# TECHNOLOGY MAPPING AND LAYOUT SYNTHESIS OF DCVS

By

Carly Wong

B. A. Sc. (Electrical Engineering) University of British Columbia, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

ELECTRICAL ENGINEERING

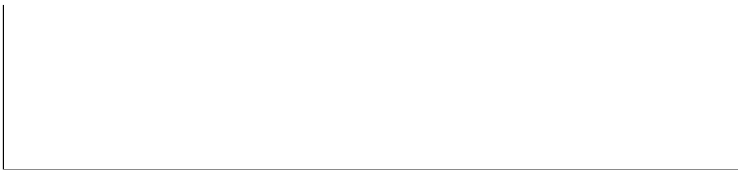
We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1992

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.



Electrical Engineering
The University of British Columbia
2356 Main Mall
Vancouver, Canada
V6T 1Z4

Date: April 28. 1992.

## Abstract

Differential Cascode Voltage Switch (DCVS) logic is a dynamic logic family that has a number of desirable properties. In particular, it is hazard-free, easy to make fully robust path delay-fault testable, and has a number of unique timing properties that make it very suitable for self-timed circuits.

This thesis investigates the problem of implementing logic with DCVS, and in particular, the automatic synthesis of DCVS circuits. This task is challenging because DCVS gates are usually significantly more complex than standard static CMOS both in terms of internal connectivity and number of inputs. Most of the work done in circuit synthesis, such as technology mapping and layout synthesis, are not directly applicable to DCVS.

We present DMAP, a technology mapping and layout synthesis system for DCVS. The system operates in three steps. First, the combinational circuit is decomposed and partitioned into sizable Boolean function clusters. Secondly, DMAP takes each cluster function, and generates a DCVS cell layout that implements that function. We develop a heuristic algorithm for finding a suitable transistor path to lay out the DCVS pull-down network. Finally, the generated DCVS cells are placed and routed inside the CADENCE environment.

Experimental results indicate that DCVS circuits can be implemented with considerably fewer cells than by conventional mapping techniques. Furthermore, the number of nets that need to be wired is typically less than twice the number of nets used in standard mapping. This is better than the intuitive assumption that dual-rail circuits require twice as many wires as their single-rail counterparts. Large DCVS circuits can be feasibly produced. DMAP is a first step towards an integrated system for dynamic circuit synthesis.

## Table of Contents

## List of Figures

## List of Tables

**Acknowledgement**

I would like to express my sincere thanks to:

Dr. Carl Seger, mentor and true scholar, who supervised me throughout the course of this work, through all the nitty gritty details, and whose incredible patience and dedication to his students and to his research will always be an inspiration to me;

Dr. Patrick McGeer, who despite being 1000 miles away, still gave tremendous encouragement, to-the-point email responses, and who taught me many things from how to give a presentation to the difference between "it's" and "its";

Dr. Dan Camporese, who first introduced me to, and encouraged me to continue in, the wonderful world of graduate studies, VLSI, and CAD of VLSI;

Dave Gagne, who taught me many tricks in EDIF and EDGE and all that fun stuff;

Elizabeth Seto, who proofread my thesis draft with great care;

Many other faculty and staff members in EE and CS, who helped to make life easier and enjoyable;

All my friends, some nearby and some far away, who listened and shared with me, who laughed and cried with me;

And last, but not least, my parents, whose unfailing love and support were behind me at all times.

I thank God for all of you for making this all possible.

# Chapter 1

# Introduction

Dynamic logic families, such as Differential Cascode Voltage Switch (DCVS) [24], have very attractive properties both at the device level as well as the design level. This thesis analyzes the problem of implementing combinational logic with DCVS. DMAP, a technology mapping and layout synthesis system for DCVS, is presented. The system takes as input, arbitrary combinational logic, and produces layout cells ready for placement and routing in the CADENCE design environment.

## 1.1 Outline of Thesis

The remainder of Chapter 1 reviews basic definitions and notations used in this thesis report. Discussion of the thesis topic begins in Chapter 2. First, the Computer-Aided Design automation process is presented. We show, in particular, how the problems of technology mapping and layout synthesis fit into the design cycle. Conventional methods of attacking these problems are reviewed. Next, DCVS logic is presented as an attractive design technology. We discuss DCVS's dynamic operation, DCVS properties, and DCVS applications. We then describe the problem this thesis seeks to solve: technology mapping and layout synthesis for **DCVS**. Previous work related to solving this problem is reviewed.

Chapter 3 describes in detail DMAP, a system which performs technology mapping and layout synthesis for DCVS. First, an overview of the system is given. Each of the system steps: circuit clustering, cell generation, and cell connecting, are elaborated on. Clustering is done to

obtain sizable functions from which to implement the DCVS gates in order to achieve minimum delay between the circuit's inputs and outputs. DCVS cells are generated in a systematic way in a one-dimensional transistor strip layout style. The cell generation step investigates the sub-problems of determining suitable DCVS logic trees, transistor path finding, and intracell routing. CADENCE placement and routing tools connect the generated cell library together to produce a final DCVS chip layout.

Experimental results of DMAP, and the discussion of these results, are given in Chapter 4. The system presented in this thesis is compared with conventional technology mapping systems in terms of various factors such as gate count, wire count, and cell area count. Finally, conclusions and future directions appear in Chapter 5.

## 1.2 Basic Definitions and Notations

This section is devoted to reviewing some common definitions and notations in graph theory and integrated circuit theory that will be used in the development of this thesis. Readers already familiar with the jargon and basics in these areas may wish to skip this section, and proceed to Chapter 2.

### 1.2.1 Integrated Circuit Theory

*Complementary Metal Oxide Silicon* (CMOS) technology has become the most common fabrication process for digital integrated circuit industry. It provides two types of transistors as the basic building blocks in digital circuit design: the *N-type* or *nMOS* transistor; and the *P-type* or *pMOS* transistor. The symbols used to represent them are shown in Figure 1.1. The N-type transistor is an active-high switching device which acts as a closed switch when its input is high, i.e., when **GATE** is high, **SOURCE** and **DRAIN** are connected. The P-type transistor is an active-low switching device which acts as a closed switch when its input is low, i.e., when

**GATE** is low, **SOURCE** and **DRAIN** are connected.

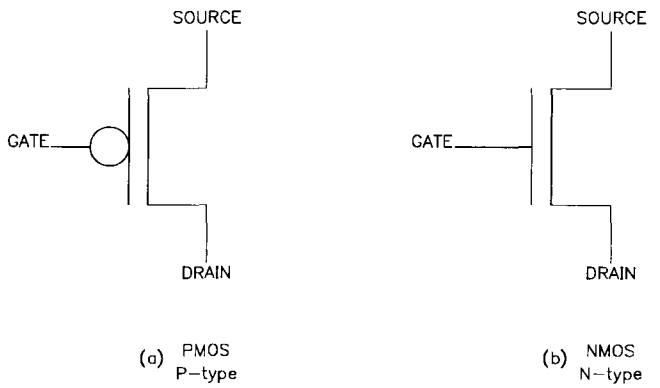



Figure 1.1: CMOS transistor symbols.

These transistors can be used in different ways to design digital systems. There are two major classifications of *design styles* or *design technologies* used with these CMOS devices: *static CMOS* and *dynamic CMOS*.

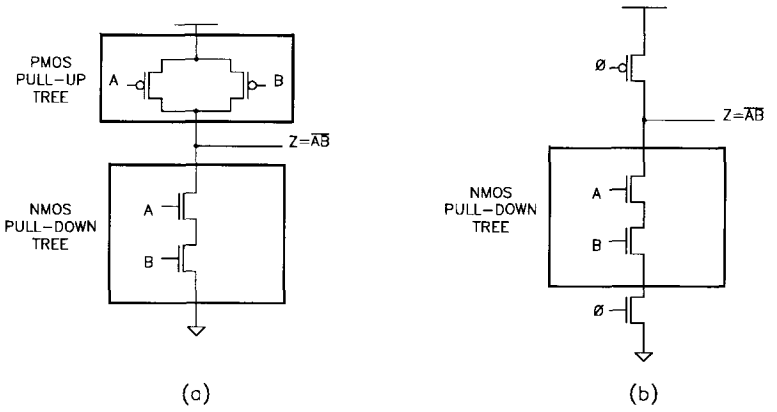



Figure 1.2: (a) Static CMOS (b) Dynamic CMOS

Static CMOS is also referred to as *fully complementary CMOS logic.* Figure 1.2(a) shows

a static CMOS 2-input NAND gate. The P-type transistor network, or *pull-up* (to power), and the N-type network, or *pull-down* (to ground), are series-parallel duals of each other. This fact suggests obvious transistor redundancy, since both networks serve to realize the same function. This led to the development of dynamic circuits.

A basic dynamic CMOS gate is shown in Figure 1.2(b). It consists of a single logic network implemented in the pull-down nMOS region (in this case, a NAND function), and operates in two stages. During the *precharge* phase when $\phi = 0$, the P-type transistor is turned on while the N-type transistor is turned off. This causes the output to be pulled high, while the pull-down network is "disabled", or non-conducting. During the *evaluation* phase when $\phi = 1$, the P-type transistor is turned off while the N-type transistor is on. As a result, the pull-down network is "enabled", or conducting.

There are two types of logic in digital circuits. *Combinational (Boolean) logic* have outputs that are a function solely of the input values, while *sequential logic* have outputs that are functions of the inputs as well as data stored in the circuit.

There are two basic sequencing techniques in digital systems. A *synchronous system* uses a fixed global clock with a period that is longer than the worst-case propagation delay. The major drawbacks of this method are loss of efficiency, clock signal routing, and difficulty in interfacing with the asynchronous world. An *asynchronous* or *self-timed system* have events initiated by completion signals created by other events. Although timing depends on circuit delays, the system is guaranteed to sequence properly by adhering to a protocol. This allows for modular design of the system, and there is no upper limit on the size of the system. However, asynchronous circuits generally require more hardware than their synchronous counterparts, and are difficult to design.

### 1.2.2 Some Computer-Aided Design (CAD) Terms

*Standard cell* or *polycell* refers to an existing library of gate cells used by CAD systems to

build circuits. A *macrocell* is a generated cell, usually implementing more complex logic than standard cells. Gate arrays and other regular structures are often considered macrocells.

A *net* is simply a set of interconnected nodes in a circuit. A *netlist* is a circuit representation (which may be a text file, or some other data format) that describes the circuit in terms of interconnections between components. These components may be transistors, gate cells, or other circuit building blocks. In other words, a netlist is a "list" of "nets" and the objects connected to the nets. An input or output *rail* is a wire coming into or out of a component, and which connects to other components. *Routing* refers to connections between inputs and outputs in a circuit. *Global* or *intercell* routing refers to connections between gates or cells in a circuit, while *internal* or *intracell* routing refers to connections between devices within a cell.

The goal of *circuit synthesis* is to reduce the effort required to design a circuit while at the same time produce circuits of comparable, if not better, quality than conventional hand-designed ones. Synthesis in CAD is used at different levels of the design hierarchy to produce different intermediate circuit representations. We will return to this in Chapter 2.

### 1.2.3 Graph Theory

Many CAD algorithms use graph representations of circuit networks. Here we define some common graph terminology. For a more comprehensive coverage of graph theory and graph algorithms, the reader is referred to [26, 52].

A *graph* consists of a finite set of *vertices* or *nodes*, $V_i$, $(i = 1, 2, ...n)$, and a finite set of *edges* or *arcs*, $e_{i,j}$, joining a pair of vertices $V_i$ and $V_j$ ($e_{i,j}$ is also said to be *incident* on $V_i$ and $V_j$). Two vertices are *adjacent* or *neighbours* if they are joined by an edge. Two edges are adjacent if they are both incident to the same vertex. A vertex is of *degree k* if there are $k$ incident edges.

A *directed graph* is a graph where all the edges have a direction (e.g., an edge can be followed from $V_i$ to $V_j$, but not from $V_j$ to $V_i$). A *directed acyclic graph* (DAG) is a directed graph with no

cycles. Two graphs are *isomorphic* if there is a one-to-one correspondence between the vertices and edges. For directed graphs, the edge directions also correspond. A *tree* is a connected graph with no cycles. A *rooted tree* $T$ is a tree with one distinguishing node, called the *root*, and the remaining nodes are disjoint subtrees $T_1, T_2, ..., T_m$. Nodes with no subtrees are called *leaves*; remaining nodes are called *internal nodes.*

A *Path* $(V_1, V_2, ..., V_n)$ is an ordered sequence of vertices such that from every $V_i$ to $V_{i+1}$ there exists an edge. A graph is *connected* if there is a path between *any* two nodes of the graph. A path where $V_1 = V_n$ is called a *cycle.* An *Euler path*, or an *Eulerian path*, in an undirected and connected graph, is a path that traverses every edge of the graph exactly once.

# Chapter 2

## Discussion of Problem

In Very-Large-Scale-Integration (VLSI) circuit design, Computer-Aided Design (CAD) tools are used to automate parts or all of the design process. Complete design systems, referred to as silicon compilers [19], automate the whole design process by taking, as input, a high level behavioral description, and producing final layout. Individual CAD tools automate part of the design process, producing various intermediate representations. The goal of design automation is to produce near-optimal results of comparable and/or better quality than manually designed circuits. The dividing lines between different levels of design abstraction are sometimes quite fuzzy, but can be generally classified as follows (see Figure 2.1):

1. Behavioral Representation. A behavioral description of a digital system describes the algorithm, flow and control of information in the system. It is usually described in a high-level programming-like language such as VHDL [5].

2. Register Transfer Level (RTL) / Logical Representation. At this level, the system is described in terms of its modules, and their interconnections. The datapath and control parts of the design are separated, and the purely combinational parts and memory storage parts of the design are also separated. The control parts are created through a process called *finite state machine synthesis.*

3. Gate Level Representation. This level of representation specifies the interconnections between logic gates. Here, a "gate" is defined as a physical component that realizes a specific logic function using transistors. For example, a gate level netlist may contain

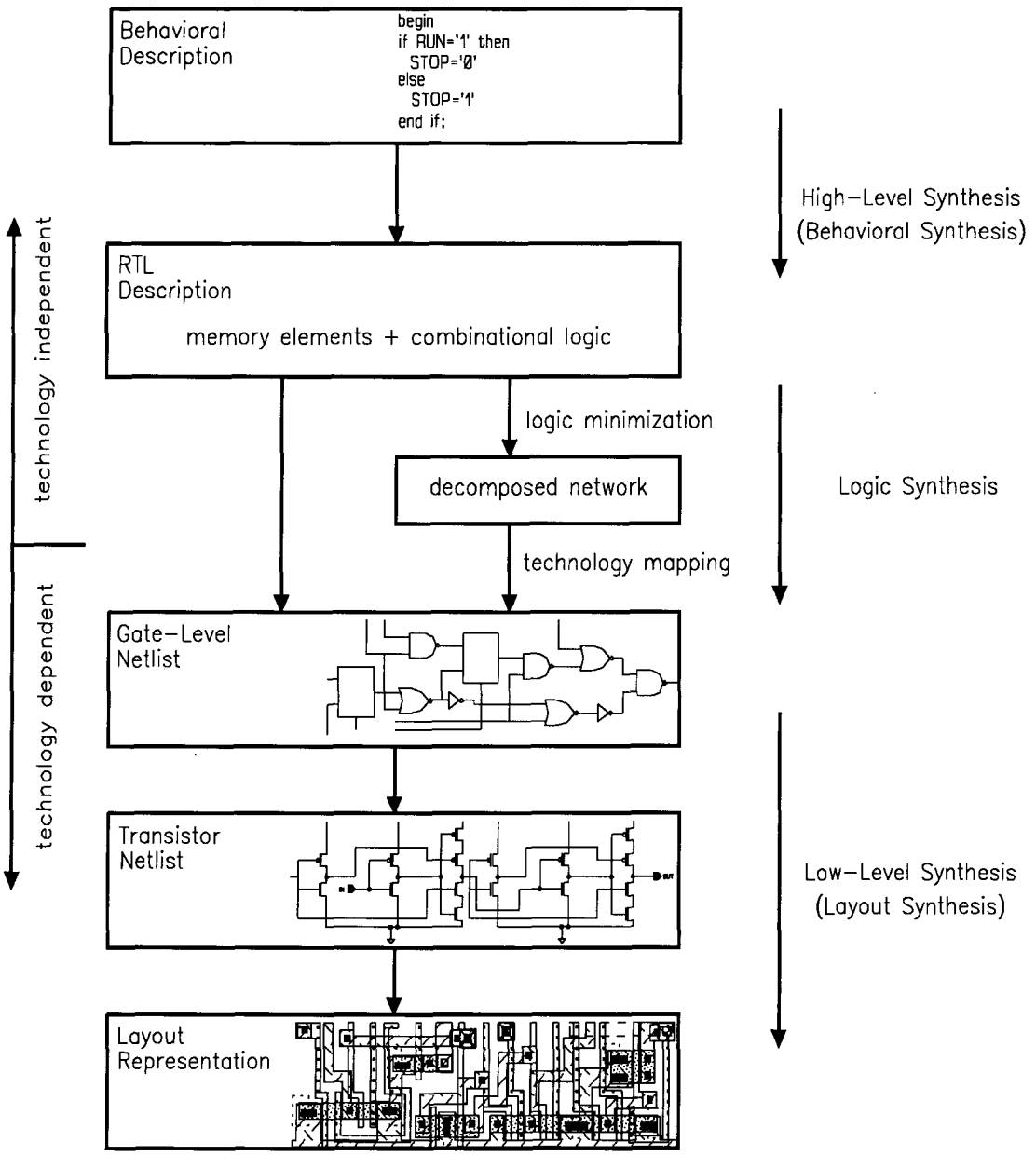



Figure 2.1: Levels of representation during digital design automation.

programmable-devices, standard-cells, or macrocells.

4. Transistor Level Representation. At this level, the complete circuit is described as interconnections between transistor elements.

5. Physical Layout. This is the physical layout representation of the design at the polygon level. The layout of a system includes the layout of the sequential and combinational parts, as well as the routing of metal wires between the components.

The process of going from level 1 to level 2 is generally referred to as *high-level synthesis* or *behavioral synthesis*, while going from level 3 down to layout is considered *low-level synthesis* or *layout synthesis*. Taking the combinational blocks from the RTL level down to the gate level is called *logic synthesis*. In this thesis, we are interested in logic synthesis and low-level synthesis. In most of the current design automation procedures, the storage elements at the RTL level of the design are assumed to be predesigned and ready to use. The task of logic synthesis is to produce a correct final gate level implementation which meets timing and testability constraints, and tries to minimize area. After this step, the combinational blocks are combined with the memory elements to form the overall design.

Logic synthesis can be divided into two approaches: two-level synthesis and multi-level synthesis [12]. Two-level logic synthesis algorithms, generally used for two-level programmable-logic-array's (PLA) implementations, concentrate on minimizing area by minimizing the number of PLA product terms [11]. Multi-level logic synthesis, on the other hand, optimizes random logic, to be implemented with non-array structured gates, such as individual NAND gates or AND-OR-INVERT (AOI) gates. Though some applications are naturally suited to two-level implementations, the multi-level realization generally offers better design freedom and flexibility.

The multi-level logic synthesis task can be considered in two consecutive stages: a technology-independent optimization step, followed by a technology-dependent mapping step. The first stage involves applying algebraic Boolean transformations on the combinational circuit, in an

attempt to minimize overall layout area of the final chip, the critical path delay time, and to maximize circuit testability [13, 11]. The second processing step is referred to as *technology mapping.* It takes an arbitrary combinational logic description, and "maps" that into some design technology dependent description, which may use standard cells or programmable devices or macrocells. In other words, it implements the circuit with specific gate choices. It performs this mapping without significantly altering the structure of the circuit network that was produced after the first optimization step.

