Advanced Research in VLSI

Proceedings of the Fourth MIT Conference
April 7–9, 1986

edited by Charles E. Leiserson

The MIT Press
Cambridge, Massachusetts
London, England

Exclusion Constraints,
a new application of Graph Algorithms to VLSI Design

Kevin Karplus
Computer Science Department
(also in the School of Electrical Engineering)
Cornell University
Ithaca, NY 14853

This paper presents a new paradigm for analyzing digital MOS circuits, based on boolean expressions for paths in the switch graph. A new circuit representation, the inverter graph, *is described.*

A new set of electrical design rules are introduced, based on path expressions in the switch graph and cycles in the inverter graph. A program has been written that generates the exclusion constraints *implied by the rules.*

Introduction

This paper presents a new set of design rules for switch circuits. Since the rules are based on graph algorithms and no information is needed about switch sizes or parasitics, the rules can be applied before layout. Problems are spotted early in the design, before corrections become expensive. The six rules in this paper are simple enough to be applied automatically.

Two graphs are used in checking the rules: the switch graph and the inverter graph. The switch graph is input by the user and the inverter graph is derived from it. Each graph is explained in more detail below.

The rules apply to both *n*MOS and *c*MOS circuits, and use a simple switch model of the circuits. Some useful circuits violate the rules, but these circuits are often not handled correctly by current CAD tools. By pointing out the places where the implicit assumptions of the tools are violated, the rules can focus attention on those parts of the circuit that need more detailed modeling.

Each rule generates boolean equations (*exclusion constraints*) that should be satisfied by the signals in the circuit. A program has been written to generate the constraints defined by the rules. Although the program can be applied to entire chips, it is probably better applied to individual sub-circuits before they have been assembled.

The next two sections of the paper discuss the switch graph and the inverter graph. The section after that discusses the representation of boolean expressions. The six rules are then introduced, and some comments are made about the program that generates constraints

Switch Graph and Path Expressions

The first five rules are checked on a *switch graph*, in which each signal is a vertex, and each switch is an edge connecting the source and drain. Edges are labeled with the signal on the gate of the switch (negated if the switch is *p*MOS). For switches in parallel, the multiple edges are merged into a single edge whose label is the OR of the labels for each switch. Static load devices (pullups) are included in the switch graph, but are specially marked as they need different treatment. A path in the switch graph represents a possible DC connection between two signals. Figure 1 shows a *c*MOS nand-gate and its switch graph.

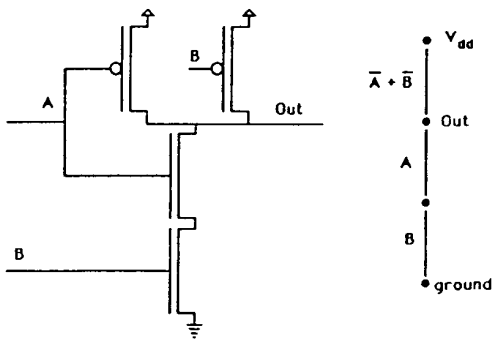

Figure 1: *c*MOS nand-gate and its switch graph.

The switch graph is a standard representation for MOS circuits, used by switch simulators, electrical rule checkers, and other tools. For compatibility with other tools, the exclusion constraint generator uses esim[8] format for input of the switch graph.

The rules are based on boolean expressions that summarize paths in the graph. Certain expressions are so useful that they are precomputed for all nodes. These expressions are the ones that summarize all paths from any node n to V_{dd} (V_n), ground (G_n), a pullup (P_n), a high input ($In1_n$), or a low input ($In0_n$).

The expressions can be quickly generated using LU-decomposition on the adjacency matrix for the switch graph[7]. Each element of the matrix is the label for the edge between the corresponding vertices ($M_{i,j}$ = label(i,j)). Since each vertex is trivially connected to itself, we add the diagonal ($M_{i,i} = 1$)). Because each switch is bi-directional, the graph is undirected and the matrix is symmetric.

