1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Performance Driven Technology Mapping
for
Cell Generators

by

Clifford B. Liem, B.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Faculty of Engineering
Carleton University
Ottawa, Ontario
August, 1991

© 1991, Clifford B. Liem

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70905-7

Canada

Thesis contains black and white and/or
coloured graphs/tables which when
microfilmed may lose their significance.
The hardcopy of the thesis is available
upon request from Carleton University Library.

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**“Performance Driven Technology Mapping
for
Cell Generators”**

submitted by Clifford B. Liem, B.Sc.
in partial fulfilment of the requirements for
the degree of Master of Engineering

Professor M.C. Lefebvre
Thesis Supervisor

Professor J.S. Wight
Chairman,
Department of Electronics

Carleton University
August 14, 1991

Abstract

Technology Mapping is the final step in Logic Synthesis and is defined as the transfer of a technology-independent logic network into a circuit realization in a given technology. This thesis discusses the design and implementation of a prototype technology mapper for cell compilers, called Rossini. Issues for the performance optimization of circuits through the utilization of complex cells in a CMOS (Complementary Metal Oxide Semiconductor) technology are addressed. Rossini improves on current mapping techniques by introducing a constraint-driven module whose parameters are defined by the abilities of the cell compiler. This *constructive* matching technique allows mapping of networks for very large cell families. New partitioning and covering approaches ensure that the mappings are also of high quality. Using this platform for experiment, a study of the impact of different sized cell families is presented. The study shows that larger cell families offer a savings in the area and delay of technology mapped circuits.

Acknowledgements

A piece of work such as this is only possible when a healthy environment of resources are available. Those resources in this case include a large number of individuals, state of the art equipment, and financial participation from several sources.

First and foremost, I wish to thank my supervisor, Professor Martin Lefebvre, who has introduced the subject to me and guided me throughout the project. His dedication to my progress has been highly motivational.

I wish to thank the faculty, staff, and my fellow graduate students of the Department of Electronics for enduring my endless problems. In particular, Richard Griffith, David Skoll, Walter Fergusson, and Mike Gagnon have been especially helpful.

Thanks are extended to Frederic Mailhot for a number of helpful technical discussions on the problem of technology mapping.

Finally, I wish to acknowledge the following sources of financial support: the Natural Sciences and Engineering Research Council (N.S.E.R.C.) through a 1967 Postgraduate Scholarship; the Department of Electronics, Carleton University; and Bell-Northern Research (B.N.R.), Ottawa. The prototype implementation and experimentation were run on a network of SUN workstations provided by the Canadian Microelectronics Corporation (C.M.C.).

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Tables	vi
List of Figures	vii
List of Appendices	ix
Chapter 1. Introduction	1
1.1 Perspective	1
1.2 Technology Mapping Overview	3
1.3 Objectives	7
1.4 Outline of Thesis	7
Chapter 2. Related Work	8
2.1 Socrates	8
2.2 Dagon	13
2.3 MIS	16
2.4 Ceres	20
2.5 Grenoble	27
2.6 Eindhoven	29
2.7 Summary and Evaluation	33
2.7.1 Representational Levels and Matching Methodologies	33
2.7.2 Partitioning and Covering Strategies	35
2.7.3 Mapping for Large Complex Cells	36
2.7.4 Suggestion for a New Approach	37

Chapter 3. The Rossini Approach	38
3.1 Outline of Mapping System	38
3.2 Tree Partitioning	39
3.2.1 Balanced Binary Decomposition	40
3.2.2 Unbalanced Binary Decomposition	42
3.3 Constructive Matching	43
3.4 Covering	48
3.4.1 Cost Models	48
3.4.2 Network Delay Modelling	48
3.4.3 Phase Assignment and Tree Covering	49
3.4.4 Tree-boundary phase assignment	52
3.5 Summary	54
Chapter 4. Logic Minimization	55
4.1 Basic Minimization Operations	55
4.1.1 Extraction	56
4.1.2 Elimination	57
4.1.3 Simplification	58
4.1.4 Resubstitution	59
4.1.5 Decomposition	61
4.2 Minimization for Delay	62
Chapter 5. Implementation and Experimental Results	64
5.1 Prototype Implementation and Experimental Procedure	64
5.2 Cell-level Area and Delay Models	66
5.3 Logic Minimization	67
5.4 Experimentation and Results	68
5.4.1 Balanced versus Unbalanced Decomposition	68
5.4.2 Tree-boundary improvement	69
5.4.3 Rossini versus MIS II	69
5.4.4 The impact of library size on performance.	70
5.4.5 Design Space Exploration	74
5.5 Chapter Summary	75

Chapter 6. Conclusions	76
6.1 Summary	76
6.2 Contributions	77
6.3 Recommendations for Future Work	78
Appendix A. Glossary of Terms and Symbols	79
Appendix B. Rossini User Manual	81
Appendix C. Sample Common Lisp Code - Matching Module	85
Appendix D. Relationship between (s,p,l,n) parameters.	89
Appendix E. String based pattern matching	92
References	94

List of Tables

Table 2.1	Results of comparison between tree mapping and graph mapping [3]. .	18
Table 2.2	Enumeration of (s,p) gates [3].	19
Table 2.3	Results using different size libraries of complex gates [3].	20
Table 5.1	Decomposition and Boundary techniques.	69
Table 5.2	Results for MIS II and Rossini using family (s,p,l,n) = (3,3,4,9). . .	70
Table 5.3	Delay optimization for different sized cell families.	71
Table 5.4	Impact of cell-families on collapsed versions of benchmarks. . . .	72
Table 5.5	Area optimization for different sized cell families.	73

List of Figures

Figure 1.1	Perspective of Logic Synthesis.	3
Figure 1.2	Multi-level Logic Network Forms.. . . .	4
Figure 1.3	DAG and Tree Exclusive-or Representations.	5
Figure 1.4	DAG and Tree mapping.. . . .	5
Figure 1.5	Partitioning of a DAG at points of multiple fanout.	6
Figure 2.1	Example rules for circuit optimization in Socrates [1].	9
Figure 2.2	Sample Pattern and matched target configuration [15].	10
Figure 2.3	The effects of different rule orderings [16].	11
Figure 2.4	Dagon Overview [11].. . . .	13
Figure 2.5	Sample Dagon pattern - family of AOIxxx gates [11].	14
Figure 2.6	Complex Pattern AOI432.	14
Figure 2.7	Dynamically covering a sample tree after matching [11].	15
Figure 2.8	Two different decompositions for a 4-input NAND gate [3].. . . .	16
Figure 2.9	Effect of DAG partitioning.	17
Figure 2.10	CMOS Complex Gate and corresponding tree [3].. . . .	19
Figure 2.11	CMOS logic implementations for Example 1 [12].	21
Figure 2.12	Recursive Shannon decompositions for f, g, and h.	22
Figure 2.13	Graphical Depiction of Functions f, g, and h.. . . .	22
Figure 2.14	NPN-equivalent set and graphical representation.	24
Figure 2.15	Matching compatibility graph for 3-Variable Boolean space [12].	25
Figure 2.16	Matching compatibility graph for Example 1.	25
Figure 2.17	Sample AOI221 cell with absorption cost [13].	27
Figure 2.18	Matches requiring/not requiring expansions [13].	28
Figure 2.19	Sample obligatory expansion [13].. . . .	28
Figure 2.20	Level-maintaining expansion step [13].	29
Figure 2.21	Second phase - conversion into cells [13].	29
Figure 2.22	Graphical representation of a Boolean expression [14].. . . .	30
Figure 3.1	Rossini Overview	39
Figure 3.2	Binary decompositions of a primitive tree.. . . .	40

Figure 3.3	Forming a Balanced Decomposition	.41
Figure 3.4	Constructing subtrees for constructive matching.	.44
Figure 3.5	A sample tree matched at its root.	.45
Figure 3.6	Constructive matching algorithm.	.46
Figure 3.7	Search Tree for Constructive Matching Algorithm	.47
Figure 3.8	Combinatorial Network with Primary Inputs and Outputs.	.49
Figure 3.9	Phase-assignment covering example.. . . .	.50
Figure 3.10	Example Tree-boundary before optimization.	.53
Figure 3.11	Circuit after Tree-boundary heuristic.. . . .	.53
Figure 5.1	Experimental Set-up	.65
Figure 5.2	Delay model of a CMOS cell.. . . .	.66
Figure 5.3	MIS II Minimization Scripts	.68
Figure 5.4	Mappings for different sized libraries.	.74

List of Appendices

Appendix A.	Glossary of Terms and Symbols	.79
Appendix B.	Rossini User Manual	.81
Appendix C.	Sample Common Lisp Code - Matching Module	.85
Appendix D.	Relationship between (s,p,l,n) parameters.	.89
Appendix E.	String based pattern matching	.92

Chapter 1: Introduction

1.1 Perspective

In the past decade, the integrated circuit community has seen parallel advancements in two prominent areas. The first is the constant improvement in integrated circuit processing technology. An example arises in CMOS (Complementary Metal Oxide Semiconductor) integrated circuits. The ability to produce smaller and smaller layout geometries are indicative of the progress of this technology. A recent advancement is the ability to create high-performance sub-micron gate length transistors. One direct result arising from this concerns series-connected transistor structures. Studies of the abilities of these technologies [1] have given a clear indication that the number of series-connected transistors may be increased from the number that current technologies allow. Whereas 2 μm designs may tolerate a maximum of five transistors in series, sub-micron technologies may allow six or seven transistors in series. This result has a drastic effect on the construction of complex digital cells. A complex cell is a combinatorial logic structure, which in CMOS forms an alternating chain of Boolean “AND” and “OR” functions with an “INVERT” at the output. As the number of allowable series-connected transistors increases, the number of feasible complex cells escalates to an alarming degree. For example a restriction of three MOS (Metal Oxide Semiconductor) transistors in series allows the construction of 87 different complex cells. The number for four transistors is 3503 and the number for five is 425 803!

The second advancement in the integrated circuit field is in the adaptation of design methodology. Designers have shifted emphasis from hand-tailored schematic capture to computer-aided synthesis. It is clear that the former advancement provides fuel for the second. Better technologies allow for more complex applications which causes the need for computer aids. Unfortunately, because of the rapid nature of the advance in integration

technology, developments in synthesis tools which tie in applications lag behind the advancement in processing technology. This thesis hopes to reduce this gap through a study of the impact of large complex cell families on the quality of synthesized integrated circuits. Recent studies on this topic have given favorable results for large cell libraries [2][3]. It is anticipated that when the sizes of cell libraries are increased in accordance with newer technologies, the trend of results will continue.

Logic Synthesis is the automatic design of digital integrated circuits from a behavioral level specification to the chip or layout level. Technology Mapping is the final step in logic synthesis and is defined as the process of converting a technology-independent logic-level network into a circuit realization in a given technology. To evaluate the effectiveness of complex cell families of technology mapping, one must take a global view of the synthesis process. For combinational logic, this view involves logic minimization and technology mapping, before layout. This study, being technology driven, gives emphasis to the mapping stage.

In addition to the effect cell family size has on the role of logic minimization and technology mapping, the possibility of such a large number of cell structures weakens the standpoint of a cell library based design methodology. The problem of creating a complete set of hand-crafted custom standard cells may become too cumbersome. The competing methodology is design using custom cell compilers. A cell compiler has many advantages, one being the flexibility of creating a variety of custom cells. However, the link between Logic Synthesis and cell compilation remains largely unexplored.

In this thesis, we examine many approaches to technology mapping to find the best way in which to execute synthesis for compiled complex cell families. The approach which is developed is based on *constructive* matching, a new method of identifying implementations for sub-trees of the logic network which can be realized in complex structures. In association with this module, are partitioning and covering modules which complete the technology mapper. These modules have been oriented for performance optimization. A

pictorial perspective, showing the *world* of Logic Synthesis as a number of sets, is in Figure 1.1.

Figure 1.1 Perspective of Logic Synthesis.

1.2 Technology Mapping Overview

In the variety of approaches to technology mapping, a common backbone can be extracted. The problem can be divided into three major tasks. The first task is parsing of the logic-level network into a representational level and preparing the network for the execution of technology mapping. The second task is the identification of sub-network sections for which physical realizations are feasible, also known as *matching*. The third task is the method of selecting when and where implementations are applied to the network, or *covering*. These tasks can have very different meanings depending on the type of approach.

Figure 1.2 Multi-level Logic Network Forms.

A multi-level logic network has various forms, two of which are shown for an example in Figure 1.2. While the targeted representation is always in netlist form, mapping can be executed from a number of representations which may or may not have been translated from some logic-level form. These representations include the DAG (Directed Acyclic Graph) representation, a tree-based representation, a Boolean representation, or another netlist representation. Sometimes, a representation can fall into more than one category. The choice of representation inevitably has a large impact on the matching and covering stages of the technology mapper.

The DAG form best represents the nature of the logic. Mapping from this level would be a desirable achievement if attained optimally, since first there are logic elements which can best be described as a DAG and are difficult to represent in other forms such as a tree (e.g. multiplexers, exclusive-or gates, gates with multiple outputs (e.g. AND-NAND gates)). For example, consider Figure 1.3. It is much easier to identify the DAG construct than the tree construct for an exclusive-or cell, since there are multiple common inputs.

Figure 1.3 DAG and Tree Exclusive-or Representations.

However, for the purpose of complex cells, this advantage has been eliminated, since complex cells can easily be represented as a tree without any loss of information.

The second advantage of a DAG is that it can be mapped globally, whereas other representations are usually formed from chopping the graph into pieces. Thus, cells which cross boundaries cannot be found. For example, consider the DAG in Figure 1.4. The AND-OR cell shown can obviously form the function for the left output of the DAG (as long as the right output is implemented in some other way). However, this can never be found if the DAG is chopped into trees, in the manner shown.

Figure 1.4 DAG and Tree mapping.

Unfortunately, optimal covering of a DAG by a set of cell DAGs has been shown to be NP-complete [4][5]; therefore, many have reduced the problem. The most popular is the tree-based representation which has been shown. It is most often obtained by partitioning the DAG into a forest of trees, by making each point of multiple fanout the root of a new tree. This procedure is shown in Figure 1.5 for a netlist DAG. In this reduction, the trade-off is the ability to globally optimize. Optimization is restricted to within trees and at tree boundaries. There then arises the question of whether it is better to try to optimally solve the subproblem or suboptimally solve the original problem.

Figure 1.5 Partitioning of a DAG at points of multiple fanout.

For the task of matching, the desired implementations may be found in a standard cell library or in the set of cells which can be formed by a compiler. In many mapping systems, a node in the network is matched to all the cells in a library which will perform the desired function. Then the third task is to select the best implementation. Other systems match sections of the network according to some cost scheme. Often then, the selection task is done in conjunction with the matching process.

When matches are performed according to a cost scheme, covering methods are usually quite simple and particular to the system. Covering networks which have been matched for all possible implementations is usually approached by dynamic programming, where a large unwieldy problem is broken into small sub-problems [6]. Other techniques may involve the use of weighted constraints.

1.3 Objectives

The first objective of this thesis is to investigate the impact of the size of complex cell families on the quality of technology mapped circuits. The quality is measured first by high performance then by area utilization. This objective will be met first by an in-depth study of existing technology mapping systems, in an attempt to form the best method of approach.

The second objective is to formulate useful performance-directed technology mapping techniques, which can be implemented in a platform for future experimentation. This prototype should be capable of fully utilizing large complex cells and should lend itself to enhancements.

1.4 Outline of Thesis

The thesis is organized as follows. Chapter 2 reviews the background work done in technology mapping as has been contributed to the literature. It describes each approach, then provides criticism for each system. Following this is a description of the foundation for a new approach. Chapter 3 describes the new approach which has been implemented in a prototype called Rossini. Chapter 4 is a tutorial overview of logic minimization. Logic minimization is a very active area of research [7][8][9] and a full study is beyond the scope of this thesis. However, a basic understanding of key operations and strategies is necessary, since minimization is usually necessary for high quality technology mapping. Chapter 5 discusses issues for the prototype implementation of the mapping system, the experimental set-up and various experimental results. Chapter 6 concludes with a summary, contributions, and recommendations for future work.

Chapter 2: Related Work

The concentration of this chapter is on six of the more prevalent technology mapping systems which have been recently introduced to the logic synthesis community. The systems under study are Socrates [10], Dagon [11], MIS [3], Ceres [12], a system developed at L'Institut National Polytechnique de Grenoble [13], and a system developed at Eindhoven University [14]. The first four systems perform technology mapping for standard cell libraries. The system from Grenoble works on a particular type of complex-cell library. The last system, from Eindhoven University directly addresses the problem of technology mapping for cell compilers, specifically, the use of complex gates.

The methods used by each synthesis system are carefully examined and outlined based on available information. The final section consists of an evaluation and criticism of the abilities of each system. This forms the foundation for a new mapping approach for complex cell families.

2.1 Socrates

Socrates (Synthesis and Optimization of Combinatorics using a Rule-based And Technology-independent Expert System) is a complete system for automating the synthesis of integrated logic. Technology mapping is performed on a netlist representational level by a Circuit Optimizer. A subject network is translated from a logic level directly to a circuit comprised of generic AND, OR, and NOT gates. The representation contains gate, pin, and connectivity information.

The optimizer improves circuit characteristics by iteratively replacing and rearranging small network divisions. It achieves this by making circuit alternatives available in the form of *if-then* rules [15][16]. Rules in the knowledge-base describe a local mapping of

one circuit configuration (target configuration) to another equivalent, yet more desirable, alternative (replacement configuration). Some examples are shown in Figure 2.1. The desirability is based on a cost function, which for a particular implementation depends on the cost requirement for that implementation and the number of rules which can subsequently be applied. The rules are broken into classes: area rules, timing rules, and general rules (a combination of area and time savings). These classes are again grouped into sub-classes which represent different phases of optimization. The rules within the subclasses are ranked by their relative merit in expected savings, which can only be approximate, because the cost depends on the location in the circuit.

Figure 2.1 Example rules for circuit optimization in Socrates [1].

There are four main operations performed during the optimization process: *matching*, *cost function evaluation*, *selection*, and *replacement*.

