Tutorial: Design of a Logic Synthesis System

Richard Rudell
Synopsys, Inc.
700 E. Middlefield Road
Mountain View, California 94043

Abstract

Logic synthesis systems are complex systems and algorithmic research in synthesis has become highly specialized. This creates a gap where it is often not clear how an advance in a particular algorithm translates into a better synthesis system. This tutorial starts by describing a set of constraints which synthesis algorithms must satisfy to be useful. A small set of established techniques are reviewed relative to these criteria to understand their applicability and the potential for further research in these areas.

1 Introduction

A **Logic Synthesis System** converts a description of a digital circuit into an interconnection of logic gates (a *gate-level net-list*). A circuit description is written in a hardware description language (HDL) such as VHDL or Verilog. These languages support descriptions at three basic levels:

- **gate level:** An explicit interconnection of gates in a given technology is specified.
- **register-transfer level:** The location of the memory elements is fixed by the designer in the description, but the logic between memory elements is specified at a higher level using Boolean and arithmetic equations, and programming language flow constructs (e.g., if-then-else and functions).
- **behavorial level:** An algorithm is described without forcing a mapping onto states in the final machine; the mapping of data variables in the algorithm onto memory elements in the circuit is left unspecified.

Logic synthesis systems accept descriptions in a hierarchical mixture of these levels and produce a net-list of gates in a target technology.

A logic synthesis system contains optimization algorithms corresponding to each of these levels. Behavorial optimization algorithms (e.g., scheduling) convert a behavorial level description into an RTL description by choosing the states in which computations are performed. RTL optimization algorithms (e.g., resource allocation) operate on the arithmetic portion of the design to choose the locations of multiplexors and the types of adders and multipliers. The output of RTL optimization is a gate-level description in a generic technology (e.g., AND, OR, and inverters). Logic optimization algorithms (e.g., identifying common factors and technology mapping) convert this generic logic description into a gate-level description in a target technology.

Optimization algorithms enable a logic synthesis design methodology. Naive translation from HDL to gates results in circuits which are too large and too slow to be interesting. Relying only on obvious transformations (e.g., deleting gates which have no fanout) is not enough to make synthesis valuable to a designer. A productivity improvement comes only from the application of optimization algorithms which provide an automatic tradeoff between many conflicting choices to choose the best transformations to meet the design goals.

Looking at recent CAD conference proceedings it is clear that synthesis-related optimization algorithms are an active area of research. It is also clear that many of the solutions are becoming highly specialized and, as a result, some are missing a global perspective of how the algorithm fits into an overall system. The goal of this tutorial is to take a step back and look at a logic synthesis system and understand the constraints that a synthesis design methodology places on the algorithms. This will identify criteria which algorithms need to address to be useful and also point out areas which are interesting from a current research perspective.

All synthesis algorithms share a common set of requirements. These are presented in Section 2. In Section 3, a set of combinational logic optimization algorithms are reviewed against these criteria to understand better how the algorithms fit into a synthesis system. Section 4 performs the same task on a set of sequential optimization techniques. While the primary focus of the synthesis systems in this paper is for ASIC (gate-array or standard cell) design, Section 5 addresses FPGA design to understand how the needs of FPGA synthesis compare to ASIC design. Finally, Section 6 summarizes the key points.

2 Synthesis Algorithm Requirements

2.1 Logical Correctness

A logic synthesis system has to produce correct circuits; i.e., circuits which are logically equivalent to the source. It seems redundant to stress that a synthesis algorithm should produce logically correct results, but because of the complexity of the algorithms and their

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

interaction, verifying the correctness of synthesis is an important problem.

Logic simulation remains the primary means to verify the initial description and is often used after synthesis to verify synthesis results. However, many users are looking to formal verification techniques (e.g., combinational circuit verification) to avoid extensive resimulation of the circuit. For this reason, some design methodologies require that the synthesis algorithms fit into a formal verification methodology; i.e., transformations on the circuit are limited to those that can be automatically and independently verified.

