

Copyright © 1993, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**NOVEL TECHNIQUES FOR HIGH PERFORMANCE  
FIELD PROGRAMMABLE LOGIC DEVICES**

by

Narasimha B. Bhat

Memorandum No. UCB/ERL M93/80

19 November 1993

**NOVEL TECHNIQUES FOR HIGH PERFORMANCE  
FIELD PROGRAMMABLE LOGIC DEVICES**

Copyright © 1993

by

Narasimha B. Bhat

Memorandum No. UCB/ERL M93/80

19 November 1993

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

## Abstract

# Novel Techniques for High Performance Field Programmable Logic Devices

by


Narasimha B. Bhat

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Ernest S. Kuh, Chair

Field programmable logic devices (FPLDs) are fast emerging as viable alternatives to mask programmed parts because of their rapid time-to-market and low costs. Their application has, however, been limited to implementing random logic, with non-critical timing specifications. This work attempts to advance FPLD usage to high-performance applications as well. A two-pronged approach is adopted. In the first part, CAD algorithms aimed at improving routability and performance of designs mapped to existing look-up table (LUT) based field programmable gate arrays (FPGAs) are developed. The concept of a two-input LUT primitive cell is introduced. This reduces the number of library patterns that require to be stored for an LUT library, and makes it feasible to extend performance-driven library-based technology mapping techniques to LUT FPGAs. The performance-driven mapping algorithm accounts for interconnect delay and provides area-delay trade-offs. Experiments on benchmark designs show the effectiveness of the new algorithms. In the second part, a new FPLD architecture is introduced. The architecture is based on the concept of time-sharing of logic and routing resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and efficient silicon usage. Real-time reconfiguration of logic and routing resources implements a given circuit in a folded pipe-line fashion, pipe-lining at the gate level. Several possible variations of the basic architecture are discussed. A simple synthesis scheme is developed and experimental results are reported. Area and timing analyses demonstrate advantages over existing FPGAs.



Prof. Ernest S. Kuh  
Thesis Committee Chairman

This work is a humble offering to my beloved parents  
**Sri Balakrishna and Smt. Premaleela Bhat**

# Contents

<b>Table of Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>xi</b>
<b>Acknowledgements</b>	<b>xiii</b>
<b>1 Introduction and Overview</b>	<b>1</b>
1.1 FPLD classification . . . . .	3
1.2 CAD for FPLDs . . . . .	5
1.3 Programmable logic system . . . . .	9
1.4 FPLD advantages . . . . .	10
1.5 Focus . . . . .	10
1.6 Thesis overview . . . . .	12
<b>2 Terminology</b>	<b>15</b>
<b>I Performance-oriented Synthesis</b>	<b>23</b>
<b>3 Overview</b>	<b>25</b>
3.1 Performance-oriented mapping . . . . .	26
3.2 Routable mapping . . . . .	27
3.3 LUTs and logic modules . . . . .	28
<b>4 Routable Technology Mapping</b>	<b>29</b>
4.1 Motivation . . . . .	29
4.1.1 Cause of incomplete routing . . . . .	30
4.1.2 Problem with existing CAD tools . . . . .	30
4.1.3 Our strategy . . . . .	31
4.2 Terminology . . . . .	34
4.3 FPGA Model . . . . .	35
4.3.1 FPGA model . . . . .	35