Matching in Socrates involves recognizing target configurations which correspond to the antecedent of a rule. It is achieved by storing the target as a *pattern*, encoded as a conjunction of conditions that must be satisfied. Gates and pins which satisfy these conditions are stored in *matcher gate registers* and *matcher pin registers* respectively.

Figure 2.2 depicts a sample pattern with a portion of a circuit that matches the pattern. In matching this particular pattern, the matcher first checks if the gate stored in gate register G_0 is a two-input NOR gate. If true, then some pin of G_0 is stored in pin register P_1 . If P_1 is connected positively (through an even number of inverters) to a two-input NAND gate pin (P_2), then it is stored in G_1 and its input pins are stored in P_3 and P_4 . If not,

it is possible that the net size may be greater than one, in which case another pin can be checked. Returning to G_0 , the pattern matcher continues to meet all the specified conditions. If a condition is not met, the matcher backtracks to the last selection it made and continues. The routine runs until the last condition is satisfied or until all possibilities have been rejected. If a pattern has been matched, the registers hold all the information necessary for the replacement module to link the replacement configuration.

1	node_class	g0	nr2
2	pinof	g0	p1
3	pinclass	p1	nr2_inputs
4	net_size	p1	1
5	con_pos	p1	g1
6	assign_con	p2	g1
7	pin_class	p2	nd2_output
8	net_size	p2	1
9	node_class	g1	nd2
10	pinof	g1	p3
11	pinclass	p3	nd2_inputs
12	pinof	g1	p4
13	pinclass	p4	nd2_inputs
14	pinof	g0	p5
15	pinclass	p5	nd2_inputs
16	net_size	p5	1
17	con_pos	p5	g2
18	assign_con	p6	g2
19	pin_class	p6	nd2_output
20	net_size	p6	1
21	node_class	g2	nd2
22	pinof	g2	p7
23	pinclass	p7	nd2_inputs
24	pinof	g2	p8
25	pinclass	p8	nd2_inputs
26	pinof	g0	p9
27	pinclass	p9	nr2_output

Figure 2.2 Sample Pattern and matched target configuration [15].

There are also conditions within the matching module that can check for timing conditions, such as the slowest input of a gate or the excess of a maximum capacitance of a network.

Cost is defined as a function of area savings of an implementation and number of rules applicable in the future.

$$\text{cost} = R * (\# \text{ applicable rules}) + S * (\text{area saved})$$

The replacement process consists of a module with three steps:

- the replacement configuration (which can be empty) is built.
- connections to the existing circuit are performed.
- the target configuration is deleted and general cleanup is executed.

Step 3 involves the deletion of superfluous gates, e.g. extra inverters, and actions includ-

ing even distribution of output loads between drivers which may have been added to increase the drive capability.

The selection process is critical to the effectiveness of the application of rules. For example, Figure 2.3 shows an example of different orderings of rules at different locations in a circuit leading to drastically different results.

Figure 2.3 The effects of different rule orderings [16].

For these plus experimental reasons, it was found necessary to use a *look-ahead* mechanism, a test on the results of a short sequence of applied rules. A control module implements this in a search strategy using a search tree. The nodes of the tree represent different implementations of the circuit, while the arcs represent required rule transformations connecting implementations. The root of the tree is the current implementation. Children of parent nodes are ordered from left to right with the leftmost child being the *best* implementation. While the tree is constructed, the cost of each node is evaluated and saved.

Naturally, the search tree can become large and expensive to construct. Thus, pruning heuristics are required:

- the number of children of a single node is limited by a parameter B (i.e. the *breadth*).
- the *depth* of the tree is limited by a parameter D_{max} .
- the area of application of a rule is restricted by a parameter N (i.e. the *neighborhood*), which is defined as the maximum length of a path connecting neighbor gates to a *center* gate.

This last item avoids situations where there are competing selections. A restricted area al-

lows for more independent application of rules.

Two parameters Δ_{class} and Δ_{cost} also affect the building and traversal of the search tree. The control module is allowed to look Δ_{class} rule classes beyond the current subclass to select competing rules and it is allowed to increase the cost function by Δ_{cost} . This is intended to prevent the routine from falling into local minima.

The first attempt using this control strategy involved the use of these parameters as fixed values. A series of experiments proved that the values $B=3$, $D=2$, and $N=1$ gave the most efficient results. There was limited success. It improved the quality of circuits optimized without the look-ahead mechanism by about 10 percent, but at a large cpu time expense. In retrospect, the amount of look-ahead needed varies greatly over the course of optimization. Look-ahead is very useful in later phases, while sometimes superfluous with powerful rules in early stages. This bred the idea of a meta-system, whereby parameters are adjusted dynamically corresponding to the state of a circuit.

In the same vein, optimization can be improved by dynamically varying the weights of the cost coefficients. For instance, since rules applied in the later stages will not likely improve the circuit as much as earlier rules, a progressively lower value for Γ could reflect this.

The new problem is then monitoring of the optimization process in order to determine values for the control parameters. This was solved in a statistical manner. A series of experiments were run to determine how *deep* and *wide* the search tree had actually been traversed during optimization. Thus, statistically frequent values can be used for control parameters. Using this technique, the run-times of Socrates were reduced to ~40% of those using rigid control parameters.

Given this 'look-ahead search and optimize' scheme, optimization for speed can be done by traversing the critical path of a circuit, using these gates as centers from which to construct local neighborhoods. Using the performance rule class, timing constraints can be

met. However, this type of optimization may require several passes, since critical paths will change throughout the mapping process.

2.2 Dagon

Dagon is a technology mapping system developed at AT&T Bell Laboratories. The writers of Dagon associate the problem of technology mapping with code generation in the field of language compilation, mapping a graph-like description of a circuit to a library of patterns for a given technology.

Dagon uses a tree-based representation, where Boolean functions are represented with operators (AND, OR) at the nodes and either phase of the signals on the edges. An overview of the Dagon approach is illustrated in Figure 2.4. A logically minimized netlist is parsed into a DAG, then partitioned into a forest of trees. The trees are then matched to technology patterns to produce the final technology bound circuit.

Figure 2.4 Dagon Overview [11].

Dagon finds minimal cost coverings of the partitioned trees by finding a list of candidate matches and choosing the best candidate. The matching stage is made possible by the *twig* tree manipulation program. The covering stage involves an evaluation of competing cost functions. Dagon guarantees an optimal match among the list of candidates.

Patterns are described in Dagon as small trees in a NAND/NOT form, which correspond to direct mappings to cells in the target technology, with possible local transformations. These pattern representations have three parts which *twig* uses (see Figure 2.5). The first part is a grammatical form of a family of pattern instances. The second part contains a cost evaluation. Use of a particular pattern would subject a cost variable to be updated with respect to the area and time penalty of the use of that tree. The timing information is dependent upon where the tree is rooted; therefore, the patterns below this root would have to be bound to specific values before this, suggesting the use of a *bottom-up* covering technique. The third section contains the action information. Figure 2.5 shows a sample pattern consisting of a family of AOIxxx (AND-OR-INVERT) gates.

```

/** AOI canonically expressed as
    not
    |
    nand_3
    |
    |
    nand nand nand */
eqn: not(nand_3(inand,inand,inand))
/*SS refers to the root of the pattern */
{
  DEFAULT_COST; /*sum up cost of children*/
  cost.cost_a[AREA] += d_get_aoi_area_cost(SS)
  cost.cost_a[TIME] += d_aoi_time_cost(SS)
  cost.cost_a[AT_K] += my_pow(cost.cost_a[TIME],
    cost_power) * cost.cost_a[AREA];
}
= {
  d_print_aoi(stdout,SS);
};

inand: nand_2(eqn,eqn)
{} /*default cost is the sum of the children*/
= {} /*action taken at root of pattern tree*/
inand: nand_3(eqn,eqn,eqn)
{} /*default cost is the sum of the children*/
= {} /*action taken at root of pattern tree*/
inand: nand_4(eqn,eqn,eqn,eqn)
{} /*default cost is the sum of the children*/
= {} /*action taken at root of pattern tree*/
inand: not(eqn)
{} /*default cost is the sum of the children*/
= {} /*action taken at root of pattern tree*/

```

Figure 2.5 Sample Dagon pattern - family of AOIxxx gates [11].

An example from this family of gates is depicted below.

Figure 2.6 Complex Pattern AOI432.

A straightforward approach of traversing the tree, trying each pattern at each node would take time $O(\text{tree_size} \times \text{pattern_set_size})$. However, *twig* achieves this in time $O(\text{tree_size})$ in an approach based on the Aho-Corasick algorithm [17]. The tree-matching problem is mapped onto a string-matching problem. String matching is discussed in Appendix E.

Having found all the candidate matches of a tree, the covering task is to find an optimal match given this search space. Dagon incorporates dynamic programming to find a least cost match. The procedure chooses the best matches for each of its subtrees in a depth-first recursive manner. The process consists of trying each pattern that has matched at a root of a subtree. The cost of matching the pattern is the cost of the pattern plus the cost of each of the subtrees with roots at the leaves of that pattern. The time for this process is $O(\text{tree_size} \times \text{pattern_set_size})$. As an example, Figure 2.7 shows pattern matches for a tree. The algorithm begins at the leaves of the tree, moving towards the root. There are no choices for least cost until the inverter at root A is visited. There is a choice between complex gate AOI21, and INV. The cost of the inverter match is INV (2 grids : cost is in width units of a standard cell) plus the cost of the subtree rooted at B (11 grids). The cost of the complex gate match is the cost of AOI21 (4 grids) plus the cost of the subtrees rooted at its leaves: two inputs (0 grids) and the ND2 at E (3 grids). The complex gate match wins with a cost of 7 grids over 13 grids.

Figure 2.7 Dynamically covering a sample tree after matching [11].

This technique is dynamic in that a match is not rigidly bound to a part of the circuit. At

any time a better match closer to the root of the tree may give a new mapping.

This method finds the optimal match for each tree; however, this method may give far from the optimal graph covering, as was discussed in the introduction.

2.3 MIS

MIS (Multilevel Logic Interactive Synthesis System), developed at University of California at Berkeley, is a logic optimization system [7] as well as a system for technology mapping. Two mapping approaches are followed. The first is the tree-based covering and the second is direct DAG covering.

A network is first minimized, then decomposed into two-input NAND gates and inverters before mapping. The patterns are also in the form of two-input NAND gates and inverters. The pattern libraries are generally large, as all non-isomorphic and irredundant forms of trees are represented. Figure 2.8 depicts an example of two non-isomorphic trees for a four-input NAND gate. In this way, large gates patterns can be matched to a variety of trees.

Figure 2.8 Two different decompositions for a 4-input NAND gate [3].

MIS tries a second DAG partitioning scheme, as well as rooting trees at points of multiple fanout, in an attempt to gain optimality across tree boundaries. Consider the example in Figure 2.9. The original partitioning method would cause breaks at branches 1

and 2, making the NOT function the root of a new tree. This could be implemented in the first realization. If, however, the trees were partitioned otherwise, the second realization may have resulted. This realization has doubled some of the logic, but has potentially given a better performing network. It is also likely to take less area, although, doubling logic will generally increase area.

Figure 2.9 Effect of DAG partitioning.

In this second partitioning method, MIS takes only primary outputs as roots. The mapping begins here, then continues until a primary input is encountered or until it reaches a node that has been previously mapped. If this node is the output of an existing cell, then it is treated as a leaf cell. If the node is internal to a cell, then the mapping continues to the input nodes of that cell. Thus, logic may be duplicated, but not in all cases. The order of mapping begins with the primary output estimated as being from the critical path, then moves towards the less critical paths' outputs.

A unique supplement to assist the matching procedure is the addition of inverter pairs to all the nodes in the network. This allows more cells to be matched at a larger number of nodes in the network. To eliminate unwanted parasitics to this procedure, a pseudo cell of a series inverter pair is added to the library with a cost of zero. Also a post processor is required to delete all double inverter occurrences. One problem using the addition of inverters is in the treatment of nodes with multiple fanouts. Inverters are inserted across a tree boundary; one belongs to the original node's tree and the others to new trees. The op-

timization of the trees does not reflect the true cost of the presence of these inverters.

Again, dynamic programming is the choice for tree covering.

The second method for technology mapping uses an algorithm for matching the network DAG with graphs of cells. The graph matching algorithm turns a cell-graph into a push down automaton. First, automata are created from the cells in the library by traversing the arcs in a cell starting at the output in a depth first manner, while adding this information to a list. During the matching process, arcs in the network are traversed in the same sequence as those in the cells. As a node in the network is visited, it is bound to a node in the cell. A match occurs at bound nodes if the fanin is the same.

Instead of using dynamic programming, the covering problem is viewed as a weighted constraint problem of all matches over the network. A large matrix represents the covering, with rows for nodes in the network and columns for each match. Heuristics have been employed as covering solutions. Approaches include greedy strategies, such as choosing matches that cover the most nodes or choosing a match that maximizes the number of nodes divided by the cost. The process is made faster with branch and bound techniques, cutting down the search space.

The results of a comparison between the tree-matching algorithm and DAG matching algorithm for various benchmark circuits are shown in Table 2.1. All but one case show a better area utilization by the tree-matching procedure. The suspected reason is that only one gate in the library is a true graph, an exclusive-or gate.

	tree mapping		graph mapping	
name	gates	area	gates	area
alupla	130	260	167	326
bw	159	334	165	322
misex1	47	88	58	113
misex2	92	187	119	251
sao2	101	245	133	271

Table 2.1 Results of comparison between tree mapping and graph mapping [3].

Technology mapping for CMOS complex gates was considered. Given the number

of complex gates satisfying a constraint on the number of devices in series, the number of 2-input NAND gate trees that are equivalent to each complex gate gives the number of patterns.

Figure 2.10 CMOS Complex Gate and corresponding tree [3].

Complex gates are combinatorial structures, such as AND-OR-INVERT gates, which may be implemented as series-parallel combinations of nMOS and pMOS transistors. The (s,p) constraint defines the maximum number of pMOS transistors from the power rail to the output (s), and the maximum number of nMOS transistors from the output to the ground rail (p). Figure 2.10 shows an example of an (s,p) = (3,3) CMOS complex gate with corresponding tree. Table 2.2 shows a tabulation of the number of complex gates for an (s,p) constraint.

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	7	18	42	90	186
3	3	18	87	396	1677	6877
4	4	42	396	3503	28435	222913
5	5	90	1677	28435	425803	6084393
6	6	186	6877	222913	6084393	154793519

Table 2.2 Enumeration of (s,p) gates [3].

The methods were tested with different sizes of complex gate libraries for a particular design. The results are shown in Table 2.3. Notice that a library consisting of all (3,3) gates provides a 25% improvement over the smaller library comprised of just 2 and 3 input NAND and NOR gates.

s	p	level	# in library	# of patterns	# mapped	# grids
2	2	1	3	3	721	2020
2	2	2	7	7	495	1568
3	3	1	5	5	551	1680
3	3	2	27	49	363	1304
3	3	5	87	118	341	1260
4	3	1	6	7	498	1574
4	3	2	62	190	311	1260
4	3	7	396	839	286	1150
4	4	1	5	9	498	1574
4	4	2	123	1171	327	1174

Table 2.3 Results using different size libraries of complex gates [3].¹

2.4 Ceres

Ceres is a technology mapper developed at Stanford University. The major point that distinguishes this approach from the previously examined methods is the representational level. Ceres uses Boolean representations; therefore, the matching algorithm is based on Boolean techniques. Covering is done using dynamic programming modified for Boolean expressions. Boolean representations handle multiple occurrences of variables much more adequately than tree-based representations, and they allow the use of *don't care* information which is useful for optimization.

The importance of don't care information lies in the ability to improve area and performance in cell-based designs. This is best demonstrated in an example. Suppose a function is described by its *on* set (i.e. the set of conditions that allow the function to have logic value 1) and its *don't care* set ² (i.e. the set of conditions in which the function may have either logic value 1 or 0).

The CMOS implementation of the function in Example 1, that uses the fewest liter-

1. (sic.) The maximum level for an s,p=(3,3) complex gate is 4 and an s,p=(4,3) complex gate is 5. This is shown in Appendix D.

2. Don't care sets are explained in Chapter 4.

als would require 6 transistors. This would be a complex gate as shown in Figure 2.11 (a) (2 extra transistors may be required to invert the output). However, a cell library which utilizes pass-transistors may have a much faster and more area-efficient multiplexer which could do the same job. An implementation of the logic $a\bar{b} + bc$ requires an extra literal (4), but uses 4 transistors as shown in Figure 2.11 (b) (this implementation may also require two extra transistors to invert b , unless $\bar{b}$ is available). It is also likely to be faster. This can be reflected in the cost function.

$$F = (a + b) c$$

$$DC = \bar{b} \bar{c}$$

Example 1

Figure 2.11 CMOS logic implementations for Example 1 [12].

Boolean matching is the process of checking the *tautology* of a target function with a function from a library. The matching problem is closely interrelated to the phase assignment problem (assigning phase of a signal for logic functionality) mainly through the cost of the implementations. If we denote a target function by $F(x_1\phi^1 \dots x_n\phi^n)$, where it has one output and n inputs where $x_i\phi^i = x_i$ for $\phi_i = 1$ and $x_i\phi^i = \bar{x}_i$ for $\phi_i = 0$. F matches a library function $G(y_1 \dots y_n)$ if either of the following equations are true:

$$F(x_i\phi^i \dots x_j\phi^j) = G(y_1 \dots y_n)$$

$$\bar{F}(x_i\phi^i \dots x_j\phi^j) = G(y_1 \dots y_n)$$

for some permutation of the variables. For example, any function $F(x_1, x_2)$ in the set $\{ x_1 + x_2, \bar{x}_1 + x_2, x_1 + \bar{x}_2, \bar{x}_1 + \bar{x}_2, x_1x_2, \bar{x}_1x_2, x_1\bar{x}_2, \bar{x}_1\bar{x}_2 \}$ can be implemented by the library func-

tion $G(y_1, y_2) = y_1 + y_2$. Note that in this case $n = 2$ and the library element can cover $n!2^n = 8$ functions.