Each element of M is an expression for paths of length 0 or 1 between the corresponding vertices. All paths from any vertex to any other vertex could be found by taking the transitive closure of M, but the resulting matrix may contain expressions too large for practical applications. We can find all paths to a particular node more cheaply. Let E_i be the vector with all zeros except a 1 in the i^{th} position. If we solve $Mx = E_i$, replacing each multiplication with an AND and each addition or subtraction with an OR, the solution vector will consist of all paths to vertex i. That is, x_j will be all paths from j to i.

When generating paths, we are not interested in paths that go through ground or V_{dd}. One way to eliminate such spurious paths from consideration would be to make the edges touching the power supplies directed edges (for example, edges might be directed out of V_{dd} and into ground). Unfortunately, directing edges makes the adjacency matrix for the graph asymmetric, and sparse matrix algorithms are much simpler for symmetric matrices (Cholevsky factorization rather than general LU-decomposition). This is the only instance I know where the bi-directionality MOS switches makes a CAD algorithm simpler.

The approach taken in the constraint generator is to partition the edges of the switch graph into four sets: the edges incident on ground, the pullups, the other edges incident on V_{dd}, and all other switches. The last group of switches are all potentially bi-directional, so have a symmetric adjacency matrix. LU-decomposition is done only on the symmetric matrix. To compute paths from all nodes (except V_{dd}) to ground we solve $MG = Gswitch$ for G, where each $Gswitch_i$ is the label on the edge from i to ground. Paths from all nodes to V_{dd} can be computed similarly ($MV = Vswitch$ and $MP = Pullup$). The

paths from V_{dd} to ground can be computed by either the dot product $G \cdot Vswitch$ or $V \cdot$ Gswitch. Note that the adjacency matrix needs to be factored only once, but that several right-hand sides may need to be solved for.

The constraint generator currently uses a slightly modified version of SPARSPAK[2] to do the matrix factoring and the solving to get path expressions. Eventually the algorithms will be re-written in C, so that the adjacency matrix and temporary arrays can be dynamically allocated. The SPARSPAK sparse matrix representation is compact and allows rapid solving for path expressions. Unfortunately, finding all neighbors of a given node (necessary for the charge-sharing rule) is slow. Furthermore, the program uses the SPARSPAK interface to build the representation, so the matrix itself is not accessible. Rewriting the sparse matrix algorithms in C should alleviate the second problem, and finding all neighbors will probably not be the bottleneck in the final tool.

Two considerations are important when using sparse matrix techniques: how sparse is the original matrix? and how sparse are the factors? Our original matrices are very sparse, with approximately as many edges as vertices. Although the degree of some nodes is high (buses for example), most nodes have degree zero or one after the V_{dd} and ground switches have been removed.

Fill-in, the difference between the number of non-zeros in the factors of the matrix and the original matrix, is a widely studied measure for sparse matrix methods. Vertex-ordering techniques are used with sparse matrix methods to try to minimize fill-in. One popular technique, quotient minimum degree (QMD), is a simple greedy method that chooses vertices one at a time, always picking the one that introduces the least immediate fill-in. Since choosing a degree 0 or 1 node in a quotient graph introduces no fill-in, QMD generates no fill-in for forests. After removing V_{dd} and ground switches, the switch graph is often a forest. Besides trees, one expects to find series-parallel graphs as components of switch graphs. When starting from a series-parallel graph, QMD never has to choose a vertex with degree higher than two, so the fill-in at most doubles the density of the matrix. (Sketch of proof: a series-parallel graph always has a vertex of degree 2 or less. The quotient graph after selecting a degree one or two vertex from a series-parallel graph is again series-parallel.)