4.3.2	qblock . . . . .	36
4.3.3	Look-up table . . . . .	38
4.3.4	Routing resources . . . . .	38
4.4	Overview of our approach . . . . .	38
4.5	RFR algorithm . . . . .	39
4.5.1	Decomposition into IOLs . . . . .	40
4.5.2	Global routing . . . . .	40
4.5.3	Checking logic feasibility . . . . .	41
4.5.4	Simulated annealing . . . . .	42
4.5.5	Re-synthesizing a packing solution . . . . .	43
4.6	Experimental results . . . . .	43
4.7	Conclusions . . . . .	48
<b>5</b>	<b>Performance-oriented Mapping</b>	<b>51</b>
5.1	Motivation . . . . .	51
5.2	Terminology . . . . .	55
5.3	Review of library-based mapping . . . . .	55
5.3.1	Tree covering algorithm . . . . .	56
5.3.2	Partitioning DAGs . . . . .	59
5.4	Why has this not been done before? . . . . .	59
5.5	TIL based LUT library . . . . .	60
5.5.1	TILs as primitive cells . . . . .	60
5.5.2	Small library . . . . .	61
5.6	dpmap Algorithm . . . . .	63
5.6.1	Input network $\rightarrow$ TINN . . . . .	63
5.6.2	Tree covering . . . . .	65
5.6.3	Handling reconvergent fanout . . . . .	65
5.7	Fast tree matching . . . . .	67
5.7.1	Principle . . . . .	68
5.7.2	Example . . . . .	69
5.7.3	Drawbacks . . . . .	72
5.8	Experimental results . . . . .	74
5.8.1	Performance-driven mapping . . . . .	76
5.8.2	Area-delay trade-off . . . . .	76
5.8.3	Minimum depth . . . . .	76
5.8.4	Minimum area . . . . .	79
5.9	Conclusions . . . . .	80
<b>II</b>	<b>Performance-oriented Architecture</b>	<b>81</b>
<b>6</b>	<b>Overview</b>	<b>83</b>

<b>7</b>	<b>Why Another Architecture?</b>	<b>85</b>
7.1	Routing issues . . . . .	86
7.1.1	Architecture approach . . . . .	86
7.1.2	CAD approach . . . . .	87
7.1.3	Summary . . . . .	90
7.2	Timing issues . . . . .	90
7.2.1	Dominant interconnect delay . . . . .	90
7.2.2	Widely varying interconnect delay . . . . .	92
7.3	CAD issues . . . . .	92
7.4	Desired features . . . . .	93
7.5	New architecture . . . . .	94
<b>8</b>	<b>New Architecture</b>	<b>95</b>
8.1	Principle of levelize and hold . . . . .	95
8.1.1	Signal propagation in logic circuits . . . . .	95
8.1.2	Levelize-and-hold . . . . .	99
8.1.3	Sequential elements . . . . .	100
8.2	<i>Dharma</i> Architecture . . . . .	100
8.2.1	High-level block diagram . . . . .	100
8.2.2	Detailed block diagram . . . . .	103
8.2.3	Clocking scheme . . . . .	105
8.2.4	Multi-chip operation . . . . .	106
8.2.5	Dynamic logic modules . . . . .	108
8.2.6	Pass-buffers . . . . .	109
8.2.7	Dynamic interconnection array . . . . .	110
8.2.8	Latches . . . . .	112
8.2.9	Level generation circuitry . . . . .	112
8.3	Operating procedure . . . . .	115
<b>9</b>	<b>Examples to Illustrate <i>Dharma</i></b>	<b>119</b>
9.1	<i>Simple Dharma</i> . . . . .	119
9.2	Combinational example 1 . . . . .	120
9.3	Combinational example 2 . . . . .	123
9.4	Sequential example . . . . .	127
<b>10</b>	<b>Variations and Modifications</b>	<b>139</b>
10.1	Dynamic logic module . . . . .	139
10.1.1	Complex LUT DLMs . . . . .	140
10.1.2	Multiplexer based DLMs . . . . .	141
10.1.3	PLA based DLMs . . . . .	141
10.1.4	Heterogeneous DLM array . . . . .	141
10.1.5	Technology dependent modifications . . . . .	143
10.2	Dynamic interconnect array . . . . .	143
10.2.1	Shift register at crosspoint . . . . .	143
10.2.2	Multiplexer based crossbar . . . . .	145