An algorithm for checking the tautology of two functions can be executed through recursive Shannon decomposition [8]. Shannon decomposition is the act of cofactoring an algebraic Boolean expression. The following is an example of the recursive Shannon expansions of three functions f , g and h .

$$\begin{aligned}
 f &= ab + bc + \bar{a}c \\
 &= a(b + bc) + \bar{a}(bc + c) \\
 &= a(b(1+c) + \bar{b}(0)) + \bar{a}(b(c) + \bar{b}(c)) \\
 &= a(b(1) + \bar{b}(0)) + \bar{a}(b(c(1) + \bar{c}(0)) + \bar{b}(c(1) + \bar{c}(0))) \\
 g &= ab + \bar{a}c \\
 &= a(b) + \bar{a}(c) \\
 &= a(b(1) + \bar{b}(0)) + \bar{a}(b(c) + \bar{b}(c)) \\
 &= a(b(1) + \bar{b}(0)) + \bar{a}(b(c(1) + \bar{c}(0)) + \bar{b}(c(1) + \bar{c}(0))) \\
 h &= ab + bc \\
 &= a(b + bc) + \bar{a}(bc) \\
 &= a(b(1+c) + \bar{b}(0)) + \bar{a}(b(c) + \bar{b}(0)) \\
 &= a(b(1) + \bar{b}(0)) + \bar{a}(b(c(1) + \bar{c}(0)) + \bar{b}(c(0) + \bar{c}(0)))
 \end{aligned}$$

Figure 2.12 Recursive Shannon decompositions for f , g , and h .

The equation $f = g$ is a tautology, since both f and g have the same logic values for all the leaves of the recursion, while h is not equal to f or g since it has different logic values in the rightmost leaves of the recursion. This is confirmed through the graphical depiction of the functions in Figure 2.13.

Figure 2.13 Graphical Depiction of Functions f , g , and h .

A match between the library element g and the target configuration f is made when the arguments in the procedure are a tautology for some ordering of the input variables not in the don't care set. At some level n of the recursion, the cofactors of the n^{th} variable of g are checked with the cofactors of all the variables of f , until a match is found. Otherwise, the procedure continues for all permutations and phase assignments. The worst case recur-

sion is $n!2^n$ decompositions. This is manageable for small values of n (i.e. library gates with few inputs), but rapidly becomes very large as n increases. For example, for $n = 4$, $n!2^n = 384$, and for $n = 5$, $n!2^n = 3840$. This makes the algorithm only practical for small values of n .

To filter the number of attempted matches, a few considerations are taken. First, the phase information for unate variables (i.e. a variable which has only one phase present) is not necessary for the determination of matching. Note that the phase information is necessary for binate variables. Also, it is easy to see that any permutation must associate each unate variable in the target function with unate variables of the library function. Similarly, binate variables (i.e. a variable which has both phases present) must be associated with binate variables. This cuts the number of permutations.

Second, variables or groups of variables that are interchangeable in the target function must be interchangeable in the library function. This is a symmetry consideration. For a given function, F , x_i and x_j are said to belong to the same symmetry class and form a symmetry set $\{x_i, x_j\}$ if

$$F(x_1 \dots x_i \dots x_j \dots x_n) \equiv F(x_1 \dots x_j \dots x_i \dots x_n)$$

A necessary condition for matching is that the target function and library element have the same symmetry classes. Therefore, symmetry information can be used to filter out impossible candidates for matching. During the matching process, the ordering of variables in a symmetry set is irrelevant, and, since only variables belonging to symmetry sets of the same size can possibly form a match, permutations need only be done over symmetry sets of the same size. And, since variables may only be interchanged if they are all unate or all binate, the number of permutations of a library element in order to produce a match can be enumerated as $\prod_{i=1}^q S_i^a! S_i^b! 2^{i S_i^b}$, where S_i^a is the number of symmetry sets of size i with unate variables, S_i^b is the number of symmetry sets of size i with binate variables, and q is the size of the largest symmetry set. The factor $2^{i S_i^b}$ considers the phase assignment of the binate variables. For example, consider a library element $MXT = a x z + b \bar{x} z + c y \bar{z} + d$

$\bar{y} \bar{z}$. This has $S^a_1=4$, $S^b_1=3$, so $4!3!2^3=1152$. This is significantly lower than the number of recursions using no symmetry information, $7!2^7=645\,120$.

Unfortunately, a target function cannot be uniquely characterized by a symmetry set when *don't care* information is considered. However, Ceres uses a method whereby symmetry sets can be used along with NPN-equivalent sets to exploit the use of don't cares. A function f is said to be NPN-equivalent to a function g if f can be derived from g through operations of:

- Negation of one or more variables of g .
- Permutation of the variables of g .
- Negation of g .

Ceres uses a matching compatibility graph $G(V,E)$, where each vertex V represents an NPN-equivalent set. For example, Figure 2.14 shows a representation of one NPN-equivalent set for three-variable space. Notice that the representative function can take on any orientation in space, and also the complement representation, i.e. all corners occupied excluding the present occupied corner(s).

Representative function		NPN-equivalent set			
$f = abc$		$\underline{a} \ b \ c$	$\bar{a} \ \bar{b} \ \bar{c}$	$\underline{a}+b+c$	$\bar{a}+\bar{b}+\bar{c}$
$\underline{a} \ b \ \bar{c}$	$\bar{a} \ \bar{b} \ c$	$\underline{a}+b+\bar{c}$	$\bar{a}+\bar{b}+c$
		$\underline{a} \ \bar{b} \ c$	$\bar{a} \ b \ \bar{c}$	$\underline{a}+\bar{b}+c$	$\bar{a}+b+\bar{c}$
		$\underline{a} \ \bar{b} \ \bar{c}$	$\bar{a} \ b \ c$	$\underline{a}+\bar{b}+\bar{c}$	$\bar{a}+b+c$
		$\underline{a} \ b \ c$	$\bar{a} \ \bar{b} \ c$	$\underline{a}+b+c$	$\bar{a}+\bar{b}+c$

Figure 2.14 NPN-equivalent set and graphical representation.

The matching compatibility graph for three variable space is shown in Figure 2.15. Before technology mapping, the vertices of the graph are annotated with elements from the library. Two vertices are interconnected by an edge corresponding to the minterms by which the two vertices differ.

Figure 2.15 Matching compatibility graph for 3-Variable Boolean space [12].

If a library element has been matched to a target function, it corresponds to a vertex in the compatibility graph. If an edge connected to that vertex corresponds to any part of the don't care set of the target function, then the vertex joined by this edge corresponds to an alternative library element *compatible* with the target function. This increases the availability of library elements for matching.

Consider Example 1 given at the beginning of this section ($F = (a + b) c$; $DC = \bar{b} \bar{c}$). The relevant excerpt of the compatibility graph is shown below (the representative function for each vertex is not unique; therefore, we have shown the most useful representation). The don't care set gives three extra possibilities for the implementation of the function F . Notice that one of these could be a multiplexer.

Figure 2.16 Matching compatibility graph for Example 1.

This example shows that the increase of available library elements introduced by the compatibility graph can lead to a better implementation of the function.

The compatibility graph is constructed once for a given value of n and annotated with elements from the library. The graph has been found useful for $n \leq 4$, but is not man-

ageable for $n > 4$. Therefore, for library elements with more than 4 variables, don't care information is not used in the matching process.

Ceres uses dynamic covering. The network is partitioned at positions of multiple fanouts into subject graphs, which are decomposed into a network of two-input gates (AND/OR decomposition). The subject graph is characterized by sub-graphs called *clusters*, having only one vertex with zero out-degree (just as a tree).

Consider the following network, which has been decomposed into two input gates:

$$\begin{aligned} f &= bx_1 \\ b &= e + x_2 \\ x_2 &= \bar{c} + d \\ x_1 &= a + c \end{aligned}$$

The corresponding possible cluster functions are:

$$\begin{aligned} k_{1,1} &= bx_1 \\ k_{1,2} &= b(a + c) \\ k_{1,3} &= (e + x_2)x_1 \\ k_{1,4} &= (e + x_2)(a + c) \\ k_{1,5} &= (e + \bar{c} + d)x_1 \\ k_{1,6} &= (e + \bar{c} + d)(a + c) \end{aligned}$$

The covering algorithm attempts to match each cluster to a library element. It computes the area cost by adding the cost of the matching plus the cost associated with each variable in the equation. Timing cost is computed in a similar way. The propagation delay through a cluster is added to the maximum of the arrival times at its inputs to give the new arrival time at the vertex.

Don't care information is updated as the covering procedure continues. For example, if b is mapped by $OR3(\bar{c}, d, e)$ then the satisfiability don't care set¹ is:

$$DC_b = b c \bar{d} \bar{e} + \bar{b} (\bar{c} + d + e)$$

1. Don't care sets are explained in Chapter 4.

2.5 Grenoble

Sakouti and Saucier [13] of L'Institut National Polytechnique de Grenoble have introduced a technology mapper which works on an autodual library of standard cells. An autodual library is a library in which complex cells are contained, such that the presence of a cell (AOIxyz) implies its dual is also present (OAIxyz). The matching phase identifies pattern trees corresponding to cells in the library in a cost scheme. The covering phase transforms previous patterns into standard cells.

This approach stems from the basis that the tree-based methods of Dagon and MIS suffer from a weakness in the decomposition procedure, where the network and library cells are transformed into a canonical NAND/NOT form. This mapper considers the original network and looks for direct isomorphisms between tree portions and standard cells.

As trees with low depth are the final objective, the standard cells are classified according to their absorption capabilities. The following definitions apply:

α = the level absorption factor, defined as the multiplicative inverse of the number of levels of the most compacted logical AOI tree of the cell.

β = the operator absorption factor, defined as the inverse of the number of operators of the same tree.

γ = the connection absorption factor, defined as the inverse of the number of connections in the same tree.

The cells are classified according to a cost function in decreasing order. The cost function is simply the product of the three absorption factors. For example, consider the AOI221 cell in Figure 2.17.

Figure 2.17 Sample AOI221 cell with absorption cost [13].

Matches are categorized in two groups: matches which require expansions and those which do not require expansions. Examples are shown in Figure 2.18.

Figure 2.18 Matches requiring/not requiring expansions [13].

There is a possibility that a node may have a fanin which is larger than the largest fanin of all the cells in the library. In this case, an obligatory expansion is required. The minimal number of layers introduced in an obligatory expansion is $\text{ceiling}(\log_{N_2} N_1)$, where N_1 is the fanin of the network node, and N_2 is the fanin of the largest cell in the library. For example, if $N_1 = 17$ and $N_2 = 3$, then $\text{ceiling}(\text{Log}_3 17) = 3$ (Figure 2.19).

Figure 2.19 Sample obligatory expansion [13].

Matchings are performed using a greedy strategy. Depth-decreasing pattern matches are those which decrease or don't increase the current number of layers. These solutions are preferred, and are therefore chosen first. If this is not possible, then matches requiring expansions are applied according to cost. The matching begins at the root node of a tree and continues toward the leaves. New nodes created during expansion, which are below the current node, are visited after the current node. New nodes created above the current node are visited on a second pass of the algorithm.

When expansion occurs, nodes having a small depth reference are pushed to a lower level, while the nodes with a large depth reference remain in place. An example is shown

in Figure 2.20. Each leaf is referenced by its depth. This type of manipulation preserves the shallowness of the trees.

Figure 2.20 Level-maintaining expansion step [13].

The second phase of the technology mapping is the conversion of patterns into actual cells in the library. This phase is organized in layers:

- For layer i : the pattern is transformed into the dual form with a virtual inverter included.
- For layer $i+1$: the inverters of the previous layers are absorbed and a direct mapping is performed.

This covering phase is shown in Figure 2.21.

Figure 2.21 Second phase - conversion into cells [13].

2.6 Eindhoven

Berkelaar and Jess [14] of the Eindhoven University of Technology have introduced an approach to technology mapping targeted for a standard cell generator. The mapper has Boolean expressions represented graphically, characterized by their longest path and largest cut set.

Figure 2.22 Graphical representation of a Boolean expression [14].

Figure 2.22 is an example of the graph representing the expression $(+ (* a b c) (* d e (+ f g)(+ h i))(* j k))$. The longest path is the largest number of edges in series (4 for this case). The largest cut set is the largest number of edges which need to be cut when crossing the graph horizontally (coincidentally, also 4 for this case). This representation is particularly useful since it represents the connectivity netlist for the pull-down circuitry of nMOS and CMOS complex-gate logic (the dual graph represents the pMOS pull-up circuitry of CMOS logic).

Any Boolean network has attributed to it an ordered pair (a,b) , which is defined in the following cases:

- if the expression is a sum, a is the largest cut set and b is the longest path of the respective graph.
- if the expression is a product, a is the longest path and b is the largest cut set of the respective graph.

The mapping algorithm is somewhat analytical, but mostly based on heuristics. The problem is approached with a one-level view of the Boolean expression to be mapped. Given an expression:

$$expr = (op \ subexpr_1 \ ... \ subexpr_n), \text{ where } op \text{ is either } * \text{ or } +$$

and a size restriction (a,b) (restricts the allowable size of an expression), a complex cell mapping can be calculated. The parameter a restricts the number of subexpressions that

may be taken for mapping to one cell. If the number of subexpressions exceeds a , then a substitute variable is introduced to take the place of a number of subexpressions, i.e.:

$$expr = (op \ subexpr_1 \ \dots \ subexpr_n)$$

becomes:

$$expr = (op \ subst \ subexpr_1 \ \dots \ subexpr_{n-i})$$

$$subst = (op \ subexpr_{n-i+1} \ \dots \ subexpr_n)$$

the substitute variable is put at the top of the list so that the recursive algorithm takes the lowest level subexpressions at the bottom of the list. Of course, this will not always guarantee the lowest level subexpressions; therefore, the list must be sorted at times. Using a heuristical rule, a subexpression of size (x_1, y_1) is considered *smaller* than a subexpression of size (x_2, y_2) , if $(x_1^2 + y_1^2) < (x_2^2 + y_2^2)$.

Given the library parameters (l, m) , Berkelaar and Jess state that it is wise to take exactly l (for $op = *$) or m (for $op = +$) subexpressions per substitution to take the largest possible standard cell (in the number of transistors in series).

Some characteristics of the algorithm are the following:

- The minimal number of substitutions can be calculated: $nr_of_subst = \text{ceiling}((n-a)/(q-1))$; $q = l$ or m . Given n subexpressions of which only n may remain, $n-a$ subexpressions must be removed. Each time q subexpressions are taken, inserting a new substitute variable. The ceiling of the division is taken since a substitution must be made even if less than q subexpressions must be taken to achieve the restriction of a .
- The minimal logical depth needed to map a function can be found: $min_depth = \text{ceiling}(\log_q n/a) + 1$; $q = l$ or m . With a depth of 1, $n = a$ subexpressions can be mapped. With a depth of 2, $n = a*q$ subexpressions can be mapped. With a depth of d , $n = a*q^{d-1}$ subexpressions can be mapped. Solving for d , the above equation results.
- If one substitution with less than l or m subexpressions must occur, the size of the *small gate* can be calculated: $size_of_small_gate = (n + nr_of_subst - a) \bmod q$; $q = l$ or m . A total of $n + nr_of_subst - a$ subexpressions must be substituted into gates. All are of size q , except one.

The above has shown the problem for one level mapping. For complicated expres-

sions, there can be many levels. As the subexpressions are mapped, the size restriction must be updated so as not to violate the library size (l,m) .

Given a top-level expression, the size restriction (a,b) is set to (l,m) or (m,l) depending on the operator. At this point, there are two cases:

- If $n > a$, then there must be one or more substitutions. This is solved as outlined above, then each of the subexpressions are mapped with size restriction $(b,1)$. (the second parameter is already completely exploited)
- If $n \leq a$, then map all n subexpressions, giving the first subexpression the size restriction $(b, a-n+1)$. Keeping account of the size it actually uses (x,y) , then map the second subexpression with size restriction $(b, a-n+1-y+1)$. The i^{th} subexpression has size restriction:

$$\left(b, a-n+1 - \sum_{j=1}^{i-1} y_j + i-1 \right)$$

In this way, each subexpression may expand until the top level size restriction is filled. If one does not use the whole restriction, the remaining space is given to the next subexpression.

Berkelaar and Jess offer two forms of post-processing. The first reduces the number of inverters. The critical path or paths are freed of redundant inverters using heuristics. This is repeated for the new critical path(s) until no further progress is possible. They report a 22% improvement in delay and a 64% improvement in number of inverters.

The second form of post-processing tries resubstitution along critical path(s), merging gates if possible by the size restriction. This is also repeated along new critical paths. With this process, sometimes logic is doubled to improve speed. They report a 13% improvement in delay.

2.7 Summary and Evaluation

Each of the technology mapping systems presented here have specific strengths offering unique solutions to this final phase of the synthesis process. However, as in many difficult problems, the strengths have companion weaknesses. The following is an attempt to objectively evaluate the systems on their relative merit in these key areas:

- network and pattern representational level
- matching methodology
- covering strategy

The merit can be measured by the effectiveness or optimality of the result with regard first to high performance, then to low area. The last system, from the University of Eindhoven, will be discussed separately, as it approaches the mapping problem from another standpoint.

2.7.1 Representational Levels and Matching Methodologies

These categories are examined together since the greatest impact of the representational level is on the ability to efficiently find matches.

The first system, Socrates, uses a netlist representation, including gates, pins, and connectivity. While this strongly represents the structure of the circuitry, it can prove to be a cumbersome representation. This is apparent when examining the described pattern matching routine. A large number of events are required to recognize simple matches, especially through the backtracking routine. Certainly, the other methods win in this category because of the simplicity of the representational levels. There is no added benefit in the extra information implicit in a gate level description.

The representational level used by Dagon and MIS is a tree structure. It is a better abstraction of the problem, while sufficient to characterize the circuit structure of the logic. This frees the mapper to use a much more efficient matching routine. One problem may lie in the pattern representation. There is a requirement to store all non-isomorphic and irredundant forms of a library element be represented. For a large library with large cells,

this number can become large very quickly, thus increasing the search space for the matching routine. Yet, a benefit of a tree-based structure is that the level of a tree can be a first order estimation of the delay.

The graph-matching used by MIS is an interesting concept, but more information would be required to assess the possibilities of using the routine. Indeed, this may be an important representation if the number of library components which are better represented as DAGs increase. However, this is not the case for complex cells.