2.2 Design-Rule Correctness

A logic synthesis system has to produce circuits which satisfy all design rules of the target technology (e.g., hold times, maximum fanout, connection rules, etc). One approach to ensure that a circuit meets a set of design rules is to patch the circuit at the end of processing so that all design rules are satisfied. Using this approach, most algorithms ignore the design rules and trust that some latter step will fix any problems which are introduced. However, for some design rules, such as a constraint on the maximum fanout of each gate in the circuit, there is an interaction between satisfying the design rule and the speed of the circuit. In this case, the synthesis algorithms must directly consider the design rules or optimization quality will suffer.

2.3 Quality: Area and Delay Tradeoff

The quality of the synthesized circuit is measured by the area, speed and power of the circuit after physical design. The unambiguous optimization goal is a circuit which meets its speed constraint and has the smallest area (or smallest power).

From a design perspective, this is the way a a synthesis system should be controlled; by changing only the timing constraints, implementations with different speed and area tradeoffs can be generated. This is greatly preferred to algorithm-specific controls which attempt to achieve the same goal in a less controlled way (e.g., a *prefer-delay* parameter for a mapping algorithm).

For these reasons, a synthesis algorithm must consider area and speed tradeoffs and must do so in a fashion which minimizes area after meeting the target speed. Rarely is the smallest or the fastest circuit acceptable, even though these cost functions are the targets of many papers on synthesis algorithms.

2.4 Delay Model Independence

It is generally accepted that static timing verification with post-layout wiring parasitics inserted into a gate-level timing model provides an accurate measure of the performance of a design. However, there is less agreement on the details of the timing model itself. Simple RC linear models have given way to more complex piecewise linear and nonlinear delay models in an attempt for more accuracy. These models also depend on a growing set of parameters such as the slope of input waveforms at each gate. Pre-layout timing models must approximate the wiring parasitics, and there is no general agreement on the best way to predict these downstream effects.

A synthesis algorithm has to consider that a detailed *sign-off* timing verification is the final judge of the circuit quality. If the algorithm uses knowledge of a specific timing model (which may differ from the sign-off timing model) or other approximations of the sign-off timing model, the algorithm must be carefully evaluated to understand what happens when the circuit is measured using the sign-off delay model.

As an example, consider measuring the delay of a circuit by the number of levels of logic in a generic technology. If an algorithm proposes a change which reduces the number of levels of logic by a factor of two, there is little doubt that a faster circuit will be produced after technology mapping regardless of the delay model used to measure the circuit.

However, to show a more subtle advantage (e.g., a 20% delay improvement), an algorithm cannot measure levels of logic or some other abstract measure of delay; a comparison must be made on the delay as measured on the design after mapping into a target technology and measured with the sign-off timing verifier. This complicates research because many times the conclusions are only as good as the back-end. The conclusions of many papers are: given our system, this was a good thing to do. It is often hard to understand whether the results would be reproducible (or even useful) in a different context.

Thus, another constraint on a synthesis algorithm is that the algorithm be abstracted from the details of the underlying timing models. At the very least, one must be aware of the delay model assumptions which are hardwired into the algorithm to understand the impact of using a different *sign-off* delay model.

2.5 Scalable

Logic synthesis algorithms must be scalable in the size of the circuit which can be processed in order to keep up with the rapid growth in circuit sizes. While a single run of a synthesis system dealt with only 1,000 gate equivalents in 1985, this number has grown to 10,000 in 1995, and will reach 100,000 or more by the year 2000 to support the design of a multi-million gate ASIC.

The constraint this places on the algorithms is harder to quantify. Because of the Boolean nature of many synthesis problems and a desire for ever-better results, it is impossible to avoid nonlinear and even exponential run-time algorithms (e.g., satisfiability). However, performance of these algorithms needs to degrade gracefully. For example, when using Binary Decision Diagrams (BDD's) [5] it is necessary to handle the case where an operation cannot complete in a graceful manner. Likewise, when using automatic test pattern generation techniques, it is necessary to allow for an aborted test after a controlled amount of run-time.

When using nonlinear algorithms, the techniques for automatic partitioning to control the run-time become as important as the details of the algorithm itself. More costly algorithms must be evaluated carefully to understand whether advantages they show on small circuits can be translated into larger circuits when partitioning is used.

