MILO: A Microarchitecture and Logic Optimizer

by

Nels Vander Zanden
Daniel Gajski

Information and Computer Science Department
University of California, Irvine
Irvine, CA. 92717

Technical Report 87-34

ABSTRACT

In this report we discuss strengths and weaknesses of logic synthesis systems and describe a system for microarchitectural and logic optimization. Our system uses a set of algorithms for synthesizing SSI/MSI macros from parameterized microarchitecture components. In addition, it uses rules for optimizing both at the microarchitecture and logic level. The system increases designer productivity and requires less design knowledge and experience from circuit engineers.

TABLE OF CONTENTS

1. Introduction

Over the past decade tremendous advances have been made in VLSI design. Greatly increased circuit complexity, however, continues to outpace engineers' abilities to develop chips with up to half a million transistors. Conventional design methods have entailed teams of highly skilled and experienced engineers providing many months of effort. More recently, companies have been turning to logic synthesis tools to help alleviate market pressures that demand lower-cost implementations with shorter development times. These tools allow less experienced designers to develop application-specific ICs (ASICs), typically gate arrays or standard cells. Further, the automated systems free a designer from the exploding number of details and provide greater time to examine high-level issues and experiment with various architectures. Thus such tools increase productivity and provide for better complexity management.

Synthesis tools employ two basic techniques in producing a final, workable design. These are refinement and optimization. Refinement is the process of transforming a behavioral description into an initial design. Usually this design consists of components such as multiplexors, decoders, and gates such as AND and OR. Optimization is the process of transforming the initial design into one that meets some set of objectives relating to time, area, etc. Often a design proceeds through a number of levels of abstractions. At each level, these two

stages are performed. For example, typically refinement and optimization can be used on a generic representation and later on a technology-specific one.

The synthesis process is shown in Figure 1. Refinement begins with the user's behavioral description. The input may take a textual or graphical form. Textual representations would be a hardware description language, such as VHDL, a set of boolean equations, or a table, such as PLA format. Graphical representations include a generic or technology-specific schematic, or a menu-driven interactive process that extracts a set of parameters from the user. Behavioral descriptions in language form require a compiler to generate an initial netlist consisting of high-level components such as decoders and registers. Diagrams entered through schematic capture may also consist of high-level components. Boolean equations, PLA format, or low-level schematics are representations for the logic level.

The behavioral description represents a black box whose inputs, outputs, and functionality are described. The representation generally does not convey any information as to the technology type or necessarily any style of implementation -- be it parallel, pipelined, etc. Since many systems accept different behavioral specifications, they must first convert them all to one central format, such as a generic schematic or set of boolean equations. This central format is then used to create a detailed representation of the design.

The design created in the refinement stage is usually by no means optimal and thus the optimization phase is called upon to improve performance. Figure 1 identifies two levels at which optimization can take place -- at the microarchitecture level and the logic level. One could add other intermediate levels. A logic generator is required to decompose the design into the lower level.

Optimization can be guided by two different strategies. The first is to optimize everything until no further improvements can be made. When optimizing for multiple constraints there may be a weighting or priority scheme to determine which design improvements should be performed. Essentially, this is a "brute-force" method as optimizations tend to be made in random parts of the circuit in no directed fashion. The second strategy is constraint driven. It relies on a set of user-entered parameters to direct the flow of optimization. For example, optimizations may be applied initially only along a critical path when attempting to meet timing constraints. Once one constraint is met, the optimizer will turn to another and work toward meeting the user's objectives. Once these are satisfied, the optimization stage may end or may apply the first strategy.

2. Previous Work

A number of automated logic synthesis systems have been previously reported: LSS [JoTr86], SOCRATES [GrBa86], and the Logic Consultant

[Kim87]. They have focused primarily on logic level optimization. The systems have used algebraic, language compiler, and expert system techniques to reduce circuit delay and gate count in combinational logic. In general, such systems are best suited for optimizing random logic that can easily be represented by boolean equations or PLA format. An exception is LSS which accepts a register-transfer language as its input, allowing entry of high-level operators. However, only limited types of optimization are performed on the high level operators before they are decomposed into gates. Once a design is at gate level it is impossible to recover high-level information that may be necessary for restructuring the design in order to meet timing or area constraints.

When circuit constraints cannot be met by further local optimizations at the gate level, a human designer will often return to a higher level and restructure the design. The gate level implementation can be rebuilt and this time may be able to meet the design constraints. For example, consider an arithmetic unit in which the critical path passes through a ripple-carry adder. Once the designer examines the gate-level implementation and determines that the critical path cannot be solved with the present design, high-level alterations may be made. One possibility would be to replace the ripple-carry adder with a carry-lookahead adder.

Previous logic synthesis systems have not had the capability to reexamine the microarchitectural design and make the necessary high-level alterations to meet low-level constraints. This paper examines the MILO system that performs such microarchitectural examinations. MILO introduces a number of novel features: it captures and optimizes designs on the microarchitecture level, uses logic compilers to expand microarchitectural components into those from a technology-dependent library of SSI/MSI components, then optimizes the expanded design. When an initial low-level implementation fails to meet the user constraints, MILO makes microarchitectural modifications, then regenerates the gate-level circuit. While examining the microarchitecture, MILO can optimize sequential logic by manipulating counters and registers. Before examining MILO in further detail, we will review the manner in which previous work has addressed the issues of automated logic synthesis.

2.1. Refinement Techniques

2.1.1. SOCRATES

Figure 2 shows the SOCRATES system. SOCRATES accepts boolean equations, PLA format, or a netlist as a behavioral description. Two-level boolean equations are used as the central format. Multi-level boolean equations can be extracted from the netlist, then run through an expander to generate the two-level format. Likewise, an extractor can be used to generate two-level

boolean equations from PLA format. An algebraic minimizer, ESPRESSO IIC minimizes the two-level equations, taking advantage of don't care conditions. The next step uses weak-division to find common subterms in the design and form multi-level equations, thereby reducing the amount of logic required to implement the function. This minimized design is written back into netlist form from which the logic optimizer will operate. The optimizer is discussed in a later section.

2.1.2. Logic Consultant

The modules comprising the Logic Consultant system are displayed in Figure 3. Like SOCRATES, the Logic Consultant accepts boolean equations, PLA format, and netlists generated from schematics as input to its system. Schematics can be created on a Mentor workstation using components from Mentor's generic library (GENLIB) or components from a technology-specific library. A decomposition module converts the input description into components from the Trimeter Generic Library (TGL) and isolates the combinational logic for processing by the minimization module. Technology-specific components are replaced with TGL components through user-entered rules in the knowledge base. For MSI components (ie., multiplexors, decoders, ALUs, etc) this replacement is optional. The user specifies whether the MSI elements should be decomposed into TGL gates (via the rule base) or should be

left "as is". Those components that are not decomposed are treated like sequential logic elements and removed from the design that is passed to the minimizer. Thus if an MSI component is not decomposed, it will not be optimized with the surrounding logic. This strategy poses a problem since in decomposing the MSI components, one loses the advantage of using them. Typically high-level components are added to a technology library because the designer of that component constructed it in such a way that it takes up less area and has greater speed than a corresponding gate implementation. But once a component is decomposed for optimization, it may be difficult or impossible to find after flattening and refactoring the circuit. However, if the components are not decomposed, no optimization is performed. As will be seen later in this paper, a better solution is to perform some optimizations at the microarchitectural level.

The minimizer module accepts combinational logic consisting entirely of TGL components from the decomposition module. The minimizer then develops a two-level SOP design and removes redundant terms. This new design will be passed to the factorization module. For some circuits it may be desirable to bypass the minimization module and go directly to factorization. This is the case when the generated two-level SOP form contains an explosive number of terms. These circuits require extensive CPU time and after factorization typically contain more components than the original design. For example,

certain multi-level circuits may require many times their number of circuit elements to be represented in a two-level format. As factorization is performed on a local basis (and not globally), one cannot guarantee that the factorization will be optimal. Hence factorization may be unable to reduce the gate count to the prior number of elements. Designs with a large number of XOR and XNOR gates are one example of this type of circuit. Thus the Logic Consultant's minimizer does not produce a better design in all cases.

The factorization module attempts to factor out common terms to produce a multi-level design. It also factors in such a way to reduce the delay along the longest path. For example, consider Figure 4. The bottom input of the AND gate is part of the longest path. This path can be shortened by factoring the 3-input AND into two 2-input AND gates. Thus some timing considerations are taken into account. Note however that the longest path may not always by a critical path. When this is the case, the assumption that it is a critical path will prevent optimization for area along that path. Hence, such a factorization strategy is not truly constraint driven.

Factorization continues until no further common terms or timing improvements can be found. The Logic Consultant factors all paths as completely as possible. Since this includes critical paths, certain factorizations must be undone at a later time.

2.1.3. LSS

The LSS system architecture is shown in Figure 5. LSS proceeds through four different description levels to produce an optimized technology-specific circuit: high-level, AND/OR, NAND/NOR, and technology specific. Each level requires a translator to produce its description format from a higher-level description. Likewise, each level has an optimizer to apply simplifying transformations. By introducing multiple levels, the designers of LSS followed the example of programming language optimizing compilers that produce several intermediate descriptions before generating assembly-language code [DaJo80]. Changes can be made through a number of levels, thereby simplifying analysis and optimization at each level. For example, simplification can be made in terms of generic components, then converted to technology-specific components for further optimization. Attempting to improve a high-level description directly into a low-level one is a much tougher task. Also by using generic levels, only the technology-specific optimizations need be rewritten when the technology changes.

LSS begins with an algorithmic representation using a register-transfer language. Using a simple translator, LSS produces a logic graph representing the design. Translation is straightforward as operators in the behavioral description are assigned to nodes in the graph. Each node is a generic

component (such as an AND gate or decoder) and is connected via the graph's edges to other components. Optimization at this level is discussed in a later section.

Another translator is called to produce the second level AND/OR description. It decomposes high-level components, such as decoders, into AND/OR/NOT gates. Optimizations are then applied at this level. The third description level, NAND/NOR consists of only NAND and NOR gates. Once again, the translator that produces this description is achieved through naive transformations that may produce unnecessary NANDs and NORs. These "extra" gates are removed by the optimizer at this level. Finally, LSS provides a translator to produce a technology-specific design using components from a technology library that consists of gate-array macros (such as NAND/NOR gates, complex AND/OR gates, etc). This technology description may then be optimized.

2.2. Optimization Techniques

After refinement, synthesis tools call upon optimizers to improve the design. Their optimization techniques fall into one of the three expert system types shown in Figure 6. Each consists of some type of blackboard and knowledge base. The blackboard is where the design resides and can be examined by the knowledge base. Included in the blackboard is usually a netlist describing the

design, statistics on area and path delays, a set of user constraints, and work space for the system to evaluate various attempts at refinement or optimization. The knowledge base consists of a set of rules and algorithmic techniques that utilize the blackboard data structures and make modifications to them. Generally algorithms are assigned structured and well-defined tasks while rules handle unusual or loosely defined problems.