Despite all the nice properties of QMD, the constraint generator uses nested-dissection to order the vertices for factoring, even though nested-dissection generates more fill-in. This is not because the QMD algorithm is slower, since little time is spent on the vertex ordering, but because nested dissection works better empirically. Fill-in, although an appropriate cost measure for numerical work, is not the most appropriate one for generating path expressions. Most of the program's running time is spent manipulating boolean expressions, and the boolean operations take time roughly proportional to the square of the length of the boolean expression. Thus vertex ordering to minimize the size of the intermediate expressions is more important than minimizing fill-in. Nested dissection does almost as well as QMD at reducing fill-in, but its divide-and-conquer approach generally gives smaller intermediate expressions.

Inverter Graph

Although paths in the switch graph provide sufficient information for five of the six rules presented below, the sixth rule applies to a different graph, the *inverter graph*. Vertices of the inverter graph correspond to signals, and the directed edges to subcircuits that act as static inverters. Each edge is labeled with the conditions needed to make the subcircuit act as an inverter. Figure 2 shows inverter graphs for some simple subcircuits.

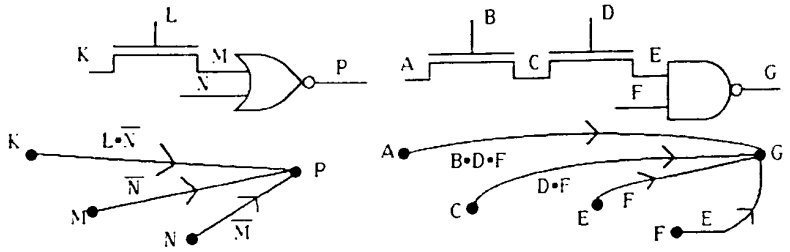

Figure 2: Inverter graphs.

The inverter graph can be generated from the switch graph. For every pair of vertices (*src* and *dest*), generate a directed edge with a label that expresses when *dest* will be directly computed as the inverse of *src*. That is, *dest* should be connected to V_{dd} and not to ground when *src* is low, and to ground, but not V_{dd} when *src* is high. Making minor modifications for inverters powered by inputs and for pullups,

we can compute the label from the switch-graph paths needed to check the other rules:

$$\begin{aligned} L(src, dest) = \; & (G_{dest} \vee In0_{dest})\big|_{src=1} \\ & \wedge(\neg G_{dest})\big|_{src=0} \wedge (\neg In0_{dest})\big|_{src=0} \\ & \wedge(V_{dest} \vee P_{dest} \vee In1_{dest})\big|_{src=0} \\ & \wedge(\neg V_{dest})\big|_{src=1} \wedge (\neg In1_{dest})\big|_{src=1} . \end{aligned}$$

The expression $L(src, dest)$ is not quite correct for labeling edges in the inverter graph, since the paths to *dest* may involve switches controlled by *dest*. The label used by the constraint generator is:

$$label(src, dest) = L(src, dest)\big|_{dest=1} \wedge L(src, dest)\big|_{dest=0}.$$

The procedure described above for generating the inverter graph looks at all vertex pairs. In the constraint generator, a more efficient technique is used. For each destination, only those vertices that appear in the expression for paths to ground $(G_{dest} \vee In0_{dest})$ are considered for sources. This eliminates only edges whose labels are obviously false. Furthermore, only vertices that appear on the gates of transistors or as outputs to the circuit are used as destinations. This restriction generates a subgraph of the inverter graph, but any cycles in the full inverter graph will have corresponding cycles in this subgraph.

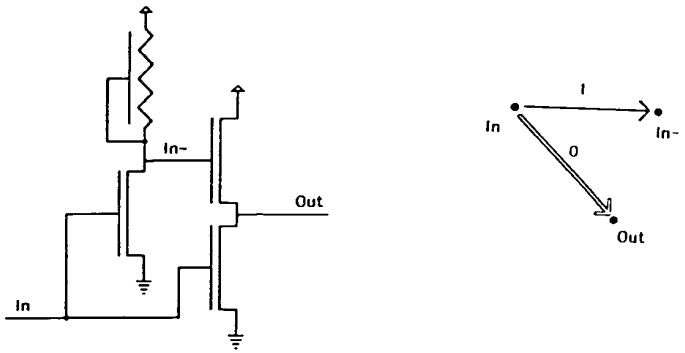

Figure 3: Push-pull inverting amplifier and incorrectly generated inverter graph.