2.6 Hierarchical Design

As previously mentioned, an HDL description is a hierarchical connection of designs. While some of the hierarchy in the initial description may be discarded during synthesis, maintaining a hierarchical description through synthesis is important. Hierarchy may be used to control the complexity of logic synthesis, by not forcing the synthesis system to optimize an entire chip as one piece of logic. Also, external factors such as physical design may require the hierarchy to be preserved for the physical design system.

An important concept when dealing with hierarchy in a synthesis system is the notion of *characterization*. Consider the design TOP in Figure 1. Assume that each of the subdesigns A, B, and C represent the largest block sizes which can be given to a synthesis system. Optimizing subdesign B requires that timing specifications be placed on TOP so that the timing environment of B can be captured automatically. This timing environment includes capturing the clock inputs and their frequencies plus *clock-to-Q* information for the inputs to B (e.g., which come from memory elements in module A) and *setup* constraints on the output of B (e.g., which feed memory elements in module C). Once this is done, optimization of B can be done in isolation and reinserted into TOP. If the characterization is done correctly, improvements during the optimization of B will directly translate into improvements for the constraints as measured for TOP.

Two problems arise with this paradigm, however, and both arise for paths which start in module A and terminate in module C.

The first is that the entire design starts as generic logic. Because no subdesign has yet been mapped into gates in the target technology, it is not possible to derive a characterization for module B. This is of particular concern for optimizations which happen early in the flow such as behavioral optimizations (e.g., scheduling) and RTL optimizations (e.g., operator sharing and selection).

The second problem is that there is freedom to chose where along a path from A to C changes will be made. This *time-budgetting* problem deals with assigning the degree of improvement needed within each module in order to force the path to meet its constraints. In particular, should A be optimized before or after B? Should the optimization of B attempt to modify the path so there is no violation in its timing or should it attempt to pick up only a smaller percentage of improvement?

Sometimes an attempt is made to avoid these problems by defining them out of existence. For example, one solution removes the time-budgetting problem by stating that each subdesign be latch-bounded; i.e., every input (or output) has a memory element so that no timing paths cross module boundaries. A second solution forces the designer to do manual time-budgetting for each piece of the design. These constraints are then passed to the synthesis system. A third solution requires that the synthesis algorithms handle the entire circuit as one piece. This removes the problem by stating that A, B, and C can always be optimized at one time.

However, none of these solutions seems to be acceptable in general. Hence, a general problem which is often overlooked is how the algorithms fit into a hierarchical design flow. For example, a tool which performs automatic time budgetting might require information from the synthesis algorithms to help it choose how to order the modules and split the timing between the modules.

2.7 Hierarchical Replacement

Another hierarchical consideration is the ability to rewrite the HDL e.g., for module C of Figure 1 and replace just module C without requiring a re-optimization of the hierarchical design. Optimization algorithms are often required to satisfy a substitution property which states that surrounding designs can be changed without invalidating the optimization of an embedded design.

This point can be made clear with an example. Imagine that a redundancy removal algorithm is able to remove all logic redundancies from B (in the context of A and C). Once this has been done, it is no longer possible to replace C with a new version which changes its functionality (e.g., to fix a design bug). To do so might render the previous optimization on B invalid. In the context of so-called *don't-care* optimization, this implies that only a subset of the don't-cares of a design can be used to optimize that design; in particular, don't-cares which arise because of the interconnection of designs cannot be used while preserving this replacement principle.

3 Combinational Logic Optimization

In this section I review a sampling of logic optimization techniques which are common in the literature against the listed criteria on synthesis algorithms.

3.1 Two-level Optimization

Two-level sum-of-products logic optimization, as exemplified by the Quine-McCluskey or Karnaugh map techniques taught in basic digital design classes [10], does not play a large role in multiple-level logic synthesis. It is rarely desireable to reduce a piece of multiple-level logic to two levels, apply two-level optimization, and then factor the two-level logic back into multiple-levels. The primary problem is the unpredictability of both the run-time and the effect on area and delay of this style of optimization.

3.2 Decomposition

Decomposition is the general problem of simplifying combinational logic by identifying factors which can be added to a circuit to reduce the logic in the circuit. Various decomposition algorithms proceed by choosing a factor, reexpressing the circuit using that new factor, and iterating until no more useful factors are found.