The approach by Grenoble also uses a tree-based data structure. In this case, matches are found according to a cost scheme. This is reminiscent of Socrates, which matches desirable structures to areas of the circuit. The cost function for matches lacks correlation with the final circuitry. Too much concern is given to using the largest cells in the library and reducing the number of inverters.

The Ceres technology mapper goes one step beyond Dagon, MIS, and Grenoble. The higher abstraction of Boolean representations is the least cumbersome, thus ideal for fast matching. Recursive Shannon decomposition is quite adequate. However, for large gates, Ceres also must find ways to match more efficiently. Ceres uses symmetry information to reduce the number of reiterations in the matching routine.

In summary of the methodologies, Boolean representations allow for the fastest matching routine, when compared with the tree and netlist representations. It also provides for utilization of symmetry information and matching of gates with multiple inputs. But, while this representation may be useful for fast matching, it may not lend itself for performance directed optimization. The directed connectivity information of a tree may be more useful. Also, the greatest advantage for using Boolean representations is the use of don't cares. This has been shown to be impractical for cells with the number of inputs > 4 . For large complex cell families, most of the cells have a large number of inputs. In any case, these two representations win over Socrates' netlist representation, which can be viewed as an *overkill*.

2.7.2 Partitioning and Covering Strategies

Partitioning and covering methods are the critical factors for the performance of the mapped circuit; therefore, they are considered simultaneously.

The metarule algorithm of Socrates has shown to be an effective method for the utilization of the knowledge base. However, it is first necessary to acquire this knowledge. The resulting circuit is highly dependent on how well these rules are classified and ranked. It appears that the Socrates system can give very good results from the methods being used. Covering along the critical path using the metaknowledge system can potentially give high performance circuits. But it is difficult to continually identify the critical path.

Partitioning of the network is done similarly in Dagon, MIS, and Ceres. Covering is restricted locally with respect to a tree. The popular choice is dynamic programming, which appears to be a natural choice for covering the tree, since it guarantees an optimal mapping for the tree. However, gaining optimality across tree boundaries is a difficult problem. There is a dependency on the type of network which is to be mapped.

MIS offers a second partitioning scheme, considering paths directly from primary outputs back to primary inputs. However, there may be a large amount of duplicated logic which can increase the area unnecessarily. Also, duplicated logic raises concern for fanout count on primary input signals. A large fanout can only degrade performance.

The weighted-constraint matrix used by MIS in the graph representation is interesting, however, no performance oriented strategies are reported.

The Grenoble mapper has the simplest covering phase. Since matches are found using a cost strategy, covering is simply a conversion to library elements, while reducing the number of inverters. For improving performance, the mapper performs depth-decreasing matches, but the lack of a performance-directed covering phase may give sub-optimal results.

In summary of the covering techniques, Socrates' expert methodology is particular

to its system. While it may be effective, it requires a large overhead in expert knowledge and is difficult to update to new technologies. In the case of tree or Boolean matching, there is little doubt that dynamic programming is the best method. For graph based representations, covering does not appear conducive to performance optimization.

2.7.3 Mapping for Large Complex Cells

In each of the prior systems, there has been an attempt to match a network to a library of cells. The Dagon and MIS approach may suffer from a weakness in the explicit pattern representation, which would explode when used with a large library. Ceres and the Grenoble mapper may be able to handle these complex cells, but Ceres lacks a suitable representation for performance optimization and Grenoble lacks a performance-directed covering method.

The group at Eindhoven University have composed a system slightly different than the other five, in that they have geared their algorithms to work with complex cells with only a size restriction. They use a graphical representation which is representative of the final circuit, but does not appear useful for a global view of timing concerns.

Berkelaar and Jess have seemingly disposed of a cost function and put this information in the order of choice for mapping. The algorithm seeks to map the largest gate possible, wherever it can. It appears that the system concentrates too much on one level of logic at a time. This may lead to increasing the logic level of the network unnecessarily. This would undoubtedly lead to poor performance of the circuit. In any case, it is difficult to suggest a performance strategy for this approach.

Although there are limitations in this approach, there is one significant contribution. Cells are constructed directly from sections of the network. This seems a natural solution to identifying large complex cells for a compiler.

2.7.4 Suggestion for a New Approach

To embody a large family of complex cells, it appears clear that technology mapping techniques must move away from *library matching* techniques and move to *construction* techniques, in order to have reasonable processing time. Approaches like that of Ceres have undoubtedly improved upon the prior matching techniques, but it is still not appropriate to handle complex cell libraries enumerated in the thousands. In the first case, a large amount of memory is required to manipulate the library. Secondly, the large library leads to a large number of attempted matches, which is manifested in long run times. Surely, there must be a way to approach the ideal solution without *trying every cell* or even almost every cell.

An approach such as that introduced by Berkelaar and Jess must be investigated. Cells can be constructed from a network under the guidance of constraints. However, one does not want to risk the possibility of low performance circuits. The availability of matched cells at each node combined with dynamic covering seems a natural choice for high performance circuits. There is sufficient basis for an approach which can embody a constructive method of matching, while still regarding dynamic programming in a covering stage.

Chapter 3: The Rossini Approach

3.1 Outline of Mapping System

The pioneering work of Keutzer on Dagon [11] and Rudell on MIS [3] has provided a three-step methodology for Technology Mapping, namely partitioning of the DAG into trees, matching to all possible implementations, followed by covering of the network. This methodology has been adopted for the Rossini approach, with the introduction of new adaptations to suit the desired goals. The key ingredient to this approach is a *constructive* matcher, which is capable of finding matched patterns which correspond to a cell family used by a cell compiler. Simple parameters direct the type of matches which are found.

With delay optimization in mind, new features are proposed for the partitioning and covering steps of the system. Included in the section on partitioning is a decomposition method which directs the form of the final circuit, which is oriented for high-performance. The covering step includes a natural extension to dynamic covering which is suitable for performance-directed phase assignment. A tree-boundary heuristic for improving the final design is also presented.

The approach is shown in a flow chart in Figure 3.1. A Boolean network is partitioned into trees which are ordered from primary inputs to primary outputs. Each of these are then mapped as follows. A tree is binary decomposed with arrival time information, matched to a complex family, then covered dynamically for the best implementation. This is repeated until all the trees have been mapped. Lastly, a tree-boundary heuristic is applied.

Figure 3.1 Rossini Overview

3.2 Tree Partitioning

Beginning with a Boolean Network which has been performance-optimized through minimization techniques, the logic is parsed into a forest of trees, each comprised of primitive AND, OR, and NOT logic functions. A new tree is introduced for each primary output as well as intermediate nodes with multiple fan-out. Then, NOT functions are *pushed* to the leaf level by repeated application of DeMorgan's Theorem. Having the inversions only at the leaves allows for easily matching large complex cells. Redundant operators (i.e. the same operators which are directly joined by an edge) are then removed. The resulting trees have depth equal to the logical depth of their function.

Each tree of primitive functions is then decomposed into a binary tree. The primitive tree itself could be used for matching, like the Grenoble approach, but this would inhibit the performance-driven covering step, because the form of the network is not set. This means the best solution at each node could not be identified. In conjunction with the

fact that a binary tree allows matches of a few as well as many inputs to be easily identified, this makes the choice of a binary tree desirable. There are many possibilities of binary trees. The selection of representation at this point is critical to the performance of the circuit, since it will have a direct impact on the types of matches found. While balanced and unbalanced decomposition techniques are not new [22][23][24], new features are presented and a novel technique for effective implementation is proposed. The binary decomposition procedure produces trees which are balanced by the level of the logic tree and unbalanced by the discrepancy in arrival times of the input signals at the leaves.

3.2.1 Balanced Binary Decomposition

Consider a case where there is no difference in arrival times at the leaf inputs of a tree. In general, it is desirable to choose a representation which allows for depth reduced covering. A very deep tree would require many logic gates to cover it from the leaves to the root. Thus, the signals would have to travel through many gates, increasing the delay. A low level tree allows the critical path(s) to be covered by as few logic gates as possible. An example of different decompositions is shown in Figure 3.2.

Figure 3.2 Binary decompositions of a primitive tree.

At nodes of the primitive tree which have more than two branches, one wishes to decompose the node into new nodes which have only two branches. In an attempt to produce the binary tree with the fewest levels possible, it is necessary to keep deep branches close to the root.

Consider the example node in Figure 3.3. The branches of this node are marked by their respective depths. Through a careful inspection of the optimal depth binary decomposition, one will notice that for the root node, the sum of the depths of the branches on one side of the tree is equal to the sum of the depths of the branches on the other side. In fact, this is true of all the nodes in the tree.

Figure 3.3 Forming a Balanced Decomposition

Thus, a general method for balanced decomposition can be proposed. The decomposition routine visits one node of the primitive tree. The depths of each branch of this node are found. Then, the branches are split into two groups, such that the difference between the sums of the branch depths in each group is as close as possible to zero. This step is then repeated recursively until each group contains only one branch. This will produce the desired solution as in the example of Figure 3.3. Subsequently, this method can be applied to all nodes with more than two branches.

Physically splitting the branches into two groups can be implemented in a variety of ways. Since in most cases, the number of branches is usually small, a straight forward method that executes in $O(2^n)$ can be used. This is simply exhausting all combinations of the branches. This can be sped-up by trying only half of these combinations, since for every combination, the elements not in this combination form a group, that is naturally in the

second group. Thus, the execution time is really $O(2^{n-1})$.

If the number of branches is very large, other speed-ups can be used. For example, if there are a large number of branches with the same depth, then not all combinations need to be tried.

3.2.2 Unbalanced Binary Decomposition

In the more general case of input signals of a tree arriving at different times, an unbalanced decomposition of the tree may be desired. Signals arriving late should be placed closer to the root than those arriving early. However some regard must still be given to depth of the branches as in the balanced decomposition.

In [23], Keutzer and Vancura introduced the use of input weighting, according to arrival times, for the purpose of tree-height reduction. A modification of this idea for binary decomposition can be proposed as an extension to the methodology described in sub-section 3.2.1. This is *unbalanced* binary decomposition, which takes the discrepancy in arrival times into account. Given an arbitrary tree, beginning at the root node, one can proceed as in the balanced case to find the depths of the branches of the node. For each branch, the latest arrival time for the deepest part of the branch is found. Thus, each branch can be weighted according to this arrival time. The smallest of these times are subtracted from each of the arrival times so that the discrepancy between arrival times is the weighting criteria. This difference in time is then multiplied by a technology dependent scaling factor k and added to the depth of the respective branch. The value of k attempts to relate a technology independent parameter (i.e. level) to a technology dependent parameter (i.e. delay through a cell). It is somewhat arbitrary, since a number of levels of logic may be covered by a range of cells and give very different delay values. It is therefore difficult to estimate a value of k which will give the most effective results. It can be estimated through a look at the delay model parameters and an empirical method to statistically find the best value of k can be used.

3.3 Constructive Matching

For constructive matching, the target *library* of cells is characterized by four simple parameters. These are the maximum number of n-type transistors in series (s), the maximum number of p-type transistors in series (p), the maximum number of levels of logic realized by a single gate (l), and the maximum number of gate inputs (n). The values (s) and (p) are influenced by technological factors. A limit of 4 is a generally accepted rule of thumb for both parameters s and p for low performance CMOS technologies, whereas it can be a much higher value in sub-micron technologies. The logic level (l) is defined as the depth of the logic tree (without redundant operators) minus one.¹ The trees of Figure 3.2 all have logic level 4. The value (l) is not restricted by the technology, but may be restricted by the abilities of the cell compiler. The number of inputs (n) per cell may be restricted by layout concerns. An exceedingly large number of inputs to a cell may introduce congestion while routing the layout. Note that a restriction on (l) and (n) is implicitly placed by defining (s) and (p). For example, for $(s,p) = (4,4)$ the maximum (l) is 6, and the maximum (n) is 16. A discussion of the relationships between cell parameters is found in Appendix D. The CMOS logic cell of Figure 2.10 has $(s,p,l,n) = (3,3,4,8)$.

The goal of matching is to find *all* the possible implementations at all the nodes in a tree, so that during covering the *best* of the matches may be chosen. The constructive matching routine achieves this goal by *building* matches at a given node in the tree. The procedure begins with a binary tree and iteratively creates trial subtrees at the root node. These subtrees may or may not form a matched pattern. New subtrees are created by building on old subtrees. Beginning with a subtree containing only the root node and two branches, new trial subtrees are constructed by adding branches to this subtree. To reduce the search space, new trial subtrees are only constructed from previously matched pat-

1. The leaves of the tree are usually not included in the calculation of the logic level; therefore, the depth of a tree (without redundant operators) is one larger than the logic level of the function it represents.

terns. Given a matched pattern, with b branches on the next level, there are $(2^b - 1)$ combinations of the branches together with the matched pattern which can form a new trial subtree. There is no repetition of trial subtrees, since branches are only added from the next lower level. An example is shown in Figure 3.4.

Figure 3.4 Constructing subtrees for constructive matching.

A matched pattern is created if a subtree does not exceed the (s, p, l, n) constraint. A dual matched pattern is created if the dual of the subtree does not exceed the (s, p, l, n) constraint. The dual of a tree is simply the tree with its operators reversed ($* \Rightarrow +$, $+ \Rightarrow *$). These are the two patterns which can form a possible implementation of the subtree in an AND-OR-INVERT structure or an OR-AND-INVERT structure. An example of a matched tree is shown in Figure 3.5.

Figure 3.5 A sample tree matched at its root.

This tree has been matched with no restriction on the size of the cell family. This collection represents all the possible implementations for the root node of this logic tree. Notice that the AOI matches will give the correct phase of the signal at the output, while the OAI matches will give the complemented phase. Ensuring that the correct number of inverters exists at cell boundaries is the function of the covering phase. The constructive matching algorithm is presented in Figure 3.6.

The function `make_match(tree, leaves)` creates an object (a match) which has a tree and a list of unsearched leaves associated with it. Notice that if a subtree and the dual of a subtree *both* make a match, then the dual match is not assigned unsearched leaves. This avoids redundant construction of subtrees, since both a subtree and the dual are always tested for matches. Also note that for families with $s = p$, if a subtree forms a match then its dual must also form a match. The Common Lisp code for this algorithm is shown in Appendix C.

```

find_matches (tree s p l n)
  series = find_number_of_series_transistors(tree)
  parallel = find_number_of_parallel_transistors(tree)
  level = find_logic_level(tree)
  inputs = find_number_of_inputs(tree)
  if (series,parallel,level,inputs are all < or = s,p,l,n)
    match_list = make_match(tree,leaves)
    if (parallel,series,level,inputs are all < or = s,p,l,n)
      match_list = match_list + make_match(dual(tree),nil)
    else if (parallel,series,level,inputs are all < or = s,p,l,n)
      match_list = match_list + make_match(dual(tree),leaves)
    end if
  for_each match of match_list
    new_matches = expand_match(match s p l n)
    match_list = match_list + new_matches
  end for_each
return match_list

expand_match (match s p l n)
  leaves_list = all_combinations(unsearched_leaves(match))
  for_each leaf in leaves_list
    trial_tree = tree(match) + leaf
    unsearched_leaves = leaves(leaf)
    series = find_number_of_series_transistors(tree)
    parallel = find_number_of_parallel_transistors(tree)
    level = find_logic_level(tree)
    inputs = find_number_of_inputs(tree)
    if (series,parallel,level,inputs are all < or = s,p,l,n)
      new_matches = new_matches + make_match(tree,unsearched_leaves)
      if (parallel,series,level,inputs are all < or = s,p,l,n)
        new_matches = new_matches + make_match(dual(tree),nil)
      end if
    else if (parallel,series,level,inputs are all < or = s,p,l,n)
      new_matches = new_matches + make_match(dual(tree),unsearched_leaves)
    end if
  end for_each
return new_matches

```

Figure 3.6 Constructive matching algorithm.

A search tree for the algorithm, applied to the tree of Figure 3.5, is shown in Figure 3.7. The trial sub-trees are shown on edges and the matches are shown at the vertices. The first matches are two-input NAND and NOR gates. This pattern has two unsearched branches, from which three new trial sub-trees are formed. The algorithm matches six new patterns, each of which have two unsearched branches. Continuing, two new trial sub-trees are constructed, each of which form two new matched patterns.

3.4 Covering

As has been discovered, the natural choice for tree covering is dynamic programming. This has been adopted in the Rossini approach.

3.4.1 Cost Models

For covering, cell-level models for each match must exist, so that a low-cost realization can be chosen. Rossini accepts any cell-level cost model. The topology (tree topology; s, p, l, n) and place in the network (fanout loading) of a matched pattern are sent to a module which computes the cost and returns this data to the covering routine. Typically, the cost is based on a weighted average of the cell area and delay, but it is not restricted. It is possible, for example, to include a testability cost for each pattern. The models which have been used in experiments are described in Chapter 5.

3.4.2 Network Delay Modelling

Delay optimization based on a linear cell-level model cannot disregard the effects of primary input/output loading on the overall circuit delay. A high primary input fanout must be taken into consideration, as the driver of this input takes on a considerable loading, increasing the delay in each of these fanout paths. Ignoring this effect is equivalent to assuming the presence of an infinitely powerful driver which is not affected by increased loading. Also, if a primary output is reused within the combinatorial network, the primary output loading must be considered, since the path within the network becomes lengthened. In order to take account of these effects during optimization, an input buffer which has the same delay characteristics as an inverter has been included at each primary input. Also a load of one fanout at each primary output is included to account for the reuse of primary outputs within the network. This model is illustrated in Figure 3.8.

Figure 3.8 Combinatorial Network with Primary Inputs and Outputs

3.4.3 Phase Assignment and Tree Covering

The purpose of phase assignment is to choose the phase of a signal which may allow the best implementation for which multiple possible realizations may exist. The objective is to select the fastest overall realization by minimizing the number of inverters on critical paths. This is a rule of thumb, and may not be true at nodes where the fanout is greater than one (tree-boundaries), since the fanout coefficient is typically smaller for smaller gates (e.g. an inverter). However, minimizing the number of inverters within a tree is naturally desirable.

Recall that the dynamic programming procedure begins at the leaves of a tree and traverses toward the root node, choosing the best implementation at each node. The best match at a node is determined by the current cost metric. Delay is calculated as the greatest arrival time at the inputs (leaves) of the gate plus the delay associated with the gate itself. Area is calculated as the total area at the leaves of the gate plus the area of the gate itself.