10.2.3	Partially hard-wired DIA . . . . .	146
10.2.4	Binomial concentrator based DIA . . . . .	146
10.2.5	DIA as a <i>Copier</i> . . . . .	149
10.2.6	Sparsely connected <i>K</i> -class DIA . . . . .	150
10.2.7	Functional DIA . . . . .	153
10.2.8	Technology dependent modifications . . . . .	154
10.3	Level expander . . . . .	154
10.3.1	Example . . . . .	154
10.3.2	Expander circuitry . . . . .	156
10.3.3	Local level generator . . . . .	158
<b>11</b>	<b>Synthesis for <i>Dharma</i></b>	<b>161</b>
11.1	Introduction . . . . .	161
11.1.1	Topological levelization . . . . .	162
11.1.2	Temporal partitioning . . . . .	164
11.1.3	Handling other designs . . . . .	165
11.2	Levelization as an integer linear program . . . . .	165
11.2.1	Upper and lower level limits . . . . .	166
11.2.2	Level variable . . . . .	166
11.2.3	Level of a node . . . . .	166
11.2.4	Fanin constraints . . . . .	167
11.2.5	Nodes per level . . . . .	167
11.2.6	Complete ILP . . . . .	167
11.3	Temporal partitioning . . . . .	168
11.3.1	Uniform DLM distribution . . . . .	168
11.3.2	Modified Kernighan-Lin heuristic . . . . .	169
11.3.3	Move selection . . . . .	171
11.4	Experimental results . . . . .	172
11.5	Conclusions . . . . .	174
<b>12</b>	<b>Architecture Analysis</b>	<b>177</b>
12.1	Area analysis . . . . .	177
12.1.1	Assumptions . . . . .	177
12.1.2	Area components of <i>Dharma</i> . . . . .	178
12.1.3	The analysis . . . . .	179
12.1.4	Comparison with Xilinx 2000 series . . . . .	181
12.1.5	Comparison with Xilinx 3000 series . . . . .	183
12.1.6	Comparison with Xilinx 4000 series . . . . .	186
12.1.7	Comparison with the AT&T ORCA . . . . .	186
12.2	Timing analysis . . . . .	187
12.2.1	Notation . . . . .	187
12.2.2	EFP timing equation . . . . .	188
12.2.3	S timing equation . . . . .	190
12.2.4	Remarks . . . . .	192
12.3	Experimental verification . . . . .	193

12.4 Conclusions . . . . .	195
<b>13 Unexplored Terrain</b>	<b>197</b>
13.1 Logic synthesis for <i>Dharma</i> . . . . .	197
13.2 Repetitive structures . . . . .	201
13.2.1 Principle . . . . .	201
13.2.2 Example . . . . .	201
13.2.3 Remarks . . . . .	203
<b>14 Conclusions</b>	<b>207</b>
<b>Bibliography</b>	<b>209</b>

# List of Figures

1.1	A PLD . . . . .	2
1.2	Xilinx LCA . . . . .	4
1.3	FPLD classification . . . . .	6
1.4	FPLD design flow . . . . .	7
1.5	FPGA cost advantage over MPGA . . . . .	11
2.1	A Boolean Network . . . . .	16
2.2	Decomposition . . . . .	17
2.3	Look-up table (LUT) . . . . .	18
2.4	Worst-case delay analysis . . . . .	21
4.1	$\mathcal{M}_1$ implemented on $\mathcal{P}$ . . . . .	32
4.2	$\mathcal{M}_2$ implemented on $\mathcal{P}$ . . . . .	33
4.3	An RST . . . . .	35
4.4	FPGA model . . . . .	36
4.5	Internals of a qblock . . . . .	37
4.6	Overview of RFR . . . . .	39
4.7	Effect of integrating physical design and logic synthesis . . . . .	45
4.8	Wire-length increase for constant qblocks . . . . .	46
4.9	Routing congestion before and after running <b>RFR</b> . . . . .	47
5.1	Symbolic representation of a 2-input LUT (TIL) . . . . .	55
5.2	Example gate library with 4 gates . . . . .	57
5.3	Tree covering for standard cells . . . . .	58
5.4	Partitioning a DAG by a break at every multi-fanout node . . . . .	59
5.5	3-input LUT as trees of 2-input LUTs . . . . .	61
5.6	Library patterns for a 4-input LUT mapping . . . . .	62
5.7	Input network $\rightarrow$ TINN . . . . .	64
5.8	4-input LUT mapping . . . . .	66
5.9	Reconverging fanout example . . . . .	67
5.10	A 5-input LUT pattern . . . . .	68
5.11	Example to illustrate fast tree-matching . . . . .	70
5.12	Computing cost $C_{13}$ . . . . .	71
5.13	Computing cost $C_{22}$ . . . . .	72