The technology mapping step depends on the design method chosen. The use of programmable devices, such as *programmable logic devices* (PLDs) and programmable gate arrays is becoming increasingly popular in the *application-specific-integrated-circuit* (ASIC) industry, and their usage is currently fueling much new research. Examples of some popular architectures are those by A.M.D, Altera, Xilinx and Actel.

If the standard cell design method is used, it is assumed that the library cells have been manually predesigned. This is the most common method of technology mapping. Automatically generated cells are also a design option. Arbitrary logic cells need to be individually synthesized from the logic description level down to cell layout. This process can be referred to as *cell layout synthesis.*

The next two sections in this chapter (Section 2.1 and Section 2.2) will discuss in greater detail the general problem of technology mapping, and in particular, mapping for standard cells, as well as the problem of layout synthesis. Subsequently in Section 2.3, we describe a relatively new design technology called DCVS. DMAP, the system described in Chapter 3 of this thesis, performs technology mapping and layout synthesis for DCVS.

## 2.1 Technology Mapping

In conventional standard cell library design, the problem of technology mapping is one of chosing the right gates from the cell library to implement the given combinational circuit, while optimizing for certain constraints such as area and speed. A good mapper should be able to adapt to changing libraries, and handle different technology-dependent cost functions. The most popular technology mapping technique is the graph covering method. We will begin by describing this technique.

### 2.1.1 Network Representation

Any Boolean network can be represented as a directed acyclic graph (DAG), such as the one in Figure 2.2(a)[1]. Each node is associated with an output variable $y_i$, and a representation of a logic function $f_i$. A directed arc from a node $i$ to a node $j$ exists if the function $f_j$ depends on the variable $y_i$. The leaves of the graph are the network inputs *or primary inputs*, while the roots are the circuit outputs, or *primary outputs*. If there is an arc from node $j$ to node $i$, node $j$ is a *fanin* of node $i$, and node $i$ is a *fanout* of $j$.

An arbitrary DAG can be partitioned into a forest of trees by making each node with fanout greater than one, the root of the new tree, as illustrated in Figure 2.2(b).

### 2.1.2 DAG Matching

It was observed that the standard cell library technology mapping problem is much like the compiler code generation problem, where a set of high level instructions are mapped onto a set of machine instructions for a particular target machine [28]. Each high level code sequence can be represented as a DAG, called a subject graph; each machine instruction for the target machine can also be decomposed into a small DAG, called a pattern graph. Each pattern DAG

[1] Henceforth, the terms *graph*, *network*, and *circuit* will be used interchangeably.

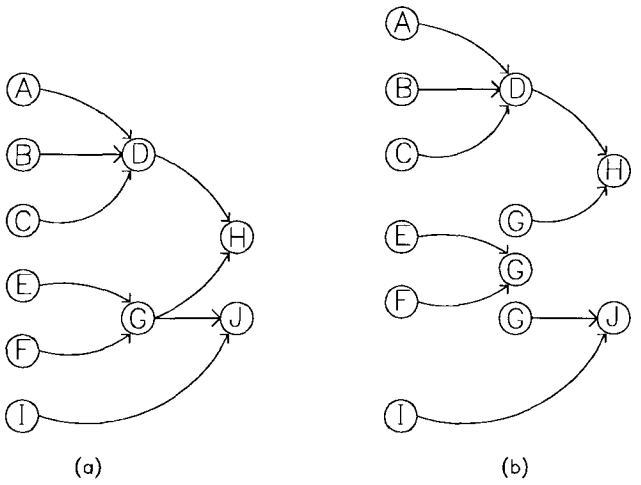



Figure 2.2: (a) DAG (b) partitioned DAG (forest of trees)

has an associated execution time cost. The optimum code is generated by finding an optimum mapping to cover the large DAG with the little pattern DAGs [6].

### 2.1.3 DAG Matching Applied to Technology Mapping

Similarly in technology mapping, each combinational circuit is decomposed into DAGs of standard form (i.e., the nodes of the graph are base functions such as NANDs and INVERTERs, or NORs and INVERTERs). If there is any node with fanout greater than one, partitioning is performed by making that node the root of a new DAG. The resulting DAGs form the set of subject graphs. Each gate in the standard cell library is represented by several small-sized DAGs decomposed from the logic function of that gate. Since there are many possible trees for one gate, only the non-isomorphic patterns are used. For example, Figure 2.3 shows two different DAG patterns for a 4-input NAND gate using 2-input NAND gates and INVERTERs. The union of the set of DAGs for each library function is used as the set of pattern graphs.

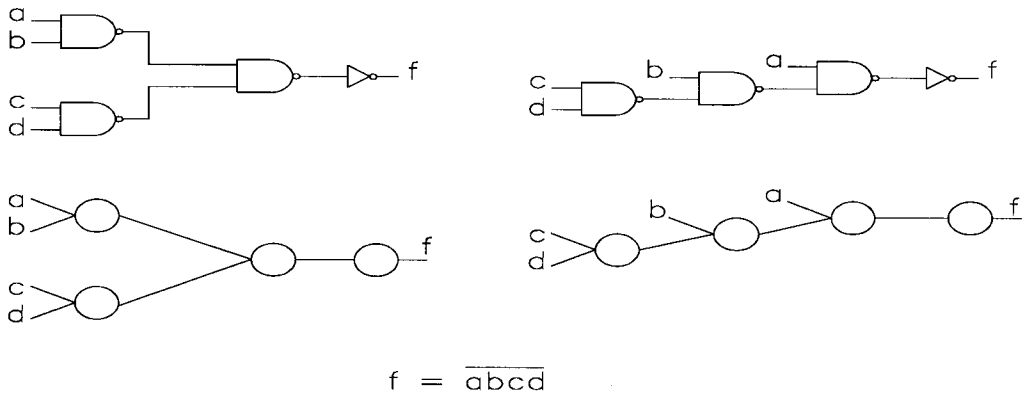



Figure 2.3: Non-isomorphic DAG representations for the function $f = \overline{abcd}$.

The technology mapping problem boils down to matching every branch in the subject graph with one of the tree patterns from the standard cell library. The problem of optimal code generation, where a subject DAG is matched by pattern DAGs, is an NP-complete problem [6]. However, in technology mapping where the DAGs are restricted to trees, the problem becomes linear in both the size of the subject tree and pattern forest. The choice of patterns depends on the technology mapper's cost evaluation, which may consider area, speed, and other parameters. Each pattern graph has an associated cost function. Achieving a complete *cover* or *map* means that every node in the subject graph must be contained in one or more of the pattern graphs, and that each input of a pattern graph is the output of another pattern graph. DAGON is a technology mapper based on DAG matching developed by Keutzer [28] at AT&T. This approach was later extended at Berkeley in the MIS logic synthesis system [10].

To illustrate the concept of graph matching, consider Figure 2.4 where technology mapping is performed on a full-adder circuit using MIS. Figure 2.4(a) shows one decomposition for a number of basic gates into base form tree patterns. Recall that there are several possible decompositions for each gate, and all of them are considered as patterns in matching. Figure 2.4(b)

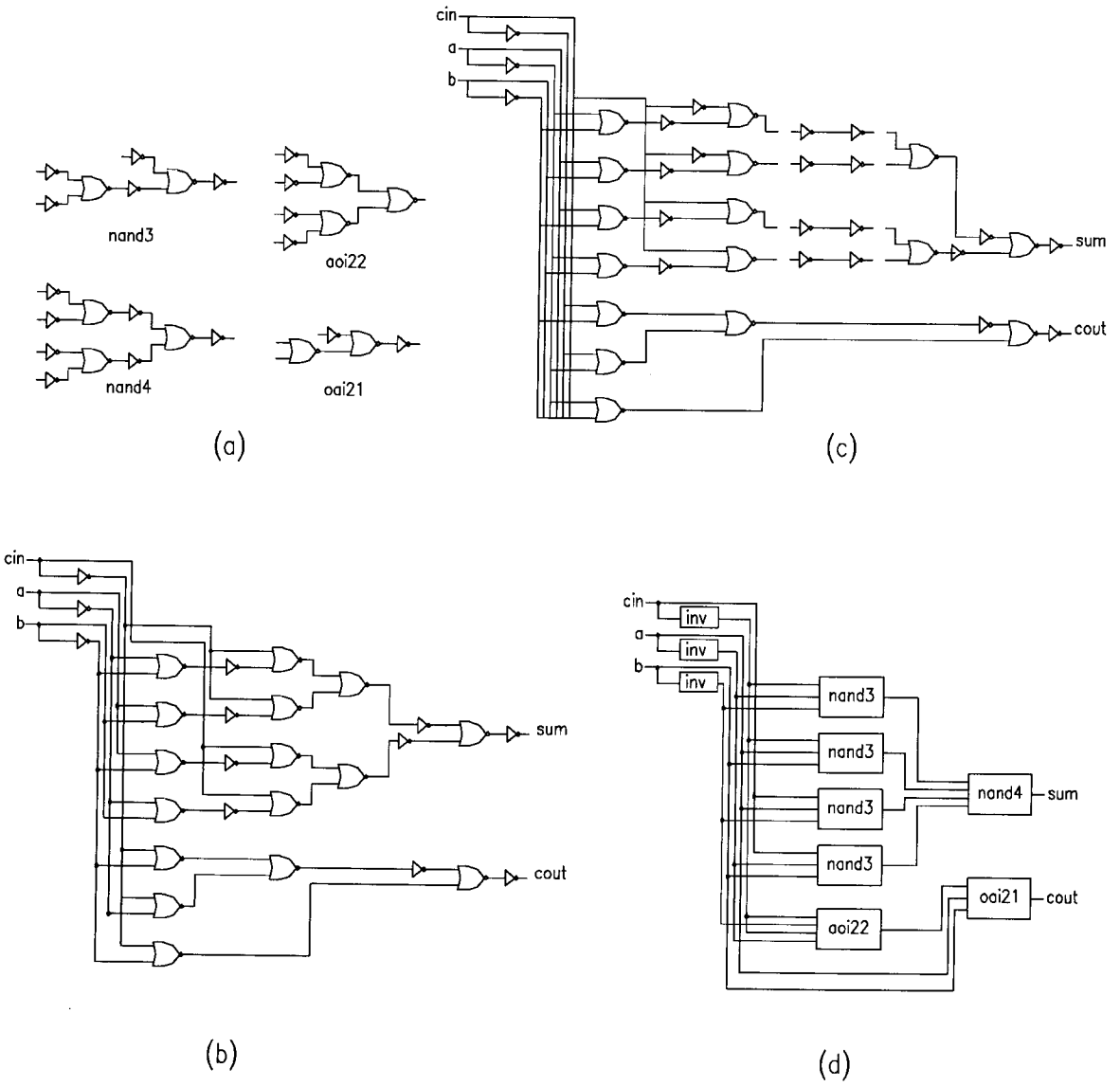



Figure 2.4: Illustration of tree matching in technology mapping.
(a) A possible tree pattern for some library gates.
(b) Decomposed adder circuit before mapping.
(c) Decomposed adder circuit with INVERTERs inserted.
(d) Adder circuit after mapping with MIS-II

shows the decomposed full-adder circuit. Technology mapping in MIS differs from DAGON primarily by the fact that MIS inserts INVERTERs in the circuit (without affecting functionality) to enhance the number of successful pattern "matches". From Figure 2.4(c), we can see that once INVERTERs are added to the circuit, a number of familiar patterns from Figure 2.4(a) can be identified in the circuit. Finally, Figure 2.4(d) shows the result of the graph matching.

After technology mapping, the structure of the circuit is fixed, and optimizations can no longer be performed on the combinational circuit as a whole. Any further optimizations would have to occur inside the gate level. For standard cells, the layout is predesigned and fixed, and nothing more can be done with the layout before global placement and routing. For generated cells, the gate's internals can be synthesized to meet still more optimization parameters. Indeed, this is one of the advantages, and objectives, of cell layout synthesis.

## 2.2 Cell Layout Synthesis

Laying out a circuit at the physical level is a tedious and time consuming process. It becomes very costly to do the layout portion of the design (draw, check, and correct layout) by hand. Therefore, it makes sense to use either layout cells from a pre-drawn standard cell library which are usually designed to be very compact and efficient, or to use off-the-shelf programmable devices. As design methods and semiconductor technologies rapidly evolve, it becomes very costly to maintain a standard cell library. This problem can be significantly alleviated if a layout synthesis program is used to generate cell layouts of reasonable area and performance. This way, the technology-dependent factors can easily be modified inside the synthesis program. Whether to generate basic standard cell libraries, or more complex cells, layout synthesis is of considerable interest. The automatic design and generation of such polycells include two main steps:

1. Determining the transistor network (i.e., transistor netlist).

2. Placing the transistors to meet the design rule specifications.

### 2.2.1 Static CMOS Cell Layout Synthesis

Most of the work done in layout synthesis has been for complementary static CMOS design. There is little published work on layout synthesis for dynamic CMOS design. In static CMOS, the first step of obtaining the transistor netlist is relatively straightforward for smaller functions, since the logic function of the cell can easily be translated to a series-parallel connection of transistors, where the pull-up and pull-down networks are duals of each other.

Significant work has been done in cell synthesis for static CMOS standard cells to perform the second step above: given a transistor netlist, lay out the network to meet design rule requirements.

### 2.2.2 Transistor Placement for Static CMOS

Uehara and vanCleemput [58] introduced the automatic generation of cell layout in one-dimensional transistor arrays. Subsequent work by other people has built on this concept. Given the transistor netlist, the CMOS static circuit is converted into two graphs, one for the pull-up (P-type transistor region), and one for the pull-down (N-type transistor region). The vertices correspond to the transistor connections (i.e., **SOURCE/DRAIN**), and the edges correspond to the transistors. Figure 2.5(a) shows an example of a CMOS logic gate $f = (AB)(C + D)$, and Figure 2.5(b) shows the two corresponding graph representations for the pull-up and pull-down networks.

Uehara and vanCleemput's algorithm for finding the layout path is essentially one of finding Euler paths in the pull-down and pull-up graphs, and then laying out the transistors in two strips according to the Euler path labeling. An algorithm for finding an Euler path is given in Appendix A. Sometimes, a graph will not have an Euler path, in which case imaginary

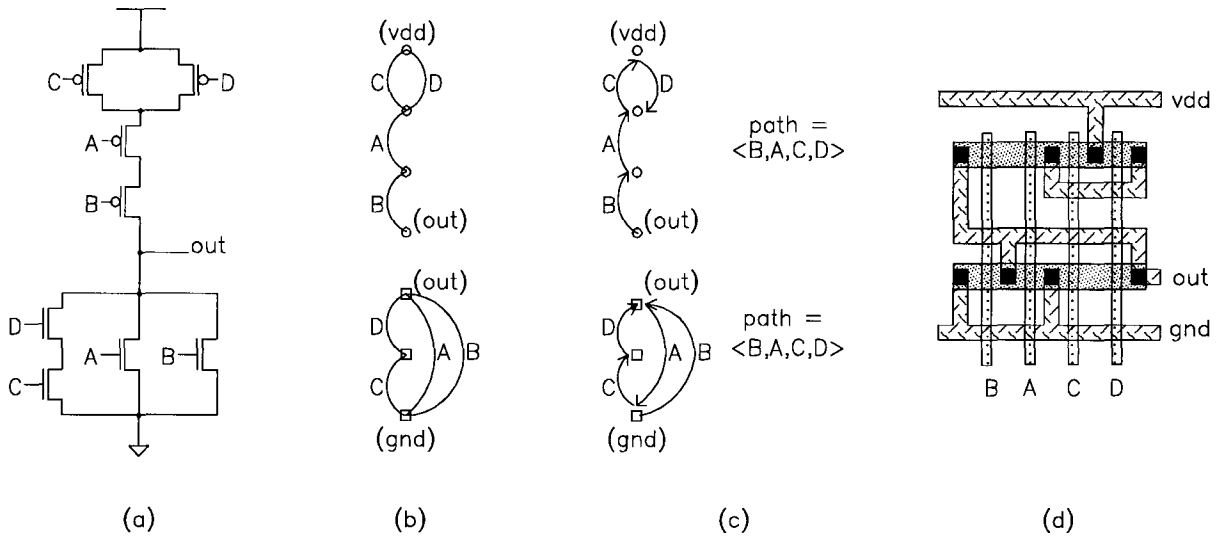



Figure 2.5:
Finding the Euler path and transistor placement:
(a) Static CMOS gate for the function $(AB) * (C + D)$.
(b) Graph corresponding to pull-up and pull-down logic tree.
(c) Euler paths with the same labeling for the pull-up and pull-down graphs.
(d) Corresponding layout of the gate with two transistor strips.

edges called *pseudo-edges* are added to the graph. The Uehara and vanCleemput method for transistor placement can be summarized as follows.

1. Find all the Euler paths that cover the two disjoint pull-up and pull-down graphs.

   The objective for finding this path is to allow consecutive transistors in the path to share a diffusion region. In other words, the source-drain connections of adjacent transistors on the path are made by abutment in diffusion. Figure 2.5(c) shows an Euler path for the pull-up graph and one for the pull-down graph.

2. Find a pull-up path and a pull-down path such that the labeling of the two paths are the same.

   Using two paths with the same labeling allows the transistor inputs to be aligned, so that the inputs can be connected with a vertical polysilicon input rail without any horizontal routing. This saves area and reduces internal routing complexity. Figure 2.5(d) shows the corresponding layout of our CMOS circuit example.

3. If the paths in Step 2 are not found, add pseudo-edges to the graph(s) so that the paths in Step 2 can be found.

   When pseudo-edges are added, the resulting Euler path is equivalent to two or more individual Euler paths joined by edge separations. When laid out, these separations become diffusion gaps. If the pull-up and pull-down transistor arrays do not have aligning pseudo-edges, one of the transistor strips will need to have a slightly wider diffusion connection to allow for alignment. Figure 2.6(a) shows a graph where a pseudo-edge needs to be added to obtain an Euler path. Figure 2.6(b) shows the added pseudo edge from node 5 to node 3 and the resulting Euler path.

An improvement of Uehara and vanCleemput's non-optimal heuristic was described in [33], where an optimal, non-exhaustive, method of minimizing the layout area of complementary

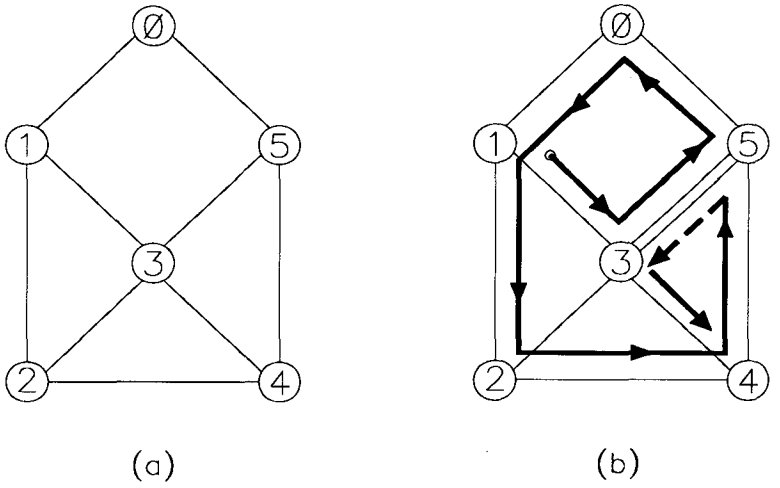



Figure 2.6: Adding pseudo-edge to find an Euler path.

series-parallel CMOS cells in the standard cell style was developed. A number of other algorithms that extended the basic Uehara algorithm to minimize diffusion or reduce internal wiring have been described in [9, 25, 44, 47, 61]. Further generalization of the Uehara and vanCleemput method was presented in [8], where the layout generation is driven by a combination of optimization criteria and composition constraints to control various aspects of the layout, such as wire length, metal utilization, and diffusion breaks.