2.2.1. Rules Only Strategy

The first type of expert system is entirely rule based. This approach generally lacks structure as rules are entered essentially independently of one another. The control unit selecting a rule to fire must examine all rules to determine which rules are applicable. Any of the rules whose conditions are satisfied can be fired. It is up to some rule selection mechanism to choose one rule from that set. An early implementation of this type of system was R1 [McDe82], a rule-based system for configuring Vax-11/780 computers. A more recent version of a strictly rule-based system is the Logic Consultant. Both of these systems use the OPS production system language to write rules for their knowledge base. R1 was written in OPS4, the Logic Consultant uses OPS83 -- the lastest OPS version. Each rule consists of a set of conditions and a set of actions to be performed when all of the conditions are met. Control in OPS is exercised through a recognize-act cycle. In this cycle all rules whose If-

conditions are satisfied are found using the Rete match algorithm [BrFa85], then one of these rules is selected to be applied. Briefly, the Rete algorithm compiles all of the rule conditions in an OPS program into a set of attributes whose values can be tested. These attributes are placed in a tree-structured sorting network. In doing so, it allows similar tests on attribute values to be shared among different rules. Further, once a test has been performed on a tree node, it is not redone until a change in data occurs upon which the attribute is dependent. These features make the Rete algorithm much more efficient than a simple pattern matcher that examines each rule's conditions on every recognize-act cycle to determine which rules apply.

From this set of applicable rules, called the conflict set, a single rule must be selected. OPS has a conflict resolution scheme to choose this rule based on a number of tests through which rules are eliminated from the conflict set. Priority is given to rules in the following order [Fo85]: rules that have not been executed, rules whose first attribute (value of the first condition) has been most recently altered, rules with the most conditions, rules that entered the conflict set most recently.

In a rule-based system incorporating such schemes, control over which rules are applied can only be achieved by adding more conditions to each rule. In R1, an additional condition was added to each rule corresponding to the stage of the

design task. The Logic Consultant system builds in limited structure by examining its set of applicable rules and making an evaluation of the gain produced by each rule. The rule with the largest gain can then be applied on the circuit. Another characteristic of strictly rule-based systems is the lack of backtracking. Neither R1 nor the Logic Consultant permit backtracking -- once a rule is applied, it will not be undone.

The Logic Consultant's first optimizer module is the cell selection module. It converts the design consisting of TGL components to technology-specific components. This module uses rules from the knowledge base that specify how one or more TGL components map into a technology-specific component.

Once a technology-specific design is derived, the design is further optimized by a technology-specific optimization module. This module utilizes rules that make equivalent circuit transformations to improve area or time. The optimizer chooses which rules to apply based upon some formula that examines time/area tradeoffs. For critical paths and the longest path, rules are selected that decrease delay but that may increase area. Along non-critical and short paths, the Logic Consultant applies rules that decrease area at the expense of time.

Certain rules may appear to increase area or time but can actually result in a decrease by applying "clean up" rules. The Logic Consultant has a classification of rules to do this. For example, if a rule adds inverters to the

circuit, the optimization module will examine a set of high-priority or "clean up" rules to determine if any of the high-priority rules can eliminate the inverters from the design. Before implementing any rule, the optimizer calculates the result of rule applied with any high-priority rules to determine how the rule will affect area and time.

When entering rules into the knowledge base, the user indicates whether a rule is high-priority. The rule classification indicates only that the rule should be examined to determine whether it can "clean up" after a regular rule application. It does not give the high-priority rule preference over a "regular" rule. This high-priority rule classification provides a limited lookahead feature of one rule.

2.2.2. Rules/Algorithms (Mixed Strategy)

The second expert system type makes use of a rule base but employs structuring through algorithmic control. The algorithm establishes a hierarchy that determines which set of rules should be examined at any point in time. This added structuring allows greater control over the type of rules that are eligible to fire. The top level of the hierarchy examines the blackboard to determine the current stage of optimization. Depending on the set of constraints that must be satisfied, various strategies may be selected for further investigation. At this stage, more information is needed to evaluate the remaining strategies. Lower

levels in the hierarchy are called upon to produce it. From this information, the high-level decisions can be reached.

Lower levels in the hierarchy inspect the blackboard in greater detail and can choose a single strategy for optimization. Inside each strategy is further hierarchy. Similar to the strategy selection, one technique in the strategy must be chosen from a number of possibilities. The chosen technique in turn calls upon still lower levels of the hierarchy to carry out a design transformation. These lowest levels include rules for manipulating data structures in the blackboard, keeping data in the blackboard up to date, and removing old or useless information.

SOCRATES is an expert system that uses a mixed strategy for optimization. The SOCRATES optimizer is written in C and runs on the VAX 11/780. Like the Logic Consultant, it consists of rules that make local transformations on the circuit. SOCRATES also has procedures to provide feedback on the time and area savings produced by a rule. Through these measurements, SOCRATES can choose which rule to apply. In addition though, the SOCRATES rule base employs a limited hierarchy to reduce the number of rules that must be considered at any given time. It organizes the knowledge base into a number of classes, such as timing or area optimizing rules. Further, each class of rules is divided into subclasses of related rules. For example, one

subclass might replace partially redundant multiplexors with NAND and NOR gates, and another might combine and synthesize those gates [GeCo85]. SOCRATES examines only rules in a particular subclass at any one time. Each class and subclass of rules is prespecified in a certain order in the knowledge base. SOCRATES then examines each rule class and subclass in this order. Optimization in SOCRATES begins with rules that improve both time and area. Then rules are applied that optimize time, possibly at the expense of area, until all timing constraints are satisfied. Finally, area optimizations are made on noncritical paths, possibly at the expense of time, until no other area improvements can be made.

A rule in SOCRATES consists of a target configuration and a replacement configuration. These configurations are patterns of components, pins, and nets that identify a particular circuit structure. SOCRATES has its own pattern matcher that determines which target configurations are present in a design. Like OPS83, it contains a recognize-act cycle in which possible rule applications are examined, one rule is selected and then applied. Because of the hierarchical rule-base, SOCRATES only needs to consider rules in the currently activated rule subclass. To eliminate rules in the conflict set, the recognize-act cycle may utilize lookahead. Each rule in the conflict set is evaluated by implementing its set of actions, then measuring the resulting effect. The rule's future effect (ie., an examination of the rules that may be applied after the initial rule

application) can be observed via a state search. This involves setting up a search tree consisting of future design states. The search tree is a graph with the nodes representing possible circuit implementations and the arcs representing rule applications. The root of the tree is the current circuit implementation. Children of nodes in the graph are ordered by desireability. The leftmost children being derived from "better" rules [CoBa85]. The path through this state graph producing the lowest cost function is the optimal sequence of rules.

Construction of the graph is performed in a depth-first manner. After each transformation, the results are evaluated by a cost function. If the resulting circuit is acceptable the next set of rule applications will be examed, the "best" rule selected and its effects determined. The process is repeated until some maximum depth is reached in the search tree. If the resulting circuit is deemed "unacceptable", SOCRATES backtracks to the node's father and examines alternative circuit transformations [GaGr84]. In constructing the search tree, SOCRATES keeps a log of changes made to the circuit by each rule application. When backtracking is required, the changes to the circuit can be quickly undone by referring to this log.

Each node of the tree will contain the cost function estimate of the circuit implementation. The lower the cost function, the better the circuit. Once the search tree has been completely built, it can be traversed to determine which

sequence of rules is best. Those rules will then be applied.

Since search trees can become massive and require large amounts of CPU time, the designers of SOCRATES introduced metarules that control the size of the search tree. Metarules are rules that contain control knowledge, as opposed to design knowledge that suggests circuit alterations. The SOCRATES metarules are based on the parameters presented in [CoBa85]. The first parameter, **B**, restricts the number of sons any node may have. Thus it limits the breadth of the search tree. The parameter **D** limits the depth of the tree by restricting the number of consecutive rule applications. Using the neighborhood control parameter, **N**, restricts rule applications to gates of path distance N from some center gate. This prevents rules that apply to different circuit regions from being considered. The parameter $\mathbf{D}_{app}$ restricts the rule application depth. Although the search tree extends to depth $\mathbf{D}_{\max}$, only a portion of some sequence of rules will be executed. A parameter Δ_{class} was introduced to limit the number of rule classes to be examined beyond the current subclass. This variable decreases the number of rules that can be applied at any given time. Finally, the parameters Δ_{cost}, **R**, and **S**, relate to the cost function. Δ_{cost} limits the increase to the cost function by a single rule application. Parameters R and S are used in the cost function to determine the desireability of applying a particular rule. The cost function takes into account the area saved and the number of rules that can be applied after a transformation is made. The

weighting of terms in the cost function affects the size of the tree by changing the desireability of various rules.

Initially SOCRATES used fixed values for these parameters, regardless of the optimization phase. However, the ideal parameters vary greatly over the course of optimization. For example, greater lookahead is required for area-saving rules than general rules. Also, little or no lookahead is required for the most powerful rules [GeCo85]. Thus based on the state of the optimization, metarules determine what values the control parameters should have. These rules supervise the control module and dynamically vary the parameters. Such a technique permits more selective use of lookahead. Control parameters can be changed depending on the rule class, rule subclass, or even an individual rule.

Through this process of lookahead, the best sequence of rule applications can be determined. By examining the future effects of a rule, SOCRATES can determine whether the decision to apply a rule was good. Poor decisions can be undone through backtracking and other rules can be considered. The designers of SOCRATES report this approach produces superior results to those where only one rule is examined and then applied [CoBa85]. Their examples indicate that the use of lookahead without metarules required roughly four times longer on average to run, producing designs with 12 percent less area on average. Adding metarules only doubled the run time and still provided the same

decrease in area.

LSS is a system that also employs limited hierarchy as it performs optimization at each of its four levels of representation. The LSS refinement stage produces an initial graph with AND/OR gates, registers, decoders, etc. The first level of optimization is performed on this network. It uses transformations that reduce the gate-level logic and transformations that use information about a high-level component to reduce the surrounding gates. Figure 7 demonstrates transformations of this type from [JoTr86]. The first rule uses knowledge about a decoder to eliminate the OR gate. The second rule in Figure 7 is a simple logic reduction transformation.

Once all level one transformations have been performed, the high-level components can be expanded into AND/OR/NOT gates. This forms the second description type, the AND/OR level. At the AND/OR level, transformations perform AND/OR simplification, common subexpression elimination, and constant propagation (ie., OR(a,1) = 1, AND(a,1) = a) [DaJo81].

The third level, NAND/NOR, introduces some technology considerations. Depending on the technology, the design will be converted to one consisting entirely of generic NAND and NOR gates. The same type of transformations that were performed at the AND/OR level are applied at this level, only with NAND/NOR simplifications in mind this time. Transformations at this level

attempt to incorporate technology tables that supply information on generic NAND/NOR gates. For example, information on each generic primitive (such as a NAND3) is maintained on its size, driving capability, delay, fan-in, etc. [JoTr86]. Hence at the NAND/NOR level it is possible to make decisions about what type of NAND/NOR gates should be used. For example, LSS can tell whether to use three-input NANDs or two-input NANDs to reduce area.