5.14	Computing cost $C_{31}$ . . . . .	73
5.15	Example to illustrate the fast matching drawback . . . . .	75
5.16	dpmap's area-delay trade-off solutions . . . . .	77
7.1	XC4000 family routing resources . . . . .	88
7.2	AT&T ORCA family routing resources . . . . .	89
8.1	A general model of a sequential logic circuit. . . . .	96
8.2	Multi-level implementation of combinational logic block. . . . .	97
8.3	Signal propagation and identification of 'idle' resources . . . . .	98
8.4	Levelize and hold . . . . .	99
8.5	High level block diagram of <i>Dharma</i> . . . . .	101
8.6	Detailed block diagram of <i>Dharma</i> . . . . .	104
8.7	Clocking schemes . . . . .	107
8.8	Circuits with $l > L$ can be implemented on multiple <i>Dharmas</i> . . . . .	108
8.9	Dynamic logic module architecture . . . . .	109
8.10	Dynamic interconnection block architecture . . . . .	111
8.11	Block diagram of level generation circuitry. . . . .	113
8.12	Timing waveforms for multi-chip operation. . . . .	115
8.13	Timing diagram to illustrate <i>Dharma's</i> operation . . . . .	117
9.1	<i>Simple Dharma</i> . . . . .	121
9.2	Schematic diagram for Example 1 . . . . .	121
9.3	DLM values for Example 1 . . . . .	122
9.4	Example 1. DIA: Primary inputs $\rightarrow$ level 1; DLMs : Level 2. . . . .	124
9.5	Example 1. DIA: Level 1 $\rightarrow$ Level 2; DLMs : Level 1. . . . .	125
9.6	Schematic diagram for Example 2 . . . . .	126
9.7	DLM values for Example 2 . . . . .	128
9.8	Example 2. DIA: Primary inputs $\rightarrow$ Level 1; DLMs : Level 4. . . . .	129
9.9	Example 2. DIA: Level 1 $\rightarrow$ Level 2; DLMs : Level 1. . . . .	130
9.10	Example 2. DIA: Level 2 $\rightarrow$ Level 3; DLMs: Level 2. . . . .	131
9.11	Example 2. DIA: Level 3 $\rightarrow$ Level 4; DLMs: Level 3. . . . .	132
9.12	Sequential Example . . . . .	133
9.13	DLM values for the sequential example . . . . .	135
9.14	Sequential Example. DIA: Primary inputs, state variables $\rightarrow$ Level 1; DLMs: Level 3. . . . .	136
9.15	Sequential Example. DIA: Level 1 $\rightarrow$ Level 2; DLMs: Level 1. . . . .	137
9.16	Sequential Example. DIA: Level 2 $\rightarrow$ Level 3; DLMs: Level 2. . . . .	138
10.1	An example of a complex LUT DLM . . . . .	140
10.2	An example of a PLA based DLM . . . . .	142
10.3	A heterogeneous DLM array . . . . .	142
10.4	An example of $L$ -crossbar which uses shift registers . . . . .	144
10.5	Hard-wire connections from a DLM to adjacent DLMs. . . . .	147
10.6	Hard-wire connections reduce $L$ -crossbar crosspoints. . . . .	148

10.7	A 15-to-4 binomial concentrator . . . . .	149
10.8	DIA as a <i>Copier</i> . . . . .	151
10.9	K-class DLMs reduce crosspoints . . . . .	152
10.10A	DIA with wire-AND capability . . . . .	153
10.11	An example to illustrate level expansion . . . . .	155
10.12	Modified <i>Dharma</i> with level expansion capability . . . . .	157
10.13	The <i>Local Level Generation</i> circuitry . . . . .	158
11.1	Different levelization techniques . . . . .	163
11.2	Temporal partitioning procedure . . . . .	170
11.3	Move selection procedure . . . . .	173
11.4	DLM distribution before and after temporal partitioning . . . . .	175
12.1	Input to output signal path for EFP clocking scheme . . . . .	189
12.2	Input to output signal path for S clocking scheme . . . . .	191
13.1	Unbalanced and balanced implementation of function $y$ . . . . .	199
13.2	Example of balanced decomposition. . . . .	200
13.3	Tiny <i>Dharma</i> with VSC. . . . .	202
13.4	An example DAG with repetitive structures. . . . .	204