In addition to this standard-cell transistor-row style, other layout styles have been studied as well, such as those in [49, 54, 50] and gate matrix styles in [62, 63, 55]. A common feature for all of them, however, is that the symmetry between the P-type transistor and N-type transistor networks in full static CMOS design is considered. For dynamic CMOS, mainly the N-type transistor region is of concern.

## 2.3 Differential Cascode Voltage Switch Logic

It is desirable to build VLSI circuits using devices of small power consumption, small area and delay. It was with these goals in mind that the logic design family, *Cascode Voltage Logic Switch* (CVSL) [24], surfaced as an attractive design technology. The logic is achieved by cascoding differential pairs of N-type transistors to form a stacked pull-down tree capable of realizing complex Boolean functions within a single gate delay. Its single-ended form, *Single-Ended Cascode Voltage Switch* (SCVS) produces one output function from a binary pull-down tree. On the other hand, its differential form, *Differential Cascode Voltage Switch* (DCVS)[2], produces both the true and complemented form of the output i.e., each cell computes both $q$ and the complement $\overline{q}$. DCVS exists in both static and clocked (or dynamic) styles as shown in Figure 2.7. In this thesis, we consider the dynamic version. Before proceeding to describe the operation of the dynamic DCVS cell, we briefly survey some previous dynamic CMOS techniques and some of their shortcomings.

### 2.3.1 Other Dynamic CMOS Technologies

First, let us consider the generic dynamic circuit of Figure 1.2(b). The initial precharge or "set-up" state of the pull-down network must be nonconducting. To achieve this, a common design practice is to make *all* the inputs low to ensure that there is no pull-down path. During evaluation, when the pull-down network conducts, the output of the network will be pulled low to ground. This implies that the output of the network must have initially been high for this transition to occur. However, if this output is connected to the input of another dynamic gate in a cascading configuration, the required "set-up" condition for the next gate cannot be achieved.

Krambeck, Lee, and Law [30] solved this dilemma by putting a static INVERTER gate at the output of the pull-down network. This is called DOMINO logic (Figure 2.8). The output $f$ of

[2]DCVS is sometimes referred to as Differential Cascode Voltage Switch Logic (DCVSL) in the literature.

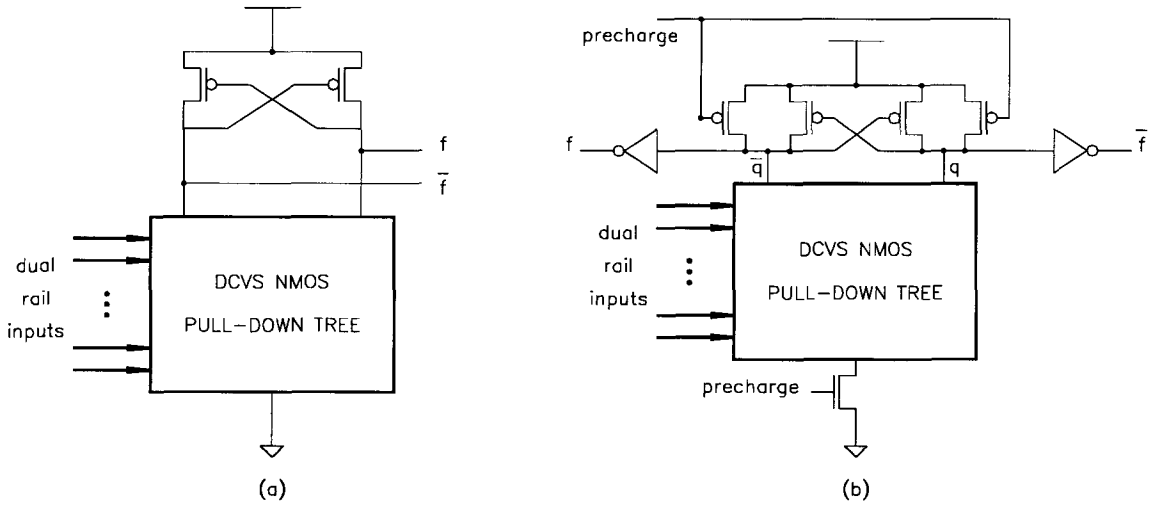



Figure 2.7: Differential Cascode Voltage Switch: (a) Static (b) Dynamic.

the INVERTER is the real output of the gate, and is precharged low. When this output is fed into subsequent DOMINO gates, this initial precharged value turns the transistors off. During evaluation when the pull-down network is conducting, $f$ goes high.

The major drawback of DOMINO is that the gate is non-inverting. DOMINO is, by itself, an incomplete logic. The pull-down network realizes the complement of some function, while the static INVERTER forces the gate output to be the uncomplemented function. This makes implementing arbitrary combinational logic awkward with DOMINO.

An alternative design which provides more logic flexibility is NORA [22], where nMOS pull-down gates and pMOS pull-up gates alternate. Figure 2.9 shows a N-type gate feeding into a P-type gate, which feeds into an N-type gate. Since the P-type gate is disabled with inverted inputs, the first N-type gate output need not be inverted. The P-type gate is precharged when $\overline{\phi}$ is high, so that the network is disabled, and the output is connected to ground.

The major drawback of NORA, however, is that the P-type gates tend to switch slower than

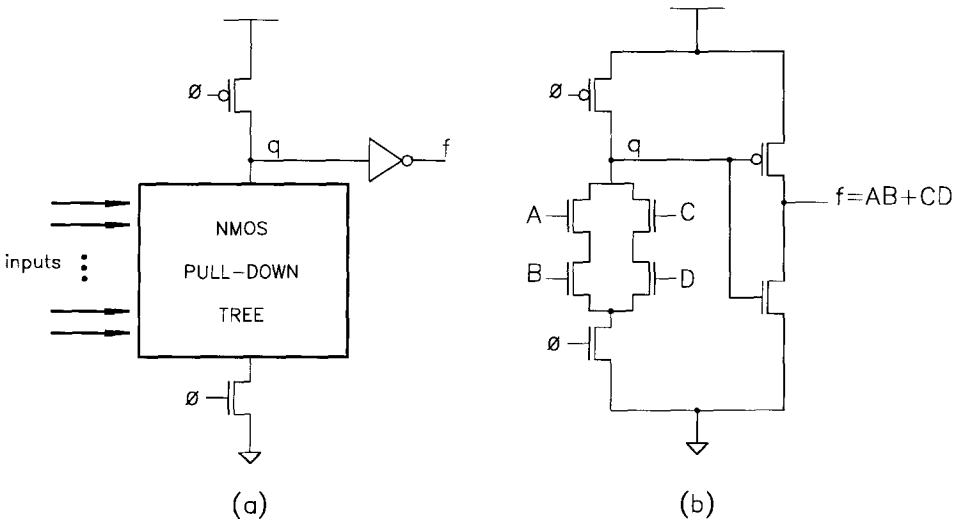



Figure 2.8: Dynamic DOMINO design:
(a) Generic DOMINO gate.
(b) Detailed schematic of a DOMINO gate for the function ($f = AB + CD$).

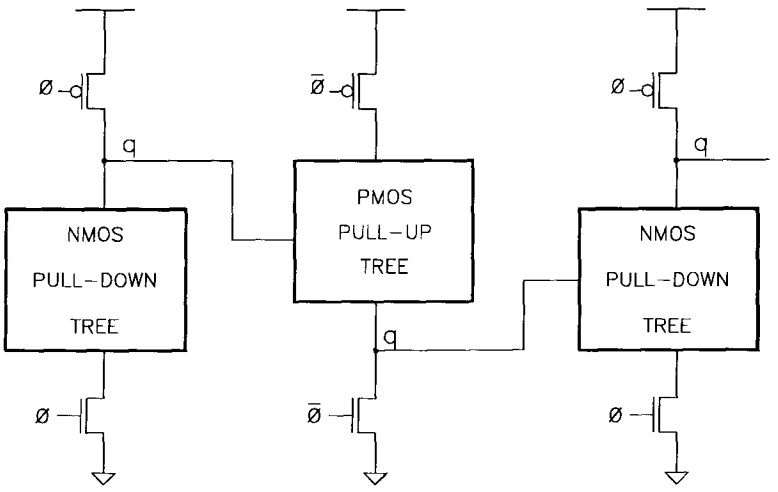



Figure 2.9: Dynamic NORA design.

the N-type gates. This is due to lower electron mobility in the p-well. Furthermore, the need to alternate the two types of gates makes NORA circuits very awkward to design. In comparison, DCVS circuits have all the advantages of fast-switching dynamic CMOS, while also being easy to design since every gate realizes both the complemented and uncomplemented form of the logic function.

### 2.3.2 DCVS Operation

A basic DCVS cell consists of: a fixed pull-up network, a function dependent pull-down network, a single pull-down transistor controlled by the precharge signal, and two output inverters. These components operate together to form a very powerful circuit building block. DCVS takes in true and complemented input control signals. Figure 2.10 shows a 3-input clocked DCVS **AND/NAND** gate. Its operation cycle is as follows.

First, the cell is precharged, i.e., the precharge signal is set low. Consequently, the outputs $f$ and $\overline{f}$ of the DCVS cell will both be pulled low. When the precharge signal goes high, depending on the values on the inputs of the function specific pull-down tree, one of the nodes $q$ and $\overline{q}$ will be pulled down, and thus either $f$ or $\overline{f}$ will go high. If the pull-down tree is designed properly, i.e., the path function for connecting $q$ to the final precharge transistor is the complement of the path function for connecting $\overline{q}$ to the same transistor, then it is easy to verify that for any DCVS cell, at most one of $f$ and $\overline{f}$ will ever be high at the same time.

In conventional dynamic design, the gates must be clocked (or precharged) at a minimum operating frequency to maintain the output values. There are two extra P-type pull-up transistors in a cascode configuration acting as "staticizers". When either $q$ or $\overline{q}$ is realized, the pair of feedback transistors cause the outputs to immediately stabilize. Hence, there is no lower limit on the frequency of the precharge signal in a DCVS cell. This is important for reliability reasons as well as when using the logic in more unconventional circuit designs. For example, very efficient self-timed circuits can be designed using DCVS logic. In such designs, the

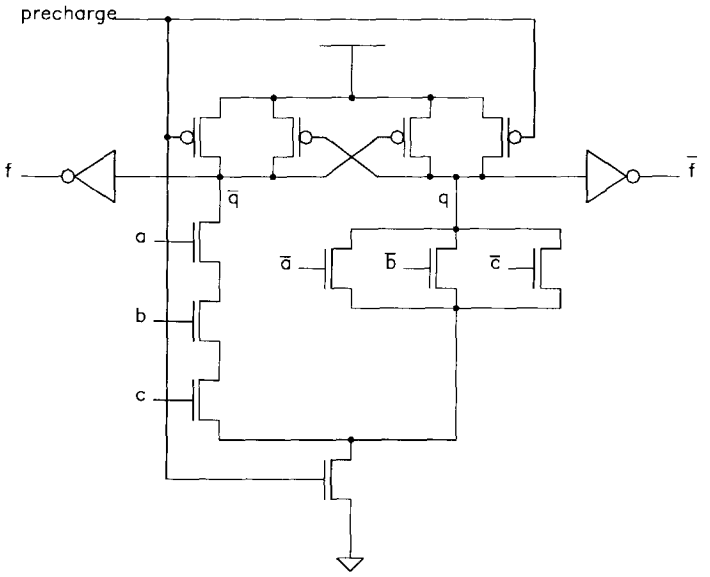



Figure 2.10: 3-Input DCVS AND/NAND gate.

precharge/evaluation cycle is initiated by the completion of some previous circuit component, whose timing may be impossible to predict.

### 2.3.3 DCVS Properties

Early interest in dynamic logic arose from a naive view of circuit layout tradeoffs. It was thought that due to the absence of a pull-up network, a dynamic logic layout, with roughly half the transistors of static logic, should be more compact than its static cousin. This argument was misleading for a couple of reasons. First, most of the area consumption in VLSI layout is due to wiring, not transistors. Secondly, prevention of latchup in CMOS technologies requires a comparatively large separation between the pull-up and pull-down regions. This in turn enforces a very confined topology on any CMOS cell, which in turn prevents either designers or layout tools from taking advantage of the relatively empty pull-up region presented by dynamic logic technologies.

Subsequently, research in dynamic logic has concentrated on the perceived advantages in power consumption. It was thought that dynamic logic might save power due to two central effects:

- In static logic, as each gate switches, a short connection between power and ground is established, with only two weak resistors in the path; this short effect is responsible for a fair amount of static logic's power consumption.
- Power is consumed each time a gate switches; in static logic, each gate switches potentially many times during a single evaluation phase. In contrast, a dynamic gate switches only once.

However, detailed performance comparisons of DCVS with other design methods performed by Chu and Pulfrey [17] revealed that DCVS power gain is unfounded. For example, when compared to static design, it was found that DCVS gains in terms of input capacitance and device count, but looses in regards to power dissipation. In comparison to other dynamic CMOS design styles (e.g. NORA and DOMINO), DCVS is faster, at the expense of increased device count and power dissipation.

Recent experiments by Meng[42] further supported this fact. Dynamic logic, due to the restoring inverter at the output of this gate, *does* short. Further, power is consumed as each capacitor in the circuit is charged during the precharge phase, and as half of these nodes are discharged during evaluation. Finally, though each gate makes a transition only once during evaluation, there are many *more* switches than in typical static networks. In fact in DCVS, half of all the gates switch during partial evaluation.

Despite the illusory advantages of power and size, it is in the area of speed and testability where dynamic logic shows proven advantages over its static counterpart. More recently, research in dynamic logics has concentrated on advantages in the fields of asynchronous design, timing verification, testability for various classes of fault, and hazard freedom.

The most obvious advantage is that dynamic logic, unlike static logic, does not glitch. A *hazard* is a momentary transition of the output that should have remained stable. A hazard can become fatal if it causes a transistion to an improper system state. A *glitch* is a hazard that is visible, such as a 0-1-0 or a 1-0-1 transition. In fact, in [37, 39], it was shown that the two essential properties that a circuit must possess in order to remain hazard-free are:

1. that it be internally noninverting; and

2. that each node be precharged to a specific value before evaluation.

Dynamic CMOS logic is currently the only logic family possessing these qualities. In dynamic gates, the outputs are affirmatively driven, and there can only be at most *one* output transistion after a precharge. For example, if the outputs are precharged to 0, then they will either stay at 0, or be driven to 1 during the evaluation stage. Freedom from glitches has some nice consequences in circuit testing.

A *delay fault* is a fault which slows down a circuit and impairs the clocked operation of the circuit. Typically, a delay-fault *test* consists of a set-up vector, $V_1$, which is first applied to the circuit, and a second vector, $V_2$, to see if the outputs change. A path delay-fault test is said to be *robust* if it detects a fault in that path regardless of whether or not there are faults in other paths. Robust delay-fault testing for integrated circuits is regarded as both difficult and extremely demanding.

Since dynamic circuits have all the nodes initialized during precharge, it can be seen intuitively that only the second test vector, $V_2$ is needed to test a path delay-fault. Together with the hazard-free property of dynamic circuits, McGeer showed that the condition for robust path delay fault testability (RPDFT) on dynamic circuits was much less demanding than on static circuits [34]. In particular, it was demonstrated that every minimal sum-of-products form was robustly path-delay fault testable, and that every multi-level circuit could be made robustly path-delay fault testable by transformation. In the era of designing for testability, RPDFT is

a very desirable circuit property.

Perhaps the simplest and most basic fault model for Boolean networks is the *stuck-at fault.* A wire is said to have a *stuck-at-0* fault if the output of the wire terminal is 0 regardless of the wire input. Similarly, a wire has a *stuck-at-1* fault if the output of the wire terminal is 1 regardless of the wire input. $\tau$-irredundant faults are a newly-identified class of stuck-at-faults where the faults do not change the function of the circuit, but instead act to slow down the computation of the circuit [40]. In [38, 39], McGeer further demonstrated that one could obtain a precise upper bound on the true delay of a dynamic logic network even in the presence of uncertainties in the individual gate delays. This can be shown to be very difficult for static logic networks[35, 36, 39]. This precision in delay estimate permits us to test for the $\tau$-irredundant faults on dynamic logic networks.

In fact, a number of other timing related questions can be more precisely addressed for dynamic circuits [40, 38]. This, plus the hazard-free property of DCVS logic and its dual-rail nature, make DCVS particularly attractive for asynchronous applications.

### 2.3.4 DCVS Applications

The 1989 Turing award lecture by I.E. Sutherland on his work on "micropipelines" [56] is an excellent example of the growing importance and potential of asynchronous circuits. A micropipeline can simply be viewed as a series of fast processing asynchronous computations, such as multiplication or division, under the control of some small logic blocks, as shown in Figure 2.11. The control logic generates a precharge signal, $P$, after the completion signal, $C$, from the computational block is activated.

Several designs have been suggested for the control block [56, 59, 48]. The control block uses a handshaking protocol, such as the H-Protocol[48], to interface with other logic block segments of the pipeline. When one computational block finishes its job, it sends an "I'm done" signal ($C$) to the control block. The control block uses this signal, together with control

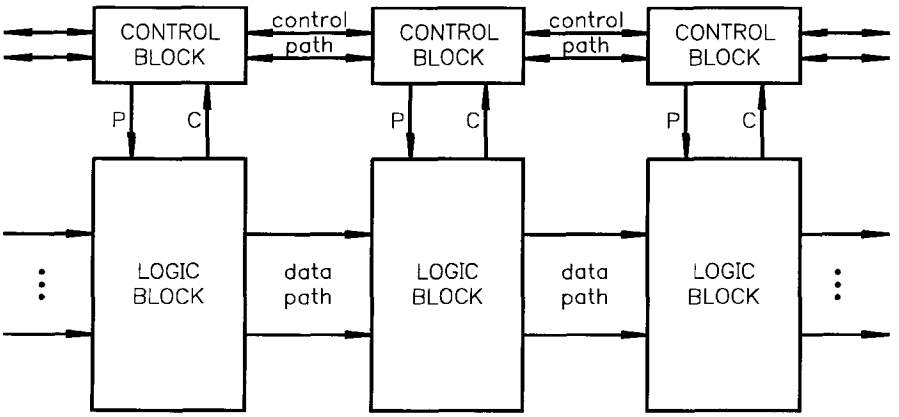



Figure 2.11: Generic asynchronous pipeline.

signals from the previous and next stage computational blocks to determine the current state of this pipeline section. Eventually, it returns a "go ahead" signal ($P$) to the logic block when it is safe for the next round of computation to begin. A completion circuit that generates $C$ can be easily implemented in DCVS by taking advantage of the gate's differential outputs, as shown in Figure 2.12. The *completion* line goes high when both outputs are stable, and remains low during precharge. A DCVS gate only produces three possible output states for $(f, \overline{f})$: (0,0), (0,1), and (1,0). Note that the pair of staticizing transistors in the DCVS gate ensures that the state (1,1) is never achieved.