The final level of description is technology specific. Transformations convert generic components to components from a technology library that may include complex gates such as AND/OR, multiplexors, etc. The transformations also enforce technology constraints such as fan-in and fan-out. To deal with complex gates, LSS makes use of tables that list the components available in each complex gate type (ie., AND/OR, decoder, OR/AND). In each category the components are ordered according to the savings created. When conflicts arise as to which transformation to apply, the one with the largest gain is selected based on this ordering.

The transformations performed at each level in LSS are executed through PL/I procedures that manipulate the logic graph. Transformations at the final two levels make use of a number of technology tables that can be readily updated for a new technology. It has been reported [JoTr86] that the use of local transformations as opposed to two-level boolean minimization tends to keep

synthesis times linear for increasing design sizes. For circuits of 200 to 2000 two-input equivalent gates, a time of one second for roughly nine gates was reported (based on IBM 3081 CPU time).

2.2.3. Algorithms Only Strategy

The final type of expert system is entirely algorithmic. It uses an algorithmic controller to determine how to apply algorithms in the knowledge base. Such a system performs the same operations always in the same order. An example of an algorithmic system is DAGON [Ke87]. DAGON performs technology binding and can optimize for time, area, or some function of the two. It utilizes programming-language compiler techniques that are strictly algorithmic. In doing so, DAGON can guarantee locally optimal matches over several thousand patterns.

Similar to language compilers, DAGON matches a graph description of a technology-independent circuit against a technology library consisting of numerous patterns. The problem is viewed as finding the best technology patterns to cover a directed acyclic graph (commonly termed DAG). A DAG is a graph containing no cycles that consists of nodes and directed edges. DAGON builds a DAG for boolean functions using nodes to represent AND and OR operators and edges labeled 0 or 1 to indicate a true or inverted output (thereby providing NAND and NOR operators as well).

A globally optimal solution to the DAG covering problem could be generated by comparing all possible technology implementations (each a collection of patterns from a technology library) for time and area. However, such a problem is NP-complete. Therefore, DAGON's first step is to partition the graph into trees. This is accomplished by making every component in the graph whose fanout is greater than one, the root of a new tree. DAGON may then find the minimal cost technology pattern for each tree, producing a locally optimal solution.

Finding a minimal cost match for a tree consists of two major tasks: recognizing the set of possible matches and determining the pattern from that set providing the minimal cost (in terms of time and area constraints). Finding applicable pattern matches is performed by *twig* [Tj86], a program designed to construct code generators for programming language compilers. It generates the set of matches in O(TREE SIZE) time. From this set the minimal cost match can be found using a recursive algorithm that determines the least cost match for each subtree.

3. Optimization Strategies

The quality of a synthesized design can vary dramatically depending upon the strategies used to optimize it. Strategies can be examined at both a high and a low level. At a high level there are essentially two types of decisions for

which a strategy must be chosen. They are: what to optimize for and what subsection of the overall design to optimize at any given time. At a low level there are also two types of decisions: what types of transformations or algorithms to apply and what order to apply them to the specified subcircuit.

The first decision depends heavily upon the user-entered constraints. Typically three types of constraints are entered: time, area, and power. The optimizer must choose which order (if any) to optimize the constraints and which constraint(s) should carry the greatest weight in directing the optimization.

The second decision concerns the portion of the design toward which optimizations should be directed. One could simply search the entire design for all the rules that are applicable and select one to apply. This has the effect of applying transformations randomly throughout the design. Generally the technique is time consuming and produces less than optimal results. Human designers break up large circuits into smaller ones, making optimization more manageable and allowing more control over the course of the optimization. This approach is best for optimization programs as well. By focusing on only a small section of the design, one not only reduces the number of rules that must be considered but also allows a rule's effect to be more closely examined -- one need only consider its effect in the subcircuit to know what happens in the larger

design.

When performing timing optimizations one tends to select a path-oriented approach. Thus the subcircuit might be a critical path. Area optimizations are not path directed except for avoiding critical or near-critical paths. In this case, critical paths could be removed to eliminate the time wasted by considering rules that affect some component that is part of a critical path.

The third decision involves the specific rules or algorithms that can be applied. This decision is in part based upon the earlier issue of what to optimize for. Certain techniques work best when optimizing for time, others when optimizing for area. As an example, consider a rule that greatly decreases delay but improves area only incrementally. Clearly such a rule is best suited for use in timing directed optimizations. An alternative rule that produces greater area improvements can be used when optimizing for area.

The final decision involves the order in which optimization techniques should be applied. The use of lookahead can aid greatly in assuring that the "best" rules will be used. Lookahead, however, can be quite time consuming and simply considering certain rules before others may be a better solution in some cases. Another example of the use of ordering can be seen in the use of algorithms. For instance, some algorithmic techniques, such as collapsing a network into two levels to reduce delay, require a great deal of time and effort.

Yet if only a small improvement is required, the effort is, in effect, wasted. Another technique, perhaps a rule-based approach producing smaller gain, could have been used.

4. Optimization Strategies for Timing

In this section, we examine strategies used in timing optimization. One approach is shown in Figure 8. First, a timing analyzer determines which paths are critical. From this information a critical path is chosen. This is usually the one whose delay is furthest from the user's specifications. Next, a point on the critical path must be selected for optimization. Two criteria are used to determine this point. The first criteria chooses the component which the most critical paths pass through. This has the effect of improving a number of critical paths with a single replacement. If there are multiple points that satisfy criteria 1, the second criteria will select the component from that set that is closest to an external input. Once a point of optimization has been chosen, a control strategy is selected. This strategy determines which type of optimizations should be attempted. The different strategies will be discussed later. The next step is to select a rule within that strategy and evaluate its cost function. If the cost of applying the rule is too great or the rule fails to achieve a sizeable gain, a new rule will be selected and evaluated. If the strategy succeeds in making the path non-critical, another critical path will be selected. On the other hand, if the

strategy has exhausted all possible rules without solving the critical path, a new strategy will be selected.

4.1.1. Control Strategies

Different control strategies are required for timing optimization. For example, critical paths whose delays differ greatly from user specifications tend to require some type of circuit restructuring. In contrast, those critical paths that are close to the specifications may require that only a few gates be replaced or only a small portion of the critical path to be modified.

4.1.2. Types of Strategies

There are a number of different strategies that can be used to improve speed. These are illustrated in Figure 9. Strategy 1 swaps equivalent signals on the same component. For example, the 3-input AND gate in Figure 9a has a different delay from each input to the output. Thus the critical path should be connected to the input with the shortest delay. Strategy 1 can be implemented by examining a technology file that contains timing information on each component. This strategy has no cost as it does not change any circuit elements. However, the gain produced is small. Thus strategy 1 is useful when the slack (difference between the actual and required times) is extremely small or when strategies that produce larger gains have been exhausted.

Strategy 2 replaces a low-power or standard-power macro with a high-power macro of greater speed. This type of strategy is only applicable to ECL logic where the amount of current can be increased to provide faster switching transistors. Strategy 2 can be implemented by examining the technology file to determine if the same macro exists with higher power, higher speed. The strategy is similar to strategy 1 as it produces only a small gain. Thus it would be used in the same situations as strategy 1. However, since strategy 2 increases the power, strategy 1 is preferred.

Strategy 3 employs factorization to reduce the delay of a critical path. Figure 9c shows how a four input AND gate can be factored to speed up path D. Implementation is a factorization program such as that found in MIS [BrRu87] or ESPRESSO [Br84]. The gain is typically small though greater gains are achieved for larger gates. In addition, using factorization along the entire critical path can add up. Power generally increases but the effect on area may vary. Strategy 3 tends to produce a slightly larger gain then strategies 1 or 2. Hence, it will be used when the slack is small.

Strategy 4 attempts to make a better macro selection that reduces time without an increase in area or power. This strategy may utilize a rule base containing only rules that will produce a gain for no cost. For instance, a rule could be written in OPS83 to perform the transformation of Figure 9d. An

alternative method is use of a hash table. Lookup in the hash table is accomplished through a key that is the truth table entry for a particular function. The hash table is typically limited to entries of up to five variables, making each hash table key a maximum of 32 bits -- a common computer word. Certain transformations cannot be represented by a hash table. For example, functions with multiple outputs, such as a decoder, or circuits with sequential logic elements, such as a counter. In these cases it is necessary to use the rule-based approach. A hash table has an advantage over the rule-based approach in that fewer transformations need to be entered. For example, Figure 10 shows two different implementations of a multiplexor that can be represented by the same hash table entry, but require two separate rules. Another advantage of hash table lookup is speed. It requires only one table lookup per function. Of course time is required to find a circuit's function but this may require less time than having to search through a set of rules to determine which are applicable. Since Strategy 4 has no cost, it will be used before strategies 2 and 3, and will be the first strategy examined for moderate gain.

Strategy 5 duplicates logic along a critical path, thereby doing the reverse of common term factorization. Like strategy 4, it can be implemented using either a rule-based or hash table approach. The gain from strategy 5 is typically small and hence the strategy would be applicable at the same time as strategies 1 - 3. As the cost in area and power tends to be greater than strategies 1 - 3, strategy

5 would be the last examined.

Strategy 6 is similar to strategy 4 except a better macro selection is made with an increase in area and/or power. It can make use of a hash table or rule based approach as well. Typically a moderate gain can be achieved through strategy 6 with a small to moderate increase in the power and area constraints. It is often considered for moderate slack improvements or for large slack improvements after large gain strategies have been examined.

Strategy 7 is the foundation of both SOCRATES and the Logic Consultant. It expands the design into two-level SOP form then minimizes by removing redundant terms. This strategy is often combined with strategy 3 to factor the circuit along non-critical paths and take advantage of common terms. This strategy is the most time consuming but can produce large gains, often with no increase or only a small increase in area and power. Thus this strategy is the preferred method for improving large slacks.

Strategy 8 is one discussed in [JoMc87]. It duplicates logic that strategy 5 can't by adding a multiplexor. Figure 9h has a function FOUT = F(A,B,C) where C -> FOUT represents the critical path. The logic network may be duplicated with the C input connected to GND in one, and VDD in the other. The real C input is then hooked up to the select input of a multiplexor. The gain from Strategy 8 is large but so is the cost if little optimization can be

performed after increasing the amount of logic. New circuit elements are added which increases both area and power. Hence, strategy 8 will be examined for a large slack but will be considered after less costly strategies have been attempted.

4.1.3. Choosing a strategy

The control strategy can be changed depending on how far the critical path is from the timing constraints. When the time difference is small, a local optimization can be attempted using some combination of strategies 1 - 4. Rules from three different categories can be examined: those that do not increase area and power, tradeoff rules that improve speed but increase either area or power, and tradeoff rules that increase both area and power. Those rules in category 1 will be considered before examining rules in categories 2 or 3. Similarly category 2 rules are preferred over those in category 3.