The main applications of the inverter graph involve finding feedback paths in the circuit (cycles in the graph). Unfortunately, the inverter graph computed above does not include all circuits that can act as inverters, and some feedback paths may pass through non-inverting amplifiers. Figure 3 shows an *n*MOS push-pull inverting amplifier and the corresponding computed inverter graph. The inverter graph computation assumes that the source varies independently of the other inputs to the subcircuit controlling the destination. When some other input depends on the source (as In- does in the figure), inverter edges may be missed. Improved methods for computing the inverter graph are being sought.

The inverter graph may be useful for more than just finding troublesome feedback paths. It has potential for parsing switch graphs into higher-level circuit descriptions. In particular, it seems promising for recognizing static and pseudo-static memory elements. A static memory element can be identified as an even cycle in the inverter graph, whose edge labels are simultaneously true. A pseudo-static memory element also corresponds to an even inverter cycle, but not all edges need be true at once. The vertices in the cycle that do not have true in-edges must be dynamic storage nodes, and all edges must be true sufficiently often. Dynamic memory elements can be identified in the switch graph (any gate isolated from all other nodes).

Boolean Expression Representation

Most of the time and memory used by the constraint generator is for manipulating boolean expressions. Several different data representations have been tried for the boolean expressions, but none are really satisfactory.

The first attempted representation was a simple and-or tree with no simplification or normalization. The expressions quickly became huge and incomprehensible.

The second attempted representation was a simple Disjunctive Normal Form (OR of ANDs). DNF has the advantage that each path generates one term, and many of the rules simply enumerate a few paths, so the expressions are fairly readable. Without simplification, DNF expressions are still rather large, since obviously false paths such as the V_{dd}-to-ground paths in static cMOS are included.

The third attempted representation was an if-then-else tree with *ad hoc* simplification[4]. The expressions were often of reasonable size, but so deeply nested they were incomprehensible. Such a representation may be suitable for internal manipulation, but is not adequate for expressing constraints to a user.

The fourth attempted representation was a modified form of DNF, which I call sorted-DNF. The terms are sorted in increasing order of length, and the literals in each term are sorted in numeric order. The ordering of the literals is arbitrary, but fixed for all expressions. No term is included if any subset of its literals is already a term of the expression. A list of *a priori* excluded terms is checked before a term is added to an expression. This allows the user to provide some information about mutually exclusive signals, avoiding computing expressions that include paths known to be uninteresting. Since redundant terms are eliminated at every boolean operation, expressions remain fairly small. To save memory, the sorted-DNF expressions are stored in a format similar to sparse matrix representations. The expression is stored as an array of pointers to the beginning of each term in an array of literals. The arrays are dynamically allocated based on estimates of the expression size. Figure 4 shows an example of the data structure.

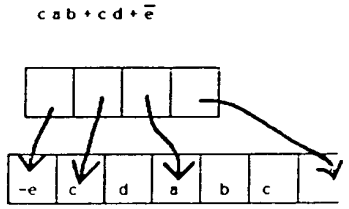

Figure 4: Sorted-DNF representation for $cab + e + cd$

Although checking the terms is expensive, the reduction in expression size seems to more than compensate. Some preliminary tests were run with the simplification done only before output, rather than at every boolean operation, and the delayed simplification made the program at least two or three times slower.

The sorted-DNF representation seems adequate for OR operations, and marginally acceptable for AND operations, but negations are quite expensive. Another variant of if-then-else trees has been implemented, using Randy Bryant's canonical form[1]. Unfortunately, this format is very sensitive to the ordering of the literals, so is not suitable for representing many expressions of unknown form. Generating human-readable expressions from if-then-else trees is also a challenging problem, though more tractable with the canonical DAGs than with arbitrary if-then-else trees.