As a natural extension to dynamic covering, it is possible to treat the phase assignment problem concurrently. This is a more elegant solution than introducing inverter pairs at every node. At a particular node, as a match is chosen as the best implementation, two

costs are given to the output, one for each phase. As the routine proceeds towards the root, the appropriate cost for a match at each leaf is taken. If a leaf contains the incorrect phase for correct logic, the routine uses the second cost value for the computation of the cost of the current match. This method leaves the possibility of ties for the best implementation at a node. The *best_match_list* keeps all the best matches for each phase of the signal. The algorithm may discard some choices as it makes decisions on its way toward the root. When a match has more than one possibility for an implementation at a leaf, the algorithm tries all possibilities of the list of leaves to find the best implementation. An example of the procedure is shown in Figure 3.9.

Figure 3.9 Phase-assignment covering example.

For simplicity, the cell family is restricted to $(s,p,l,n) = (2,2,1,2)$, delay is chosen as the optimization metric, and the following gate delay characteristics are assigned arbitrarily, all with zero fanout coefficient:

- NOR gate $\Rightarrow$ 2.5 time units ; NAND gate $\Rightarrow$ 2 time units ; INVERTER $\Rightarrow$ 1 time unit.

The covering begins at nodes 1 and 2 (the lowest level). The `best_match_list` has a single entry of an inverter. For node 1, the `phase_1` delay cost is zero (inverter absent) and the `phase_2` delay cost is one (inverter present). For node 2, this is reversed, the `phase_1` delay cost is one and the `phase_2` delay cost is zero. At this point, no choice is made for the presence or absence of the inverter. The choice is deferred to higher levels, when these nodes are inputs to a cell. At node 3, there are two matches, a NAND gate and a NOR gate. The inputs to the gates are given the required coverings to ensure the correct logic. The NAND implementation has a `phase_1` delay cost of 3 and a `phase_2` delay cost of 4 (with an inverter on its output), while the NOR implementation has a `phase_1` delay cost of 4.5 and a `phase_2` delay cost of 3.5. These represent the best implementations for node 3. At node 4, no choice is made as at nodes 1 and 2. At node 5, there are again two matches. Each gate chooses the correct phase at its inputs. This gives mappings of M1, M2, M3, and M4, with the following delay costs:

	M1	M2	M3	M4
• phase 1	6	7.5	6	6.5
• phase 2	5	6.5	7	7.5

If f is the output of an intermediate tree (i.e. not needed as a primary output), then the best mapping of M1 (phase 2) is used. Thus the signal is inverted ($\bar{f}$), and trees dependent on this tree would use $\bar{f}$ as a leaf. However, if the uninverted signal f is required as a primary output, then the best mapping for phase 1 (M1 or M3) is used as the implementation. If f were not the output of this tree, then matches M1 and M3 would be stored as the best matches for this node, and the mapping would continue to higher nodes.

3.4.4 Tree-boundary phase assignment

It is possible to introduce phase assignment heuristics for tree-boundaries *during* technology mapping [19], however, the difficulties stem from inaccuracies in fanout assignment. As signals are assigned to different phases, *new* tree-boundaries are usually introduced. Thus, the delay estimations, used to map the tree which fans into the boundary, were *inaccurate* since the fanout values are changed. Therefore, an optimal covering of the fanin tree is no longer guaranteed.

Given only the tree boundaries introduced in the original partitioning scheme, phase-assignment during tree covering guarantees the best choice of matches for the fastest result within the tree. Following is a simple post-processing step which is likely to globally improve the area and timing of the mapped circuit. While this step is very small in approaching a general solution to the tree-boundary phase assignment problem, it does guarantee to equal or better the quality of the mapped circuit after tree-mapping.

Consider the tree-boundary node in Figure 3.10. The fanin tree has an inverter before its output. It would be possible to connect nodes b, c, and/or d to node a for the same signal; however, in doing so, the fanout count on node a has increased, thus, changing the delay estimations for the cell which has a as an output. Also, the fanout dependence is likely to be smaller for an inverter than any other cell, and the cell with output a is a cell other than an inverter. This can result in drastic changes to the delay of the circuit, likely making it larger. Noting this, the fanin tree can be left alone, since it is unlikely for it to have an inverter at the output, since the fastest possible covering has been chosen for either phase of the signal (sub-section 3.4.3).

Figure 3.10 Example Tree-boundary before optimization.

However, it is possible to reduce the number of inverters in Figure 3.10. Nodes *c* and *d* can be connected to node *b* and the inverters feeding *c* and *d* can be eliminated. Subsequently, fanout on node *x* is reduced and the fanout on node *b* is increased. This manipulation will decrease the area (i.e. the savings of the eliminated inverters), and is likely to improve the timing of the circuit. As stated earlier, the cell with output *x* is unlikely to be an inverter. The fanout dependence is likely to be larger than that of inverter. The savings of the reduced fanout on node *x* will always equal or outweigh the cost of the increased fanout on node *b*. The new circuit is shown in Figure 3.11.

Figure 3.11 Circuit after Tree-boundary heuristic.

The only drawback to this manipulation is that after processing, the propagation delay for the entire circuit must be recalculated. Another possibility is to initially record the delay values symbolically, then dynamically adjust the values as these type of manipulations are made.

3.5 Summary

The Rossini approach is based on a new constructive matcher which is capable of finding all the possible implementations at a given node. The cell matches are constrained by four simple parameters which reflect the capability of the given technology and limitations of the cell compiler. This matching module is embedded in a performance-driven technology mapping system consisting of a partitioning and covering algorithms. The partitioning algorithm includes techniques which build upon existing ideas for maintaining the shallowness of the logic trees during decomposition, with regard given to discrepancy in arrival times. The tree-covering algorithm provides a new natural extension to dynamic covering which elegantly treats the phase assignment problem. Finally, a simple tree-boundary heuristic is introduced which is likely to globally improve area and delay.

Chapter 4: Logic Minimization

A multi-level logic network is a multiple set of logic equations directing a fixed number of primary input signals to primary output signals. Before technology mapping, it is generally necessary to perform logic optimization operations on the Boolean network in an attempt to tailor the network to the user's needs. For area optimization, the size of the network is reduced with regards to a technology independent metric such as the number of literals. For delay optimization, the metric may be the number of levels of logic. These metrics are estimates and can sometimes have little correlation with the final circuit. Recently, in the synthesis community, there is an increasing awareness of the need for more technology dependent metrics which may include the routing factor and cell timing characteristics. The problem of finding a suitable cost objective is an enormous challenge.

The following is an introduction to logic minimization techniques. It merely *scratches the surface* of a very fruitful field of research. The discussion is included here, because minimization is generally necessary before technology mapping to obtain optimal results.

4.1 Basic Minimization Operations

There are five basic logic minimization operations [9]. These can be categorized in two groups: algebraic operations and Boolean operations. Algebraic operations are manipulations involving the variables of the equations, and can generally be applied to other types of algebra. Boolean operations are manipulations specific to Boolean algebra, and replace old expressions with new logically-equivalent expressions. During the synthesis process, a sequence of these operations is applied to minimize the Boolean network. The types of operations are:

1. *Extraction* - an algebraic operation, where a subexpression common to two or more expressions is extracted and used to create a new expression. Subexpressions are found using kernels and cubes [7][19].
2. *Elimination* - an algebraic operation, where an intermediate expression is *pushed back* into an expression. This is the inverse to extraction.
3. *Simplification* - a Boolean operation, where a logic function is replaced by a logically equivalent *simpler* expression. This is usually done with the use of *don't care* sets [8][25].
4. *Resubstitution* - either an algebraic or Boolean operation where an intermediate expression is checked (i.e. divided into) to see if it is a factor of an existing expression. If so, the intermediate expression is resubstituted into the existing expression.
5. *Decomposition* - an algebraic operation, where one large expression is broken down into a number of smaller expressions.

4.1.1 Extraction

The problem of identifying common subexpressions to two or more functions can be difficult in multi-level logic, since the size of the set of divisors for each function can be very large. The basic way of finding common expressions is achieved by forming partial factorizations of each function, then identifying algebraic commonality.

A logical expression may be represented in a sum-of-products (SOP) form: $f = f_1 + f_2 + \dots + f_n$. Each product term is called a *cube*. An expression is said to be *cube-free* if no cubes may be factored out of it. The set of *kernels* of a function f is the set of all quotients f/c such that the f/c is cube-free and c is a cube, but not necessarily one of $f_1, f_2, \dots, f_n$. Note that a kernel must contain at least two product terms; therefore, a single cube is not cube-free.

The identification of common divisors can be made through the use of kernels. Given a function, the cube-free divisors are made up of common cubes found in the kernel sets. Subsequently if all cube-free divisors are found, then all that remain are single cube divisors. These may be found straightforwardly. This is best explained through an example. Consider these three Boolean equations in SOP form:

$$\begin{aligned}f &= AE + BE + CDE \\g &= AD + AE + BD + BE + BF \\h &= BC\end{aligned}$$

The respective kernel sets for each equation are:

$$\begin{aligned}K(f) &= \{ A + B + CD \} \\K(g) &= \{ A + B, D + E, D + E + F \} \\K(h) &= \emptyset\end{aligned}$$

Note that the expression $A + B$ is contained in the kernel set of f and g ; therefore, a new variable may be introduced, X , and the network becomes:

$$\begin{aligned}X &= A + B \\f &= EX + CDE \\g &= DX + EX + BF \\h &= BC\end{aligned}$$

Common cube extraction is now a simple task of identifying cubes common to more than one function. An obvious choice is the cube EX .

$$\begin{aligned}X &= A + B \\Y &= EX \\f &= Y + CDE \\g &= DX + Y + BF \\h &= BC\end{aligned}$$

Again, operations are performed according to some cost savings, such as literal savings.

4.1.2 Elimination

Elimination is simply the inverse operation to extraction. *Pushing back* X and Y in the previous example is the elimination of these nodes. Usually, there is some restriction on the elimination process driven by a cost consideration. For example, all intermediate nodes with a fanout of 1 may be eliminated. Another example is the elimination of all intermediate nodes. This is known as *collapsing* the network to two levels. The resulting Boolean network has only primary outputs on the left-hand side of the equations. In many cases, fully collapsing a network is not practical because the network simply becomes too large.

4.1.3 Simplification

In simplification, the idea is to replace an expression with a logically equivalent *simpler* expression. For multi-level logic, room for simplification is usually less than obvious. For example, consider the following equations:

$$\begin{aligned} F &= A \bar{B} + B C \\ B &= \bar{C} + D \end{aligned}$$

This can be simplified to:

$$\begin{aligned} F &= A C + B C = (A + B) C \\ B &= \bar{C} + D \end{aligned}$$

with the savings of one literal.

The simplification can be computed with the use of the *don't care* set [26], which is associated with the network. This is a set of states which cannot possibly occur. Don't care sets originate either internally or externally to the network. There are two types of internal don't care sets. The *satisfiability* don't care set is the set which is necessary for the satisfaction of the network equations. Given an equation $f = g$, the satisfiability don't care set is $\{f \oplus g\}$ or $\{\bar{f}g + f\bar{g}\}$ (i.e. an exclusive-or function). This simply states that both sides of the equations must have the same logic value; therefore, any expression to the contrary must be a state which cannot occur. The second type of internal don't cares is the *observability* don't care set, which is comprised of states which are not observable at primary outputs. Consider the following example:

$$\begin{aligned} F &= X Y + T \bar{Z} + \bar{T} \bar{Y} \\ T &= \bar{X} Y + X \bar{Z} \end{aligned}$$

If F is the only primary output of the Boolean network and T is an intermediate variable, then we can form a set of input conditions, such that the value of T is not observable at the output F . Since T is not seen at the output, we can use these conditions to modify T . The observability don't care set for T is:

$$\begin{aligned} \text{ODC}(T) &= X Y + (\bar{Z} + \bar{Y}) + !(\bar{Z} + \bar{Y}) \\ &= X Y + \bar{Z} \bar{Y} + Z Y \end{aligned}$$

Note that the observability don't care set is only associated with a particular node (T), in

contrast with the satisfiability don't care set, which is global. In practice, the observability don't care set is difficult to compute, since there can be many implicit dependencies throughout the network.

Don't cares may also be introduced to the network externally. These may originate from such sources as unoccurable states from sequential logic structures.

The simplification of the previous example can be obtained by examination of the satisfiability don't care set of the equation for B.

$$DC(B) = \bar{B} \bar{C} + \bar{B} D + B C \bar{D}$$

It is possible to use any part of this don't care set to simplify the equation for F. We can take the first cube of the set and expand it in minterms for A, B, and C.

$$DC(B) = A \bar{B} \bar{C} + \bar{A} \bar{B} \bar{C} + \dots$$

We can now write the first equation as:

$$\begin{aligned} F &= A \bar{B} + B C \\ &= A \bar{B} C + A \bar{B} \bar{C} + A B C + \bar{A} B C \end{aligned}$$

Since $A \bar{B} \bar{C}$ is part of the don't care set, we can remove it from the equation for F. With appropriate groupings, the equation then becomes:

$$F = A C + B C = (A + B) C$$

In practice, the size of the don't care set can become too large to manage. Filters are necessary to remove expressions which are doubtful to give future simplifications.

4.1.4 Resubstitution

For resubstitution, one needs to check existing expressions to see if they can become a divisor to other expressions. For example:

$$\begin{aligned} X &= AC + AD + BC + BD + E \\ Y &= A + B \end{aligned}$$

The function Y can be resubstituted in the equation for X, since it is a divisor. Thus, the network becomes:

$$X = Y (C + D) + E$$

$$Y = A + B$$

More specifically, this is an algebraic resubstitution, and can be carried out using some of the techniques used in extraction. Algebraic substitution, the process of finding substitutes, is sometimes referred to as weak division.

Boolean resubstitution exploits the Boolean properties of the equations. Consider the following example:

$$\begin{aligned} F &= A B + \bar{A} C + B C \\ X &= \bar{A} + B \end{aligned}$$

This network can be simplified using algebraic resubstitution, i.e. the equation for F becomes:

$$F = A B + X C$$

However, Boolean resubstitution allows the equation to be written as:

$$F = X (A + C)$$

The process of finding Boolean substitutes (also known as strong division), is much more expensive than algebraic substitution, but is potentially much more powerful. This is usually the case when comparing Boolean operations to algebraic operations. Strong division can be done using similar techniques used in don't care simplification. The satisfiability don't care set of the equation for X is:

$$\begin{aligned} DC(X) &= \bar{X} \bar{A} + \bar{X} B + X A \bar{B} \\ &= \bar{X} \bar{A} B + \bar{X} \bar{A} \bar{B} + \bar{X} A B + X A \bar{B} \\ &= (\bar{X} \bar{A} B + \bar{X} \bar{A} \bar{B} + \bar{X} A B + X A \bar{B})(C + \bar{C}) \end{aligned}$$

If we expand the equation for F to include the variables in the don't care set, the equation becomes:

$$\begin{aligned} F &= X A B + \bar{X} A B + X \bar{A} B C + \bar{X} \bar{A} B C + X \bar{A} \bar{B} C \\ &\quad + \bar{X} \bar{A} \bar{B} C + X A B C + \bar{X} A B C \end{aligned}$$

Removing don't care cubes, the result is:

$$F = X A B + X \bar{A} B C + X \bar{A} \bar{B} C + X A B C$$

Adding two terms from the don't care set, we can simplify:

$$\begin{aligned} F &= X A B + X A \bar{B} + X \bar{A} B C + X A \bar{B} C + X \bar{A} \bar{B} C + X A B C \\ &= X A + X C = X (A + C) \end{aligned}$$

It is easy to see why Boolean resubstitution is computationally intensive.

4.1.5 Decomposition

After extraction, it is often necessary to break down an equation into smaller pieces to aid in technology mapping. Decomposition is basically a *factoring* operation. Divisors of an expression are removed and placed as new equations in the network. The approach to finding commonality has been shown in sub-section 4.1.1 on extraction; therefore, this method can simply be applied recursively to one expression for decomposition.

As an example, consider the following equation:

$$X = A D F + A E F + B D F + B E F + C D F + C E F + G$$

There are a lot of common factors; therefore, this can be written in factored form:

$$X = (A + B + C) (D + E) F + G$$

It can also be written in decomposed form:

$$X = Y Z F + G$$

$$Y = A + B + C$$

$$Z = D + E$$

Note that in general, the factored form is not unique. Typically, one is interested in the smallest factored form, which may be the one with the fewest literals. A cost strategy can determine that “large” factors are removed earlier than small factors.

4.2 Minimization for Delay

While logic minimization is a mature science, the change in emphasis from low area goals to low delay goals have opened new doors for improvement. It is apparent that all of the basic minimization operations were motivated through concerns for least area; therefore, the effect on delay remains in question. Manipulations such as simplification and elimination are likely to increase the performance of the combinational logic, while extraction and resubstitution introduces more levels to the logic, which may or may not deter from the timing of the logic.

A strategy for the order of application of operations is presented in [9]. It is outlined as follows:

1. Apply a fast simplification procedure to the incoming data.
2. Apply algebraic resubstitution for maximum sharing of intermediate functions.
3. Apply elimination. Push-back intermediate nodes of low value.
4. Apply Boolean simplification.
5. Extract common sub-expressions.
6. Apply elimination again.
7. Apply simplification again.
8. Decompose into functions small enough for the target technology.

This is a rough strategy and can be expanded or shrunk in sections. The next chapter shows a MIS II script in Figure 5.3. This script roughly follows the presented strategy, although some steps are excluded and step 5 is broken into sections where larger subexpressions are removed before smaller ones.

While this strategy is likely to be very good for area reduction, again, it is difficult to evaluate its impact on performance. In any case, we can see that the strategy is likely to produce many logic trees with low literal count and low depth. Since we are concerned about the successful use of large complex cells, it is likely that the use of this minimization strategy could lead to trees which are too small to be exploited by large complex gates. Other types of minimization can surely be used for delay minimization.