A *cube factor* [8] is a logic factor which is a simple product of literals. For example, given:

$$\begin{aligned} F &= ade + bdeh + cde + f \\ G &= bgh + cg + dg + aef \\ H &= aeg + bc \end{aligned}$$

the cube factor $X = ae$ can simplify the equations to:

$$\begin{aligned} F &= aed + bdeh + cde + f \\ G &= bgh + cg + dg + Xf \\ H &= Xg + bc \end{aligned}$$

A *kernel* [4] is a sum-of-products logic factor. Considering the previous example, the kernel $Y = bh + c$ can further simplify the equations to:

$$\begin{aligned} F &= (a + Y)de + f \\ G &= (Y + d)g + Xf \\ H &= Xg + bc \end{aligned}$$

Cube and kernel factors correlate to less area in the design but also have a strong impact on the delay. Cube and kernel factors can reduce the delay by reducing the fanout of gates, but this comes at the cost of increasing the levels of logic. It is critical to consider factoring algorithms which consider this area and delay tradeoff, but few published algorithms have dealt with this problem. Dealing with this tradeoff and remaining delay-model independent is also difficult.

Cube and kernel factoring are called *algebraic* methods. Their strength is that they rely heavily on the form of the original circuit to identify the factors, and these factors can be identified efficiently even in large circuits. Older *Boolean* factoring algorithms (e.g., *Ashenhurst* [1] or *Roth-Karp* [14] decomposition) have made a comeback recently with the application of BDD's as an implementation technique. While these algorithms have the potential for much better optimization, they have not clearly demonstrated better results on a wide range of circuits, nor have they been able to scale with increasing circuit complexity.

3.3 Iterative Improvement

A second class of techniques for optimizing a circuit rely on iterative improvement. They take an initial circuit structure and make local modifications to reduce the amount of logic or logic depth of the circuit.

The simplest form of these is combinational redundancy removal. A combinational redundancy is a connection in a circuit which, if removed, does not alter the Boolean behavior of the combinational portion of a circuit. Redundancy removal is the process of testing each circuit connection and removing any which are redundant thereby simplifying the circuit. This is typically iterated until all connections have been tested and none has been found redundant.

Automatic test pattern generation (ATPG) techniques are used to test each connection and return a status of either irredundant, redundant, or abort. Although deriving a test for a single fault is a hard problem, techniques have been developed which are efficient and still result in very few aborted tests [16].

More complex forms of iterative improvement are transduction [13, 12], rewiring[7, 6] and global flow[3]. Each of these attempts to optimize a circuit by first adding redundant connections to some gates followed by deleting (presumably a different set of) redundant connections. While the basic process is well understood, detailed approaches differ in how they decide whether connections can be made and broken and in how they decide where to modify the circuit.

There are two basic techniques for deciding whether a redundant connection can be added or deleted. The first relies on ATPG techniques. In this approach, a connection is made if it can be proven redundant by the test generator. Other connections are deleted if they are found redundant. A second approach relies on more global information, computed using a symbolic data structure (e.g., BDD's).

Iterative improvement techniques, in general, are powerful. They use the full power of Boolean relationships to make what are sometimes nonobvious (at least to a designer) changes to a circuit. They have the potential for dramatic area reductions in many cases, but this can come at a cost in delay, especially in the case where a signal deep in the circuit is reused early along a critical path. Similar to the decomposition algorithms, these algorithms often operate on a generic technology representation, so predicting their impact on delay is difficult. While optimizing for area or delay is hidden deeply in the heuristics which decide when and were to move connections, it is fair to say that techniques published so far for area / delay tradeoffs are somewhat ad-hoc.

Techniques which rely on ATPG algorithms tend to scale well with circuit size, at the cost of optimality when a connection must be assumed testable when the test generator aborts. BDD-based techniques can be made scalable by partitioning large circuits into smaller pieces, but again, this comes at a cost of optimality.

3.4 Technology Mapping

Technology mapping is the subproblem of choosing gates from the primitives in a given technology to replace the generic logic. A common paradigm for technology mapping relies on DAG-covering and fanout optimization. DAG-covering chooses the gates within the circuit and fanout optimization solves the problem of distributing the signals between the gates.