DCVS naturally lends itself to very efficient self-timed circuits. Since asynchronous logic blocks can compute as fast as the signals propagate through the circuit, micropipeline designs have superior speed advantages over centrally-clocked systems. Meng and others have designed a series of circuits, such as multipliers and arithmetic-logic-units, using DCVS signal completion. These circuits have been demonstrated to operate at speeds comparable to their synchronous

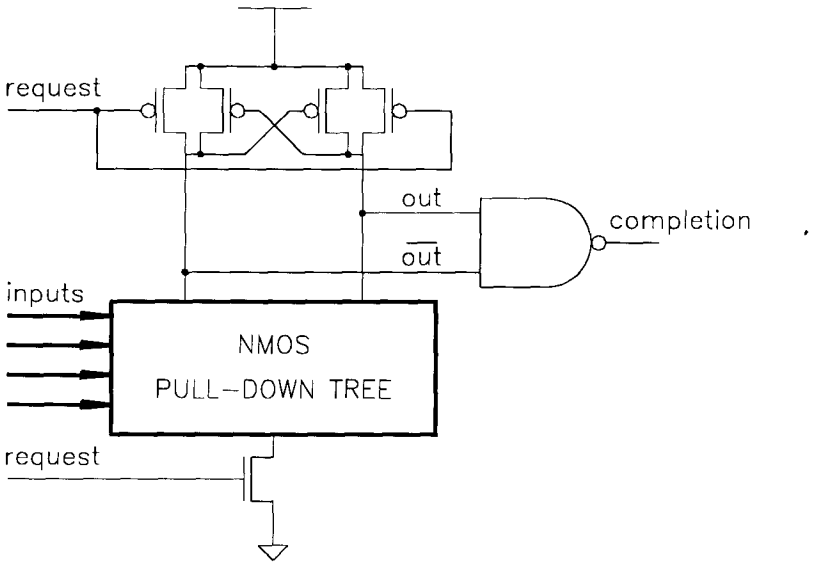



Figure 2.12: DCVS completion signal generation.

counterparts [41, 43, 27]. More recently, Williams has devised a self-timed IEEE floating-point division ring which operates in 100 nanoseconds using this technique[60]. However, in these works the routing of the DCVS differential lines required significant active area penalty. Furthermore, most of them were built from basic building blocks, such as adders, which were hand designed. Automatic design synthesis for DCVS is needed for building micropipelines for other interesting applications. In particular, we need computer tools to help build any arbitrary combinational DCVS logic blocks.

## 2.4 Technology Mapping and Layout Synthesis for DCVS

Although DCVS technology has generated much interest due to its attractive features, the problem of technology mapping for DCVS has not been thoroughly studied. Implementing a DCVS circuit is challenging due to the fact that the logic is complementary. Consequently, all signals in the circuit must be duplicated. Intuitively, this leads to twice the number of internal

nets and connections, and causes global routing area to increase by a factor of two or more. Since wiring usually take up more area on a chip than the transistors themselves, this would appear to make DCVS logic highly area-inefficient. However, this issue is not that clear-cut. It is possible that the larger area could be compensated for by fewer cells of larger complexity.

There are two possible avenues for DCVS design. We can either design a very large cell library containing very many complex cells and use the standard tree-matching algorithm for technology mapping, or we can develop a synthesis system that can generate an efficient layout for an arbitrary DCVS cell given only the Boolean function the cell is to implement.

Since DCVS gates can realize complex Boolean functions efficiently, standard cell libraries cannot fully take advantage of this feature. Using a standard mapper with a very large library has several shortcomings. First of all, since a complete cell library for all, say 5-input gates, would contain somewhere in the range of a billion cells, we would clearly have to compromise and use an incomplete cell library. Hence, the mapper would often be forced to use smaller cells than absolutely necessary, simply because the cell needed was not in the library. On a more practical level, it is very time-consuming and costly to design and maintain such a large cell library. Finally, experimental results [18, 29] have indicated that tree matching-based mappers can be computationally expensive in matching large cells. In conclusion, using cell libraries of limited size and functionality is inappropriate for DCVS mapping.

The second approach of cell generation is more appropriate for DCVS. The goal then, is to generate fewer cells of larger complexity. This approach will maximize the functionality of each cell and optimize transistor usage, while minimizing the number of cells that need to be routed.

### 2.4.1 The Problem

Given a circuit description in terms of a set of Boolean equations, we are interested in its final realization using DCVS cells as building blocks. In general then, there are four steps required

in transforming the general combinational circuit (assuming some technology-independent optimization has been done) into a finished DCVS circuit layout:

1. Decomposing the original large circuit into a set of smaller Boolean functions.
2. Designing the pull-down tree for the DCVS cell corresponding to each such Boolean function.
3. Laying out each DCVS cell.
4. Placing the cells and connecting them together.

### 2.4.2 Previous Work in DCVS Synthesis

Although technology mapping and automatic layout synthesis have been studied in general, no synthesis system has been developed specifically tailored to DCVS logic. In particular, we are not aware of any system that can take an arbitrary combinational circuit and decompose it into a collection of "as-large-as-possible" DCVS cells and then automatically generate the needed cells.

Various automated layout generation tools have been developed over the years. However, these tools have mostly been tailored towards generating layout for standard CMOS cells, i.e., cells that have both function dependent pull-up and pull-down transistors. Consequently, the techniques used in these systems are not directly applicable to DCVS layout generation. Furthermore, DCVS cells are usually significantly more complex than standard CMOS cells both in terms of internal connectivity and in terms of number of inputs.

Some previous work has been done towards solving the second step of obtaining compact DCVS pull-down trees at the transistor netlist level given a Boolean function [16]. Also, the close relation between binary decision diagrams and the pull-down tree of DCVS cells was pointed out earlier [13].

ACORN is an automated physical design system developed at IBM by Yoffa and Hauge for DCVS macro design that does the second and third steps of the problem [65, 64], namely, designing the pull-down trees and laying them out. The system uses the brickwall approach for the physical layout of the DCVS trees, where a two-dimensional tree array style is used. First, it performs tree placement, followed by local customizations to achieve variable alignments to improve intercell wirability. The goal is to obtain high density DCVS layout by reducing congestion caused by the complex internal tree wiring between transistors. ACORN exploits the fact that for any Boolean function, there are several distinct tree implementations, and that could lead to different array arrangements. Figure 2.13 shows the function ($f = AB + C$) implemented by six different trees. Each square is a transistor controlled by the input labeled inside the square.

The program chooses the trees to achieve common input alignments between all pairs of adjacent trees in each of the input rails. This rail optimization allows for signal bussing, hence reduces metal wiring between blocks. To illustrate the concept of local optimization, consider two neighbouring DCVS trees as shown in Figure 2.14(a). If the two circuits are wired as is, the connections would be as in Figure 2.14(b). However, by rearranging the transistors of one of the trees Figure 2.14(c), the wiring between trees is simplified Figure 2.14(d).

In [53], a better DCVS placement technique is presented, which builds upon the algorithms in ACORN to achieve a layout system that is provably good. The main drawback with these systems is that they assume that the initial decomposition step has been performed, i.e., they assume that the user already has partitioned the circuit into suitable "chunks". Since the size and characteristics of the final DCVS cell are very difficult to estimate a priori, the partitioning will likely be non-optimal.

Placement and routing is a significant concern in DCVS design because of the differential input and outputs. In the brickwall layout approach, no channels are used for the routing. Instead, all wires pass above the macro regions. ACORN performs this with a wiring tool also

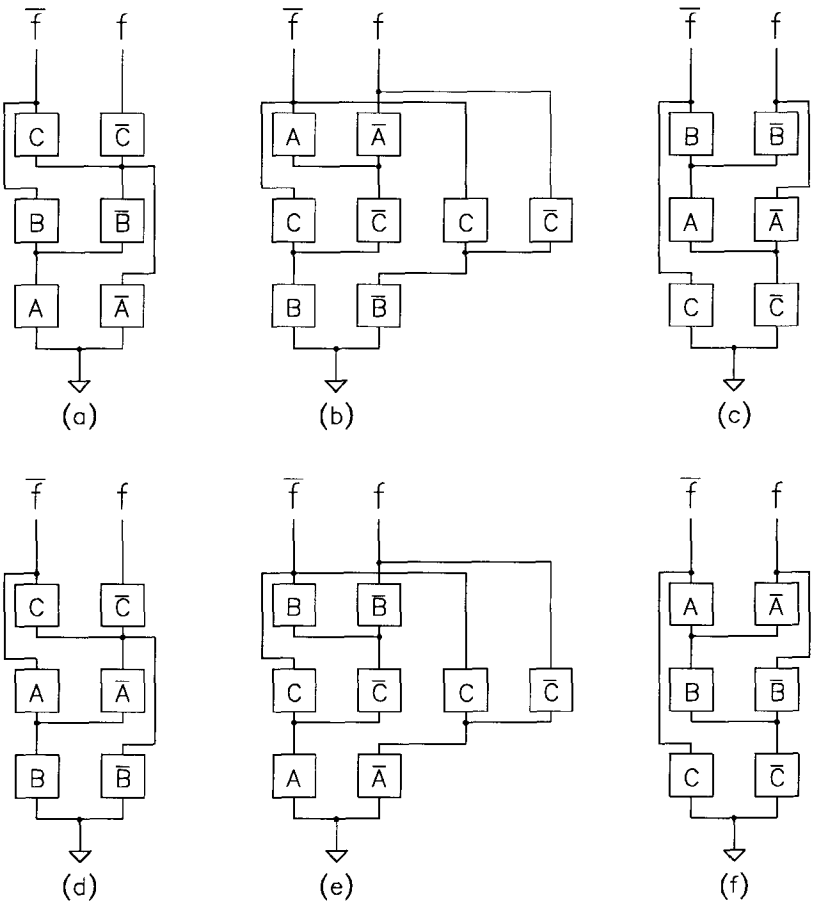



Figure 2.13: Six different DCVS trees for the function ($f = AB + C$).

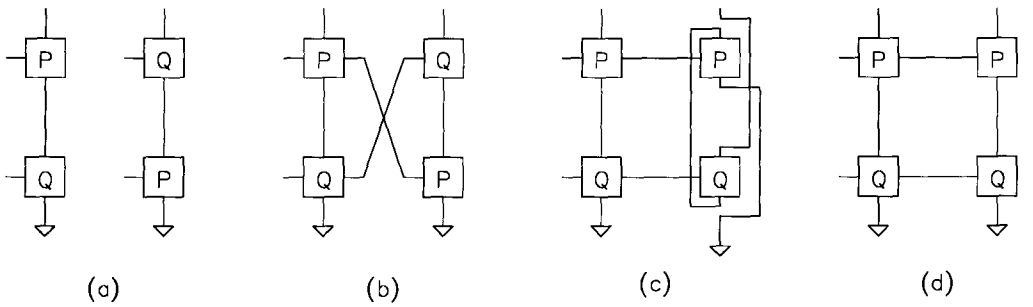



Figure 2.14: Illustration of local optimization in ACORN:
(a) Two adjacent DCVS trees after synthesis.
(b) Trees wired together.
(c) Trees after transistor rearrangement.
(d) Trees re-wired after local optimization.

developed at IBM [20]. On the other hand, in DCVS polycell designs, routing channels are allocated. The placement and routing stage is no different for DCVS logic than for most other CMOS standard cell approaches. The next chapter presents a system which incorporates steps 1 to 4 of the DCVS problem, and which uses a polycell routing method.

## Chapter 3

## DMAP: A DCVS Technology Mapping and Layout Synthesis System

DMAP is a technology mapping and layout synthesis system for DCVS. Figure 3.1 shows the flow diagram for the DMAP system. This system consists of two major modules: the clustering module and the cell generation module. DMAP employs the CADENCE Design System [3] to perform placement and routing.

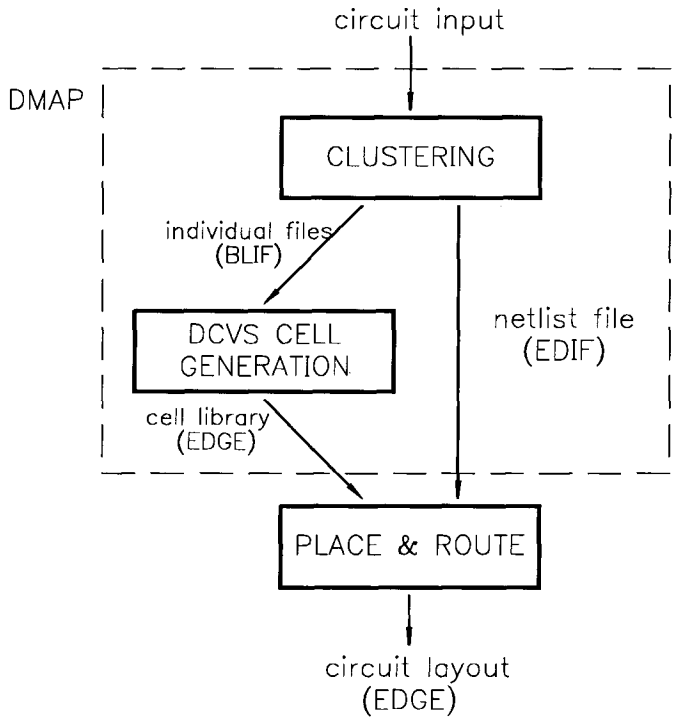



Figure 3.1: DMAP system flow diagram.

## 3.1 System Overview

The goal of the system is to take, as input, a technology independent circuit description, and produce DCVS layout cells ready for placement and routing in CADENCE.

The DMAP program is incorporated into Berkeley's MIS-II program, and hence can take an arbitrary circuit input in one of MIS-II's many input formats, such as BLIF (Berkeley Logic Interchange Format [2]). Once read into the system, the circuit is decomposed, and then partitioned into single-output, high functionality clusters (or modules). The cluster size is controlled by layout constraints. In particular, the system makes sure that each cluster can be laid out in the chosen cell style without violating any design rules. For example, the cluster size is constrained so that the final cell will have internal transistors that can be routed in the given layout style. The clustering module produces two outputs: a global netlist file which specifies the interconnection between the clusters (specified in EDIF, Electronic Design Interchange Format [4]), and circuit descriptions (at the logic function level) of the individual clusters (specified in BLIF).

The individual clusters are input to the DCVS cell generation module, where the layout of each cell is determined, and deposited in an intermediate layout representation (EDIF). After this, a CADENCE (EDGE) layout representation for each of the cluster functions is generated. The output of this cell generation module is a library of DCVS cell layouts realizing complex functions.

Finally, the CADENCE automatic placement and routing program takes the cell library (EDGE representations), together with the netlist specification of the whole circuit (EDIF), and produces the final circuit layout (EDGE representation). Each of these steps: clustering, cell generation, and cell connecting, are now expanded.

## 3.2 First Step: Clustering

The problem of circuit clustering or logic partitioning has been of interest for quite some time. In the days of small scale integration (SSI), it was desired to partition a complex system into physical components (chips) subject to delay constraints. Since inter-chip delay is significant, it was desired to minimize the maximum delay (along the critical path) through the whole design during partitioning. The optimization was carried out using objective functions such as total wire length, wire length in the longest closed path, or the wire length in the longest path from a network source to a network sink. To minimize cost, the number of chips should also be as small as possible. This lead to the earliest form of the "clustering" problem. That is, the partitioning of a circuit into chips and the routing between them. The first solution to this problem was an algorithm presented by Lawler, Levitt and Turner [31][1] for combinational networks. They assumed a "unit delay" model, defined as follows:

- all gate delays are zero;
- the delay between gates in the same cluster is zero;
- the delay between gates in different clusters is one.

Subsequently, when the scale of integration increased, clustering was applied at many other levels of the design hierarchy. Touati applied clustering to technology independent circuit delay optimization [57]. Rather than identifying critical paths, his idea was to cluster the circuit to minimize the maximum number of clusters on a path. Each of these clusters are collapsed onto a single node, then simplified. Recently, Murgai et.al. [45] generalized Lawler's algorithm for a general delay model. This is needed for high capacity clusters (such as LSI and VLSI chips), which very likely have critical path delays inside the cluster that are comparable to the total delay of the system.

[1] Henceforth referred to simply as Lawler's algorithm

Network clustering has also been applied in the automated design of programmable devices, such as programmable gate arrays (PGA's) [46]. There are a limited number of configurable logic blocks (CLBs) on one chip. For example, a typical Xilinx [1] chip has about 320 of these CLBs to implement random logic. This is further constrained by a maximum limit on the number of inputs to each CLB and the total wiring connections that are available between the CLBs. The clustering goal is to minimize the number of blocks used, reduce routing complexity, and minimize delay on the critical path. The goal of DCVS clustering is quite similar.

### 3.2.1 Circuit Clustering in DMAP

DMAP partitions the decomposed input circuit onto clusters that can be realized as DCVS gates. The clustering in DMAP is constrained by the following:

- If all the nodes in the cluster were to form the function of a DCVS pull-down tree, the number of internal routing tracks needed for internal connections would not exceed a maximum limit.
- The number of inputs to any cluster does not exceed a maximum limit.

The resulting clusters are rooted trees, each of which is collapsed onto a single function (or node) that is implemented as a DCVS pull-down tree at the transistor level. Since the delay between transistors within a cluster is negligible compared to the delay between clusters, it is safe to assume the unit delay model defined earlier. Hence, it seems that some variation of Lawler's algorithm, which will be explained in detail below, is quite suitable for our needs.

### 3.2.2 Lawler's Algorithm

Lawler's algorithm, in its basic form, clusters the circuit to minimize the delay throughout the network. The clustering procedure is essentially a labeling algorithm, where the label of node $i$,

$\lambda(i)$, represents the longest delay along any path from a network input to $i$. Hence, the longest delay in the network is the largest label value. Let $w_i(k)$ be the sum of the weights of all the predecessor nodes of node $i$ with label $k$, and $w_i$ be the weight of node $i$. The weight of a node is a value associated with some property of the node. A node is added to a cluster of the same label, $w_i$ plus $w_i(k)$, where $k$, the label of node $i$, does not exceed a maximum $M$. Figure 3.2 shows a graph after clustering with the Lawler's algorithm for $M = 3$ and $w = 1$ for all nodes. The nodes that belong to the same cluster are drawn together, and the label for each node is shown in parenthesis beside the node. The numbering of the nodes is in the order that they are traversed in the algorithm. The labeling algorithm is outlined as follows:

1. Label all input nodes 0.

2. Find an unlabeled node $i$, all of whose predecessors have been labeled.

   Let $k$ be the largest label of any predecessor.
   If $(w_i + w_i(k) \leq M)$
   $\lambda(i) = k$
   Else $\lambda(i) = k + 1$.