Recently, some work has been done directly addressing delay concerns [27][28]. The “speed_up” algorithm by Singh et al. [27] works using critical path analysis. Delay optimization is achieved by partially collapsing logic along critical paths. It also performs local decomposition for timing by placing late arriving signals closer to the output. Although results are shown to be favorable, it suffers from two limitations. The optimization routine is based on a network of 2-input NAND gates and inverters. Therefore, cost values are dependent on this model. It is difficult to predict the effect on the final circuit, especially since the cells which will be used are not known. The second limitation is on the types of networks under optimization. For circuits which do not have a well-defined critical path, the optimization may have ill effects.

The work done by Touati et al. [28] is closer to a general solution for performance optimization. Their delay model is not as limited as in the previous case. The number of logic levels is used as a rough indication of delay, and the introduction of *clusters* on a path is used as a second indication. The work is similar to that of Singh [27], in that delay minimization is achieved through partial collapsing of expressions. However application is not limited to critical paths, but more so, to areas of criticality. The clustering algorithm controls the grouping of clusters according to connectivity criteria. Then, appropriate clusters are collapsed in an attempt to improve delay.

Chapter 5: Implementation and Experimental Results

This chapter outlines the implementation of the proposed algorithms for technology mapping in a working prototype. The mapper is presented in an experimental context for evaluation of individual algorithms and comparison with an existing technology mapper. Finally, the prototype has been used to investigate the impact of cell family size on the quality of mapped networks.

5.1 Prototype Implementation and Experimental Procedure

The approach for cell family technology mapping described in Chapter 3 has been implemented in a prototype called Rossini. Rossini is written in Common Lisp for fast prototyping, an interactive user interface, and the compatibility with existing software (the Picasso cell compiler [21]). The interactive interface is useful for quickly assessing the impact of different routines, but can also be used in batch mode, where a script controls the synthesis process. The mapper is comprised of four modules:

1. Parser - This is the input module of the technology mapper. A technology independent Boolean network is read and organized in the database for connectivity and fanout information. Partitioning into a forest of trees is implemented.
2. Matcher - The constructive matching routine is implemented. A tree can be matched for all the possible patterns at each node.
3. Cover Module - The dynamic programming covering routine is implemented. A function to compute the delay based on a pattern's topology is included. A tree can be covered for area or delay goals.
4. Top-level Module - This is the main level module which controls the mapping process. It controls the order in which trees are mapped and the order of applying decomposition, matching, and covering. Also included are the tree-boundary improvement rou-

tine, functions which give statistical information about the present network, and output functions.

Also necessary for the prototype is a set of utility functions, organized in a general library. These include functions to find tree attributes. For example, given an arbitrary tree, find the level, inputs, cut-set, path-set, or deepest leaves. There are tree manipulation routines (e.g. push inversions to leaf level, find the dual of a tree, decompose unbalanced, change tree format, etc.). The trees are represented as embedded lists, with the operator as the first element and the branches as the rest of the elements. The functions usually perform in a recursive manner, working on a single list at a time.

As was mentioned previously, it is often necessary to perform logic minimization before technology mapping. MIS II is used to perform the logic minimization. Then technology mapping is tested independently in MIS II and Rossini. The experimental set-up is shown in Figure 5.1.

Figure 5.1 Experimental Set-up

5.2 Cell-level Area and Delay Models

For cost metrics, we have chosen cell area and propagation delay. For area, the cost of a cell is simply calculated as the number of inputs to the cell plus one. This is reflective of the number of grids the layout of the cell would use.

The conventional delay model typically used for technology mapping expresses the delay through a logic gate as the sum of an intrinsic portion plus the fanout loading multiplied by a fanout coefficient. This is a simplification of a very complicated delay dependence on a number of related and/or unrelated factors, which, for CMOS, may include input transition time, output resistive and capacitive loading, the size of the transistors involved, cell topology, and layout parasitics. Characterizing the performance of such a cell is an art in itself, as there are a number of factors in the layout, which determine its performance. A very simple topology model is proposed, which may be easily updated to a more complicated model as the ability to characterize improves [20]. The delay through a cell is estimated considering only the values of s and p . For example, see the cell in Figure 5.2. This cell is replaced by a series-connected construct of transistors, determined by the values of s and p . The value of s determines the maximum possible fall transition delay, while the value of p determines the maximum possible rise transition delay.

Figure 5.2 Delay model of a CMOS cell.

The test technology is characterized by empirically obtaining intrinsic delay values and fanout coefficients for series-connected constructs with the possible values of s and p

transistors in series. Then, given a cell with the characteristics (s, p) and the fanout, the fall delay time and the rise delay time can be found in a look-up table. The worst (i.e. maximum) of these two values is used as the delay through a cell.

Parameters for the gate delay model were obtained using SPICE¹ (Simulation Program with Integrated Circuit Emphasis). The level 3 transistor model values for the nMOS and pMOS devices were based on a CMOS 1.2 μ process. Working with a simple inverter, the sizes of the devices were adjusted until the input transit time was roughly the same as the output transit time. The input capacitance of this gate served as one fanout loading. Then, worst case intrinsic delays and fanout factors were found by simple curve-fitting of results obtained for series-connected structures with 1,2,3 or 4 nMOS or pMOS transistors in series. These values were supplied to Rossini and incorporated in the genlib libraries for MIS Release #2.2.

5.3 Logic Minimization

The forthcoming sections which include experimental results are based on logic networks from the 1991 MCNC (Microelectronics Center of North Carolina) International Workshop on Logic Synthesis benchmark set. These multi-level logic benchmarks were prepared by various logic minimization scripts performed by MIS II before technology mapping. A total of six networks were mapped for each benchmark. Then, the best mapping from all the minimizations was reported.

The three basic networks were the unminimized network and those prepared by the two scripts shown in Figure 5.3. Also in each of these cases, the *speed_up* [27] command was used, giving three more networks. Unfortunately, the partial collapsing algorithm of Touati [28] was not available for experimentation. A full explanation of the commands used in Figure 5.3 can be found in [7].

1. The program used was HSPICE © Meta-Software, Inc.

sweep; eliminate -1	eliminate -1
simplify	qkx
eliminate -1	gcx
sweep; eliminate 5	
simplify	
resub -a	
qkx -abt 30	
resub -a; sweep	
gcx -bt 30	
resub -a; sweep	
qkx -abt 10	
resub -a; sweep	
gcx -bt 10	
resub -a; sweep	
qkx -a1	
resub -a; sweep	
gcx -b	
resub -a; sweep	
eliminate 0	
decomp -g *	
eliminate -1; sweep	

Boolean Script

Simple Script

Figure 5.3 MIS II Minimization Scripts

The Boolean script is the standard MIS minimization script used for area reduction. This is often useful for very large networks. The simple script is an attempt to preserve the structure of the network, so as not to impose a delay penalty. Only the common expressions are removed from the equations.

5.4 Experimentation and Results

5.4.1 Balanced versus Unbalanced Decomposition

The two decomposition techniques were tested on the benchmark set to see the impact of arrival time information on the resulting circuit. Given the delay parameters, the range of transit times for various levels of logic was determined. Bounding this range, we determined that the value of k should lie between 1 and 25. Statistically, we found a value of $k = 2.5$ to be a value where the arrival times and tree levels were well balanced.

Results using a test family of $(s,p,l,n) = (3,3,4,9)$ are shown in Table 5.1. Delay is improved by an average of 3.5% for an area expense of 1.7%. Notice that there is rarely improvement for the very small benchmarks, as the trees are small and binary decomposition does not play a large role.

5.4.2 Tree-boundary improvement

Results showing the effect of the tree-boundary improvement procedure on the unbalanced mappings are also shown in Table 5.1. Note that delay is always equal or better than the previous mapping. On average, area has improved by 10.6% and delay has improved 3.3%.

Network	1 - Balanced area delay(r)	2 - Unbalanced area delay(ns)	3 - Boundary Improvement area delay(ns)	Column 1 to 2 Δ area Δ delay	Column 2 to 3 Δ area Δ delay
apex6	2208 3.530	2196 3.530	1790 3.119	-0.5% 0%	-18.5% -11.6%
C1355	1424 5.113	1408 5.028	1148 5.019	-1.1% -1.7%	-18.5% 0%
C432	992 11.068	976 10.784	862 9.593	-1.6% -2.6%	-11.7% -11.0%
C499	1366 4.816	1376 4.816	1124 4.816	+0.7% 0%	-18.3% 0%
C880	1177 4.728	1173 4.482	1089 4.482	-0.3% -5.2%	-7.2% 0%
cm151a	87 1.400	87 1.400	71 1.400	0% 0%	-18.4% 0%
con1	45 0.904	45 0.904	43 0.904	0% 0%	-4.4% 0%
cordic	192 1.505	198 1.505	190 1.505	+3.1% 0%	-4.0% 0%
count	595 3.169	589 3.169	509 2.722	-1.0% 0%	-13.6% -14.1%
dalu	3306 11.468	3334 11.343	2852 10.367	+0.8% -1.1%	-14.5% -8.6%
duke2	1646 4.442	1664 3.971	1484 3.926	+1.1% -10.6%	-10.8% -1.1%
example2	1076 3.545	1068 3.303	952 3.199	-0.7% -6.8%	-10.9% -3.2%
f51m	35 2.641	409 2.543	343 2.420	+3.0% -3.7%	-16.1% -4.8%
i4	54 1.077	510 1.077	510 1.077	-5.9% 0%	0% 0%
i5	897 2.625	1121 2.326	1121 2.326	+25.0% -11.4%	0% 0%
inc	544 3.235	540 2.935	502 2.935	-0.7% -9.3%	-7.0% 0%
mux	119 1.586	119 1.586	109 1.586	0% 0%	-8.4% 0%
pcler8	376 2.501	404 2.501	362 2.395	+7.5% 0%	-10.4% -4.2%
squar5	233 2.421	235 2.243	211 2.237	+0.9% -7.4%	-10.2% -0.3%
x1	895 2.382	911 2.285	809 2.108	+1.8% -4.1%	-11.2% -7.8%
z4ml	169 1.744	175 1.594	159 1.567	+3.6% -8.6%	-9.1% -1.7%
Average				+1.7% -3.5%	-10.6% -3.3%

Table 5.1 Decomposition and Boundary techniques.

5.4.3 Rossini versus MIS II

Rossini was tested against MIS II on the same benchmark set. Comparative results are shown in Table 5.2. Each optimized network was tested in MIS as well as in Rossini; therefore, no bias in optimization is given to one technology mapper over another. Both of the delay optimization routines available in MIS (i.e. map -m 1 ; map -n 1) were tested, and the delay was measured, externally, using the delay model described in sub-section 3.4.2. The mappings with the smallest critical delay resulting from any optimized/non-optimized network are reported. The target library is the family with (s,p,l,n) = (3,3,4,9) since this is the largest complete library with $s = p$, that MIS II could handle on our system

without running into memory problems.¹

Network	MIS II			Rossini			Δ area	Δ delay
	area	delay(ns)	cpu(s)	area	delay(ns)	cpu(s)		
apex6	1934	3.516	62.5	1790	3.119	185.6	-7.5%	-11.3%
C1355	1142	5.367	136.9	1148	5.028	128.0	+0.5%	-6.3%
C432	1002	9.478	53.9	862	9.593	119.7	-14.0%	+1.2%
C499	1088	5.135	87.0	1124	4.816	121.3	+5.4%	-6.2%
C880	1187	5.037	57.9	1089	4.482	108.2	-8.3%	-11.0%
cm151a	69	1.400	21.8	71	1.400	75.4	+2.9%	0%
con1	45	0.983	42.3	43	0.904	52.1	-4.4%	-8.0%
cordic	190	1.539	41.5	190	1.505	54.2	0%	-2.2%
count	475	3.416	32.0	509	2.722	70.5	+7.2%	-20.3%
dalu	3468	12.596	139.8	2852	10.367	291.2	-17.3%	-18.4%
duke2	1338	4.153	60.9	1484	3.926	411.5	+10.9%	-5.5%
example2	1018	3.489	119.6	952	3.199	114.8	-6.5%	-8.3%
f51m	331	2.551	41.8	343	2.420	76.2	+3.6%	-5.1%
i4	622	1.156	53.0	510	1.077	93.3	-18.1%	-2.2%
i5	1442	2.403	70.5	1121	2.326	96.9	-22.3%	-3.2%
inc	406	3.095	27.2	502	2.935	59.6	+23.7%	-5.2%
mux	103	1.625	34.6	109	1.586	56.3	+5.8%	-2.4%
pcler8	298	2.579	26.2	362	2.395	76.2	+21.5%	-7.1%
squar5	219	2.515	25.6	211	2.237	63.4	-3.7%	-11.1%
x1	719	2.226	34.6	809	2.108	99.1	+12.5%	-5.3%
z4ml	141	1.615	22.7	159	1.567	53.7	+12.8%	-3.0%
							+0.2%	-6.8%

Table 5.2 Results for MIS II and Rossini using family (s,p,l,n) = (3,3,4,9).

For this target family, the Rossini approach has improved over the algorithms of MIS II by an average 6.8% for an area expense of 0.2%, although individual results varied widely. The reader should note that this claim is made only with regard to the delay models which have been described from sub-sections 5.2.

5.4.4 The impact of library size on performance.

As a test for the impact of the use of different sized families, Rossini and MIS II were tested for various sized libraries on the same benchmark set. The results for families (s,p,l,n) = (2,2,2,4), (3,3,4,9), and (4,4,6,16) are compiled and shown in Table 5.3. These values represent the best implementations from either mapper, with critical delay as the first priority and area as the second. The mappings denoted by 'M' were those produced by MIS II. The rest of the mappings were produced by Rossini. The run-times are shown to

1. Experiments were run on a SUN SPARCserver 4/390 with 32 MB RAM and 110 MB Virtual Memory.

give an indication of the cpu penalty incurred by the larger families.

Network	Family 2 (2,2,2,4) area delay(ns) cpu(s)	Family 3 (3,3,4,9) area delay(ns) cpu(s)	Family 4 (4,4,6,16) area delay(ns) cpu(s)	Family 2 to 3 Δ delay	Family 2 to 4 Δ delay	# of Trees	Deepest Tree	Weighted level
apex6	1974 3.476 170.8	1790 3.119 185.6	1768 3.119 684.9	-10.3%	-10.3%	177	9	3.2
C1355	1143 5.053 108.6	1148 5.028 128.0	1164 4.988 220.1	-0.5%	-1.3%	152	4	1.7
C432	948 9.227 102.5	1002 9.478 53.9 M	866 8.884 399.9	+2.7%	-3.7%	75	9	1.6
C499	1124 4.816 114.0	1124 4.816 121.3	1148 4.777 127.6	0%	-0.8%	158	4	1.7
C880	1077 4.935 106.4 M	1089 4.482 108.2	1053 4.430 173.1	-9.2%	-10.2%	117	6	1.5
cm151a	64 1.570 13.8 M	71 1.400 75.4	71 1.400 76.8	-10.8%	-10.8%	4	7	2.3
con1	45 0.954 58.5	43 0.904 52.1	43 0.904 56.0	-5.2%	5.2%	2	4	4.0
cordic	190 1.540 54.7	190 1.505 54.2	180 1.489 95.3	-2.3%	-2.3%	16	4	2.1
count	607 3.056 75.2	509 2.722 70.5	499 2.790 87.7	-10.9%	-8.7%	38	5	2.1
dalv	3184 10.809 297.3	2852 10.367 291.2	2794 10.308 440.7	-4.1%	-4.6%	257	4	1.7
duke2	1450 4.146 23.9 M	1484 3.926 411.5	1434 3.905 481.5	-5.3%	-5.8%	141	5	1.2
example2	996 3.208 94.9	952 3.199 114.8	952 3.208 106.9	-0.3%	0%	104	5	1.6
f51m	351 2.488 12.9 M	343 2.420 76.2	311 2.249 107.5	-2.7%	-9.6%	28	6	1.8
i4	766 1.300 19.3 M	510 1.077 93.3	426 1.040 120.0	-17.2%	-20.0%	6	4	3.0
i5	1580 2.461 23.7 M	1121 2.326 96.9	1109 2.326 104.1	-5.5%	-5.5%	92	4	1.9
inc	456 3.093 7.2 M	502 2.935 59.6	490 2.935 66.6	-5.1%	-5.1%	57	2	1.1
mux	113 1.625 66.7	109 1.586 56.3	108 1.496 69.4	-2.4%	-7.9%	3	6	2.7
pcler8	390 2.496 63.6	362 2.395 76.2	340 2.395 75.9	-4.1%	-4.1%	29	4	1.8
squar5	219 2.339 57.4	211 2.237 63.4	205 2.237 58.8	-4.4%	-4.4%	17	6	2.0
x1	801 2.196 25.8 M	809 2.108 99.1	787 2.167 117.7	-4.0%	-1.3%	6	6	1.6
z4ml	175 1.638 58.1	159 1.567 53.7	139 1.457 77.8	-4.3%	-11.1%	14	4	1.6
				-5.0%	-6.3%			

Table 5.3 Delay optimization for different sized cell families.

An improvement in delay, albeit a small one, was displayed when targeting larger cell families. In some cases the savings is very small, but it can be as high as 20%, when moving from family 2 to family 4. In a few rare cases, the larger cell family shows a worse mapping for a network than a smaller family. This is likely due to the heuristic nature of the tree-boundary procedure. It may improve the delay significantly for a mapping using a small library and not so drastically for the larger library.

Improvements for larger cell families can be attributed to two main sources. The first is the dependence on the type of benchmark to be mapped. The attributes listed in the rightmost columns can give some intuition about the type of network. The number of trees and deepest tree are self-explanatory. The *weighted level* (w) is defined as the average of all the trees in the network, weighted by the number of trees at each level, i.e.:

$$w = \frac{1}{n} \sum_i l_i \times d_i$$

where n is the number of trees with distinct level in the network, and t is the number of trees with level d . The manipulation scripts have generally produced networks which have low depth. The weighted levels are relatively low when compared to the depths of the largest cells in each cell family. Thus, there are few gates in the larger families which offer more logic depth savings, and thus, savings in delay.

To demonstrate this effect, collapsed versions (in factored form - not two-level) of some of the benchmarks are shown in Table 5.4.¹ While these networks offer a poorer performance than their counterparts in Table 5.3, they do demonstrate a greater percentage savings when using larger cell families. This is due, in part, to the larger trees in the network, which is intuitively shown through the larger weighted levels.