3.4.1 DAG-Covering

The DAG-covering approach to technology mapping first reduces the circuit to a simple form (e.g., two-input NAND gates) and reduces the gates in the library to the same form. Choosing the gates is then the problem of *covering* the NAND gates in the circuit by the configurations of gates which are present in the library. This approach is motivated by algorithms which can produce optimum covers for the case when the circuit to be covered is a simple tree.

Tree-mapping suffers from several problems according to our criteria, even ignoring the common complaint that cross-tree optimizations should be considered. The first is that the algorithms are linear only for an area cost function. The best algorithms for an area versus delay tradeoff on a single tree have pseudo-polynomial time complexity which quickly becomes expensive. Also, most algorithms for delay which have been presented (e.g., [15]) need intimate knowledge of the delay models to be successful.

Ad-hoc techniques, such as the rule-based approach of Socrates[9] solve many of these problems, but they have a tendency to get stuck in suboptimal local minima. New algorithms for technology mapping are clearly needed and they need to address these specific shortcomings to be interesting.

3.4.2 Fanout Optimization

Fanout optimization is the problem of distributing an output of a gate to its fanouts with minimum cost [2]. Fanout optimization is not hard for for area optimization, where the only problem is dealing with maximum fanout constraints. However, using trees of inverters and buffers it is possible to exploit two distinct properties to speed up a circuit:

- increase the drive capability to drive a signal to many destinations.
- isolate critical signals to reduce the load on the critical path

The first applies in the case when all of the signals are needed at their destination at the same time. The second case applies when a single signal is critical and all others are noncritical. Fanout optimization becomes difficult, however, when many signals are near critical. Solutions for fanout optimization also need to handle both the average case of 3 fanouts per gate as well as when the number of signals to drive may be over 100.

3.5 Discrete Sizing

Discrete sizing is the logic synthesis equivalent of transistor sizing. The assumption is that there are a discrete set of sizes for each gate and the goal is to select a size for each gate to optimize the cost function of minimum area under a delay constraint. In the synthesis context there is a further assumption that the details of the timing model are unknown to the sizing algorithm. While this is a simple statement of the problem, discrete sizing has not been the topic of many research papers in synthesis. Transistor sizing has been more popular, but transistor sizing algorithms typically assume a continuous variable for the size of each transistor and have detailed knowledge of how the size of each transistor translates into its delay. It is not clear how to apply these results to discrete sizing.

4 Sequential Logic Optimization

Sequential optimizations are relatively new to synthesis systems. A major barrier faced by all sequential optimizations is the constraint of verifiability. Behavioral synthesis, retiming, and other useful sequential optimizations cannot be verified using the formal verification techniques available today. While many of the individual steps can be verified (e.g., verifying a retiming on a circuit is easy, if no other operations have been performed), there is little hope for a completely independent functional verification analogous to combinational circuit verification.

4.1 FSMs

Similar to two-level logic minimization, application of traditional finite-state machine optimizations such as state-assignment, state-minimization, and state-machine decomposition play little role in logic synthesis systems today. The problem is that these techniques are based on two-level logic descriptions; this limits the size of logic which can be processed and forces the circuit to be partitioned before applying FSM techniques. Another problem with FSM optimization is that it operates on an abstract level with little correlation to the final area and speed of the final circuit. As a result, it is not possible to make detailed area versus speed tradeoffs.

4.2 Retiming

Retiming is a technique for repositioning the memory elements in a circuit while leaving the combinational gates fixed to optimize the speed and area of the circuit. Leiserson et al. [11] presented algorithms for retiming, including the critical problem of area minimization (i.e., fewest number of memory elements) under a delay constraint. Because retiming operates directly on a multiple-level sequential circuit, has potential for speed versus area tradeoffs, and is largely delay-model independent, it satisfies many of the requirements for a synthesis algorithm.

Retiming does suffer from its ability to scale to large circuits. While optimal retiming for minimum area can be solved in polynomial-time, the complexity is $O(V^3)$ for a circuit with V gates. Recent results [17] have shown that retiming is feasible for circuits with up to 50,000 gates by exploiting the sparse nature of typical circuit graphs. However, partitioning is required to retime circuits which are much larger than this size.