3. Repeat Step 2 until there are no more nodes.

### 3.2.3 Modified Lawler's Algorithm

The DMAP clustering algorithm differs from Lawler's in three ways. First, as the reader may notice from Figure 3.2, Lawler's method produces some inefficient single-noded clusters. That is, there are single nodes whose fanins and fanouts are of a different label. Without violating cluster-constraints, nodes 6, 7, and 8 can be put into one cluster; node 10 can be included into the 9 and 11 cluster; nodes 4, 5, and 12 can be one cluster. The DMAP algorithm recognizes these cases. Secondly, DMAP uses a different capacity constraint. In fact, DMAP verifies that the clusters can be implemented in less than the maximum number of tracks allowable by a cell

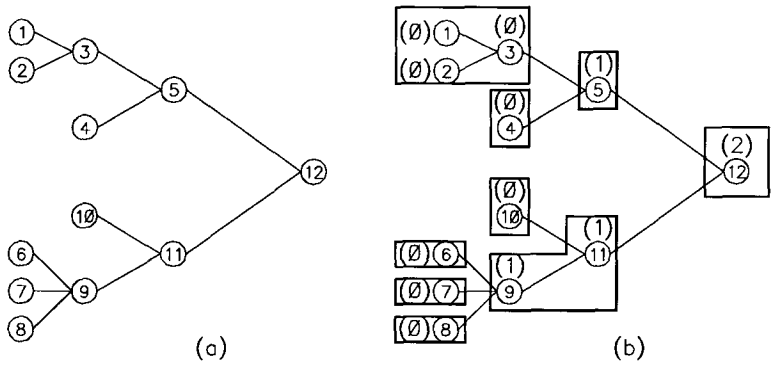



Figure 3.2: Illustration of Lawler's clustering algorithm ($M = 3$).

of standard height and that the number of cluster inputs does not exceed a maximum. Finally, once the labeling count has been advanced during the node traversal procedure, the completed clusters are collapsed and generated immediately.

Incorporating these changes then, we develop a "modified Lawler's algorithm". Given any node $n$ in a circuit, we define $T_k(n)$ as the tree built by taking $n$ as root, and including all nodes labeled $k$ which will ultimately fanin to $n$. The algorithm is as follows:

1. Label all input nodes 0.

2. Find an unlabeled node $i$, all of whose predecessors have been labeled.

   Let $k$ be the largest label applied to any predecessors.
   Let $tree = T_k(i)$.
   If (Satisfy_Constraints($tree$))
   $\lambda(i) = k$
   Else
   $\lambda(i) = k + 1$
   Foreach $p$ = fanin($i$)
   Let $test\_tree = T_k(i)$ plus the node $p$

If (Single_Node($p$)) AND (Satisfy_Constraints(*test_tree*)) )
    $\lambda(p) = \lambda(i)$
Else If ($\lambda(i) > \lambda(p)$)
    Generate_Cell($p$)

3. Repeat Step 2 until there are no more nodes.

The function Satisfy_Constraints($n$) checks to see if the tree with root $n$ containing only those predecessor nodes with the same label, has a logic function that will result in less than the maximum number of allowed routing tracks when run through a somewhat simplified version of the cell generation routine. For a 1.2 micron double metal technology, for example, this upper track limit is 7. Generate_Cell($n$) will take the tree with root $n$ containing nodes of the same label, collapse the tree into one function, and generate the DCVS layout cell for that function. This operation will be described in Section 3.3.

Figure 3.3 illustrates the DMAP clustering algorithm. Part (a) shows the tree to be clustered, and Part (b) shows the results of clustering. First of all, nodes 1 to 14 are labeled 0. Suppose that $T_0(15)$ does not satisfy the cluster-constraints, and node 15 is labeled 1. At this point, node 15's two fanin trees ($T_0(7)$ and $T_0(14)$) are collapsed and the first two clusters, C1 and C2, are generated. Next, nodes 16 to 30 are labeled 0 because they all fit into one cluster. The next node to be labeled is node 31. The largest predecessor label is $k = 1$ (node 15). $T_1(31)$ consists of nodes 15 and 31 (C1 and C2 are considered as inputs to node 16 now). Nodes 16-30 have been labeled 0, and hence are not part of the tree $T_1(31)$. It is found that this tree satisfies the cluster constraints, and so $\lambda(31) = 1$. Before proceeding to label the next node, $T_0(30)$ can be collapsed and the corresponding cell can be generated to form C3. Similarly, nodes 32 to 61 are labeled, and C5 and C6 are generated. This time, the tree consisting of nodes 46 and 62 ($T_1(62)$) does not satisfy the cluster constraints, and so $\lambda(62) = 2$. Then C7 is generated. Although node 46 is a single-node cluster, its label cannot be incremented without violating some cluster constraints. C8 is generated. Then, nodes 63 to 72 are labeled 0, and node 73 is labeled 0. Lastly, we label node 74. At this point, $k = 2$. The tree consisting of nodes 74 and

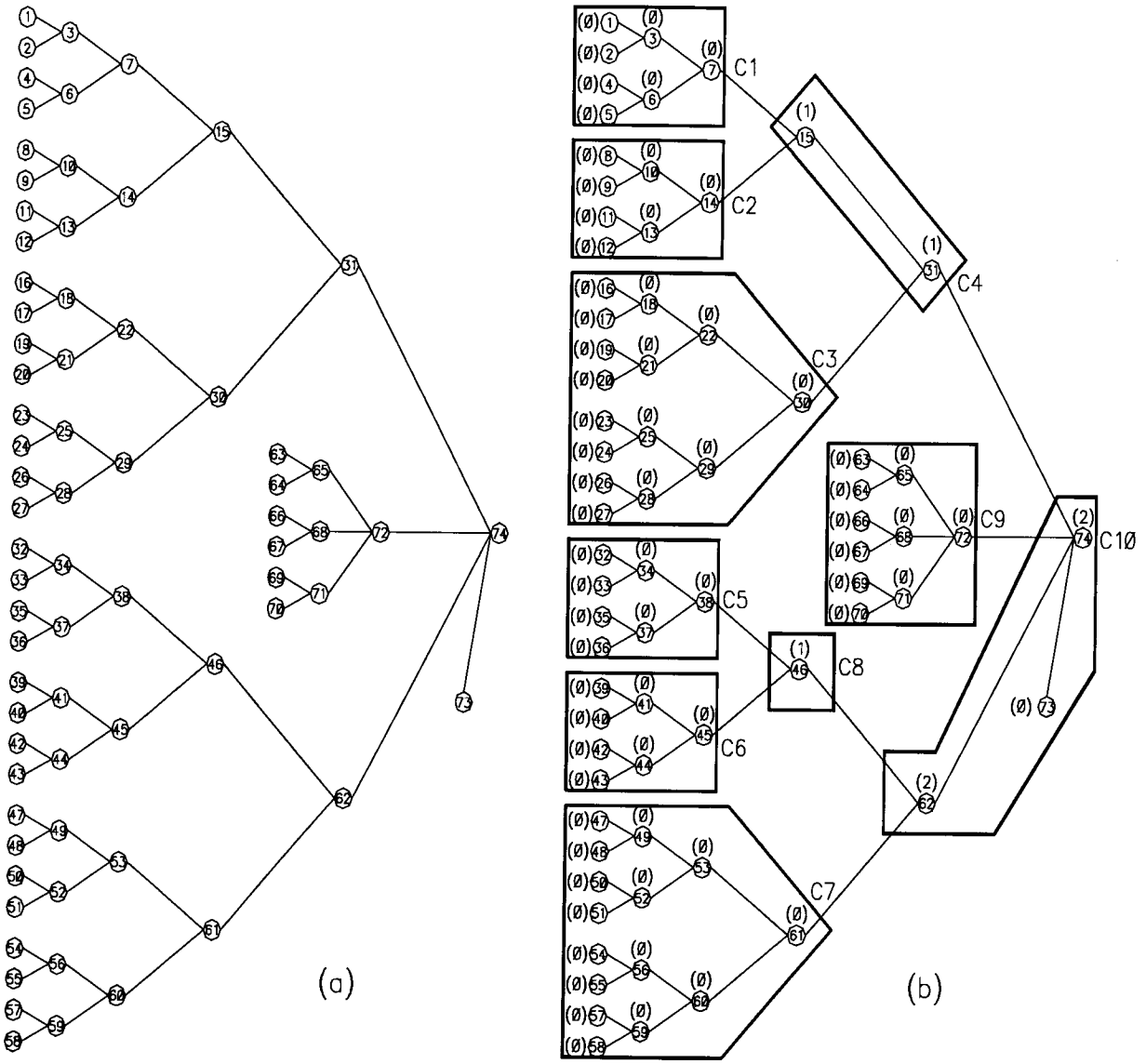



Figure 3.3: Illustration of DMAP clustering: (a) original DAG (b) DAG after clustering.

62 is tested, and found to satisfy constraints, and so $\lambda(74) = 2$. C4 and C9 are generated. It is found that adding node 73 to $T_2(74)$ still satisfies constraints. So, finally, the cluster C10, consisting of nodes 62, 73, and 74 is formed and generated.

In the example just shown, DMAP clustered the circuit into ten clusters. During the process, two single-noded clusters were encountered (node 46 and node 73), but only node 73 could be re-labeled and included into a bigger cluster. It is likely that node 46 already contained fairly high functionality. The longest wire delay in the network is 2 since the largest label is 2. For example, the path $(C6, C8, C10)$ has a wire delay of 2 and a gate delay of 3.

Finally, after all the nodes have been labeled and all the cells have been generated, the last step is to output a global netlist file which specifies the connection between the generated cells. Generate_Cell($n$) takes a single output function of single-rail inputs and produces a DCVS cell with two output functions of dual-rail inputs. Therefore, the original circuit just after clustering must be *dualized*, i.e., to correctly duplicate the appropriate rails so the connections to the dual-rail DCVS cells are correctly made. For example, in Figure 3.3, a netlist of 10 cells, 9 internal nets, 32 inputs, and 1 output becomes a netlist of 10 cells, 18 internal nets, 64 inputs, 2 outputs, plus one additional input for the precharge signal. Any DCVS circuit will have about twice as many outputs and roughly twice as many inputs, as its single-rail counterpart.

## 3.3 Second Step: Cell Generation

The goal of the DCVS cell generation step is as follows. Given an arbitrary Boolean function $f$, lay out the pull-down tree as well as the other transistors needed to construct a DCVS gate that implements $f$. By nature of the DCVS structure, this also means implementing $\overline{f}$. In order to facilitate automatic synthesis, a structured layout style is used. Figure 3.4 shows the layout of a generated 3-input XOR DCVS gate. The shaded and dotted regions are the device wells, while the unshaded and dotted regions intersecting them are polysilicon. The unshaded striped regions are metal2, while the unshaded cross-hatched regions are metal1. The solid

regions are either contact or via. The cell consists of two parts: a standard left-half, and a generated right-half. The two halves are connected by the power and ground rails and three routing tracks: one for the connection to the source of the N-type precharge transistor; one to $q$; and one to $\overline{q}$ (refer to Figure 2.7).

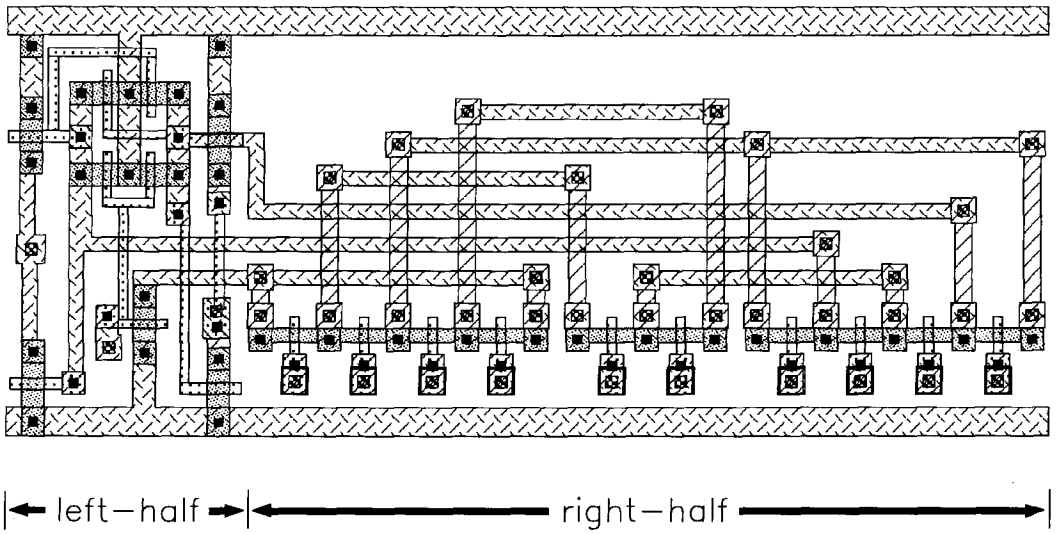



Figure 3.4: Layout of a generated DCVS 3-input XOR cell.

The left-half contains nine transistors: the two precharge and cross-coupled P-type transistors, the single N-type precharge transistor, and the two static INVERTERs (one N-type transistor and one P-type transistor). This part is hand-designed and is the same for all the DCVS cells.

The right-half of the cell is the layout of the DCVS pull-down tree. It includes a single strip of transistors placed along the bottom of the cell and the routing tracks for transistor connections consuming the remaining space. A one-dimensional transistor layout style is adopted. The DMAP program takes several steps to generate such a layout, each of which will be elaborated

on in the following subsections:

- Given the logic function, obtain a DCVS pull-down transistor netlist.
- Given the pull-down netlist, obtain a transistor layout path for this netlist.
- Given the transistor layout path, obtain the track connections between transistors.
- Output the physical layout.

### 3.3.1 Obtaining the DCVS Logic Tree

There is a very natural relation between *binary decision diagrams* (BDD) [7] and the pull-down tree of a DCVS. BDDs are an alternative way of representing logic functions. A BDD is a directed acyclic graph with two paths directed away from every node: one for the node asserted true, and one for the node asserted false. The root of the tree is the function to be implemented, the leaves are 0 or 1 terminals, and the nodes are function variables. Bryant introduced *ordered* BDD [14, 15], which requires that for any path from the root to a leaf node, the variables must have the same ordering. Figure 3.5(a) shows an unordered BDD, and Figure 3.5(b) is an ordered BDD. Most of the BDD research has focused on ordered BDDs. If unspecified, a "BDD" will be assumed to be ordered. Although in this thesis we will use algorithms for ordered BDDs, it should be noted the relationship between BDD and DCVS logic is just as valid for *unordered* BDDs. Should future algorithms be developed for unordered BDDs, they can be used for our purpose as well.

To obtain a DCVS pull-down tree for a function $f$, the basic idea is to turn the BDD representing $f$ "upside down". After this, the following conversions are made. The "1" terminal is labeled $q$, and the "0" terminal is labeled $\overline{q}$. The "1" branch of a variable is labeled with the variable name, while the "0" branch of the variable is labeled with the complemented variable name. The root node of the BDD is labeled *precharge*, and will be connected to the precharge transistor that conducts to ground. The other nodes are labeled arbitrarily, and correspond to

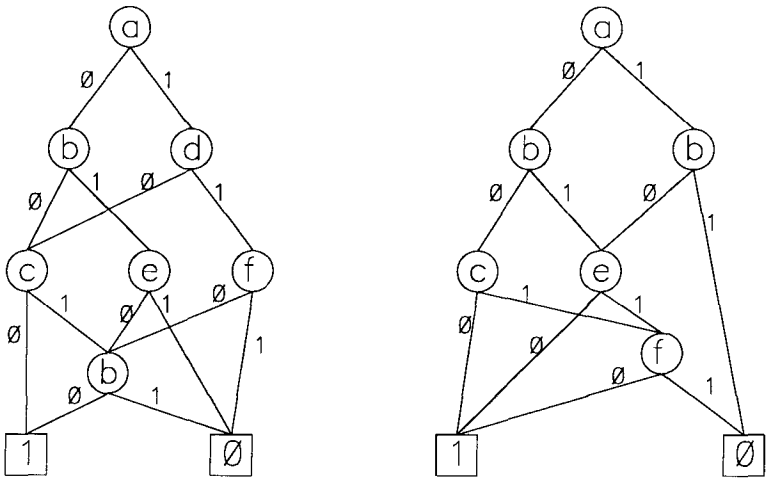



Figure 3.5: (a) Unordered BDD (b) Ordered BDD

electrical nodes in the pull-down circuit. Each edge of the BDD becomes a transistor controlled by one of the DCVS cell inputs.

To illustrate the transformation of a BDD to the pull-down tree of a DCVS gate, consider Figure 3.6. Part (a) is a possible BDD for a 3-input XOR function, and part (b) is the same BDD turned upside down. Part (c) shows the relabeling, and part (d) shows the corresponding DCVS pull-down tree. Each BDD node converts to two transistors controlled by the complemented and uncomplemented form of the node variable. Given this straightforward transformation, one way of synthesizing a DCVS cell is to derive a small BDD representation for the desired function and then carry out the transformation.

The order that variables appear in a BDD can significantly affect the size of the BDD [15]. Figure 3.7 shows two different BDDs for the same function $x_1x_2 + x_3x_4$, using different variable ordering. Clearly, a poor initial choice of variable ordering can cause undesirable effects.

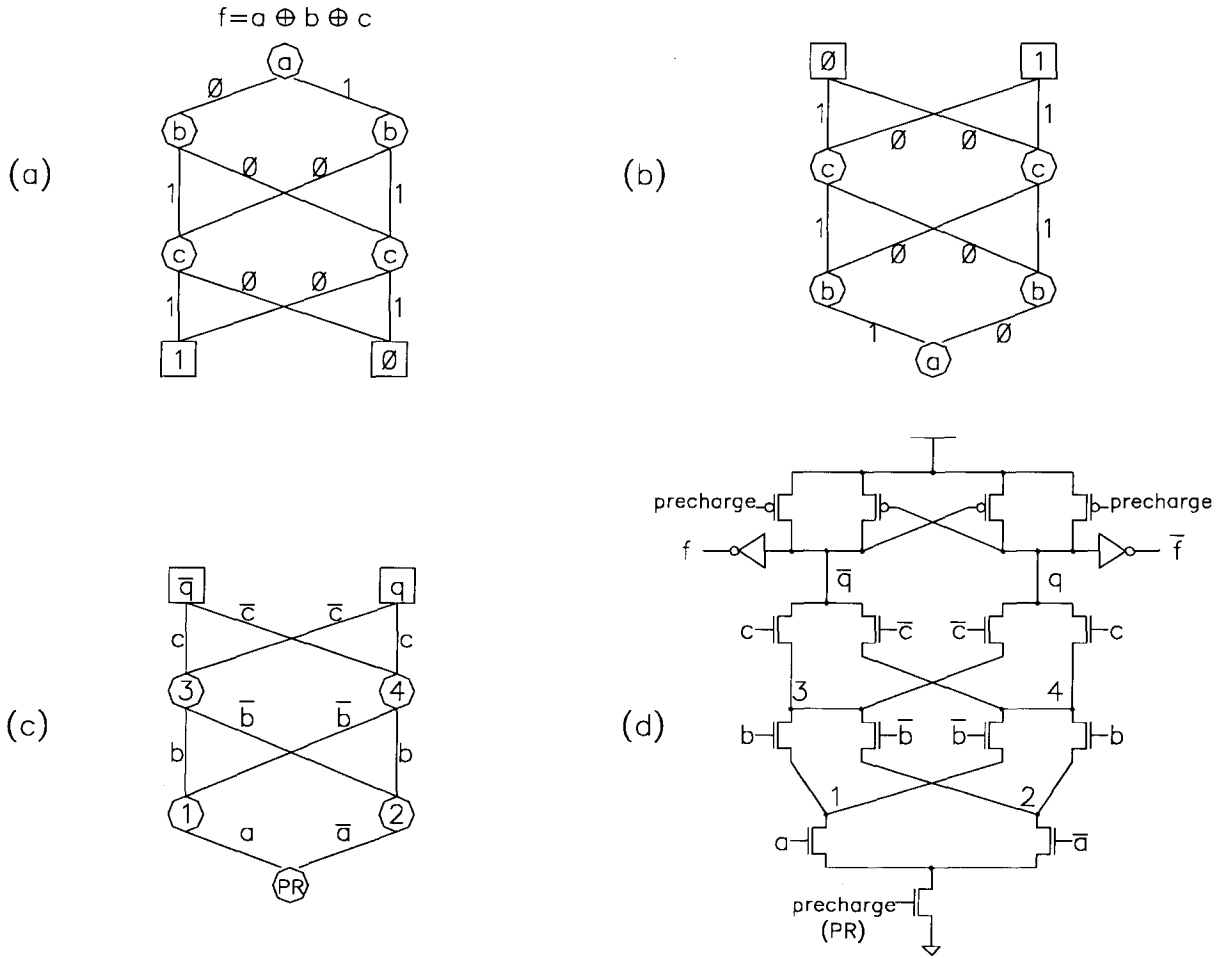



Figure 3.6: Converting a BDD to a DCVS pull-down tree:
(a) 3-Input XOR BDD.
(b) "upside down" BDD.
(c) Relabeled BDD.
(d) Corresponding DCVS gate schematic.