Within each rule category, the smallest rule with the maximum gain will be chosen. Thus rules involving only two inputs will be examined first. If one of them meets the timing specifications, that rule will be applied. Otherwise rules with three inputs must be examined. This process continues until all rules have been examined at which time a new strategy must be applied.

When the time difference is great or all other strategies fail, the circuit can be minimized into a two level circuit using strategy 7. It can then be expanded

through weak division into multiple levels as in strategy 3.

5. A Framework for Microarchitectural Optimization

Systems such as LSS, SOCRATES, and the Logic Consultant decrease the design knowledge required by a designer as they reduce technology dependence. In addition, they increase a designer's productivity by performing optimizations previously performed by the designer. Logic gates, however, represent only about 20% of a typical design. The remaining portion consists of sequential logic, RAM, ROM, as well as higher-level combinational components such as arithmetic units that make use of MSI-type logic such as adders. Such logic cannot readily be represented by a set of equations or by low-level schematics. Thus these tools provide technology independence for only a portion of the design.

In order to extend technology independence and provide for even greater productivity it is necessary to start the synthesis process at a microarchitectural level. At this higher level a designer enters a circuit with components such as arithmetic units, counters, registers, RAM units, and of course the random logic that connects these components. In this manner, one can enter a crude design with correct functionality and have an optimized technology-specific design produced.

Each of the microarchitectural components can be defined by a set of parameters. These parameters may include functional and structural information. For example, an arithmetic unit can be classified in terms of the functions it performs: addition, subtraction, incrementation, and decrementation. It can also be classified in terms of its structure: carry propagate or carry lookahead. The parameters, along with component interconnection information can be used to perform local transformations, similar to the manner that SOCRATES performs transformations at the gate level.

At the microarchitectural level it is difficult to know with any accuracy what the actual time delay and area are. However, if an optimizer is to make area/time tradeoffs, the design statistics must be known. Essentially, there are two methods to acquire this information. The first method is to have an estimator that makes use of a technology-specific file. It may be possible to create a formula that when passed the component parameters, produces a reasonable estimate of the time and area required by the technology. The second method generates a low-level generic design based on the component parameters. This design may then be mapped into a technology-specific circuit via a technology library.

Once a design has been optimized at the microarchitectural level, the low-level circuit must be synthesized. The high-level components tend to have a

very regular structure. For example, a 32-bit adder can be decomposed into eight 4-bit adders. Such decomposition can best be expressed in terms of algorithms. The algorithms can be generalized to incorporate a set of parameters from the user. The basic structure of the design is always the same -- parameters only require small modifications or additions be added to the basic structure. This form of algorithmic decomposition contrasts with synthesis at a low level which has an irregular structure. Such irregularity brought about the use of rules which could deal best with the unstructured nature of the problem.

6. MILO System Architecture

In this section we describe MILO. It uses a number of novel concepts: it captures and optimizes the design at the microarchitecture level, then uses compilers to expand the microarchitectural components into MSI/SSI library macros (not gates) and optimizes this compiled design. The system architecture of MILO is shown in Figure 11. Input to the MILO system is a netlist generated through schematic entry or by a compiler for the VHDL hardware description language. Also included in the input are parameters for path delays, area, and power consumption that must be met by the design optimizers. The components entered through schematic capture may be any combination of generic, technology-specific, and logic compiler generated components. For each microarchitectural component (such as a register or ALU), there is a logic

compiler.

6.1. Logic Compilers

Each compiler consists of two programs, each written in C. The first program, the symbol compiler, generates a symbol for the high-level component that can be used in microarchitecture capture. The second program, the design compiler, generates the design for the microarchitectural component and creates a schematic for it so that the user may inspect the design.

Symbol compilers are selected from the menu in the schematic capture program. The menu displays the compilers available and the designer selects one using the mouse. Once the compiler is called, it presents the user with a number of options as to how the high-level component should be built (such as number of inputs, the speed desired, or the number of input loads an output must drive). A symbol for the component is produced by the symbol compiler and can then be placed in the user's design. MILO's current set of logic compilers is shown in Figure 12.

Once a symbol has been generated, the design compiler will be called. There is a different design compiler for each technology (ie., CMOS, ECL). Thus compilers for ECL will tend to use NOR and OR gates whereas CMOS compilers will use NAND and AND gates. The generic library contains a set of standard gates, such as AND, NAND, and XOR, as well as some MSI components, such as

2-bit counters, 4-bit adders, and 4 to 1 multiplexors. The generic library is displayed in Figure 13.

Each design compiler consists of an algorithm that describes the type of components to use and how to connect them. For example, consider the simple case of an OR compiler for ECL. The algorithm for generating an i-input OR gate is given below.

```
See if the requested design already exists in the database. If so, exit.
Otherwise, proceed.
Place i input pins in level 1
level_count = 2
num_outputs = i
num_left_over_outputs = i
while (num_outputs > 1)
    total_num_gates_added = 0
    /* Process each level */
    while (num_left_over_outputs > 1)
        Find an OR gate in the database with num_or_inputs such that:
            num_or_inputs <= num_left_over_outputs
        num_gates_to_add = i/num_or_inputs
        Add num_gates_to_add OR gates to level level_count
        num_left_over_outputs = num_left_over_outputs -
                        (num_gates_to_add * num_or_inputs)
        total_num_gates_added = total_num_gates_added + num_gates_to_add
    End While
    num_outputs = num_left_over_outputs + total_num_gates_added
    level_count = level_count + 1
End While
```

The design compilers build circuits in a hierarchical fashion. This allows one design compiler to call another, or even itself. For example, consider the register compiler. It permits the user to select a number of functions such as

load, shift right, and shift left. To achieve these functions the design compiler places a multiplexor in front of each flip-flop. In the course of creating the register, the register compiler will call the multiplexor compiler to produce the 4 to 1 mux required.

6.2. Technology Mapper

The technology mapper converts components from a generic library into those from a technology-specific library. Thus it can take designs from the logic compilers and produce a design using components from a gate-array or standard cell library. The technology mapper uses a lookup table to replace a generic component with the corresponding technology-specific component or set of components. Each technology library has a different lookup table. During the conversion process, various design rules may be violated (such as a component's fanout). These must be detected and corrected by the electric critic. This critic is part of the logic optimizer and is discussed in the later section.

6.3. Microarchitecture Critic

The microarchitecture critic operates on the generic netlist produced through schematic capture or behavioral compilation. It makes local transformations at the microarchitectural level, replacing small sections of the design with equivalent functions. An example of a microarchitecture

transformation is shown in Figure 14. Rules at the microarchitectural level are based primarily on the parameters that describe each component as well as their interconnection to other components. For example, the rule for the transformation in Figure 14 is shown in Figure 15. The antecedent (IF section) of the rule identifies two components whose functionality parameters are "adder" and "register". The interconnections are then compared to ensure the pattern is of the form in Figure 14. The consequent (Else section) replaces the two components with a single instantiation of a counter. The symbol for the new counter is generated by a call to the counter compiler, providing it with the set of parameters (# inputs = N bits, Reset = YES). The design for the counter will be generated when it is required for lower-level analysis.

As mentioned earlier, it may be difficult at this level of abstraction to make decisions regarding certain types of transformations without much knowledge of the true design statistics (such as speed, area, and power). To get design statistics, the critic calls upon the logic compilers to generate the low-level generic designs for the desired microarchitectural components. A technology mapper converts these generic components into those from a technology specific library. Statistics can then be generated from this design. Now the microarchitecture critic can examine the high-level design for area/time tradeoffs. Note that statistics are only necessary when making such tradeoffs. For instance, the optimizer must know which paths fail to meet the user timing

constraints in order to make the necessary delay improvements -- which typically result in increased area. On those paths for which the user has entered no timing constraints, no timing statistics are required as the critic will attempt to minimize the area without regard to time.

The microarchitectural optimization process in illustrated in Figure 16. The design in Figure 16 is initially seen as consisting of a 4-bit adder, 2 to 1 multiplexor with a 4-bit slice, and a 4-bit register with shift-right capability. Assuming that the user has entered a time constraint from the input A to the output C, the path will be broken down into the hierarchy by calling upon the logic compilers. The compilers produce the designs ADD4, MUX2:1:4, and REG4. The adder and multiplexor compilers each produce designs that are ready to be sent to the technology mapper. As mentioned previously, the register compiler makes a call to the multiplexor compiler -- in this case to produce component MUX2:1:1. This introduces an additional level of hierarchy. Each primitive generic component can then be mapped into its corresponding technology-specific design. The technology-specific designs produced are for designs ADD4, MUX2:1:4, and MUX2:1:1. At this point, the microarchitecture critic has the timing statistics for the high level modules of ADD4 and MUX2:1:4. The statistics for module REG4 are still not known, however, as its technology-specific design has not yet been generated. One can now be produced, inserting the already mapped version of MUX2:1:1 into the REG4

design and finishing the technology map by making technology-specific replacements of the flip-flops. Finally, the optimizer has the time delays for each module and if the path from A to C is found to be critical, time improvements can be made at the expense of area. One example of a tradeoff would be changing the parameters of the adder to instantiate a carry-lookahead model.

6.4. Logic Optimizer

When the microarchitecture critic can make no more optimizations, the design is passed to the logic optimizer. The logic optimizer makes technology-specific transformations on the design. It consists of three optimizers and five experts that work toward meeting the user's design specifications. The optimizers: time, area, and power, reduce delay, area, and power, respectively. They make use of the five experts: logic critic, timing critic, area critic, power critic, and electric critic. Each critic is rule-based and examines the blackboard to make suggestions to the control module. Figure 17 shows an example rule from each of these critics. The logic critic contains rules that always decrease delay and area. Thus rules in this knowledge base always improve the design. The timing critic has rules for increasing speed but at the expense of area and power. Likewise, the area and power critics decrease area and power, respectively, while increasing the other constraints. The electrical critic consists of rules that spot and correct electrical errors in the circuit. In this manner it is

very much like an electronic rule checker.

The logic optimizer operates in a hierarchical fashion. It optimizes the design for each microarchitectural component before the designs are combined to form one large design. In this manner, the logic optimizer begins by optimizing designs at the lowest level of the hierarchy, then the design at the next highest level can be expanded in terms of its lower-level designs and that design can be optimized. This process is illustrated in Figure 18. Optimization begins with the technology-specific versions of designs ADD4, MUX2:1:4, and MUX2:1:1. At this point no transformations can be performed on the designs so optimization proceeds to the next highest level and examines the design REG4. In REG4, each multiplexor and flip-flop set can be combined into a single technology-specific element, providing a decrease in area. Finally, the designs for REG4, MUX2:1:4 and ADD4 can be expanded to produce the design for the highest level design, ABADD. Again, there is optimization at this level, combining the 2-1 multiplexors with the already multiplexed flip-flops of REG4 -- making use of high-level macros that have 4-1 multiplexors combined with a flip-flop.