5 FPGAs

This paper so far seems focused on ASIC design systems. However, all of the same constraints apply to FPGA design systems, except that FPGA design systems hit the problems several years later.

Consider the idea of delay model independence. Many papers on delay optimization for FPGA's focus on the number of levels of LUT's as as a measure of the circuit delay. But this model of delay is less accurate when considering FPGA's which have hard-wired connections between LUT's (e.g., Xilinx 4000 or Xilinx 5200 parts) or other hierarchical routing architectures (e.g., Altera Flex 8000 parts). Also, the wire delay in many FPGA's is strongly dependent on the number of fanout. As FPGA chip sizes become larger, the delay models used in ASIC design start to look more appropriate. Hence, today's delay-based FPGA optimization algorithms are at risk of being obsoleted by the use of more complex timing models for FPGA design.

Consider the constraints of scalability and hierarchical design. Most FPGA's today have complexities of at most 10,000 ASIC-equivalent gates, meaning that these problems do not exist. However, as FPGA'S approach the 100,000 gate level, the same complexity issues hit FPGA synthesis systems. Time-budgetting and hierarchical design methodologies will have to be

used, resulting in many of the same constraints faced by ASIC synthesis systems. Likewise, algorithms based on techniques such as BDD's and Boolean decomposition techniques will no longer be applicable without partitioning.

6 Summary

A general theme in this paper is that while delay optimization and area versus delay tradeoffs are the most important aspect of synthesis algorithms, very few published techniques do a good job at this tradeoff. The reasons are easy to see. Delay optimization, especially in a delay-model independent fashion, is difficult to do, except in the last stages of optimization. Many of the techniques that are used are *ad-hoc* and do not fit into a nice theoretical framework.

A second motivation of this paper is to point out that while the general framework for synthesis algorithms is clear, there is still potential for more research in many aspects of logic synthesis algorithms.

References

[1] R. Ashenhurst. The decomposition of switching functions. In *Proceedings International Symp. Theory of Switching*, pages 74–116, April 1959.

[2] C. L. Berman, J. L. Carter, and K. F. Day. The fanout problem: From theory to practice. In *Advanced Research in VLSI, Proceedings of the Decennial Caltech Conference on VLSI*, pages 69–99. MIT Press, 1989.

[3] L. Berman and L. Trevillyan. A global approach to circuit size reduction. In *Advanced Research in VLSI, 5th MIT Conference*, pages 203–214. MIT Press, 1988.

[4] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings International Symposium on Circuits and Systems (ISCAS-82)*, pages 49–54, 1982.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE *Trans. Comp.*, C-35(8):677–691, August 1986.

[6] Shih-Chieh Chang and Malgorzata Marek-Sadowska. Perturb and simplify: Multi-level boolean network optimizer. In *Proceedings International Conference on Computer-Aided Design (ICCAD-94)*, November 1994.

[7] K.-T. Cheng and Luis A. Entrena. Multi-level logic optimization by redundancy addition and removal. In *European Conf. on Design Automation (EDAC-93)*, February 1993.

[8] D. Dietmeyer and Y. Su. Logic design automation of fan-in limited nand networks. IEEE *Trans. Comp.*, C-18(1):11–22, January 1969.

[9] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings 23th Design Automation Conference (1986)*, pages 79–85, June 1986.

[10] F. Hill and G. Peterson. *Introduction to Switching Theory and Logical Design.* John Wiley & Sons, Inc., 3rd edition, 1981.

[11] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. In *Journal of VLSI and Computer Systems*, pages 41–67, 1983.

[12] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proceedings International Conference on Computer-Aided Design (ICCAD-89)*, pages 556–559, November 1989.

[13] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney. The transduction method. IEEE *Trans. Comp.*, 38(10):1404–1424, October 1989.

[14] J. Roth and R. Karp. Minimization over boolean graphs. IBM *J. Res. Develop.*, 6(2):227–238, April 1962.

[15] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, University of California, Berkeley, 1989.

[16] M. Schulz, E. Trischler, and T. Sarfert. Socrates: A highly efficient automatic test pattern generation system. IEEE *Trans. Comp. Aided. Design*, CAD-7(1):126–137, January 1988.

[17] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proceedings International Conference on Computer-Aided Design (ICCAD-94)*, November 1994.