# List of Tables

4.1	Updating $U_i$ and $U_o$ , considering input nets . . . . .	41
4.2	Updating $U_i$ and $U_o$ , considering output net . . . . .	42
4.3	Results of running <b>RFR</b> for XC3000 . . . . .	48
5.1	3 level benchmark circuits . . . . .	53
5.2	4 level benchmark circuits . . . . .	53
5.3	Previous library sizes, for $K$ -input LUT mapping . . . . .	60
5.4	Number of $K$ -input LUT patterns . . . . .	61
5.5	<b>dpmmap</b> 's performance-driven mapping results . . . . .	76
5.6	<b>dpmmap</b> 's minimum-depth mapping solutions . . . . .	78
5.7	<b>dpmmap</b> 's minimum-area mapping solutions . . . . .	79
7.1	Dominant interconnect delay . . . . .	91
11.1	DLM distribution statistics before and after temporal partitioning . . . . .	176
12.1	Number of configuration bits . . . . .	179
12.2	Number of pass transistors . . . . .	180
12.3	Number of routing lines . . . . .	180
12.4	Number of latches . . . . .	180
12.5	<i>Dharma</i> 's area comparison with XC2064 . . . . .	182
12.6	<i>Dharma</i> 's area comparison with XC2018 . . . . .	183
12.7	<i>Dharma</i> 's area comparison with XC3090 . . . . .	184
12.8	Gate equivalent area comparison with XC3090 . . . . .	185
12.9	$K$ -class DIA fares better . . . . .	185
12.10	<i>Dharma</i> 's configuration bits comparison with XC4000 . . . . .	186
12.11	<i>Dharma</i> 's configuration bits comparison with AT&T ORCA . . . . .	187
12.12	<i>Dharma</i> 's delay comparison with Xilinx 3000 series . . . . .	194

TABLES TO BE REFERRED TO

1	.....	1
2	.....	2
3	.....	3
4	.....	4
5	.....	5
6	.....	6
7	.....	7
8	.....	8
9	.....	9
10	.....	10
11	.....	11
12	.....	12
13	.....	13
14	.....	14
15	.....	15
16	.....	16
17	.....	17
18	.....	18
19	.....	19
20	.....	20
21	.....	21
22	.....	22
23	.....	23
24	.....	24
25	.....	25
26	.....	26
27	.....	27
28	.....	28
29	.....	29
30	.....	30
31	.....	31
32	.....	32
33	.....	33
34	.....	34
35	.....	35
36	.....	36
37	.....	37
38	.....	38
39	.....	39
40	.....	40
41	.....	41
42	.....	42
43	.....	43
44	.....	44
45	.....	45
46	.....	46
47	.....	47
48	.....	48
49	.....	49
50	.....	50
51	.....	51
52	.....	52
53	.....	53
54	.....	54
55	.....	55
56	.....	56
57	.....	57
58	.....	58
59	.....	59
60	.....	60
61	.....	61
62	.....	62
63	.....	63
64	.....	64
65	.....	65
66	.....	66
67	.....	67
68	.....	68
69	.....	69
70	.....	70
71	.....	71
72	.....	72
73	.....	73
74	.....	74
75	.....	75
76	.....	76
77	.....	77
78	.....	78
79	.....	79
80	.....	80
81	.....	81
82	.....	82
83	.....	83
84	.....	84
85	.....	85
86	.....	86
87	.....	87
88	.....	88
89	.....	89
90	.....	90
91	.....	91
92	.....	92
93	.....	93
94	.....	94
95	.....	95
96	.....	96
97	.....	97
98	.....	98
99	.....	99
100	.....	100

## Acknowledgements

I am grateful to my research advisor, Prof. Ernest. S. Kuh, for his guidance and constant support during my stay at Berkeley. His encouragement has been the driving force behind my graduate studies.

Prof. Robert. K. Brayton took on the responsibility of being my ‘unofficial’ research advisor, and guided my last year at Berkeley. He encouraged the *Dharma* idea and suggested that it be patented. I would like to thank him for many useful discussions, suggestions and help.

I thank Prof. Ian Adler, of IEOR, for being on my qualifying examination and thesis committees. Thanks to Prof. Jan Rabaey for being on my qualifying examination committee and giving me useful suggestions.

Thanks to Dr. Dwight Hill for being my mentor at AT&T Bell Labs, and introducing me to FPGA research. Massoud Pedram taught me how to do research. I am indebted to him for his support and friendly guidance. Thanks to Kamal Chaudhary for many hours of stimulating discussions on a wide variety of topics.