In tests involving mainly AND and OR operations, Bryant's format proved to be both slower and bulkier than sorted-DNF. Tests involving extensive negation have not been done yet. The current version of the constraint generator can be compiled to use either sorted-DNF or Bryant's canonical form, but eliminating *a priori* excluded paths is not implemented for Bryant's canonical form.

The Rules

The rules for generating constraints are still fairly primitive. They are an attempt to capture various rules-of-thumb in a more formal system. Violating a constraint generated by a rule does not necessarily mean a circuit is unusable, but careful analysis or analog simulation is needed to ensure digital operation. Some of the rules point out potential problems with a circuit, others point out weaknesses in existing CAD tools.

The rules have been deliberately kept few and simple. This is in direct contrast to expert system approaches, in which complex rules are allowed to proliferate. Undoubtedly a more "intelligent" system could be built with many special-case rules, but the aim of this research is to explore the potential of a particular paradigm (path expressions) rather than to thoroughly analyze one or two circuits. Eventually we hope to define some provable correctness properties for MOS circuits.

The rules can be summarized as follows:

Rule 1: Avoid shorting power.
Rule 2: Avoid changing the inputs.
Rule 3: Avoid parallel pullups.
Rule 4: Avoid gates in the middle of pulldown chains.
Rule 5: Avoid charge-sharing.
Rule 6: Avoid odd inverter cycles.

Rule 1 is the primary well-formedness criterion for static cMOS and pre-charged circuits. Rule 2 prevents sneak paths that can cause

modules to behave incorrectly when connected. Rule 3 points out where special inverter ratio computations are needed. Rule 4 detects un-intentional bi-directional pass transistors. Rule 5 detects some situations in which circuit behavior is not digital. Rule 6 detects oscillation and some non-digital circuit behavior.

Rule 1: Avoid shorting power.

Every path in the switch graph from V_{dd} to ground must have at least one open switch on it (that is, $G_{V_{dd}} = 0$). Rule 1 generates constraints mainly for cMOS circuits and pre-charged circuits. For cMOS, the power-short constraints are the primary criteria for the well-formedness of logic gates. For pre-charged-circuits (nMOS or cMOS), the power-short constraints are usually satisfied by using a clocking discipline[3,5].

For the cMOS nand gate in Figure 1 rule 1 requires the tautology $(\neg A + \neg B) \cdot A \cdot B = 0$.

Rule 2: Avoid changing the inputs.

Connections are prohibited from a low input of a circuit to V_{dd} or a pullup, from a high input to ground, or from an input to an input with a different value, that is:

$$\forall \text{ inputs } n: \quad n \cdot (G_n + In0_n) = 0$$
$$\neg n \cdot (V_n + P_n + In1_n) = 0.$$

This rule tries to prevent "sneak paths" back through the inputs of a circuit. If such sneak paths are allowed, the behavior of a system can not be reliably computed from the behavior of individual subcircuits. Not all circuits that violate rule 2 are wrong. For example, datapath modules commonly use buses for both input and output. Note that this rule is best checked on small components of a chip, since the chip as a whole will rarely violate it.

Figure 5 shows a transmission-gate exclusive-or circuit that causes problems in many switch simulators. Rule 2 generates two constraints, $A \cdot A\text{-} \cdot B = 0$ to prevent shorting high input B to ground, and $A \cdot A\text{-} \cdot \neg B = 0$ to prevent shorting high input A to low input B. Since A- is delayed with respect to A, problems should be expected

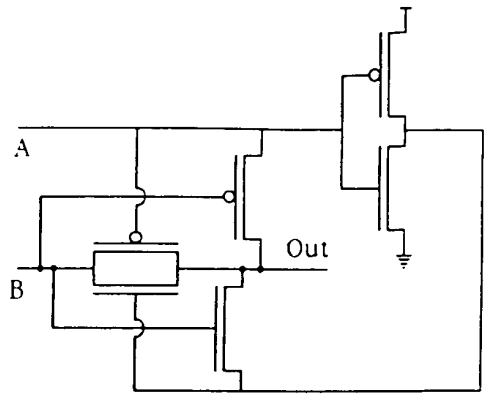

Figure 5: Transmission-gate exclusive-or that violates rule 2.

when B is high and A rises (the first constraint), and possibly when B is low and A rises (the second constraint).

