on Circuits and Systems*, vol. CAS-26, no. 9, September 1979, pp. 675-684.

[oust83] J. Ousterhout, "Crystal: A Timing Analyzer for NMOS VLSI Circuits", *Proc. Third Caltech Conference on VLSI*, 1983, pp. 57-70. ISBN 0-914894-86-2.

[pati75] S. Patil, "An Asynchronous Logic Array", *Project MAC Tech. Memo TM-62*, May 1975.

[pati79] S. Patil and T. Welch, "A Programmable Logic Approach for VLSI", *IEEE Transactions on Computers*, vol. C-28, September 1979, pp. 594-601.

[patt81] D. Patterson and C. Sequin, "RISC I: A Reduced Instruction Set Computer", *ACM SIGARCH Proceedings of the 8th Annual Symposium on Computer Architecture*, vol. 9.3, May 1981, pp. 443-457.

[pomp82] M. Pomper, W. Beifuss, K. Horninger and W. Kaschte, "A 32-bit Execution Unit in Advanced NMOS Technology", *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 3, June 1982, pp. 533-538.

[pret85] J. Pretorius, A. Shubat and C. Salama, "Analysis and Design Optimization of Domino CMOS Logic with Application to Standard Cells", *IEEE Journal of Solid-State Circuits*, vol. SC-20, no. 2, April 1985, pp. 523-530.

[pugh85] B. Pugh, Private Communication., January 1985.

[risc82] R. Risch, "Staggered Input Networks: An Approach to Automatic Logic Decomposition", *Proc. International Symposium on Circuits and Systems*, May 1982, pp. 55-57.

[rube83] J. Rubenstein, P. Penfield and M. Horowitz, "Signal Delay in RC Tree Networks", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. CAD-2, no. 3, July 1983, pp. 202-211.

[rude85] R. Rudell and A. Sangiovanni-Vincentelli, "ESPRESSO-MV: Algorithms for Multiple-Valued Logic Minimization", *Proc. Custom Integrated Circuits Conference*, May, 1985.

[sans81] W. Sansen, W. Heyns and H. Beke, "Layout Automation Based on Placement and Routing Algorithms", in *Computer Design Aids for VLSI Circuits*, P. Antognetti, D. O. Pederson and H. DeMan (editor), Sijthoff and Noordhoff, 1981, pp. 470. ISBN 90-286-2701-4.

[schm80] M. Schmookler, "Design of Large ALUs Using Multiple PLA Macros", *IBM Journal of Research and Development*, vol. 24, no. 1, January 1980, pp. 2-14.

[shoj82] M. Shoji, "Electrical Design of BELLMAC-32A Microprocessor", *Proc. IEEE International Conference on Circuits and Computers*, September-October 1982, pp. 112-115.

[shoj85] M. Shoji, "FET Scaling in Domino CMOS Gates", *Proc. International Symposium on Circuits and Systems*, June 1985.

[sisk82] J. Siskind, J. Southard and K. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions", *Proc. Conference on Advanced Research in VLSI*, January 1982, pp. 28-40.

[smit82] K. Smith, T. Carter and C. Hunt, "Structured Logic Design of Integrated Circuits Using the Storage/Logic Array (SLA)", *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 2, April 1982, pp. 395-406.

[souk81] J. Soukup, "Circuit Layout", *Proceedings of the IEEE*, vol. 69, no. 10, October 1981, pp. 1281-1304.

[sout83] J. Southard, "MacPitts: An Approach to Silicon Compilation", *Computer Magazine*, December 1983, pp. 74-82.

[sout82] E. Soutschek, M. Pomper and K. Horninger, "PLA Versus Bit Slice: Comparison for a 32 Bit ALU", *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 3, June 1982, pp. 584-586.

[suzu73] Y. Suzuki, K. Odagawa and T. Abe, "Clocked CMOS Calculator Circuitry", *IEEE Journal of Solid-State Circuits*, vol. SC-8, December 1973, pp. 462-469.

[toku83] T. Tokuda, K. Okazaki, K. Sakashita, I. Ohkura and T. Enomoto, "Delay-Time Modeling for ED MOS Logic LSI", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. CAD-2, no. 3, July 1983, pp. 129-134.

[wein67] A. Weinberger, "Large Scale Integration of MOS Complex Logic: A Layout Method", *IEEE Journal of Solid-State Circuits*, vol. SC-2, no. 4, December 1967, pp. 182-190.

[whal84] S. Whalen, "CMOS Adder Designs for High Performance Microprocessors", Master's Report, Department of EECS, University of California, Berkeley, August 1984.

[wing85] O. Wing, "Automated Gate Matrix Layout", *Proc. 1985 International Symposium on Circuits and Systems*, June 1985.

[wyat83] J. Wyatt, C. Zukowski, L. Glasser, P. Bassett and P. Penfield, "The Waveform Bounding Approach to Timing Analysis of Digital MOS IC's", VLSI Memo No. 83-148, Department of EECS, Massachusetts Institute of Technology, Cambridge, July, 1983.