Since the logic compilers produce near-optimal designs, little optimization is required -- for the most part a cleanup of the technology-mapper's design (such as inverter elimination, or merging of components -- as was seen in the optimization of REG4 in Figure 16). Also, use of the logic compilers has allowed

us to use higher-level generic components (such as multiplexors, and adders) in our designs than the Logic Consultant or SOCRATES that decompose designs completely into gates in order to optimize. By having the higher-level components, we are better able to make use of the high-level technology-specific elements that are typically found in gate-array and standard cell libraries. Once a design has been completely decomposed into gates it can be quite difficult to recover these high-level components. Yet if the components are not decomposed, the design will not be extensively optimized as the systems tend to rely upon algebraic techniques for minimization which require the decomposition of high-level components. Often times the silicon implementations of these high-level library components have been manipulated by experienced designers and take full advantage of "tricks" in the technology -- thereby reducing the actual area as well as the delay. Because of this, many times these "specialized" components were placed in the library to be used instead of gates -- used not only for design ease, but because of speed and area improvements that resulted. MILO allows full advantage to be taken of such high-level designs.

By doing more work at a higher level of abstraction and keeping high-level information for a longer period of time, MILO requires less work at the lower levels where much greater work is required to perform the optimization. This is particularly true since at a lower level, there are many times the number of components and hence it can be more difficult to determine the best route to

follow for optimization.

7. Results

MILO is currently running on SUN 3 workstations under the UNIX operating system. The logic compilers and optimization strategy selectors are written in C while the rule set is written in OPS83. We are currently in the process of adding rules to the knowledge base. To test our system, a number of small examples were run at both a gate level and a microarchitecture level. An ECL gate-array library was used by the technology mapper to create technology-specific designs. Our results are comparable to circuits produced by human designers.

Figure 19 shows the results for eight circuits entered with generic components. The optimized results are contrasted to technology-specific circuits entered by a human designer. **Generally MILO was able to improve designs 2 to 40 percent in terms of time and area.** Designs entered at a microarchitecture level required 4 to 15 logic compiler generated components. Improvements at the microarchitecture level were less dramatic than those at the logic level. This primarily resulted since the microarchitecture designs tended to be more regular in nature -- involving components such as counters and adders. Never the less, MILO showed sizeable improvements. In addition, the use of MILO greatly reduced the complexity of designing the circuits. The next step in testing MILO

will be to run a large chip design and compare the results with those of a human in terms of area, time, power, and productivity.

8. Conclusions

We have examined previous work in automated design synthesis that has been devoted to logic optimization and low-level synthesis. The systems have used boolean minimization techniques with rules or language compiler techniques with rule-like transformations. MILO is a system that permits design entry at a microarchitecture level. It optimizes designs first at the microarchitecture level and incorporates feedback from lower-level design analysis. After microarchitectural improvements have been made, the focus of optimization moves to the gate level.

We have combined a rule-based technique with algorithms to synthesize low-level logic from a set of parameters for a microarchitecture component. In doing so, the MILO system improves design productivity and requires less design knowledge and experience from design engineers.

9. Acknowledgements

This work was supported in part by Applied Micro Circuits Corporation and State of California MICRO program. We would also like to extend thanks to Ted Hadley, Frank Vahid, and Mehdi Amani for their work on the logic

compilers.

REFERENCES

[BrFa85]
Brownston, L., Farrell, R., Kant, E., and Martin, M., "Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming", *Addison Wesley Publishing Company*, 1985, pp. 228-239.

[Br84]
Brayton, R., et al., "ESPRESSO IIC: Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, Netherlands, 1984.

[BrRu87]
Brayton, R., Rudell, R., Sangiovanni-Vincentell, A. and Wang, A., "MIS: A Mutiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987.

[CoBa85]
Cohen, W., Bartlett, K., and de Geus, A., "Impact of Metarules in a Rule Based Expert System for Gate Level Optimization", *Proc. IEEE Int'l. Symp. on Circuits and Systems*, May 1985.

[Chu65]
Chu, Y., "An ALGOL-like Computer Design Language", *Communications ACM*, Oct. 1965.

[DaJo80]
Darringer, J., Joyner, W., "A New Look at Logic Synthesis", 17*th Design Automation Conference*, 1980.

[DaJo81]
Darringer, J., Joyner, W., Berman, C., and Trevillyan, L., "Logic Synthesis Through Local Transformations", *IBM J. Res. Develop.*, 25, no. 4, July 1981.

[EnNa85]
Enomoto, K., Nakamura, S., Ogihara, T., and Murai, S., "LORES-2: A Logic Reorganization System", *IEEE Design & Test*, October 1985.

[Fo85]
Forgy, C., "OPS83 User's Manual and Report", 1985.

[GaGr84]
Garrison, K., Gregory, D., Cohen, W., and de Geus, A., "Automatic Area and Performance Optimization of Combinational Logic", *Proc. IEEE Int'l. Conference on Computer-Aided Design* , 1984.

[GeCo85]
de Geus, A. and Cohen, W., "A Rule-Based System for Optimizing Combinational Logic", *IEEE Design & Test*, August 1985.

[GrBa86]
Gregory, D., Bartlett, K., de Geus, A., and Hachtel, G., "SOCRATES: A System for

Automatically Synthesizing and Optimizing Combinational Logic", 23*rd Design Automation Conference*, 1986.

[JoMc87]
Johannsen, D., McElvain, K., and Tsubota, S., "Intelligent Compilation", *VLSI Systems Design*, April 1987.

[JoTr86]
Joyner, W., Trevillyan, Y., Brand, D., Nix, T., and Gundersen, S., "Technology Adaptation in Logic Synthesis", 23*rd Design Automation Conference*, 1986.

[Kim87]
Kim, J., "Artificial Intelligence helps cut ASIC Design Time", *Electronic Design*, June 11, 1987.

[Ke87]
Keutzer, K., "DAGON: Technology Binding and Local Optimization by DAG Matching", 24*th Design Automation Conference*, 1987.

[McDe82]
McDermott, J., "R1: A Rule-Based Configurer of Computer Systems", *Artificial Intelligence*, 19(1) (1982).

[Tj86]
Tjiang, S., "Twig Reference Manual", January 1986.