I would like to thank Dr. Ulrich Lauther and Prof. Margaret Marek-Sadowska for helpful discussions. Thanks to Rajeev Murgai and Narendra Shenoy for useful feedback on my thesis proposals. Premal Buch, Charles Hough, Michael Jackson, Shen Lin, Minshine Shih, Arvind Srinivasan, Debbie Wang, Mei Xiao and Tianxiong Xue (David) made life in Kuh’s group interesting and lively. Thanks to all of them. Thanks to Premal for reading my thesis. Special thanks to Tahani Sticpewich for helping me with travel grants, mailing, and most everything, and making life a lot easier. I would also like to thank George Carvalho, Vijay Maheshwari and Tan Wee-Chiew.

Special thanks to Sriram Krishnan, Amit Marathe, Rajeev Murgai, Jagesh Sanghavi and Narendra Shenoy for their friendship and company. Thanks to Deepak and Rekha, Arvind and Sandhya, and Kamal and Vijay for many enjoyable moments. My heartfelt gratitude to my local guardians (and cousins) Dr. Suresh and Rekha Nayak, and their family, for providing me with much needed emotional support and love, and giving me a home away from home. Thanks to my cousins Dr. Rajendra and Anita Bhat and their family.

My uncle, Dr. Umeshraya Pai, and aunt, Nalini, deserve special mention. Their love and sincere well-wishes are really appreciated. They carried their role of surrogate parents to the full. My parents, Balakrishna and Premaleela Bhat, and my brothers Prashanth and Srikanth, gave me monumental support, encouragement, and enthusiasm in my work.



My wife, Shashikala provided me encouragement and support, especially when I needed it most. I am forever indebted to all of them.

This research was supported in part by the SRC, under grant number 92-DC008 and 93-DC008. This support is gratefully acknowledged.

# Chapter 1

## Introduction and Overview

This thesis is about field programmable logic devices (FPLDs). As their name implies, these devices are integrated circuits that can be programmed (i.e., configured) by the user to implement digital logic circuits. First introduced in the mid-70's, they provided a means for rapid prototyping of simple (consisting of a few 100 gates) logic circuits. Over the years, these simple prototyping devices have evolved into sophisticated devices which can be programmed to implement logic circuits having several thousand gates.

Programmable logic technology was dominated in the past by devices based on a PLA-type architecture. The basic PLA architecture consists of a programmable AND block (popularly called AND plane) and a programmable OR block (OR plane). A given circuit to be implemented using such a device, is first represented in a two-level AND-OR form. The PLA AND-plane is then configured to implement the AND part, and the PLA OR-plane is configured to implement the OR part. Variations and evolutions of this basic architecture have resulted in several kinds of programmable devices. Some have a fixed OR plane instead of a programmable one; some use a NOR-NOR representation or NAND-NAND representation, some use more complex gates instead of a basic AND gate for the product term, similarly some use complex gates instead of the basic OR gate for the sum term and some provide features such as invertible outputs, registered outputs, configurable I/O, etc. All these devices, which *look* like a PLA, are popularly called PLDs. Figure 1.1 shows a logic diagram of a popular PLD.

With advances in IC technology, it has become feasible to produce larger and larger devices based on the PLA-style architecture. However, in terms of silicon usage and speed of circuit operation, it is not efficient to implement large logic circuits by means of