Problems arise in switch simulation because the vertex for A- is on the path from B to ground, making A- sometimes unknown. The unknown value propagates forward through *Out* and backward through B and A. If A and B are forced by the simulator, the backward propagation is not detected until the circuit is used as part of a larger circuit. Rule 2 may need to be modified to flag cases where a vertex on a path is used as an atom in the path expression.

Rule 3: Avoid parallel pullups.

No signal vertex may be simultaneously connected to two different pulled-up nodes and to ground. This rule is intended to make the usual simple pullup over pulldown ratio calculations accurate. Most inverter ratio checking programs examine each pullup independently, and determine the maximum resistance of any pulldown path to ground. If two pullups are connected in parallel, their saturation currents add, and any pulldowns carrying the combined current must be wider. Instead of checking rule 3, ratio checking programs could be modified to take possible parallel pullups into consideration.

The constraints generated by rule 3 for an array of the two-port dynamic RAM cells of Figure 6 can be satisfied by putting some restrictions on ram usage:

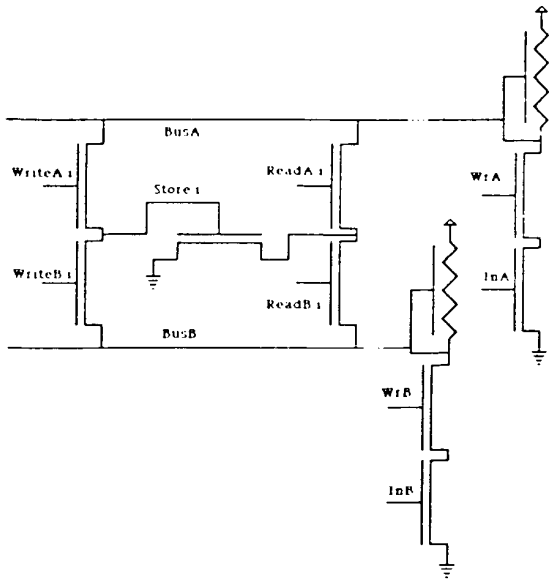

Figure 6: Two port RAM.

- Simultaneous reads and writes on the same bus are prohibited.
- Writing into a cell from both buses simultaneously is prohibited.
- Reading the same cell from both buses simultaneously is prohibited when the storage transistor is on.

Although the last seems an unnatural restriction, the storage pulldown would have to be wider than usual if both buses read from it at the same time.

Rule 4: Avoid gates in the middle of pulldown chains.

If the gate of a transistor is connected to a pulldown tree, the connection should be through the pulled-up node, not through a lower node in the tree. For *c*MOS, a gate connected to either an *n*MOS pulldown tree or a *p*MOS pullup tree should be connected through the node where the two trees meet.

The circuit shown in Figure 7 will be interpreted by some programs as having a uni-directional information flow through the pass transistor. This is usually intended, but is correct only if **PASS** and **CLEAR** are mutually exclusive.

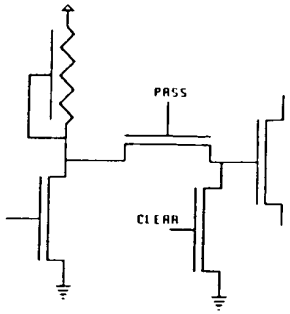

Figure 7: Gate in middle of pulldown chain.

Rule 5: Avoid charge-sharing.