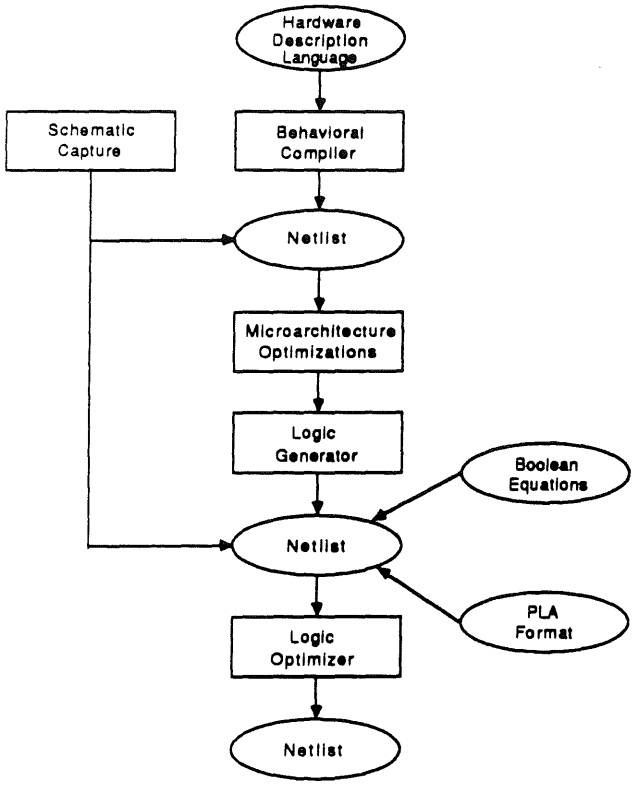


Figure 1: Logic Synthesis Process

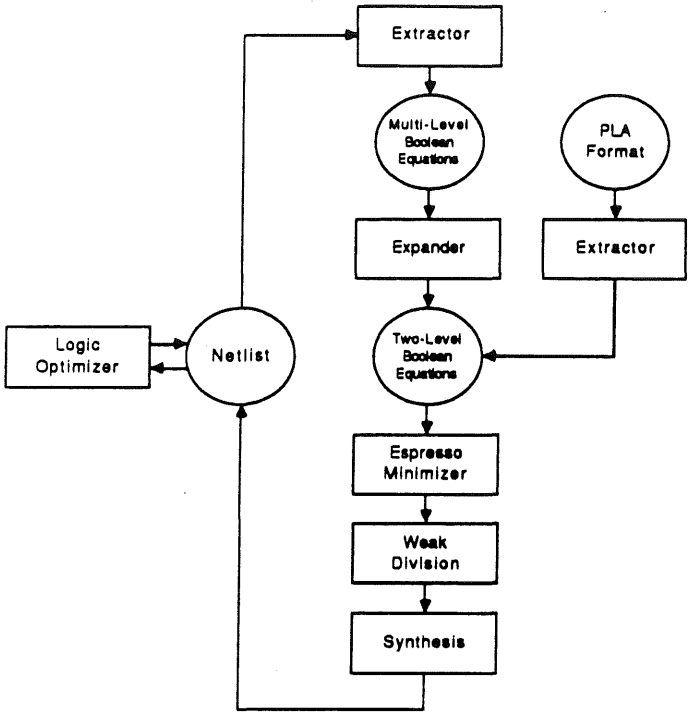


Figure 2: SOCRATES System Architecture

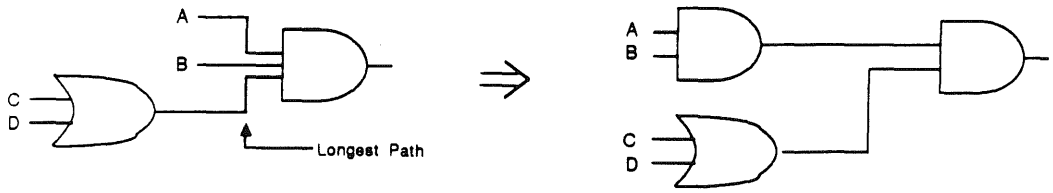


Figure 4: Factorization for Timing Improvements

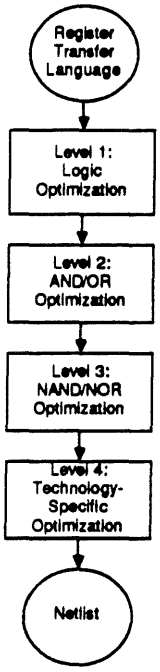


Figure 5: LSS System Architecture

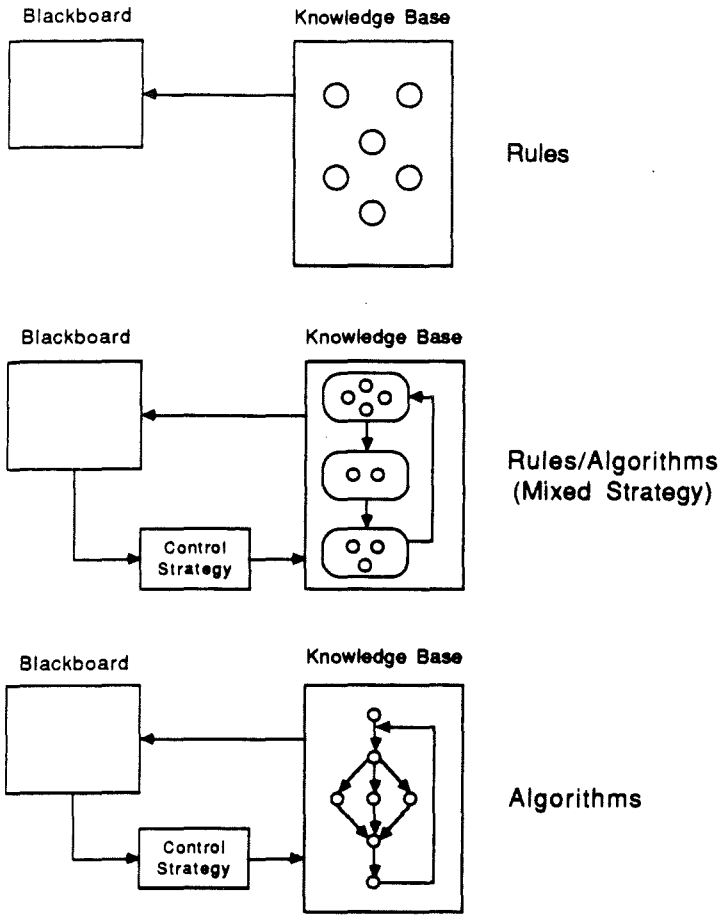


Figure 6: Expert Systems

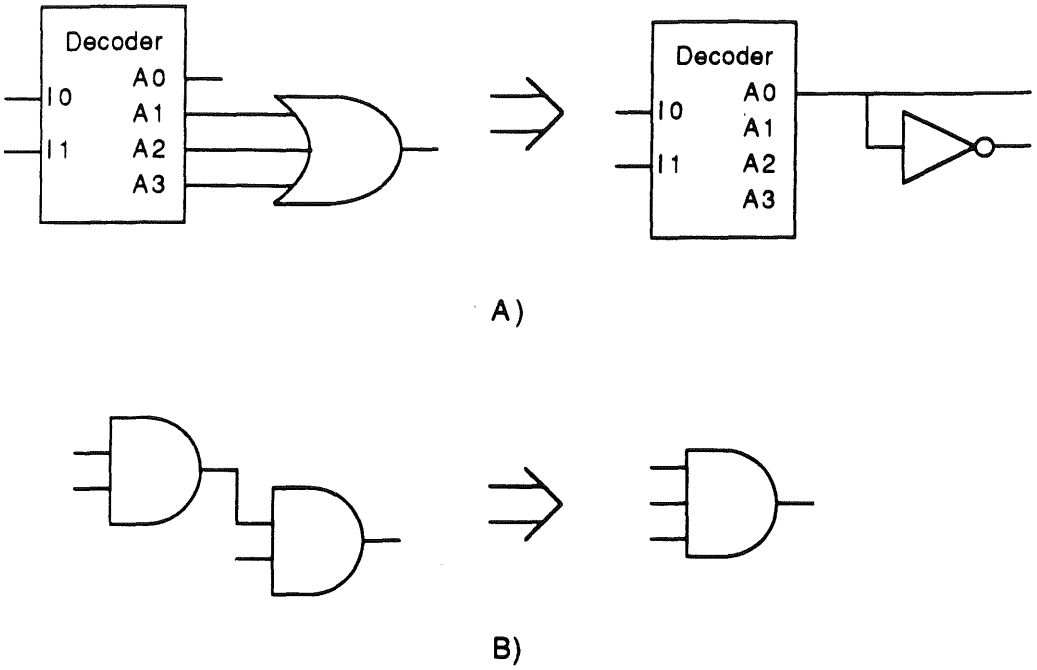


Figure 7: LSS Level 1 Optimizations

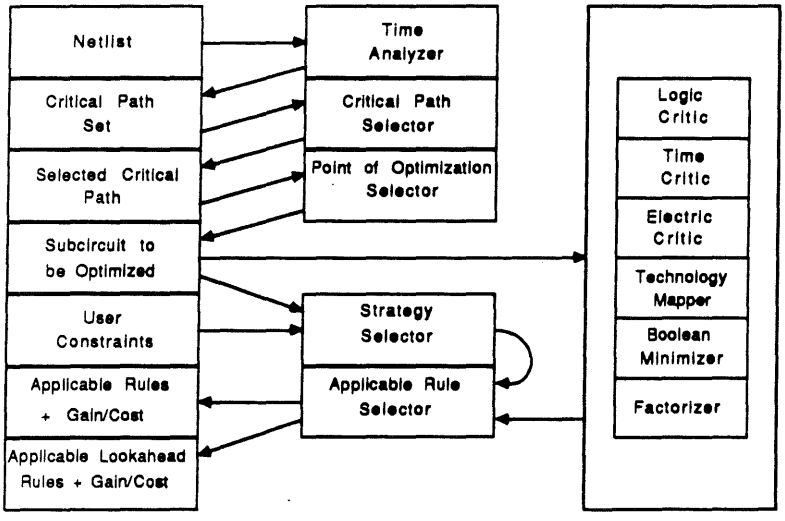
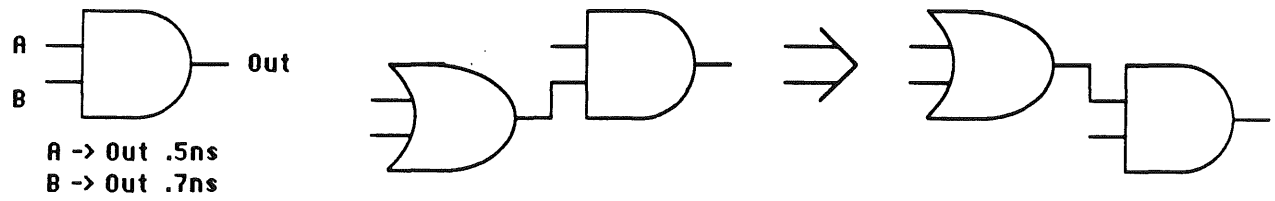
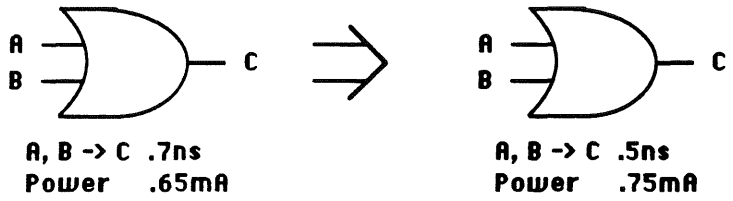


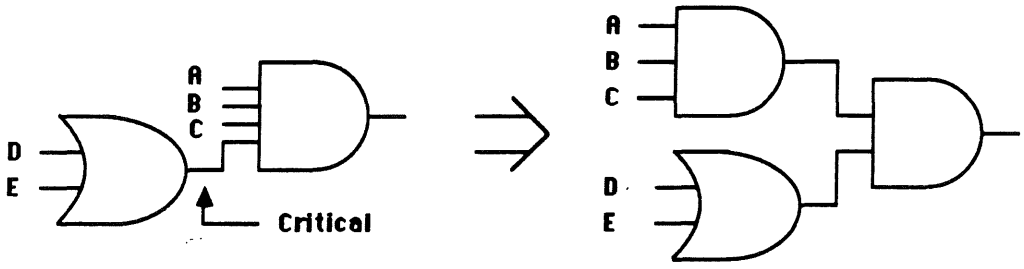
Figure 8: Time Optimizer



(a) Swap equivalent signals on the same component

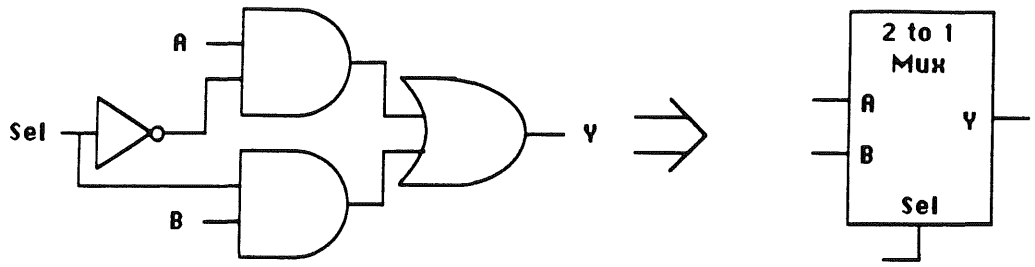


(b) Replace macro with one of higher speed

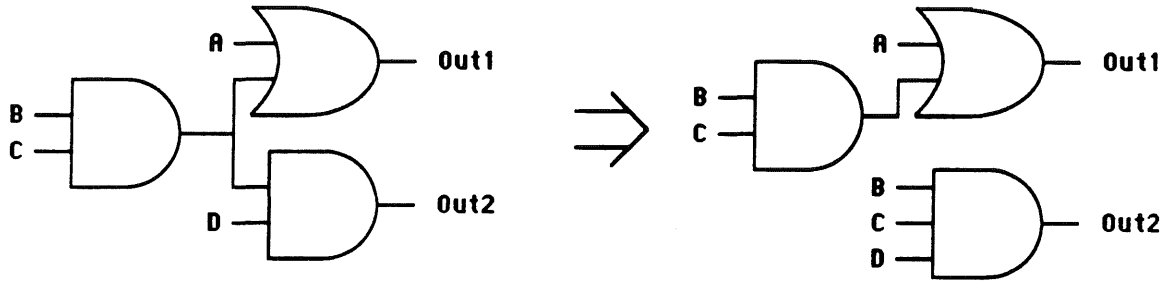


(c) Factor

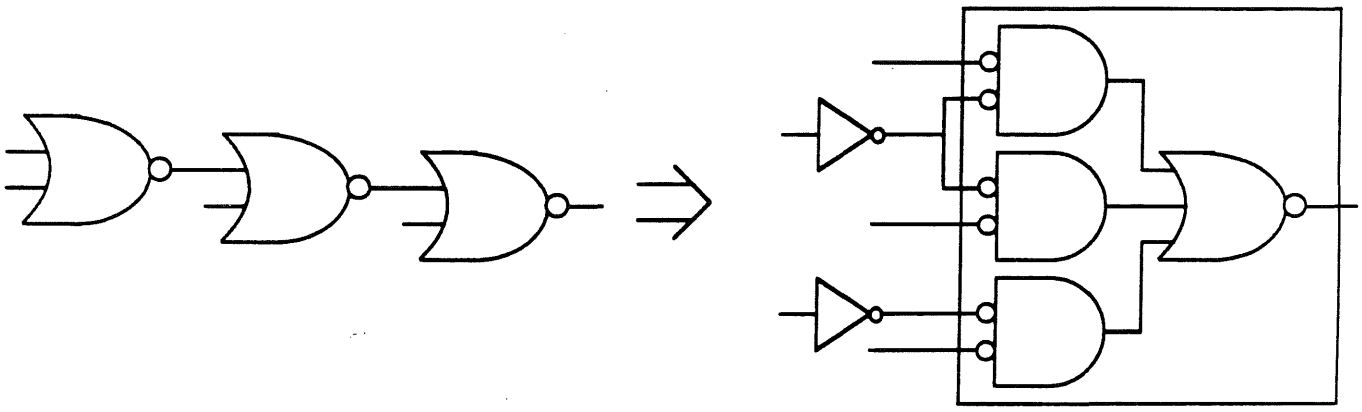
Figure 9: Strategies for Reducing Delay Along Critical Paths