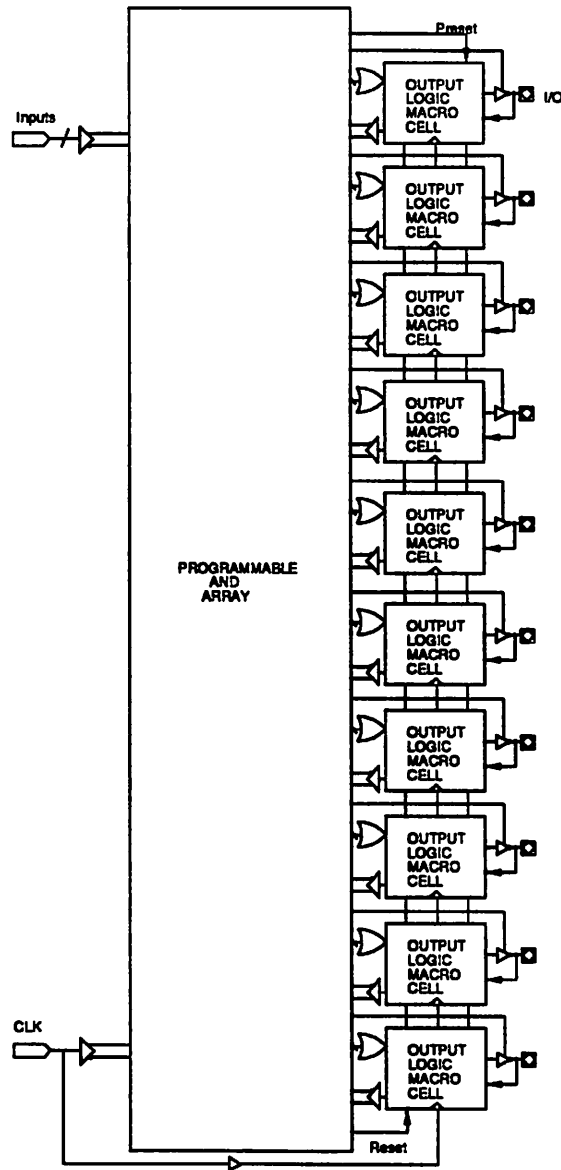


Figure 1.1: Logic diagram of the 22V10, a popularly used PLD

---

an AND-OR type of two-level implementation. In the case of large circuits, a multi-level implementation can be a better solution. Just as the foundry-based design styles evolved from the PLA into the standard cell and gate-array styles, so also the programmable logic technology evolved from PLDs into Field Programmable Gate Arrays (FPGAs).

FPGAs consist of a two-dimensional array of configurable logic cells, connected by means of a configurable interconnection network. The first of the current FPGA families, the logic cell array (LCA) was introduced by Xilinx, San Jose, CA, USA in 1985. In 1988 Actel introduced a configurable array architecture based on the structure of a channeled gate array. Today, the third generation of FPGAs is in the market. Figure 1.2 shows the Xilinx LCA.

FPLDs vary not only in architecture, but also in the programming technology. Early PLDs use bipolar technology, similar to PROMs, and are one-time programmable. These are manufactured with all connections intact, and programming consists of “blowing off” undesired connections. The configurable connections have a fuse link that is “blown off” in the programming process. With advance in technology, PLDs with erasable programming, like the Altera EPLD [Altera 85], are now available. These devices use EEPROM technology, and can be programmed several times. FPGAs also come in two flavors: the one-time programmable and the re-programmable kinds. Actel uses a new technology, called the anti-fuse, which is used to make a connection (as opposed to break a connection). The Actel devices are one-time programmable. Xilinx uses an SRAM technology to store the logic cell and interconnection configurations; the Xilinx devices can be programmed many times.

## 1.1 FPLD classification

There are several FPLD vendors, and many different kinds of devices in the market today. In Figure 1.3, an attempt has been made to classify the various FPLDs based on their underlying basic architecture. The chief purpose of Figure 1.3 is to give a feel for the different devices already existing, and also put in perspective, the new FPLD architecture being introduced in this thesis. It should be noted that Figure 1.3 is not intended to be complete, nor is it the only way to classify FPLDs. No standards for FPLDs have been established, and vendors have come up with ad-hoc nomenclatures that could be quite confusing. For example, a PLA-type architecture, with the output of the OR plane feeding