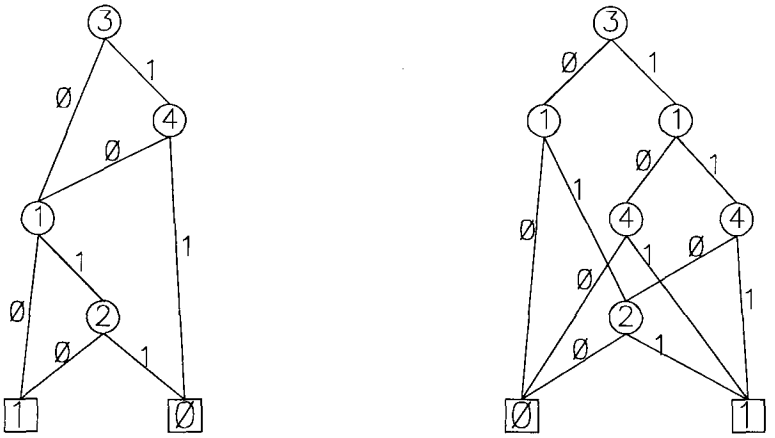



Figure 3.7: Example of BDDs for the same function using different variable orderings

DMAP creates the DCVS pull-down tree by building an *ordered* BDD of optimal variable ordering using the algorithm given in [21]. During the clustering process, a simplified variable ordering heuristic algorithm [32], implemented in Berkeley's MIS-II logic synthesis program, is used to check if a node belongs in a cluster.

### 3.3.2 Obtaining the Transistor Layout Path

Given the transistor netlist for the pull-down tree and the chosen one-dimensional transistor layout scheme, the transistors must be laid out in an manner that will give good cell area and speed, as well as facilitate automatic global placement and routing. The problem of finding the best layout path to minimize the number of internal transistor connections is very difficult to solve exactly. Hence, DMAP uses heuristics that appear to give good results.

The basic idea is to perform a priority-based graph walk. The heuristic first tries to find a

path that will lead to any node that has already been encountered but which still has transistors connected to it that has not been placed. This seeks to localize all the connections made to a single node, so that as many different node connections can be shared on one track line as possible. Given the above, the heuristic then tries to find a path along adjacent nodes to maximize diffusion sharing. This is essentially equivalent to finding an Euler path [58] in the transistor graph. If this cannot be done, a pseudo edge separation is inserted, which results in making the cell width greater. The pseudo code for the algorithm is given below. Initially, all nodes are marked as INACTIVE. A node become ACTIVE once a transistor connected to the node has been placed.

```
Find_Transistor_Path(n) {
    While (Edge Set is not empty) {
        If (degree(n)=0)
            Mark n as FINISHED;
            S = set of ACTIVE nodes;
        Else
            AN = set of ACTIVE neighbours
            If (AN is not empty)
                S = AN;
            Else
                RA = set of ACTIVE nodes excluding n;
                if (RA is not empty)
                    S = set of neighbours closest to any node in RA;
                Else
                    S = set of neighbours;
        n_next = node in S with smallest degree;
        Output edge (n,n_next);
        Remove edge (n,n_next) from Edge Set;
        Add n_next to the set of ACTIVE nodes;
        n = n_next;
    }
}
```

In Figure 3.8 we illustrate the first steps in the algorithm for a 3-input XOR function assuming

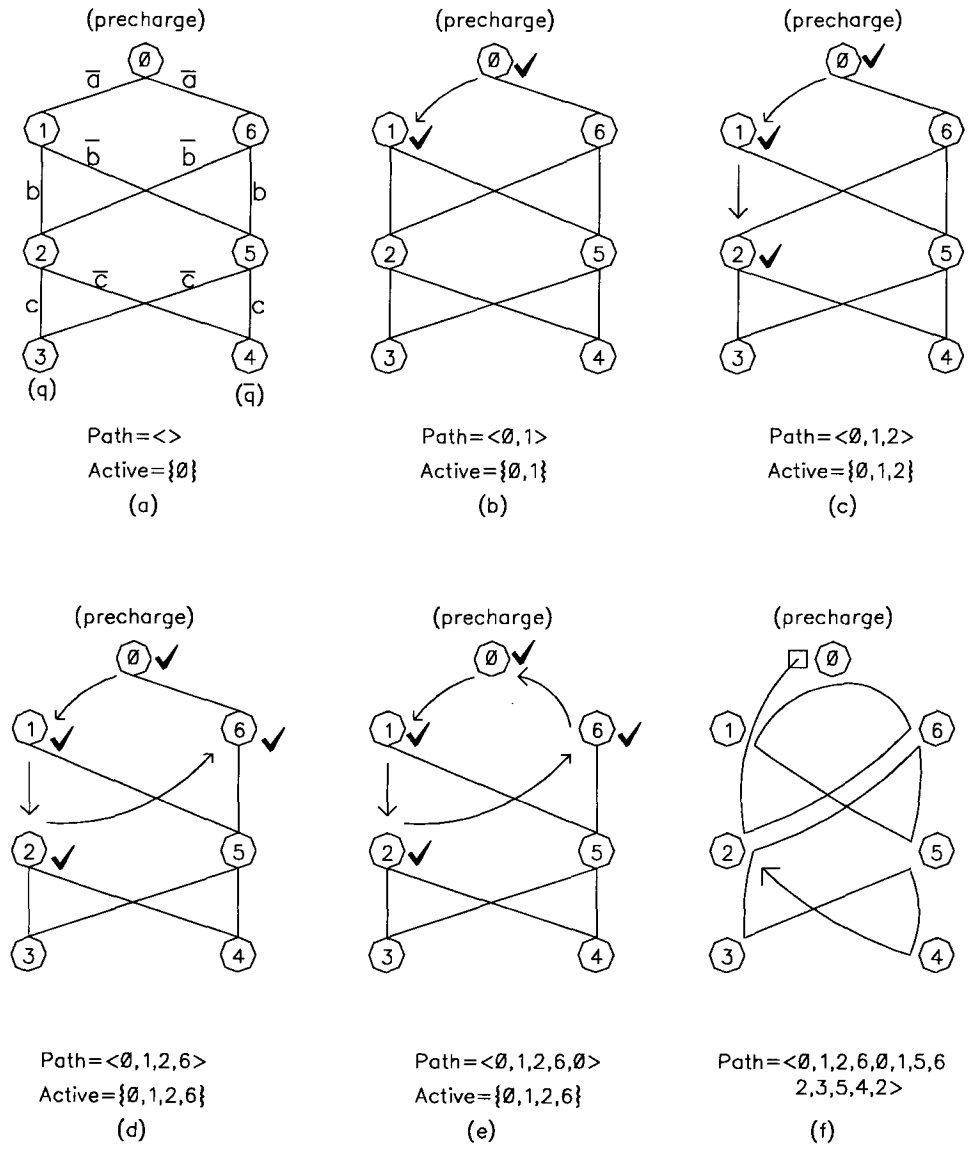



Figure 3.8: Illustration of DMAP's transistor path finding algorithm.

the path is started from node 0. ACTIVE nodes are marked with a "tick". Since node 0 has degree 2, and has no ACTIVE neighbours, the next node can either be node 1 or node 6—both of which have the same degree. Arbitrarily, node 1 is chosen (Figure 3.8(b)). Node 1 has degree 2, and does not have any ACTIVE neighbours. The set of active nodes, excluding node 1, only contains node 0, i.e., $RA = \{0\}$. There are two shortest paths from node 1 to node 0: $< 1, 2, 6, 0 >$ and $< 1, 5, 6, 0 >$. The set of neighbours closest to any node in $RA$ is hence $\{2, 5\}$. Arbitrarily, node 2 is chosen, and the Edge and ACTIVE Sets are updated (Figure 3.8(c)). The next nodes chosen are nodes 6 and 0 (Figure 3.8(d) and Figure 3.8(e)). At this point a pseudo edge separation must be added from node 0 to an ACTIVE node with smallest indegree—in this case node 1. Continuing the procedure, the final path, $< 0, 1, 2, 6, 0, 1, 5, 6, 2, 3, 5, 4, 2 >$, is shown in Figure 3.8(f). In summary, the 3-input XOR function is laid out as a DCVS cell in which the pull-down tree uses ten transistors with only two pseudo edge separations added.

### 3.3.3 Obtaining the Track Connections

After the transistor layout path is obtained, the next step is to connect the transistors together. DMAP allows the upper portions of the cell's right-half for such connections. This is done in a series of tracks parallel to the transistor strip. For our standard design height of 75 design microns, a maximum of 7 tracks are allowed. As mentioned at the beginning of this section, 3 of these tracks are already used up: one of the nodes to the pull-down precharge, one for the positive function realization, and one for the complement of the function. This leaves 4 tracks free for general connections. Once the transistor path is known, the connection points between transistors is fixed. The problem is to place these connections in a way that will share as many tracks as possible, hence using as few tracks as possible.

DMAP achieves this by using the Left-Edge-Algorithm (LEA) first given by Hashimoto and Stevens in [23], and later employed by Berkeley's YACR-II channel router [51]. The connections between transistors can be viewed as wire segments ($I_i$), whose ends $[x_i, y_i]$ are connected by

a vertical piece of wire down to the transistor row. As in channel routing, the object is to place these segments on the fewest number of tracks without overlap. The algorithm, which guarantees minimum tracks, is essentially as follows:

1. Search the set of intervals for the element which has the smallest lower bound.
2. This element is assigned to the first track and removed from the interval set.
3. Search the element in the interval set which has the smallest lower bound which is greater than the upper bound of the previously chosen interval.
4. This interval is assigned to the track and removed from the interval set.
5. Repeat Steps 3 and 4 until no intervals are left that satisfy the requirement.
6. If the interval set is non-empty, go to Step 1 to start on the assigning for the next track.

Figure 3.9(a) shows 7 track segments, sorted according to left-edge. Figure 3.9(b) illustrates the tracks after LEA is applied. The horizontal axes of the graphs are the connection points on the transistor path (which determine the cell-width); the vertical axes are the routing tracks (which determine the cell-height). The segments 1, 4 and 8 are chosen to put on the first track; segments 2 and 6 share the second track; and segments 3 and 5 are on the third track.

### 3.3.4 Layout Issues

Once the transistors and track connection positions are all known, the right-half portion of the cell can be added together with the standard left-half portion of the cell. The cell information is written onto a layout specification file in EDIF, which can be read into CADENCE. Figure 3.10 shows some typical generated DCVS cell layouts.

Our designs use a 1.2 micron, Northern Telecom CMOS double metal technology. The cells are of standard height (75 design units) and variable width. This allows for 7 internal routing

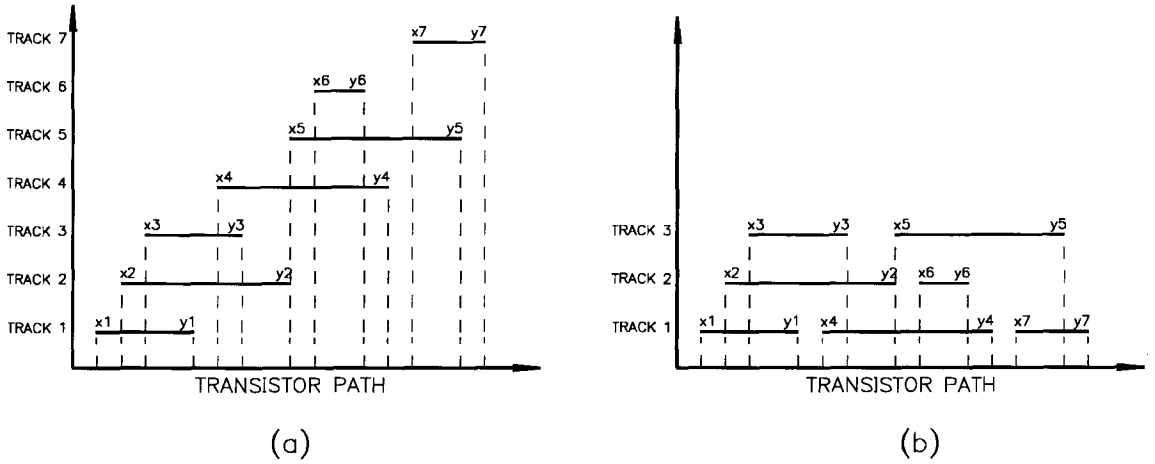



Figure 3.9: Illustration of Left-Edge-Algorithm.

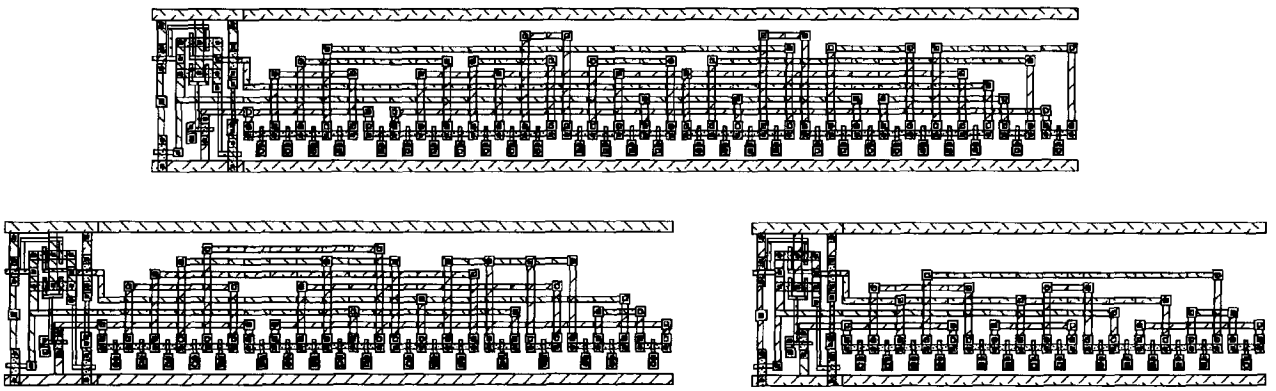


Figure 3.10: Some generated DCVS cell layouts.

tracks in the right-half portion of the cell. Metal 1 is used for horizontal routing of track segments, while metal 2 is used for the vertical connection of these segments to the transistors. The function inputs are pulled to the bottom of the right-half portion of the cell. The precharge and two output lines are at the left-half portion of the cell. Jumper pins are inserted where possible to facilitate global routing.

The right-half portion of the cell can waste space in the routing region. Also, the one-dimensional array of transistors tends to produce long and short cells, which may make it difficult for the placement and routing program. However, there are a number of advantages for us adopting this layout style.

Having a standard left-half that is independent of the pull-down tree laid out in the right-half allows for flexible changes in the pull-up network and pull-down network. For example, it may be desired to give the static INVERTERs stronger driving power (since DCVS cells typically drive many inputs). The size of the INVERTERs transistors can be changed without affecting the rest of the system. On the other hand, new ways for laying out the pull-down tree can be easily adopted into the system. This style allows for future extensions of our system for other forms of dynamic logic.

The one-dimensional transistor strip allows the cells to be "porous" with respect to global routing. In other words, the input pins act as feedthrough pins, where the input signals can pass through the pin from the top and bottom directions. Furthermore, the one-dimensional strip allows for transistor sizing. One of the advantages of cell synthesis is the freedom to vary the transistor sizes for better performance.

## 3.4 Third Step: Connecting Cells Together

At this stage, we have a set of DCVS layout cells specified in EDIF files, and a circuit netlist description of the interconnection between these cells, also in EDIF format. After some CADENCE-specific preprocessing steps (Appendix B documents these "fixes" in full detail and glory), the layout files are converted to EDGE cell representations. CADENCE then reads in the global netlist file, searches for the layout instances, and performs placement and global routing. The result is a complete DCVS circuit layout.

# Chapter 4

## Experimental Results

The system described in the previous chapter was implemented in the C-programming language, inside the MIS-II programming environment. Exhaustive experiments were done to evaluate the performance of the system. The primary concern in DCVS synthesis is the final circuit area, which consists of:

- the total area of the generated cells, and
- the total area of global routing.

In DMAP, the area of each cell is determined by the number transistors in the layout strip and the number of pseudo-edges. The total area of generated cells is also affected by the number of cells that are produced by the clustering function. The number of internal routing tracks used, give an indication of how well the cell area is being utilized. The global routing area is determined by the number of cells that need to be wired together, as well as the number of net connections between these cells. We now look at experimental results and examine some of these factors that influence the final area.

### 4.1 Evaluating DMAP's Stand-Alone Performance

DMAP was run with industrial benchmark circuits from MCNC (Microelectronics Corporation of North Carolina) and ISCAS (International Symposium on Circuits and Systems). Table 4.1 and Table 4.2 summarize the statistics of these experiments. Column 2 shows the average

number of transistors in the layout path, while Column 3 shows the number of pseudo-edges in the layout path. Typically, each cluster contains approximately 20 transistors, which implement reasonably large sized functions. The small pseudo-edge to transistor count ratio in Column 5 indicates that the path finding algorithm is able to find fairly good layout paths. Column 4 shows the average number of routing tracks that are used for transistor connections in that cell. It appears that most of the cells do not actually use up all the allowable tracks. This suggests that better use of space in the right-half portion of the cell can be made. We will discuss methods of improvement later.

## 4.2 Comparing DMAP with Conventional Mapping

Since DMAP maps *single* rail networks onto *dual* rail circuits, there will always be an area penalty for using DCVS. In order to understand what the size of the penalty is, we compare DCVS circuits produced by DMAP with static CMOS circuits mapped with MIS-II using the UBC-Northern Telecom CMOS4 standard cell library.

Table 4.3 and Table 4.4 summarize the cell and internal net count of the two approaches. The number of cells used by DMAP is considerably fewer than that used by the standard cell method. Often DMAP uses only one fifth as many cells. This means that although DCVS designs require dual rails, fewer cells need to be routed. Furthermore, the total number of nets that need to be wired, i.e., unique signals that need to be routed to each cell, is typically less than twice the number of nets in standard mapping. In fact, a number of circuits have *fewer* total nets. This is better than the intuitive assumption that dual-rail circuits require twice as many wires as their single-rail counterparts.