(d) Better macro selection that does not increase area or power



(e) Duplicate logic



(f) Better macro selection at the expense of area/power

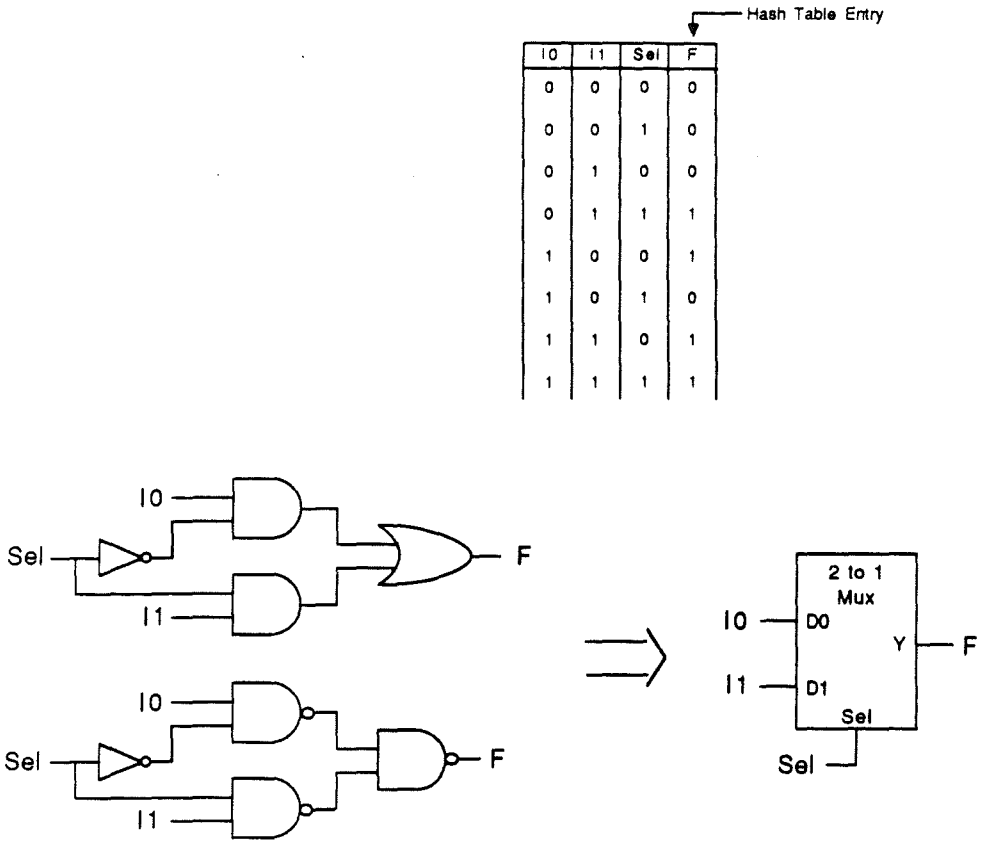
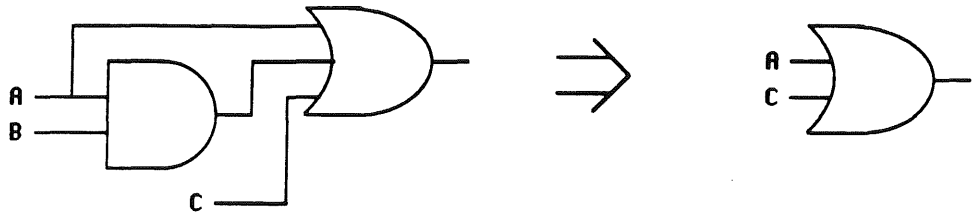
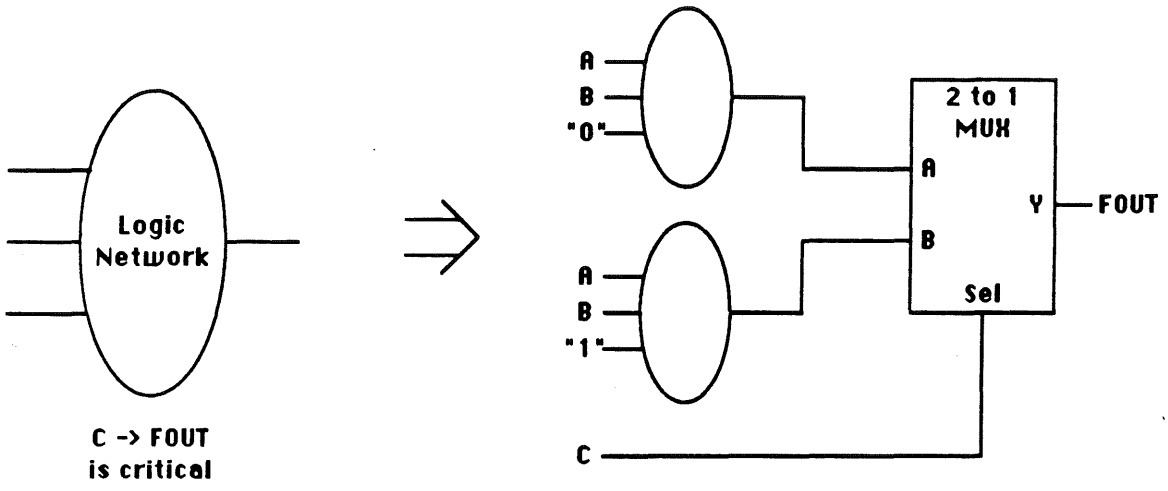


Figure 10: Use of a Hash Table to Reduce the Number of Rules



(g) Minimize



(h) Duplicate logic with multiplexor

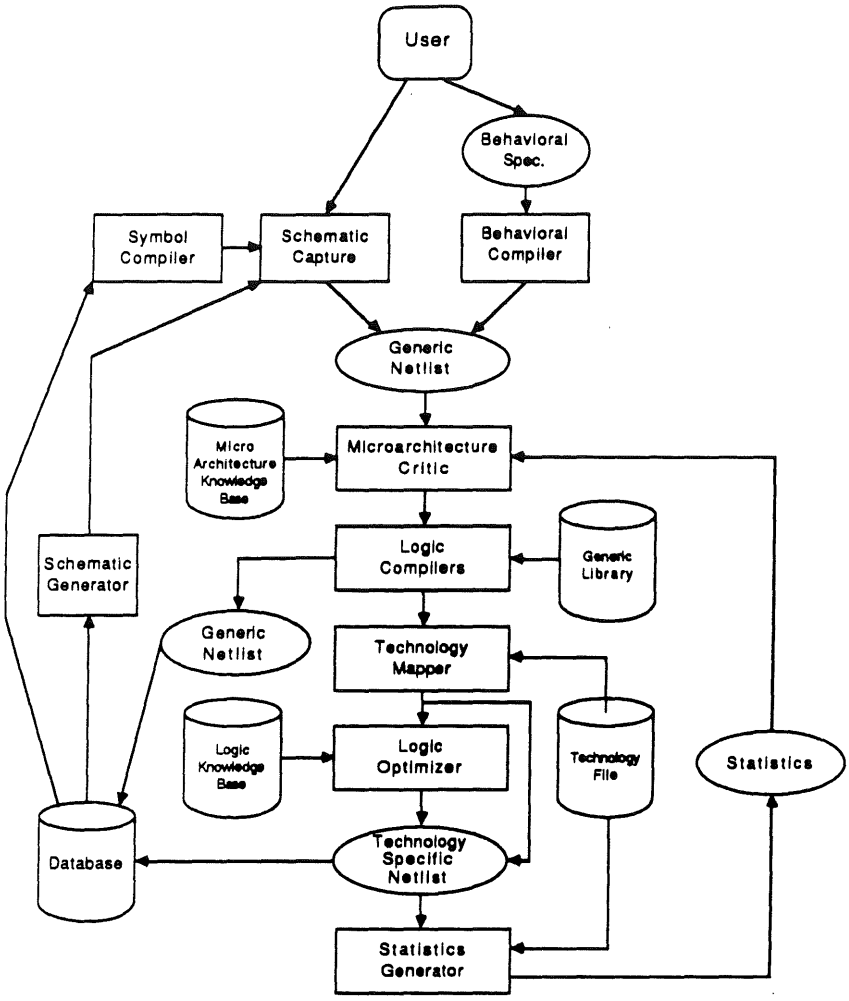


Figure 11: MILO System Architecture

```
GATES
        (function (= AND, OR, INV, NAND, NOR, XOR, XNOR),
         #inputs.
        )

MULTIPLEXOR
        (# bits,
         control (= enable),
         # inputs.
        )

DECODER
        (# bits,
          control (= enable).
        )

COMPARATOR
        (# bits,
         function (= >, <, =, ...).
        )

LOGIC UNIT
        (function (= AND, OR, INV, NAND, NOR, XOR, XNOR),
         # inputs.
        )

ARITHMETIC UNIT
        (# bits,
         function (= +, -, INC, DEC),
         mode (= ripple, carry-lookahead).
        )

REGISTER
        (# bits,
         type (= latch, edge triggered),
         function (= load, shift),
         control (= set, reset, enable).
        )

COUNTER
        (# bits,
         function (= load, up, down),
         control (= set, reset, enable).
        )
```

Figure 12: The Set of Logic Compilers

Generic Macros

AND	2,3,4
OR	2,3,4
NAND	2,3,4
NOR	2,3,4
XOR	2,3,4
XNOR	2,3,4
INV	
BUF	
VDD	
VSS	
MUX	2 To 1
	4 To 1
DECODER	1 To 2
	2 To 4
ADDER	1 Bit
	4 Bit
	4 Bit with Carry Lookahead
COMPARATOR	2 Bit
	4 Bit
COUNTER	2, 4 Bit with Up/Down/Reset/Load/Enable
REGISTER	1 Bit
	with Inverting/Noninverting/Set/Reset/
	Edge Triggered/Level Sensitive