Network	Family 2 (2,2,2,4) area delay(ns)	Family 3 (3,3,4,9) area delay(ns)	Family 4 (4,4,6,16) area delay(ns)	Family 2 to 3 Δ delay	Family 2 to 4 Δ delay	# of Trees Depth Time Weighted level
cm151a	132 1.967	112 1.787	106 1.731	-9.2%	-12.0%	2 7 6.5
f51m	409 3.287	355 3.074	319 2.937	-6.5%	-10.7%	8 9 5.1
i4	562 1.591	490 1.109	426 1.040	-30.3%	-34.6%	6 12 8.3
i5	1410 5.870	1220 5.253	1194 5.205	-10.5%	-11.3%	66 15 9.2
inc	619 4.318	533 4.117	475 4.040	-4.7%	-6.4%	11 13 8.1
squar5	367 3.746	321 3.476	301 3.409	-7.2%	-9.0%	8 10 5.9
x1	1203 2.773	1063 2.488	1029 2.467	-10.3%	-11.0%	35 10 3.7
z4ml	245 2.989	211 2.661	201 2.570	-11.0%	-14.0%	4 9 6.3
				-11.2%	-13.6%	

Table 5.4 Impact of cell-families on collapsed versions of benchmarks.

Therefore, it can be concluded that cell-families with deep cells offer a substantial savings in delay only if the trees in the network are correspondingly deep. There then arises the question of whether delay-oriented logic minimization will ever perform in such a manner as to produce deep logic trees. It is anticipated that the algorithms for delay optimization presented in [28] will produce deep logic trees, since they are based on partial collapsing. From Table 5.4, one can see that full collapsing produces deep logic trees.

The second source of the effect on the impact of family size can be attributed to the

1. Not all benchmarks are collapsible, as they become too large to manage in memory.

delay parameters which are being used. These results are specific to the CMOS 1.2 μ target technology. If the delay parameters are set for a submicron process, the results will likely be completely different than these.

The information that is most useful from these tables may lie in the impact from family 2 to 3, when compared with the jump from family 3 to 4. There is a more noticeable decrease in the former, which may indicate that this CMOS 1.2 μ is suitable for applications using cells with three transistors in series, and benefits very little more from including cells with four transistors in series.

However, it is possible that there will be a significant additional savings when interconnection is taken into account, since a reduced number of cells leads to fewer interconnections between cells. Interconnect savings can improve area utilization and reduce capacitive and resistive loading, improving the delay.

As a point of interest, Table 5.5 is a compilation of mappings of the same benchmark set using area optimization in both mappers.

Network	Family 2 (2,2,2,4) area delay(ns)	Family 3 (3,3,4,9) area delay(ns)	Family 4 (4,4,6,16) area delay(ns)	Family 2 to 3 Δ area	Family 2 to 4 Δ area
apex6	1701 5.093	1459 3.677 M	1391 6.139	-14.2%	-18.2%
C1355	1024 6.959 M	998 7.222 M	1024 7.178	-2.5%	0%
C432	578 10.503	502 16.952 M	466 17.786	-13.2%	-19.4%
C499	1024 6.959 M	998 7.222 M	1024 7.178	-2.5%	0%
C880	912 9.208 M	748 10.828	714 10.456	-18.0%	-21.7%
cm151a	46 1.660	42 1.411	34 1.831	-8.7%	-26.1%
con1	43 1.010	37 1.100	31 1.179	-14.0%	-27.9%
cordic	169 1.825	141 1.881	131 2.341	-16.6%	-22.5%
count	305 6.320 M	247 6.381 M	247 6.444	-19.0%	-19.0%
dalu	2514 14.445 M	2180 17.346	2042 17.243	-13.3%	-18.8%
duke2	1083 4.303 M	831 5.645 M	762 6.292	-23.3%	-29.6%
example2	771 4.487 M	688 4.772 M	671 4.810	-10.8%	-13.0%
f51m	284 8.915 M	215 10.966 M	206 13.238	-24.3%	-27.5%
i4	466 1.804	330 1.460	298 1.770	-29.2%	-36.1%
i5	396 3.489	396 3.489	396 3.489	0%	0%
inc	436 3.287 M	331 3.800 M	311 5.017	-24.1%	-28.6%
mux	103 1.639 M	96 2.026	74 1.879	-6.8%	-28.2%
pcler8	199 4.151	173 4.893	171 4.742	-13.1%	-14.1%
squar5	187 2.348 M	136 3.118	132 2.831	-27.3%	-29.4%
x1	733 2.531	553 2.894	515 2.894	-24.6%	-29.7%
z4ml	96 2.649	82 2.789	78 3.014	-14.6%	-18.8%
				-15.2%	-20.4%

Table 5.5 Area optimization for different sized cell families.

5.4.5 Design Space Exploration

A design space graph is shown for the benchmark circuit 'C880' in Figure 5.4, resulting from the compilation of the many mappings in area and delay optimization mode. This resulted in *clouds* of possible implementations of the circuit. The lower left contour of each cloud is shown, representing the best implementations in an area/delay trade-off curve. Those produced by MIS are denoted by 'M', the rest were produced by Rossini.

Figure 5.4 Mappings for different sized libraries

Note that the mappings for $(s,p,l,n) = (4,4,6,16)$ are all performed by Rossini, since MIS is not capable of using this cell family.

Unfortunately, since at the moment, Rossini maps entirely for low area or entirely for low delay, it is difficult to move along the area-delay curve. This is the reason the curves are not smooth. However, this graph clearly depicts the trends of the use of larger libraries. The larger libraries offer a savings in both area and delay.

5.5 Chapter Summary

The Rossini technology mapping prototype has offered a platform for experimentation with logic benchmarks. Using delay parameters for a CMOS 1.2 μ process and a test library, Rossini has been shown to perform favorably when compared to MIS II for delay optimization. It produces mapped networks which perform better by an average 6.8% for an average area expense of 0.2%. This performance enhancement can be attributed to the new decomposition, phase assignment, and tree-boundary algorithms presented in Chapter 3.

Experimentation with delay optimization has shown that compared with the family with $(s,p,l,n) = (2,2,2,4)$, that the family with $(s,p,l,n) = (3,3,4,9)$ offers an average improvement of 5.0% in delay and the family with $(s,p,l,n) = (4,4,6,16)$ offers an average improvement of 6.3% in delay. It is anticipated that logic minimization which produces deeper logic trees would offer a further savings. It is also anticipated that delay parameters for a submicron CMOS process would again show a further savings.

The same experiments, run for area optimization, have shown an average savings of 15.2% in area for $(s,p,l,n) = (3,3,4,9)$ and an average savings of 20.4% in area for $(s,p,l,n) = (4,4,6,16)$, when compared with the family with $(s,p,l,n) = (2,2,2,4)$.

Chapter 6: Conclusions

6.1 Summary

Through a long development, this study has been successful in both of its objectives. By the introduction of a new approach to technology mapping, the goal of finding a method to evaluate the impact of different sized cell families has been met. The new approach stems from limitations found in present mapping ideologies for use with complex cell families. Those which have been introduced for use with a standard cell library have weaknesses when dealing with a very large set of cells. Those which have been introduced for use with large cell families have little performance optimization capabilities. The Rossini approach is based on a new constructive matching method, which is capable of quickly forming patterns which correspond to complex cells which may be crafted by a cell compiler. The timing quality of the technology-bound circuit is maintained through performance-driven partitioning and covering techniques.

The study of the impact of cell family sizes has demonstrated the increased quality of technology-mapped circuits when larger cells are available. For the target CMOS 1.2 μm technology and compared with a complete family with two transistors in series, a complete family with three transistors in series offers an average savings of 5.0% in delay, while a complete family with four transistors in series offers an average savings of 6.3% in delay. The savings has been attributed to the ability of cells with a high logic level to reduce the depth of the logic networks. It is expected that the trends in the progress of fabrication technology and logic minimization capability will increase this percentage savings in delay. In addition, large complex cells offer a substantial savings for area optimization. Also for the target CMOS 1.2 μm , a complete family with three transistors in series offers a 15.2% savings in area and a complete family with four transistors in series offers a

20.4% savings in area over a complete family with two transistors in series.

As for the secondary objective, new techniques in decomposition, phase assignment, and tree-boundaries handling have been introduced for the technology mapping process. This lays the groundwork for further enhancements. The approach has been implemented in a modular Common Lisp system, and new methods may be tested through the addition of new modules.

6.2 Contributions

The Rossini mapping approach has made a key contribution to technology mapping for complex cell families. Rossini can perform high quality synthesis for any size family. Mapping for complex cells is made possible by the introduction of a *constructive* matching module, which iteratively builds sub-trees corresponding to patterns under the restriction of simple parameters. The ability to handle large cell families does not limit the system's ability to produce high performance circuitry.

New ideas for enhancing the ability of mapping systems have been introduced. The unbalanced decomposition procedure has shown an ability to generally improve the timing of circuits with minimal area increase. The tree phase assignment heuristic has proven to be an elegant method of treating phase assignment during the covering of a tree. Although young in its development, the simple tree-boundary heuristic proved effective in implementation.

The presented system has allowed the investigation of different sized cell families. A useful study of the impact of the size of cell families on the quality of technology mapped circuits has been included. The system has also opened doors for new directions of investigations.

6.3 Recommendations for Future Work

With the present platform for experimentation, there is room for a number of short-term investigations. The most obvious is experimentation with delay parameters corresponding to shorter gate-length technologies. If the studies contained in [1] are a reflection of the behavior of real technologies, then the trend of large savings for large cell families will continue. Also, experiments dealing with different sized cell families could be examined. For example, the impact of constraining the l and n parameters would be of interest for compiler and layout limitations, respectively.

New performance-oriented logic minimization techniques can be used as precursors to the technology mapper. Specifically, the minimization algorithms described in [28] may enhance the utilization of complex cells, and further current trends.

In the medium term, upgrades to the techniques used in this prototype can be introduced. New performance-driven tree-partitioning techniques can be investigated. Since fanout dependence becomes of more concern with newer technologies, the tree-boundary fanout heuristic can be improved. In addition fanout optimization can be extended to include buffer optimization and sink allocation [29][30]. On the practical side, matching techniques for cells with multiple common inputs may be added. These may be handled by 'leaf-DAG's', as has been done in MIS [19].

For long term goals, the scope of the approach can be enlarged. The delay model can be modified for the inclusion of parasitics. Electrical optimization in the form of transistor-sizing would be a welcome asset. Usually, this is looked upon as a separate issue, relating closer to the layout synthesis problem. However, reducing the gap between logic and layout synthesis may soon become an intense area of research. The current combinatorial techniques can be extended to handle sequential logic, which would involve a revisit to the partitioning procedure so that it can handle general-directed graphs. Also, the mapper can be extended to include floor-planning concerns such as placement of the cells on the layout level. In the context of all these goals, testing issues should also be addressed.

Appendix A. Glossary of Terms and Symbols

- binary decomposition:** Manipulation of a tree such that each node has a fanin restricted to two.
- binate variable:** A logic variable, where both phases of the signal are present.
- CMOS:** Complementary Metal Oxide Semiconductor.
- complex cell:** A combinatorial logic structure, which performs alternating Boolean "AND" and "OR" functions ending in an "INVERT" in a CMOS implementation.
- complex gate:** see complex cell.
- covering:** The selection of previously identified implementations which can perform the given logic function in the best way.
- cube:** A product term of a logic expression written in sum-of-products form.
- cube-free:** A logic expression out of which no cubes may be factored. This excludes single cubes.
- DAG:** Directed Acyclic Graph.
- DC:** Don't Care Set.
- don't care set:** The set of input conditions in which a Boolean expression may have either logic values
- k:** A scaling factor which relates the delay through a cell to the logic level of a tree.
- kernel:** A quotient of a logic expression f/c , such that f/c is cube-free and c is a cube.
- l:** The limit of the logic level of a complex cell.
- literal:** A Boolean variable which appears in either complemented or uncomplemented form.
- MCNC:** Microelectronics Center of North Carolina.
- MIS:** Multilevel Logic Interactive Synthesis System - a synthesis tool developed at Berkeley University.
- matching:** The identification of technology implementations which can perform the given logic function.

- n: The limit of the number of inputs of a complex cell.
- NPN-equivalent: Two logic functions which are equivalent through negation of its variables, permutation of its variables, or negation of the function itself.
- off-set: The set of input conditions which cause a Boolean expression to have logic value zero.
- on-set: The set of input conditions which cause a Boolean expression to have logic value one.
- p: The limit of the number of p-type transistors in series in a CMOS complex cell.
- partitioning: The parsing and sectioning of a DAG into trees.
- phase-assignment: The selection of logic phase for a particular signal.
- s: The limit of the number of n-type transistors in series in a CMOS complex cell.
- SOP: Sum-Of-Products logic form.
- speed_up: An algorithm [27] which performs delay optimization for logic.
- tautology: A Boolean expression which is true under all conditions.
- tree: A logic structure, where nodes represent Boolean operations and boundary nodes represent logic variables. The fanout of a node is restricted to one, whereas the fanin is unbounded.
- unate variable: A logic variable, where only one phase of the signal is present.
- *: Boolean Logic "AND" function.
- + Boolean Logic "OR" function.
- ! Boolean Logic "NOT" or "INVERT" function.
- $\bar{*}$: Boolean Logic "NAND" or "AND-INVERT" function.
- $\oplus$: Boolean Logic "Exclusive-Or" function.

Appendix B. Rossini User Manual

The Rossini Technology Mapper can be invoked from the Common Lisp interactive interface by loading the main module. It contains the following commands:

- Parsing commands:
 - (read_eqn 'fn) - read and parse the logic network contained in the file fn.eqn. The network must be in 'eqn' format and headed by the primary inputs and outputs.
 - (write_eqn 'fn) - writes the current network in eqn format. If a file name is not included, the network will be written to the screen.
 - (eliminate_single_terms)
 - eliminate terms with a single input. Such a term will be pushed to its destination trees, unless it is a primary output, then it will be pulled to its source (if possible).
 - (eliminate n) - eliminate terms with a fanout less than or equal to the value of n. A value of 1 is used to form all trees with a fanout greater than one.
 - (dump_database)
 - dumps terms in memory to the screen. This is a debugging utility.
- Main-level commands:
 - (map_net parameter s p l n)
 - technology maps the network in memory, for a complex family with constraints (s,p,l,n). The parameter = 0 for area optimization, and 1 for delay optimization.
 - (write_m_code 'fn)
 - writes code for Matlab¹ graphics to fn.m. This is a utility for exploring area-delay design space graphs.
 - (write_mapped_eqn 'fn)
 - writes the mapped network in eqn format. A 'pseudo' eqn format for which each line represents a logic cell. If a file name is not included, the network will be written to the screen.
 - (improve_area) - apply the tree-boundary heuristic. Removes extra inverters at tree-boundaries after technology mapping in a way that will not degrade the performance of the network.
 - (write_blif 'fn) - writes the mapped network to fn.blif in Berkeley Logic Interface Format (the netlist form). The area and delay of a BLIF network can be evaluated using 'read_blif.cl'.

1. Matlab is a trademark of The MathWorks, Inc.

- (write_dag 'fn) - writes mapped network to fn.dag. This file can be processed by DAG¹ which produces a simplified connectivity netlist in Post-Script form.
- (cmc) - count mapping cost. This is not valid after the tree-boundary heuristic is applied (use read_blif.cl).
- (stats) - give statistics for the network. It displays the number of primary inputs and outputs, the number of trees, a summary of the number of trees with corresponding level, and the weighted level of the network.
- (summary) - give a summary of the cells used in the mapped network organized by the parameters (s,p,l,n)

A typical design flow may proceed as follows:

***** The Rossini Technology Mapper (v1.0) *****

```

<Rossini> (read_eqn 'cmb)
(TERM1 TERM2 TERM3 TERM4 TERM5 TERM6 TERM7 TERM8 TERM9 TERM10 ...)

<Rossini> (write_eqn)
INORDER = a b c d e f g h i j k l m n o p;
OUTORDER = q r s tt;
q=(!p0)*j*k*1;
r=p+(!m0);
s=(!o)+(!m0);
tt=(!s0)*(!n)*(!o)*(!p);
p0=(!b)+(!a)+(!n0)+(!o0);
m0=((!v0)*g*f*e)+((!v0)*(!h)*f*e)+((!v0)*(!h)*(!g)*e)+((!v0)*(!h)*(!g)*(!f));
s0=f+e+(!q0)+(!r0);
v0=(!u0)+(i*(!h))+(j*(!i))+(k*(!j));
n0=c*d*e;
o0=f*g*n*i;
q0=(!g)*(!h)*(!i);
r0=(!j)*(!k)*(!l)*(!m);
t0=(m*(!l))+(n*(!m))+(o*(!n))+(!p)*e);
u0=((!t0)*(!l))+((!t0)*k);

<Rossini> (eliminate_single_terms)
Number of Terms Eliminated: 0

<Rossini> (eliminate 1)
Number of Terms Eliminated: 7

<Rossini> (stats)
16 Primary Inputs. 4 Primary Outputs.
7 Terms.
```

1. DAG is distributed by AT&T Bell Laboratories.

Terms with Level

4 x 1
2 x 2
1 x 3

Weighted level: 1.5714285

<Rossini> (map_net 1 3 3 4 9)

7 Terms to Map

1.. 2.. 3.. 4.. 5.. 6.. 7..

Mapping complete.

((MT15) (MT36) (MT67) (MT90) (MT93) (MT96) (MT119))

<Rossini> (cmc)

Netlist Area: 181

Optimizing Area: 181

Average delay: 1.5904951

Critical delay: 2.0766401

<Rossini> (improve_area)

Warning - write the mapped network to a blif file and
use read_blif.cl for correct delay values.