A signal that is used on the gate of some transistor must be either isolated from all other nodes (storing charge) or connected to V_{dd}, ground, or a pulled-up node. When two nodes isolated from power become connected to each other, the charge on the nodes is shared between them. If the nodes initially have different values, the result may be an illegal intermediate voltage. Static storage nodes (nodes on even cycles in the inverter graph) also must be isolated to avoid charge-sharing.

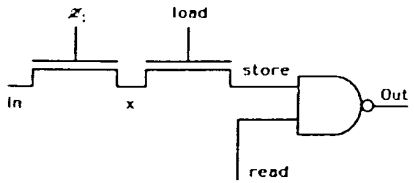

Figure 8: Charge-sharing violation.

The circuit in Figure 8 illustrates the charge-sharing rule. The input is stored at node *STORE* when *LOAD* is high during phase 1. *LOAD* may go high before ϕ_1, sharing the charge between *STORE* and *X*. If the *LOAD* and *READ* signals are simultaneously high, the illegal value can be propagated to the output. The generated constraint is $\neg LOAD + (LOAD \cdot \phi_1) = 1$.

Rule 6: Avoid odd inverter cycles.

In the inverter graph, at least one edge in each odd cycle must be inactive. This rule prohibits circuits that oscillate or present intermediate voltage outputs. Such circuits are particularly troublesome for switch simulators, which can get stuck in infinite loops looking for the non-existent digital equilibrium state. Ring oscillators and RAM bias generators routinely violate rule 6, and need detailed analog simulation to ensure correct design. Figure 9 shows a circuit with an odd inverter cycle, which generates the constraint $\boldsymbol{B} \cdot \boldsymbol{E} = 0$.

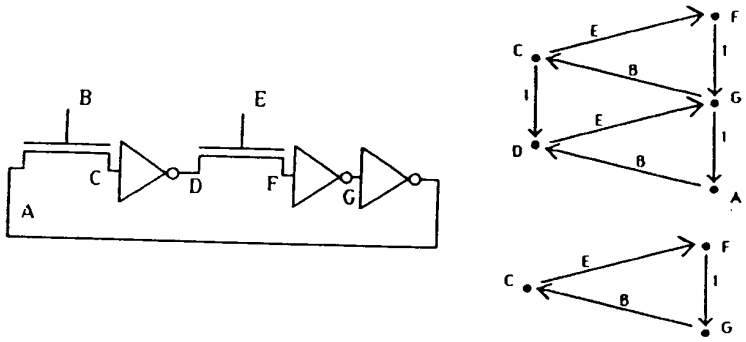

Figure 9: Odd inverter cycle, with full inverter graph and subgraph examined by constraint generator.

Some proponents of two-phase clocking prohibit all cycles in the inverter graph[5]. The extra restriction does not seem necessary, and prohibits many useful circuits (including most static RAM cells). The dynamic feedback loops in Figure 10 (based on an example in [5]) are easily detected by rule 6, which requires $\boldsymbol{A} \cdot \boldsymbol{D} \cdot \boldsymbol{I} \cdot \neg \boldsymbol{I}- = 0$ and $\boldsymbol{B} \cdot \boldsymbol{D} \cdot \neg \boldsymbol{I} = 0$. Because of circuit delays, the constraint may be violated when $\boldsymbol{A}$ rises.

Current Work

An exclusion constraint generator has been built. Its inputs are a list of switches describing the circuit (esim format) and a set of known constraints on the signals of the circuit. The output from the tool is a list of additional constraints that must be satisfied for the circuit to