Figure 13: Generic Component Library

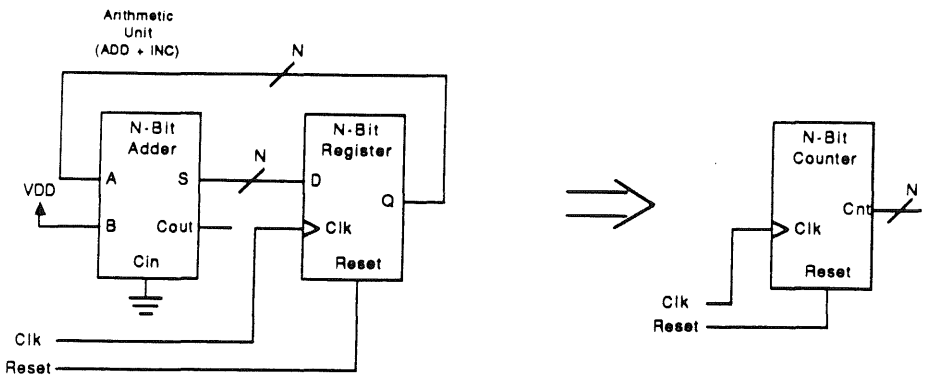


Figure 14: A Microarchitecture Optimization

If there is a component C1 with functionality = adder
AND there is a component C2 with functionality = register
AND the SUM output of C1 is connected to input D of C2
AND the Q output of C2 is connected to one of the C1 data inputs
AND the second C1 data input is set to VDD
AND the Cout output of the adder is not connected
AND C2 has a Reset pin
Then
Call the counter compiler with parameters:
# inputs = # inputs of C1
Reset = YES
to produce component C3
Replace C1 and C2 with C3

Figure 15: Microarchitecture Rule

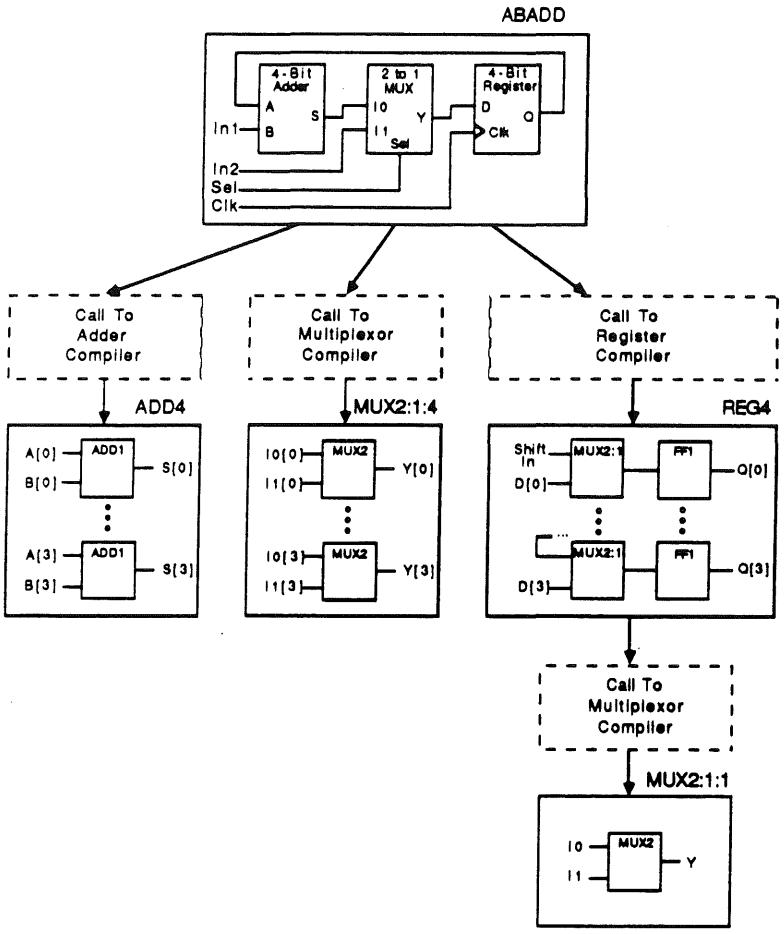
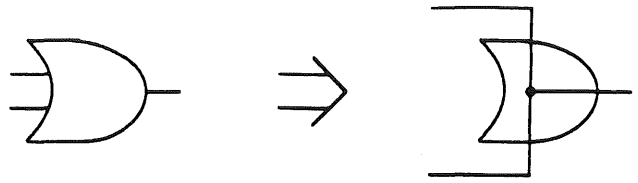
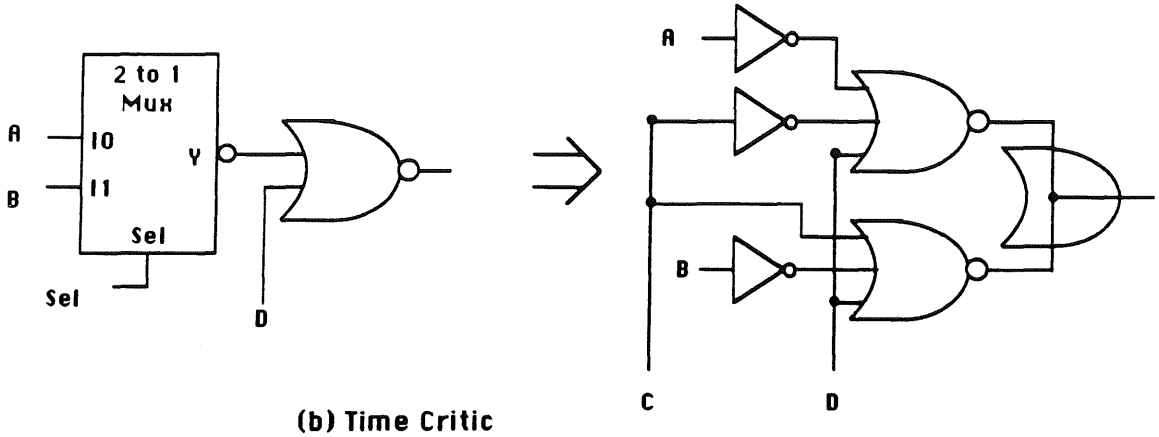


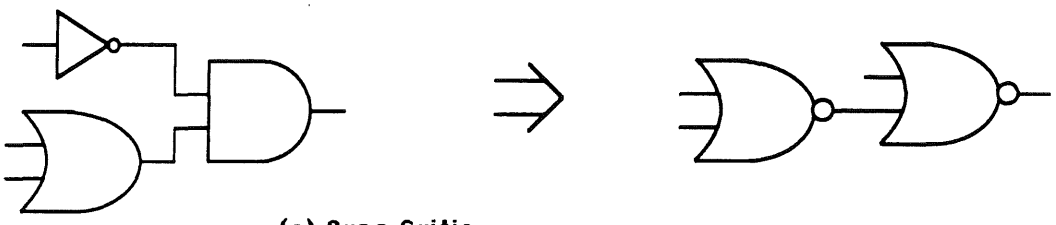
Figure 16: Microarchitecture Optimization Process



(a) Logic Critic

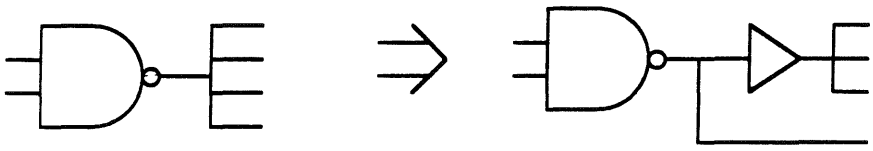


(b) Time Critic



(c) Area Critic

(d) Power Critic



(e) Electric Critic

Figure 17: Design Optimizer Rules

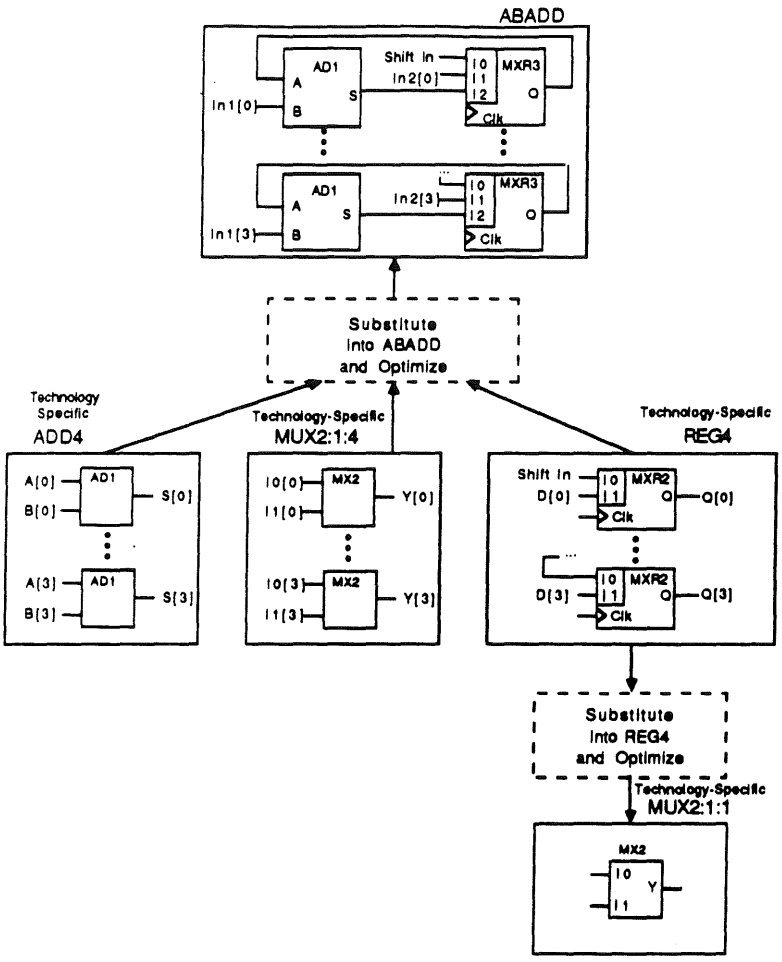


Figure 18: Design Optimizer Process

Design	Complexity (gates)	Delay (ns) Human	MILO	Percent Improv-ment	Area (cells) Human	MILO	Percent Improv-ment
1	48	19.76	14.80	25	12.0	9.0	25
2	52	3.84	2.97	23	9.0	7.5	17
3	13	2.52	1.65	35	3.5	3.0	14
4	47	5.07	3.25	36	8.0	5.0	38
5	18	3.82	3.10	19	4.0	3.0	25
6	288	19.40	18.50	5	52.0	44.0	15
7	442	18.37	16.22	12	92.0	85.0	8
8	149	15.65	14.45	8	63.5	62.0	2

Figure 19: MILO Test Cases