Table 4.5 and Table 4.6 summarize some cell area comparisons that were made between the two approaches. For most circuits, the total DCVS cell area is slightly more than twice the total cell area of the standard cells. This is a relatively good result considering the simplicity of the chosen cell layout style. It is likely that a more sophisticated layout style will improve

this significantly.

| Circuit Name | Average No. Transistors in Pulldown Tree | Average No. Pseudo Edges | Average No. Internal Routing Tracks | Pseudo-edge/ Transistor Ratio |
|---|---|---|---|---|
| C1355 | 16.8 | 3.8 | 5.8 | 0.2 |
| C17 | 9.0 | 1.5 | 5.5 | 0.2 |
| C1908 | 16.9 | 3.9 | 5.9 | 0.2 |
| C2670 | 16.7 | 3.4 | 5.8 | 0.2 |
| C3540 | 17.4 | 3.5 | 6.0 | 0.2 |
| C432 | 18.1 | 4.0 | 6.1 | 0.2 |
| C499 | 17.8 | 3.8 | 5.8 | 0.2 |
| C5315 | 20.1 | 4.1 | 6.0 | 0.2 |
| C6288 | 20.3 | 4.2 | 6.2 | 0.2 |
| C7552 | 19.7 | 3.9 | 5.9 | 0.2 |
| C880 | 14.1 | 3.0 | 5.1 | 0.2 |

Table 4.1: DMAP generated cells for the ISCAS benchmark circuits.

## 4.3 After Placement and Routing

A few circuits were placed and routed with the CADENCE placement and routing tool. They were found to have a total circuit area much greater than the corresponding standard cell circuit. For example, the DUKE2 DCVS circuit was found to be about 3.5 times larger in area than the standard cell circuit (see Figure 4.1 and Figure 4.2). Note that the DCVS chip has twice the number of pads as the single rail chip. The ruler scale beside the core area are the same in both Figures, and can be used to estimate the core areas. However, it is meaningless to make such comparisons, as the total area is very much dependent upon the placement and routing program, and which parameters the program optimizes for. Figure 4.3 and Figure 4.4, which show the core area of the circuit MISEX2, exemplify how much the placement and routing can affect the circuit area.

| Circuit Name | Average No. Transistors in Layout Path | Average No. Pseudo-edges in Layout Path | Average No. Internal Routing Tracks | Pseudo-edge to Transistor Ratio |
|---|---|---|---|---|
| 5xp1-hdl | 9.9 | 2.2 | 4.9 | 0.2 |
| 5xp1 | 15.1 | 3.2 | 5.6 | 0.2 |
| 9sym-hdl | 14.7 | 3.1 | 5.7 | 0.2 |
| 9sym | 18.6 | 4.5 | 5.6 | 0.2 |
| 9symml | 13.8 | 3.1 | 5.7 | 0.2 |
| alupla | 22.5 | 4.8 | 6.4 | 0.2 |
| bw | 13.8 | 3.2 | 5.7 | 0.2 |
| con1 | 10.3 | 2.2 | 5.3 | 0.2 |
| duke2 | 21.6 | 5.4 | 5.3 | 0.3 |
| f2 | 10.0 | 3.0 | 6.0 | 0.3 |
| f51m-hdl | 9.4 | 2.1 | 4.8 | 0.2 |
| f51m | 16.6 | 3.8 | 5.8 | 0.2 |
| misex1 | 12.1 | 2.9 | 5.2 | 0.2 |
| misex2 | 19.8 | 4.9 | 5.0 | 0.2 |
| misex3 | 14.4 | 3.6 | 5.3 | 0.2 |
| misex3c | 20.9 | 5.1 | 5.7 | 0.2 |
| 5xp1 | 15.1 | 3.2 | 5.6 | 0.2 |
| rd53-hdl | 19.3 | 3.0 | 6.0 | 0.2 |
| rd53 | 19.3 | 3.0 | 6.0 | 0.2 |
| rd73-hdl | 19.3 | 4.3 | 6.0 | 0.2 |
| rd73 | 21.0 | 5.0 | 6.2 | 0.2 |
| rd84-hdl | 15.7 | 3.6 | 5.6 | 0.2 |
| rd84 | 20.8 | 5.1 | 6.0 | 0.2 |
| sao2-hdl | 20.1 | 4.8 | 5.8 | 0.2 |
| sao2 | 18.1 | 4.4 | 5.4 | 0.2 |
| seq | 20.0 | 4.6 | 6.2 | 0.2 |
| vg2 | 19.4 | 3.8 | 6.0 | 0.2 |
| z4ml-hdl | 16.9 | 3.9 | 5.6 | 0.2 |
| z4ml | 17.0 | 3.9 | 5.7 | 0.2 |

Table 4.2: DMAP generated cells for the MCNC benchmark circuits.

| Circuit Name | MIS-II Mapped (a) | | DMAP Mapped (b) | | (b)/(a) | |
|---|---|---|---|---|---|---|
| | # cells | # nets | # cells | # nets | # cells | # nets |
| C1355 | 732 | 1250 | 134 | 1875 | 0.2 | 1.5 |
| C17 | 6 | 12 | 2 | 18 | 0.3 | 1.5 |
| C1908 | 643 | 1164 | 170 | 1862 | 0.3 | 1.6 |
| C2670 | 1036 | 1852 | 233 | 2648 | 0.2 | 1.4 |
| C3540 | 1478 | 2721 | 393 | 5172 | 0.3 | 2.0 |
| C432 | 260 | 473 | 131 | 1088 | 0.5 | 2.3 |
| C499 | 558 | 892 | 98 | 1744 | 0.2 | 2.0 |
| C5315 | 2033 | 3992 | 656 | 9182 | 0.3 | 2.3 |
| C6288 | 2402 | 4755 | 810 | 9510 | 0.3 | 2.0 |
| C7552 | 3036 | 6200 | 652 | 10544 | 0.2 | 1.7 |
| C880 | 435 | 760 | 148 | 1140 | 0.3 | 1.5 |

Table 4.3: DMAP vs. MIS-II in cell and net count for ISCAS benchmarks.

However, as technology allows for more than two levels of interconnect in the layout, it becomes possible to perform routing on top of the cells. In this case, only the cell areas would be of concern. DCVS uses considerably fewer cells than standard mapping. By focusing on improving the layout style, the DCVS area penalty can be reduced.

| Circuit Name | MIS-II Mapped (a) | | DMAP Mapped (b) | | (b)/(a) | |
|---|---|---|---|---|---|---|
| | # cells | # nets | # cells | # nets | # cells | # nets |
| 5xp1-hdl | 135 | 275 | 28 | 224 | 0.2 | 0.8 |
| 5xp1 | 82 | 130 | 25 | 253 | 0.3 | 1.9 |
| 9sym-hdl | 138 | 225 | 57 | 479 | 0.4 | 2.1 |
| 9sym | 203 | 462 | 44 | 664 | 0.2 | 1.4 |
| 9symml | 161 | 383 | 48 | 504 | 0.3 | 1.3 |
| alupla | 137 | 246 | 44 | 704 | 0.3 | 2.9 |
| bw | 222 | 491 | 36 | 328 | 0.2 | 0.7 |
| con1 | 23 | 41 | 6 | 44 | 0.3 | 1.1 |
| duke2 | 465 | 1174 | 114 | 2120 | 0.2 | 1.8 |
| f2 | 24 | 44 | 4 | 32 | 0.2 | 0.7 |
| f51m-hdl | 78 | 124 | 30 | 237 | 0.4 | 1.9 |
| f51m | 143 | 273 | 27 | 286 | 0.2 | 1.0 |
| misex1 | 69 | 149 | 16 | 183 | 0.2 | 1.2 |
| misex2 | 89 | 234 | 17 | 286 | 0.2 | 1.2 |
| misex3 | 708 | 1674 | 229 | 2656 | 0.3 | 1.6 |
| misex3c | 514 | 1244 | 117 | 1989 | 0.2 | 1.6 |
| rd53-hdl | 56 | 88 | 3 | 30 | 0.1 | 0.3 |
| rd53 | 50 | 98 | 3 | 30 | 0.1 | 0.3 |
| rd73-hdl | 90 | 141 | 24 | 230 | 0.3 | 1.6 |
| rd73 | 136 | 317 | 21 | 284 | 0.2 | 0.9 |
| rd84-hdl | 122 | 190 | 45 | 410 | 0.4 | 2.2 |
| rd84 | 303 | 665 | 50 | 765 | 0.2 | 1.2 |
| sao2-hdl | 307 | 538 | 172 | 2924 | 0.6 | 5.4 |
| sao2 | 131 | 305 | 33 | 478 | 0.3 | 1.6 |
| seq | 2093 | 4456 | 685 | 7980 | 0.3 | 1.8 |
| vg2 | 87 | 159 | 30 | 293 | 0.3 | 1.8 |
| z4ml-hdl | 69 | 140 | 23 | 220 | 0.3 | 1.6 |
| z4ml | 54 | 92 | 21 | 93 | 0.4 | 1.0 |

Table 4.4: DMAP vs. MIS-II in cell and net count for MCNC benchmarks.

| Circuit Name | MIS-II Total Cell Area (a) | DMAP Total Cell Area (b) | (b)/(a) |
|---|---|---|---|
| 5xp1-hdl | 151 | 303 | 2.0 |
| 5xp1 | 287 | 385 | 1.3 |
| 9sym-hdl | 252 | 852 | 3.4 |
| 9sym | 47 | 821 | 1.7 |
| 9symml | 401 | 684 | 1.7 |
| alupla | 246 | 964 | 3.9 |
| bw | 503 | 516 | 1.0 |
| con1 | 41 | 67 | 1.6 |
| duke2 | 1249 | 2447 | 2.0 |
| f2 | 44 | 45 | 1.0 |
| f51m-hdl | 159 | 310 | 2.0 |
| f51m | 293 | 453 | 1.5 |
| misex1 | 159 | 206 | 1.3 |
| misex2 | 234 | 336 | 1.4 |
| misex3 | 1774 | 3419 | 1.9 |
| misex3c | 1256 | 2425 | 1.9 |
| rd53-hdl | 100 | 56 | 0.6 |
| rd53 | 104 | 86 | 0.8 |
| rd73-hdl | 159 | 461 | 2.9 |
| rd73 | 326 | 436 | 1.3 |
| rd84-hdl | 211 | 720 | 3.4 |
| rd84 | 685 | 1033 | 1.5 |
| sao2-hdl | 538 | 3433 | 6.4 |
| sao2 | 317 | 602 | 1.9 |
| seq | 5263 | 13590 | 2.6 |
| vg2 | 405 | 571 | 1.4 |
| z4ml-hdl | 102 | 393 | 3.9 |
| z4ml | 122 | 360 | 3.0 |

Table 4.5: DMAP vs. MIS-II in cell area for MCNC benchmarks.

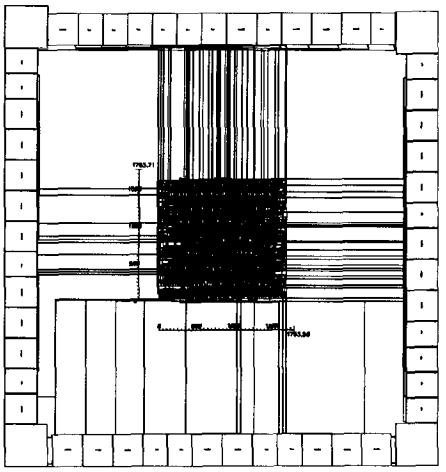


Figure 4.1: MIS-II mapped chip.

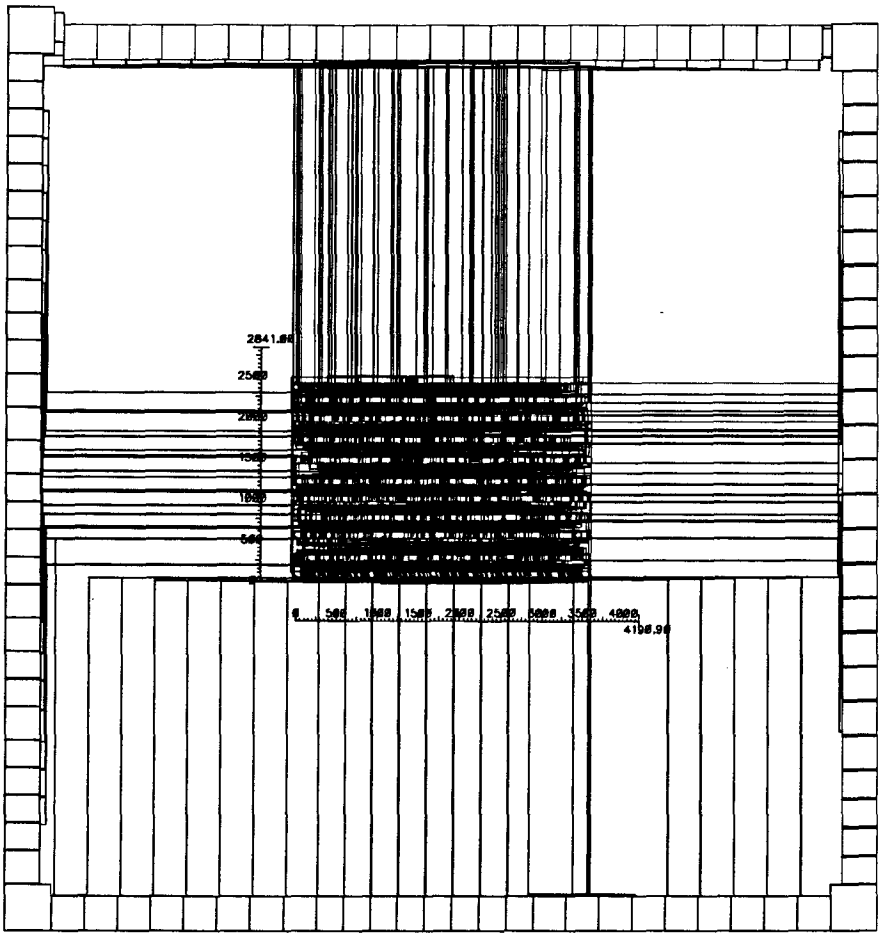


Figure 4.2: DMAP mapped chip.

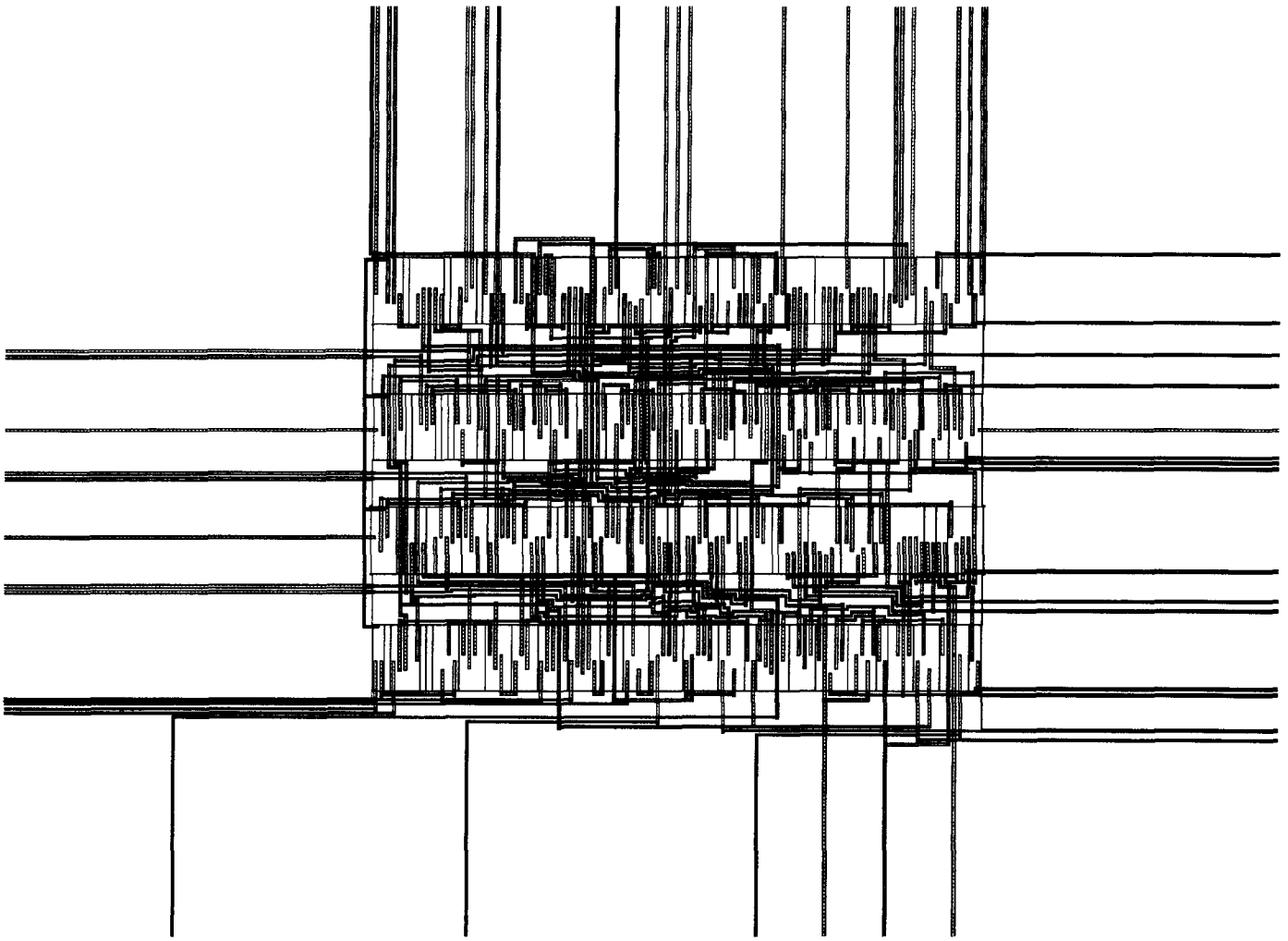


Figure 4.3: Core area of a MIS-II mapped chip.