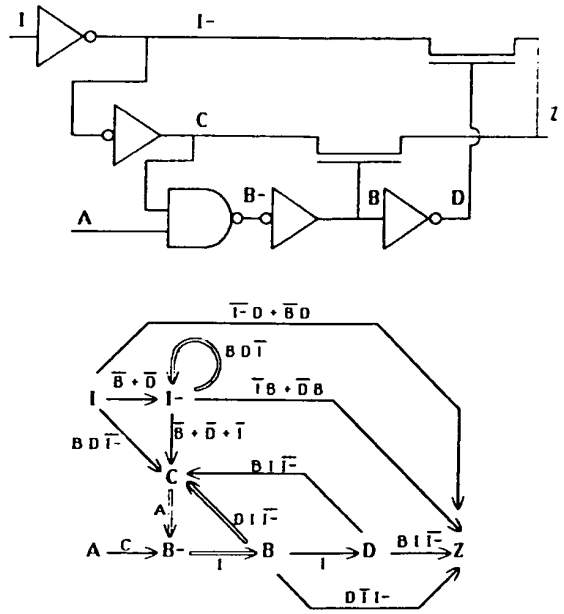

Figure 10: Circuit with dynamic inverter cycles.

meet the exclusion rules. The constraints are labeled according to the rule that generated them, so that the appropriate corrective action can be taken.

The current tool checks the rules that exclude paths to distinguished nodes in the switch graph (rules 1–4). Rule 4 is checked only for *n*MOS circuits, since pullups are easier to recognize than the node between an *n*MOS and a *p*MOS tree. Minor modifications are needed to check rule 5, since the neighbors of each gate node have to be enumerated. The switch graph representation currently used is probably adequate, but is inaccessible inside SPARSPAK. Rewriting the appropriate SPARSPAK routines in C will make the sparse matrix representation directly accessible. The labels for the edges of the inverter graph are computed, but the graph is not stored, so cycle finding cannot be done. Standard algorithms for enumerating cycles in a directed graph will probably be used[6].

The constraint generator is fast for small examples, but is somewhat slow on full chips (see Table 1). Most of the time is spent simplifying the boolean expressions. Some circuits make the tool run

very slowly; for example, transmission-gate exclusive-or circuits cause particularly slow constraint generation (fadd and tgate-mult in Table 1). The tgate-mult circuit seems to suffer from having exponentially many paths between nodes, since the program uses enormous amounts of memory and crashes while still doing the matrix factoring.

circuit	transistors	CPU seconds
fadd	22	32
hw1-3	121	5
cntr10	200	5
tgate-mult	210	> 666
multiplr	278	15
hw4	292	13
chip	445	26
digitar	3475	> 780

Table 1: CPU time (Vax 11/780) for constraint generator

Conclusions

We have presented a new paradigm for analyzing MOS switch circuits, including a new set of design rules that are simple to check and and easy for designers to learn. A tool has been built to check the rules, and has been tested on several circuits by novice designers. It appears to provide concise, understandable information about the limitations of the designs.

Acknowledgements

This work has been funded by NSF grant DCR-8503262 and by an IBM Faculty Development Award. Thanks to Liisa Raiha for implementing Randall Bryant's if-then-else trees for the constraint generator. Special thanks to John Gilbert for enlightening discussions on sparse matrix techniques.

References

[1] Randal E. Bryant. *Graph-based Algorithms for Boolean Function Manipulation*, Carnegie Mellon Computer Science Technical Report CMU-CS-85-135.

[2] Alan George and Joseph Liu. *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs NJ, 1981.

[3] Kevin Karplus. *A Formal Model for MOS Clocking Disciplines*, Cornell Computer Science Technical Report 84-632, Aug. 1984.

[4] Greg Nelson and Derek Oppen. "Simplification by Cooperating Decision Procedures," *ACM Transactions on Programming Languages and Systems*, **1(2)**, October 1979, 245–257.

[5] David Cooke Noice. *A Clocking Discipline for Two-phase Digital Integrated Circuits*, Stanford PhD Thesis, January 1983. University Microfilms 8314482.

[6] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms, Theory and Practice*, Prentice Hall, Englewood Cliffs NJ, 1977.

[7] Robert Tarjan. "A Unified Approach to Path Problems." *Journal of the Association for Computing Machinery* **28**(3), July 1981, 577–593.

[8] Chris Terman. *esim* switch simulator. Distributed with University of California at Berkeley and University of Washington VLSI tool distributions.