Number of inverters removed: 11

<Rossini> (write_mapped_eqn)

INORDER = a b c d e f g h i j k l m n o p;

OUTORDER = q r s tt;

t0=!(mt13_not*mt14);

mt13=!((n_not0+m)*(m_not0+l));

mt14=!((p_not0*e)+(o*n_not0));

v0=!(((t0_not0*l_not0)+(t0_not0*k))*mt34_not);

mt34=!((j_not0+i)*(k_not0+j)*(i_not0+h));

m0=!(mt65*mt66);

mt65=!((v0_not*mt57_not)+(v0_not*mt58_not));

mt66=!((v0_not*mt59_not)+(v0_not*mt60_not));

mt57=!(e*h_not0*f);

mt58=!(e*g*f);

mt59=!(mt55*h_not0);

mt60=!(e*h_not0*g_not0);

mt55=!(g+f);

q_not=!(mt88*mt89);

mt88=!(mt84+mt85);

mt89=!(mt86+mt87);

mt84=!(j*a*f);

mt85=!(i*b*e);

mt86=!(l*d*h);

mt87=!(k*c*g);

r=!(p_not0*m0);

s=!(o*m0);

tt_not=!(mt117*mt118);

mt117=!(mt113+mt114);

mt118=!(mt115+mt116);

mt113=!(e_not0*mt109);

mt114=!(m_not0*f_not0*g_not0);

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

```

mt115=!(h_not0*mt111);
mt116=!(j_not0*i_not0*k_not0);
mt109=!(o+n);
mt111=!(p+1);
mt13_not=mt13;
n_not0=!n;
m_not0=!m;
p_not0=!p;
v0_not=!v0;
mt34_not=mt34;
t0_not0=!t0;
l_not0=!l;
i_not0=!i;
j_not0=!j;
k_not0=!k;
mt57_not=mt57;
mt58_not=mt58;
mt59_not=mt59;
mt60_not=mt60;
h_not0=!h;
g_not0=!g;
q=!q_not;
tt=!tt_not;
e_not0=!e;
f_not0=!f;

```

<Rossini> (summary)

```

21 x (a)' s=1 p=1 l=1 n=1
9 x (ab)' s=2 p=1 l=1 n=2
7 x (a+b)' s=1 p=2 l=1 n=2
9 x (abc)' s=3 p=1 l=1 n=3
1 x ((a+b)(c+d))' s=2 p=2 l=2 n=4
3 x ((ab)+(cd))' s=2 p=2 l=2 n=4
1 x (((ab)+(cd))e)' s=3 p=2 l=3 n=5
1 x ((a+b)(c+d)(e+f))' s=3 p=2 l=2 n=6

```

```

21 gates with n=1
16 gates with n=2
9 gates with n=3
4 gates with n=4
1 gates with n=5
1 gates with n=6

```

```

46 gates with l=1
5 gates with l=2
1 gates with l=3

```

Total number of gates: 52

<Rossini> (write_blif 'cmb)
T

<Rossini> (write_dag 'cmb)
T
<Rossini>

Appendix C. Sample Common Lisp Code - Matching Module

```

;*****
; Match.cl
;
; - A 'constructive' matcher for technology mapping. Iteratively
; constructs sub-tree patterns under the constraints of (s,p,l,n).
;
; Cliff Liem - Synthesis Group
; Department of Electronics
; Carleton University
;
; working version - November 1990
; last updated - April 1991
;*****

(defun help_match ()
  (format *terminal-io* " ~%")
  (format *terminal-io* " Matching commands:~%")
  (format *terminal-io* " (match term s p l n) -find all the matches for all nodes of the
term~%")
  (format *terminal-io* " with constraints s p l n~%")
  )
;*****
;
; 'match' finds all the matches of a term, using the node list.
;
;*****
(defun match (term s p l n)
  (send term :build_node_list)
  (count_gensym 'mm 0) ; initialize match name counter
  (let ((count_matches '(+)))
    (setq count_matches (append count_matches
      (mapcar #'(lambda (x)
        (let* ((node (symbol-value x))
              (level (send node :level))
              (equation (send node :equation))
              (matches_found '()))
          (if (= level 0) ; remove inverter
              (if (listp equation)
                  (setq matches_found (list (make_match (cadr equation)
                                                         2 1 1 0 -1 nil nil)))
                  (setq matches_found (list (make_match equation
                                                         0 1 1 0 -1 nil nil)))
                  )
              (setq matches_found (find_matches equation s p l n))
              )
          (send node :set-matches matches_found)
          (length matches_found)))
        (send term :nodes))))
    (eval count_matches))
  )
;*****
;
; 'find_matches' exhaustively tries each combination of branches of a
; tree and finds all the matches to patterns constrained by
;
;   the maximum path size (s)
;   the maximum cut size (p)
;   the depth (l)
;
;   and the number of inputs (n)
;
; the tree's nodes must be labelled (use method 'build_node_list')
; the top tree is assumed to be two matches (the dual)
;
;*****

```

```
(if (= dual cost 0) (setq nn_equation (dual nn_equation)))
```

```
(setq trial_tree (subst nn_equation y trial_tree))
(setq trial_leaves
  (append (remove y trial_leaves) (cdr nn_equation)))
(setq unsearched_leaves
  (append unsearched_leaves (cdr nn equation))))
```

```
x)
{let* ((flat_tree (flatten trial_tree))
      (dual_tree (dual flat_tree))
      (not_dual_cost (abs (- dual_cost 2)))
      (series (find cutpath '* flat_tree)))
```

```

        (parallel (find_cutpath '+ flat_tree))
        (level (find_level flat_tree))
        (inputs (find_inputs flat_tree)))
    (if (and (<= level 1) (<= inputs n))
        (if (and (<= series s) (<= parallel p))
            (progn (setq new_match_list
                        (append new_match_list
                              (list (make_match flat_tree dual_cost series parallel
                                                level inputs trial_leaves
                                                unsearched_leaves))))
                    ; avoid redundant search of leaves of trees and
                    ; duals, so set unsearched leaves to nil if
                    ; the tree matches.
                    (if (and (<= parallel s) (<= series p))
                        (setq new_match_list
                            (append new_match_list
                                    (list (make_match dual_tree not_dual_cost
                                                    parallel series level inputs
                                                    trial_leaves nil))))))
                    (if (and (<= parallel s) (<= series p))
                        (setq new_match_list (append new_match_list
                                                    (list (make_match dual_tree not_dual_cost
                                                            parallel series level inputs
                                                            trial_leaves unsearched_leaves))))))
                    ))))
    (all_combinations search_leaves))
    (send match :set-unsearched_leaves nil)
    new_match_list)
)
;
; 'make_node' creates a new instance of the flavor node.
;
(defun make_node (level fanout)
  (let ((node_name (count_gensym 'nn)))
    (set node_name (make-instance 'node :level level
                                   :fanout fanout))
    node_name)
)
;
; Flavor Definitions
;
(defflavor node
  (equation level fanout matches best_match)
  () :settable-instance-variables)
(defflavor match
  (equation s p l n term_out term_leaf double leaves unsearched_leaves
    dual_cost area_cost dual_out_tt dual_delay_cost dual_delay_value
    out_transit_time delay_cost delay_value)
  () :settable-instance-variables)
;
; 'make_match' creates a new instance of flavor match in the database.
;
(defun make_match (equation dual_cost s p l n leaves unsearched_leaves)
  (let ((match_name (count_gensym 'mm)))
    (set match_name (make-instance 'match
                                   :equation equation
                                   :dual_cost dual_cost
                                   :s s
                                   :p p
                                   :l l
                                   :n n
                                   :leaves leaves
                                   :unsearched_leaves unsearched_leaves))
    match_name)
)

```

```

;
; build sorted node list according to level using 'initialize' (tree)
;
(defmethod (term :build_node_list) ()
  ()
  (let* ((node_list (sort (initialize (nth 2 equation)) #'< :key
    #'(lambda (x) (send (symbol-value x) :level))))
    (root_node (symbol-value (car (last node_list)))))
    (send self :set-nodes node_list)
    (if (= fanout 0)
      (send root_node :set-fanout 1)
      (send root_node :set-fanout fanout))
    )
  )
)
;
; 'initialize' creates a list of node objects for all the nodes of the term.
; This is required for both the matching and covering routines.
;
(defun initialize (tree)
  (let* ((level (find_level tree))
    (node (make_node level 1))
    (sv_node (symbol-value node))
    (node_list (list node))
    (new_tree tree))
    (if (listp tree)
      (if (equal (car tree) '')
        (send sv_node :set-level 0)
        (progn
          (setq new_tree (list (car tree)))
          (mapc #'(lambda (x)
            (let ((child_list (initialize x)))
              (setq new_tree (append new_tree (list (car child_list))))
              (setq node_list (append node_list child_list))
            ))
          (cdr tree))
        ))
      (send sv_node :set-equation new_tree)
      node_list)
  )
)

```


Appendix D. Relationship between (s,p,l,n) parameters.

The parameters s and p restrain the maximum number of n-type MOS transistors and the maximum number of p-type MOS transistors in series in a CMOS circuit implementation. This circuit realization may be represented in two ways which aid in manipulation. The first is a series-connected graph, where an edge represents the controlling signal on the gate of a transistor and a vertex represents an interconnection between transistor sources and drains. This representation is useful for measuring the technology parameters s and p . The maximum path set (s) is the maximum number of edges from sink to source, and the maximum cut set (p) is the maximum number of edges cut from left to right (see Figure 2.22) The second representation is a tree structure. The nodes of the tree represent boolean operations and the leaves represent signals (see Figure 2.10) This representation is useful for effective mapping methods, since the level of the tree can aid in delay optimization. The level of the tree corresponds directly to the level of the logic.

One can build a graph which can fulfill s and p in one of two ways or in a combination of these two ways. The first is to fill one parameter independent of the other. For example, if $s = 3$ and $p = 3$, we can begin with $p = 1$ and add edges until $p = 3$. Subsequently, we can repeat any one of these constructs until $s = 3$. Similarly, we can begin with $s = 1$ and add edges until $s = 3$. Then we repeat these units until $p = 3$. This method is shown in Figure E.5. The corresponding trees to this manner of building graphs will be in sum of products (SOP) form (fill s then p), or product of sums (POS) form (fill p then s). These are the lowest level trees possible with $l = 2$.

Note, if the parameters are filled in each step, then this manner gives the cells with the most possible inputs. For this example, the largest n is 9. It is easy to see that we can generalize this. Given s and p , the largest possible n is s multiplied by p .

Figure E.5 Building graphs, parameters independent.

The second manner in which to build a graph is to fill the parameters of s and p *at the same time* or dependently. Again, let $s = p = 3$. Beginning with $s = p = 1$, we can intercede this edge with a second edge, thus increasing s and p . If we continue this procedure recursively, then we quickly fill s and p . The method is shown in Figure E.6.

Figure E.6 Building graphs, parameters dependent.

This method will result in a graph with the dec_l corresponding tree. For this example, $l = 4$. This can be generalized. Given $s = p$, the maximum possible l is $(s + p - 2)$. For $s \neq p$, note that the procedure can be cut off; one parameter is filled, while the other parameter is not yet filled. Given $s \neq p$, the maximum possible l is $(2 * \min(s, p) - 1)$.

Naturally, a combination of these two methods will also give graphs which do not exceed the constraints of s and p . This leads to a great number of cell patterns as s and p increase. Discussions on the number of graphs for an (s,p) -family are found in [3] and [19]. The values, enumerated for $(s,p) = 1$ through 6, is found in Table 2.2.

Appendix E. String based pattern matching

This algorithm is a string matching routine, introduced as a search technique for text [17]. It can be modified to work with trees, since trees can be described as strings. Although Keutzer [11] is not explicit on how trees are represented as strings, one can speculate, since the network has been parsed into two-input NAND gates and inverters, that a tree such as Figure A.1 can be represented as “! $\bar{a}$! $\bar{a}$ bcd”, a *depth-first* representation. The basis of the algorithm is that a pattern is converted to a finite automaton, which works as a pattern matching machine.

Figure A.1 Example tree

A pattern matching machine may be constructed as a series of functions. An example showing the pattern functions for the strings {hers, his, he, she} is shown in Figure A.2.

Figure A.2 String pattern matching machine

The goto function takes as input a state and a symbol, then outputs a new state or a failure message. For example, in Figure A.2(a), the start state would be 0. The goto function

$g(0,h) = 1$, indicating if in state 0 and having encountered symbol h , then move to state 1. If there is no arrow, that indicates a fail, for example, $g(1,\sigma) = \text{fail}$ for $\sigma \neq \{e, i\}$. In the case of a failure, the failure function shows where the next state is to be visited. If a key state is encountered, the output function indicates the found pattern.

For greater efficiency with this matching procedure, the goto and failure functions may be combined to a *next* function. The next function avoids making unnecessary failure transitions. For example, if the procedure is in state 4, and does not encounter an e , then the failure function points to state 1. The consideration of e on state 1 is then superfluous, since it has been established that an e has not been encountered. In fact, if $\{\text{his}\}$ were not a keyword, the entire transition to state 1 would be unnecessary. The next function, depicted in Figure A.3, eliminates unnecessary failure transitions. The period stands for any symbol other than those above it.

	input symbol	next state
state 0:	h	1
	s	3
	.	0
state 1:	e	2
	i	6
	h	1
	s	3
state 9: state 7: state 3:	.	0
	h	4
	s	3
	.	0
state 5: state 2:	.	8
	h	1
	s	3
	.	0
state 6:	s	7
	h	1
	.	0
state 4:	e	5
	i	6
	h	1
	s	3
	.	0
state 8:	s	9
	h	1
	.	0

Figure A.3 Next move function

A library of patterns for gate matching can be constructed in the same way as the string patterns, since it is in a tree structure. This matching automaton considerably increases the performance over trying each pattern at each node.

References

- [1] T. Sakurai and A.R. Newton, "Delay Analysis of Series-Connected MOSFET Circuits", *IEEE Journal of Solid-State Circuits*, vol. 26, no. 2, Feb. 1991, pp. 122-131.
- [2] K. Keutzer, K. Kolwicz, M. Lega, "Impact of Library Size on the Quality of Automated Synthesis", *Proceedings of the International Conference on Computer-Aided Design*, 1987, pp. 120-123.
- [3] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Technology Mapping in MIS", *Proceedings of the International Conference on Computer-Aided Design*, 1987, pp. 116-119.
- [4] J. Bruno, R. Sethi, "Code Generation for a One Register Machine", *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 23, 3, 1976, pp. 502-510.
- [5] A. Aho, et al., "Code Generation for Expressions with Common Subexpressions", *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 24, 1, 1977, pp. 146-160.
- [6] G. Brassard and P. Bratley, *Algorithmics, Theory and Practice*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1988, pp. 142-168.
- [7] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 11, Nov. 1987, pp. 1062-1081.
- [8] R. Brayton, et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
- [9] D. Gajski, *Silicon Compilation*, Addison-Wesley Publishing Co., 1988; Ch. 7 : "The Yorktown Silicon Compiler", R. Brayton, R. Canposano, G. DeMicheli, R. Otten, J. van Eijndhoven.

- [10] D. Gregory, K. Bartlett, A. deGeus, G. Hachtel, "Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic", *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, June 1986, pp. 79-85.
- [11] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching", *Proceedings of the 24th ACM/IEEE Design Automation Conference*, June 1986, pp. 79-85.
- [12] F. Mailhot and G. DeMicheli, "Technology Mapping Using Boolean Matching and Don't Care Sets", *Proceedings of the European Design Automation Conference*, 1990, pp. 212-216.
- [13] K. Sakouti, G. Saucier, "A fast and effective technology mapper on an autodual library of standard cells", *Proceedings on the IFIP Working Conference on Logic and Architecture Synthesis*, May 1990, pp. 137-150.
- [14] M.R.C.M. Berkelaar, J.A.G. Jess, "Technology Mapping for Standard-Cell Generators", *Proceedings of the International Conference on Computer-Aided Design*, 1988, pp. 470-473.
- [15] A. deGeus, W. Cohen, "A Rule-based System for Optimizing Combinational Logic", *IEEE Design and Test of Computers*, 2, 8, Aug. 1985, pp. 22-32.
- [16] W. Cohen, K. Bartlett, A. deGeus, "Impact of Metarules in a Rule-based Expert System for Gate Level Optimization", *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 1985, pp. 873-876.
- [17] A. Aho, M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search", *Communications of the ACM*, 18, 6, June 1975, pp. 333-340.
- [18] G. DeMicheli, D. Ku, F. Mailhot, T. Truong, "The Olympus Synthesis System", *IEEE Design & Test of Computers*, 6, 10, October 1990, pp. 37-53.
- [19] R. Rudell, *Logic Synthesis for VLSI Design*, Ph. D. Dissertation, U.C. Berkeley, Memorandum No. UCB/ERL M89/42, 26 April 1989, 223 p.
- [20] R. Shum, *A New Timing Model for the Characterization of CMOS Cells*, Master's Thesis, Carleton University, pending.

- [21] M. Lefebvre, *CMOS Leaf Cell Synthesis*, Ph. D. Dissertation, Carleton University, April 1989, 143 p.
- [22] G. Hachtel, R. Jacoby, C. Morrison, "TECHMAP: Technology Mapping with Delay and Area Optimization", *Proceedings of the International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, Grenoble, May 1988.
- [23] K. Keutzer, M. Vancura, "Timing Optimization in a Logic Synthesis System", *Proceedings of the International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, Grenoble, May 1988.
- [24] P. Paulin, "Logic Decomposition Algorithms for the Timing Optimization of Multi-Level Logic", *Proceedings of the International Conference on Computer Design*, Cambridge, MA., October 1989.
- [25] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Multilevel Logic Minimization Using Implicit Don't Cares", *I.E.E.E. Transactions on Computer-Aided Design*, Vol. 7, No. 6, June 1988, pp.723-740
- [26] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis", *Proceedings of the I.E.E.E.*, Vol. 78, No. 2, February 1990, pp. 264-300.
- [27] K. Singh, A. Wang, R. Brayton, A. Sangiovanni-Vincentelli, "Timing Optimization of Combinational Logic", *Proceedings of the International Conference on Computer-Aided Design*, 1988, pp. 282-285.
- [28] H. Touati, H. Savoj, R. Brayton, "Delay Optimization of Combinational Logic Circuits through Clustering and Partial Collapsing", *Proceedings of the M.C.N.C. International Workshop on Logic Synthesis*, May 1991.
- [29] K. Singh and A. Sangiovanni-Vincentelli, "A Heuristic Algorithm for the Fanout Problem", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990, pp. 357-360.
- [30] H. Touati, C. Moon, R. Brayton, "Performance-Oriented Technology Mapping", *Proceedings of the sixth MIT VLSI Conference*, 1990, pp. 79-97.

END

28-05-92

FIN