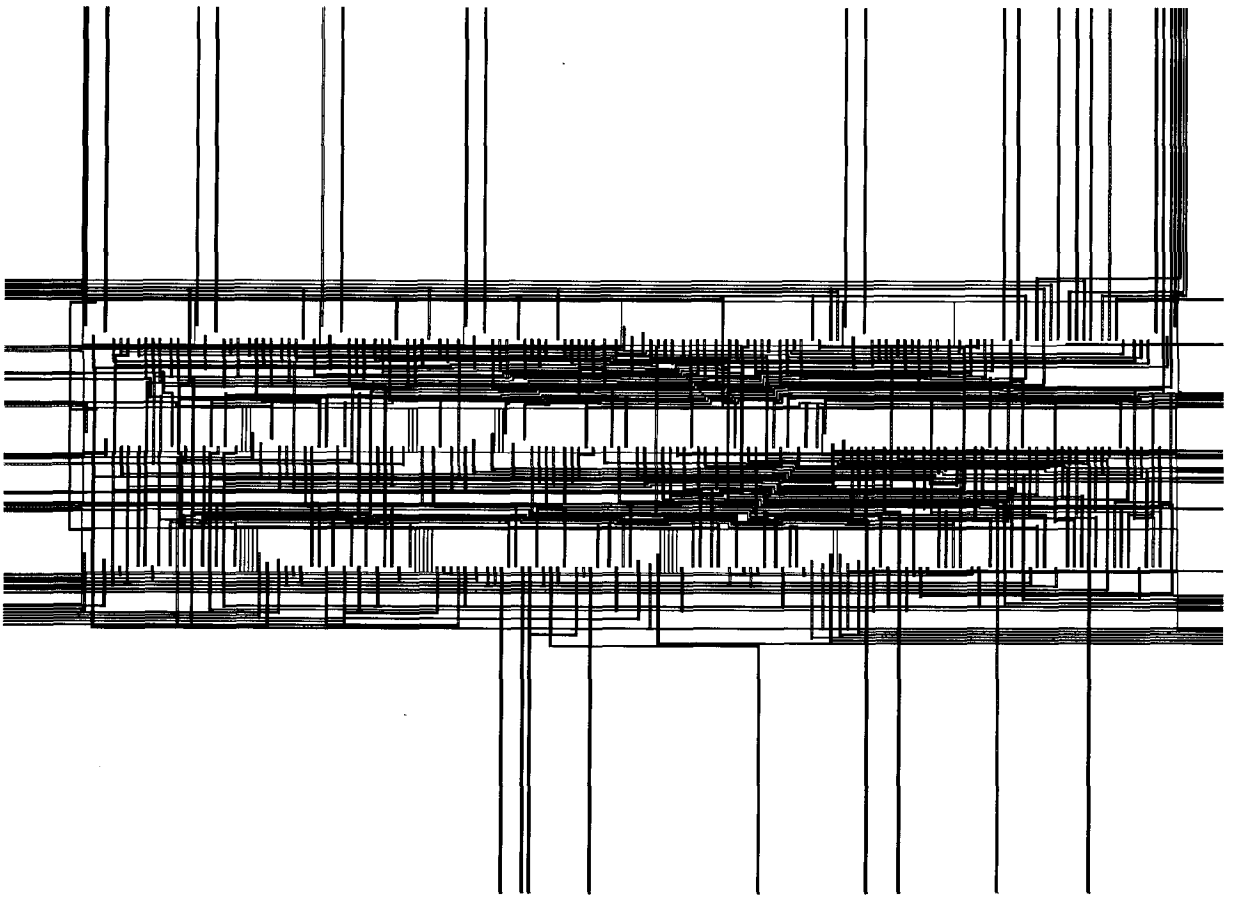


Figure 4.4: Core area of a DMAP mapped chip.

| Circuit Name | MIS-II Total Cell Area (a) | DMAP Total Cell Area (b) | (b)/(a) |
|---|---|---|---|
| C1355 | 1250 | 2272 | 1.8 |
| C17 | 12 | 20 | 1.6 |
| C1908 | 1164 | 2903 | 2.5 |
| C2670 | 1852 | 3895 | 2.1 |
| C3540 | 2721 | 6806 | 2.5 |
| C432 | 473 | 2466 | 5.2 |
| C499 | 1012 | 3409 | 3.4 |
| C5315 | 4103 | 12936 | 3.2 |
| C6288 | 4755 | 16135 | 3.4 |
| C7552 | 6350 | 12598 | 2.0 |
| C880 | 811 | 2143 | 2.6 |

Table 4.6: DMAP vs. MIS-II in cell area for ISCAS benchmarks.

# Chapter 5

## Conclusions and Future Work

This thesis addresses the problem of using CAD to implement combinational logic with DCVS. First, we looked at the design automation process for VLSI, and more specifically, the problems of technology mapping and layout synthesis. We examined dynamic DCVS logic, its operation, its distinguishing properties, and DCVS applications. The DCVS synthesis task was analyzed, and we outlined a system which produces DCVS circuits. DMAP, a technology mapping layout synthesis system for DCVS was presented in detail. The system performs the tasks of circuit clustering, transistor path finding, and cell layout connecting. Experimental results were given, and their implications discussed.

### 5.1 Main Contributions

The major contributions of this thesis are summarized and highlighted as follows.

- The problem of DCVS circuit synthesis was formulated. DCVS circuits are difficult to create because of the intricate internal connections and the dual-rail inputs and outputs associated with each cell.
- We outlined a solution to this problem by introducing an integrated system called DMAP. By generating as large as possible DCVS cells, the final circuit uses considerably less cells than if conventional technology mapping were used. As a result, the area penalty of using DCVS can be reduced because there are less wires to route in the circuit.

- The DMAP system was designed and implemented. This system takes any combinational logic block and outputs a DCVS circuit layout. More specifically, it performs the tasks of technology mapping and layout synthesis. Large DCVS circuits can be produced.

- An algorithm was developed for synthesizing the layout of the DCVS pull-down block in a systematic and straightforward manner. In fact, this algorithm can be applied to layout the pull-down block of *any* dynamic logic, such as NORA and DOMINO. At present, the synthesis of dynamic cell layout is still unchartered territory. DMAP's layout style can be easily extended to synthesize other types of dynamic cells.

- An algorithm for circuit clustering was developed based on Lawler's node-labeling procedure. It achieves minimum delay through the network and partitions the circuit into modules of sizable functions, subject to a set of constraints. This is a very useful algorithm that can be extended for use in other areas, such as the FPGA (*field programmable gate array*) technology mapping problem.

## 5.2 Future Directions

DCVS circuits are costly in terms of total area. However, we believe that the performance of DMAP can be improved through further work. In obtaining the DCVS pull-down tree, ordered BDDs were used. It is possible that unordered BDDs may give better results. Also, other methods of obtaining the DCVS pull-down tree, such as that in [16] can be experimented with. It is unclear which decomposed circuit form would better facilitate clustering. In DMAP, 2-input NANDs and 2-input NORs were used; other circuit decompositions should be experimented with as well. Using a more sophisticated layout style can improve the total cell area results.

One of the benefits with using a layout synthesis approach, rather than a standard cells approach for technology mapping, lies in the flexibility in cell style. In particular, a layout synthesis system can be modified to alter the sizes of different transistors in order to speed up

the circuit. In fact, the circuit speed can often be dramatically improved by increasing certain transistor widths. For the layout style we have chosen, it is easy to see how the output driving INVERTERs can be made wider without significantly increasing the cell area. Furthermore, it is also possible to reduce the delay in the pull-down network, and thus decrease the delay in the DCVS cell itself, by modifying the width of some of the input transistors. This later modification is also possible without a significant area penalty. For a practical DCVS technology mapper, this "size modification" addition to the mapper is necessary if the maximum speed advantage of DCVS logic is to be fully utilized.

Combining the ideas of dynamic logic synthesis, and the partitioning of circuits into blocks for programmable arrays, one interesting direction of development would be "dynamic programmable gate arrays". It is possible that the right-half portion of the cell can be replaced with a programmable gate array that can allow for any DCVS function to be programmed. Each "configurable logic block" will have a standard pull-up layout section to implement the dynamic precharging functions of the gate. The current one-dimensional transistor row layout can be extended to a structured array approach with input rails and routing rails. The circuit can be clustered into DCVS logic blocks in much the same way partitioning needs to be done for FPGAs. Also, it may be possible to consider the alignment of input variables between the arrays to facilitate global routing.

Dynamic devices such as DCVS are growing in importance and potential. As asynchronous theory is further developed and more discoveries into dynamic behavior are made, the use of DCVS will become increasingly popular. We need intelligent and flexible computer tools to help build combinational dynamic logic blocks. This thesis is an advance in that direction.

## Bibliography


[1] *Xilinx Programmable Gate Array User's Guide.* Xilinx, Inc., 1988.

[2] *Berkeley Logic Interface Format.* University of California, Berkeley, 1989.

[3] *Cadence Design Systems Users Manual.* 1989.

[4] *Electronic Design Interface Format 2 0 0.* 1989.

[5] *VHDL Hardware Description Language IEEE Standard 1076/87.* 1992.

[6] A.V. Aho and S.C. Johnson. Optimum code generation for expression trees. *ACM*, 7:488–501, 1976.

[7] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, CAD-27(6):509–516, August 1978.

[8] R. Bar-Yehuda, J.A. Feldman, R.Y. Pinter, and S. Wimer. Depth-first-search and dynamic programming algorithms for efficient CMOS cell generation. *IEEE Transactions on Computer-Aided Design*, 8(7):737–743, July 1989.

[9] J. Bhasker and S. Sahni. Optimal linear arrangement of circuit components. *Journal VLSI Computer Systems*, pages 87–109, 1987.

[10] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[11] R.K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[12] R.K. Brayton, G.D. Hatchel, and A. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[13] R.K. Brayton and C. McCullen. The decomposition and factorization of boolean expressions. *IEEE International Symposium on Circuits and Systems*, pages 49–54, May 1982.

[14] R. Bryant. Symbolic manipulation of boolean functions using a graphical representation. *Design Automation Conference*, pages 688–694, November 1985.

[15] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, CAD-35(8), August 1986.

[16] K.M. Chu and D.L. Pulfrey. Design procedures for differential cascode voltage switch circuits. *IEEE Journal of Solid-State Circuits*, SC-21(6):1082–1087, December 1986.

[17] K.M. Chu and D.L. Pulfrey. A comparison of CMOS circuit techniques: Differential cascode voltage switch logic versus conventional logic. *IEEE Journal of Solid-State Circuits*, SC-22:528–532, August 1987.

[18] E. Detjens, G. Gannot, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Design Automation Conference*, pages 116–119, 1987.

[19] D. Gajski (Editor). *Silicon Compilation.* Addison-Wedison Publishing Company, Inc., 1988.

[20] P.C. Elmendorf. KWIRE: A multiple-technology user-reconfigurable wiring tool forVLSI. *IBM Journal of Research and Development*, 28:603–612, 1984.

[21] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, CAD-39(5):710–713, May 1990.

[22] N.F. Goncalves and H.J. DeMan. NORA:a racefree dynamic CMOS technique for pipelined logic structures. *IEEE Journal of Solid-State Circuits*, pages 261–266, 1983.

[23] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. *Design Automation Conference*, 1971.

[24] L.G. Heller, W.R. Griffin, J.W. Davis, and N.G. Thomas. Cascode voltage switch logic: A differential CMOS logic family. *IEEE International Solid State Circuits Conference*, pages 16–17, February 1984.

[25] D.D. Hill. Sc2 - a hybrid automatic layout system. *IEEE International Conference on Computer-Aided Design*, pages 172–174, 1985.

[26] T.C. Hu. *Combinatorial Algorithms.* Addison Wesley, 1982.

[27] G. Jacobs and R.W. Brodersen. Self-timed integrted circuits for digital signal processing applications. *VLSI Signal Processing III, IEEE PRESS*, November 1988.

[28] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. *Design Automation Conference*, pages 341–347, 1987.

[29] K. Keutzer. Impact of library size on the quality of automated synthesis. *Design Automation Conference*, pages 120–123, 1987.

[30] R. H. Krambeck, C. M. Lee, and H-F. S. Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid-State Circuits*, SC-17(3):614–619, June 1982.

[31] E. Lawler, K. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Transactions on Computers*, pages 47–57, January 1969.

[32] S. Malik. Logic verification using binary decision diagrams in a logic synthesis environment. *IEEE International Conference on Computer-Aided Design*, November 1988.

[33] R.L. Maziasz and J.P. Hayes. Layout optimization of CMOS functional cells. *Design Automation Conference*, pages 544–551, 1987.

[34] P. C. McGeer. Robust path delay-fault testability for dynamic CMOS circuits. In *IEEE International Conference on Computer Design*, 1991.

[35] P. C. McGeer and Robert K. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *Design Automation Conference*, 1989.

[36] P. C. McGeer and Robert K. Brayton. Provably correct critical paths. In *Decennial CalTech VLSI Conference*, 1989.

[37] P. C. McGeer and Robert K. Brayton. Hazard prevention in combinational circuits. In *Hawaii International Conference on the System Sciences*, 1990.

[38] P. C. McGeer and Robert K. Brayton. Timing analysis on precharged-unate networks. *Design Automation Conference*, pages 124–129, 1990.

[39] P. C. McGeer and Robert K. Brayton. *Integrating Functional and Temporal Domains in Logic Design: The False Path Problem and its Implications.* Kluwer Academic Publishers, 1991.

[40] P. C. McGeer, Robert K. Brayton, Richard L. Rudell, and Alberto L. Sangiovanni-Vincentelli. Extended stuck-fault testability for combinational networks. *MIT Conference on Advanced Research in VLSI*, pages 239–259, 1990.

[41] T. Meng. *Synchronization Design for Digital Systems.* Kluwar Academic Publishers, Norwell, Massachusetts, 1991.

[42] T. Meng. private communication. 1992.

[43] T. Meng, R.W. Brodersen, and D.G. Messerschmitt. Implementation of high sampling rate adaptive filters using asynchronous design techniques. *VLSI Signal Processing III, IEEE PRESS*, November 1988.

[44] R. Muller and T. Lengauer. Linear algorithms for two CMOS layout problems. *Proc. Aegean Workshop on Computing*, July 1986.

[45] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli. On clustering for minimum delay/area. *IEEE International Conference on Computer-Aided Design*, pages 6–9, November 1991.

[46] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. *Design Automation Conference*, pages 620–625, 1990.

[47] R. Nair, A. Bruss, and J. Reif. *Linear Time Algorithms for Optimal CMOS Layout.* Elsevier, New York, 1985.

[48] N. Newell. Asynchronous control protocol for a DCVS logic micropipeline. *unpublished manuscript*, 1991.

[49] T. Nogatch and T. Hedges. Automated design of CMOS leaf cells. *VLSI Systems Design*, pages 66–78, November 1985.

[50] C.J. Poirier. Excellerator: Custom CMOS leaf cell layout generator. *IEEE Transactions on Computer-Aided Design*, 8(7):744–755, July 1989.

[51] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro. A new symbolic channel router: YACR2. *IEEE Transactions on Computers*, CAD-4(3):208–219, July 1985.

[52] E.M. Reingold, J. Nievergelt, and N. Deo. *Cominatorial Algorithms.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[53] M. Sarrafzadeh. Tree placement in cascode-switch macros. *Integration, the VLSI Journal*, (5):127–139, 1991.

[54] G. Saucier and G. Thuau. Sytematic and optimized layout of mos cells. *Design Automation Conference*, 1985.

[55] C.C. Su and M. Sarrafzadeh. Optimal gate-matrix layout of CMOS functional cells. *Integration, the VLSI Journal*, 9:3–23, 1990.

[56] I.E. Sutherland. Micropipelines. *ACM*, pages 729–738, 1989.

[57] H.J. Touati, H. Savoj, and R.K. Brayton. Delay optimization of combinatioal logic circuits by clustering and partial collapsing. *IEEE International Conference on Computer-Aided Design*, pages 118–121, November 1991.

[58] T. Uehara and W.M. vanCleemput. Optimal layout of CMOS functional arrays. *IEEE Transactions on Computers*, C-30(5):305–312, May 1981.

[59] T. Williams. A zero-overhead self-timed iterating ring implementating 54b division with 45ns to 160ns latency. *IEEE VLSI Workshop, Orlando Florida*, 1991.

[60] T. Williams. Analyzing and improving throughput in self-timed circuits and rings. *Proceedings of TAU '92*, 1992.

[61] S. Wimer, R.Y. Pinter, and J.A. Feldman. Optimal chaining of CMOS transistors in a functional cell. *IEEE Transactions on Computer-Aided Design*, CAD-6:795–801, 1987.

[62] O. Wing. Automated gate matrix layout. *IEEE International Symposium on Circuits and Systems*, pages 681–685, 1982.

[63] O. Wing. Interval-graph-based circuit layout. *IEEE International Conference on Computer-Aided Design*, pages 84–85, 1983.

[64] E.J. Yoffa and P.S. Hauge. Acorn: A system for CVS macro design by tree placement and tree customization. *IBM Journal of Research and Development*, 28:596–602, 1984.

[65] E.J. Yoffa and P.S. Hauge. Acorn: A local customization approach to DCVS physical design. *Design Automation Conference*, pages 32–38, 1985.

# Appendix A

## Algorithm for Finding an Euler Path

The following is a C-like pseudo-code algorithm for finding an Euler Path in a graph.

```
typedef struct vertex_struct vertex_t;
struct vertex_struct {
    char *name;
    linked list of edges Edges;
};
typedef struct edge_struct edge_t;
struct edge_struct {
    vertex_t *vertex1, *vertex2;
    char *label;
    linked list pointers *list_entries[3]; /* One on each vertex, and one globally */
};

linked list of edges global_edges;
edge_t *
traverse_arc(m)
vertex_t *m;
{
    edge_t *edge;

    edge = first arc incident on m to any node with at least two unused edges;
    if there are no such edges then
    edge = first arc incident on m
    delete edge from all three lists it appears on;
    return edge;
}

#define other_terminal(edge, vertex) (vertex == edge->vertex1?edge->vertex2:edge->vertex1)

euler_path(n)
vertex_t *n;
{
    edge_t *edge;
    edge = traverse_arc(n);
    m = other_terminal(edge, n);
    push edge on path;

    while(m != n) {
    edge = traverse_arc(m);
    push edge on path;
```

```
    m = other_terminal(edge, m);
    }

    while there are unused edges (edges left on global) {
    foreach edge on path {
        if either vertex of edge has any unused edges {
        let m be that vertex;
        path1 = euler_path(m);
        insert path1 on path after edge;
        }
    }
    }
    return path;
}
```

# Appendix B

## CADENCE Specific Notes

This document describes the processing steps that need to be done in CADENCE to convert the EDIF files output from the DMAP clustering program, to final chip layout incorporating of all the cells. The reader is assumed to be familiar with CADENCE and CADENCE's programming language SKILL.

The SKILL command ic("info-filename") takes as input a "info-filename" which contains the clustering summary file produced by the clustering program for each circuit. The command consists of two steps:

1. Fix Cells; To fix the individual cells so that they can be used for our purposes in CADENCE.
2. Make Chip; To create the chip out of those cells, and the lobal circuit netlist.

### B.1 Fixing Cells

1. **Fix jumper Pins: fixPin(blockname)**

   In CADENCE, IO pins are drawn in metal, and are distinguished as "terminals" by using the "pin" layer of that metal. In EDIF, IO pins are specified terminal ports, but there is no way to specify that the pin contains a geometry of pin layer (the clustering program simply gives it a bogus "wire" layer).

   The program edif2edge converts the EDIF file to EDGE Layout Rep. fixPin() searches for all the terminals in the Layout Rep., and creates the geometry of the metal piece. Access direction specifies which directions the global router can access the pin or port. These are added to the pins.

   All jumpers are named "jumperX" where X is the unique number in the cell. Jumpers are also declared as ports in EDIF, of layer "wire". They are searched for the string "jumper" in the name, and then made a pin.

2. **Generate Symbol Rep: gensCell(blockname)**

   This function opens the Layout Rep of the cell, reads in the input and output pin names, and creates lists for them. These two lists are used in calling the SKILL function GenS which creates

a symbol with this specified IO pins.

3. **Generate Abstract Rep: GenAb(blockname)**

   This function generates an Abstract Rep from the blockfile using the function abgen.

4. **Fix abstract Rep**

   After GenAb() is run, we need to change the "Representation/type" property of the Abstract Rep to "standard", so that it can be properly recognized.

## B.2 Making the Chip

After all the cells have been fixed, the following is done:

1. Run "edif2edge" on the global EDIF file. This creates a Netlist Rep of the circuit.
2. Run the SKILL function "globalize" on the Netlist Rep,

   so that CADENCE will join the vdd and gnd with the global vdd and gnd during P&R.
3. Generate a Symbol Rep for the circuit from the Netlist Rep. The function gensCct() scans the input block for terminals, and creates a list for the input and output pins. This is used to call the function GenS() to generate Symbol.
4. Generate a new schematic that will be the chip, containing the circuit netlist (placed as a Symbol), plus all the IO pads.

   The function AddPads(block "symbol" chip "schematic") opens a new Rep called chip/schematic. In this Rep, it creates and instance of block/symbol. Then it scans for all terminals in block/symbol, and adds input and output pads as required, including the gnd and vdd pads.