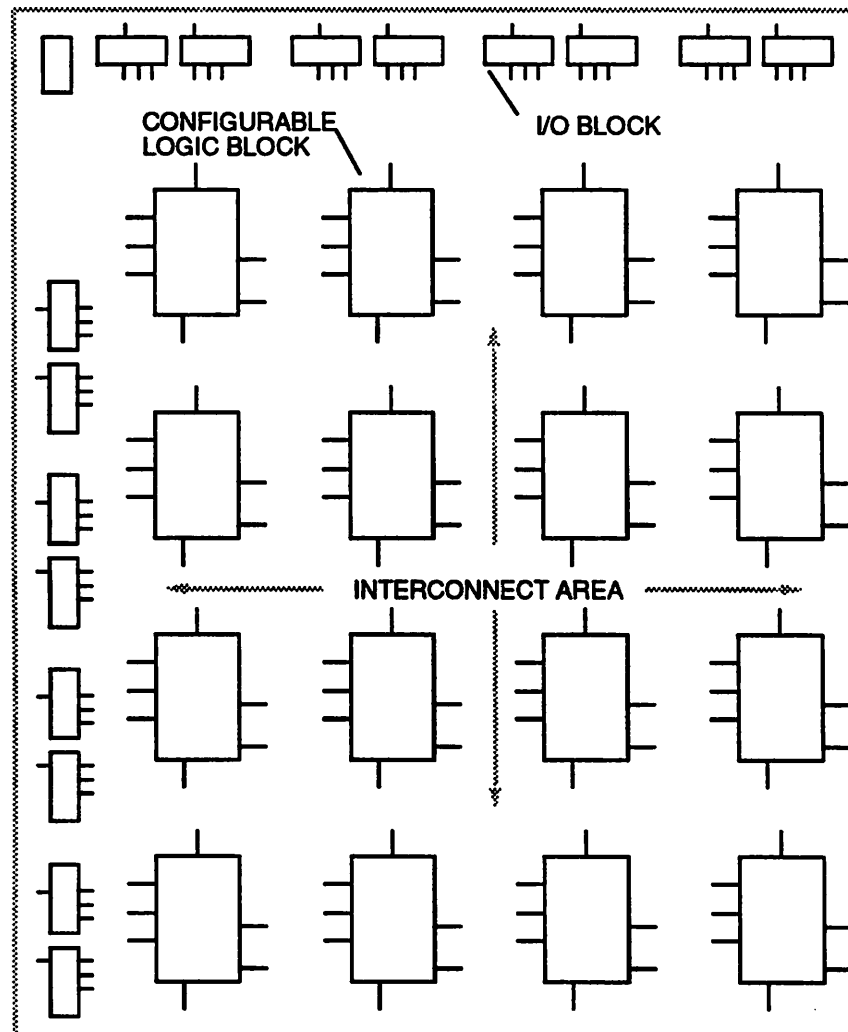


Figure 1.2: The Xilinx LCA

back into the AND plane can be used to implement multi-level logic circuits, and FPLDs with such an architecture are also called FPGAs by their vendors. However, for the purposes of this thesis, 'FPGA' is used to mean a device having a two-dimensional array architecture, consisting of configurable logic cells.

Figure 1.3 is based on Charles Small's [Small 91] high-density FPLD family tree. For this classification, only FPLDs with capacity to implement large logic circuits are considered. Prior to the new FPLD architecture of this thesis, the devices could broadly be classified into two: those that use the PLA-style architecture (classified as PLDs in the figure), and those that use the two-dimensional array (classified as FPGAs in the figure). PLDs can be further sub-divided based on the AND plane and OR plane architectures and FPGAs can be sub-divided based on the granularity of the logic cell. The *Dharma* architecture is a new programmable style, which performs a real-time re-configuration of the logic cells and interconnection patterns, so as to implement a multi-level circuit in a two-level manner. Detailed description of *Dharma* forms part II of this thesis. Since the *Dharma* style does not fall under either the FPGA or the PLD, a new branch at the root needs to be grown. It is possible that the *Dharma* architecture style will foster several devices varying in type of logic cell structure, interconnection techniques, etc.

## 1.2 CAD for FPLDs

The highly complex nature of current FPLDs necessitates the need for computer aided design (CAD) tools to design with, and program, such devices. The large number of possible interconnection strategies, and also the many different ways of distributing a given logic circuit among the configurable logic cells of FPLDs, makes it almost impossible to manage manually. Hence CAD tools become a necessary accessory to FPLDs.

Figure 1.4 is a flowchart to illustrate the various phases in the design of a circuit using FPLDs. In the figure, the logic optimization, mapping into FPLD modules, placement and routing, and the device programming steps are accomplished using CAD tools. The logic optimization and mapping steps are together referred to as logic synthesis. The design begins with a circuit specification, in terms of a register-transfer level description or a circuit schematic diagram. Using additional tools (not shown in Figure 1.4), one can also specify the circuit by means of a high level description (behavioral specification). The logic optimization tool takes as input the register-transfer level description. Logic

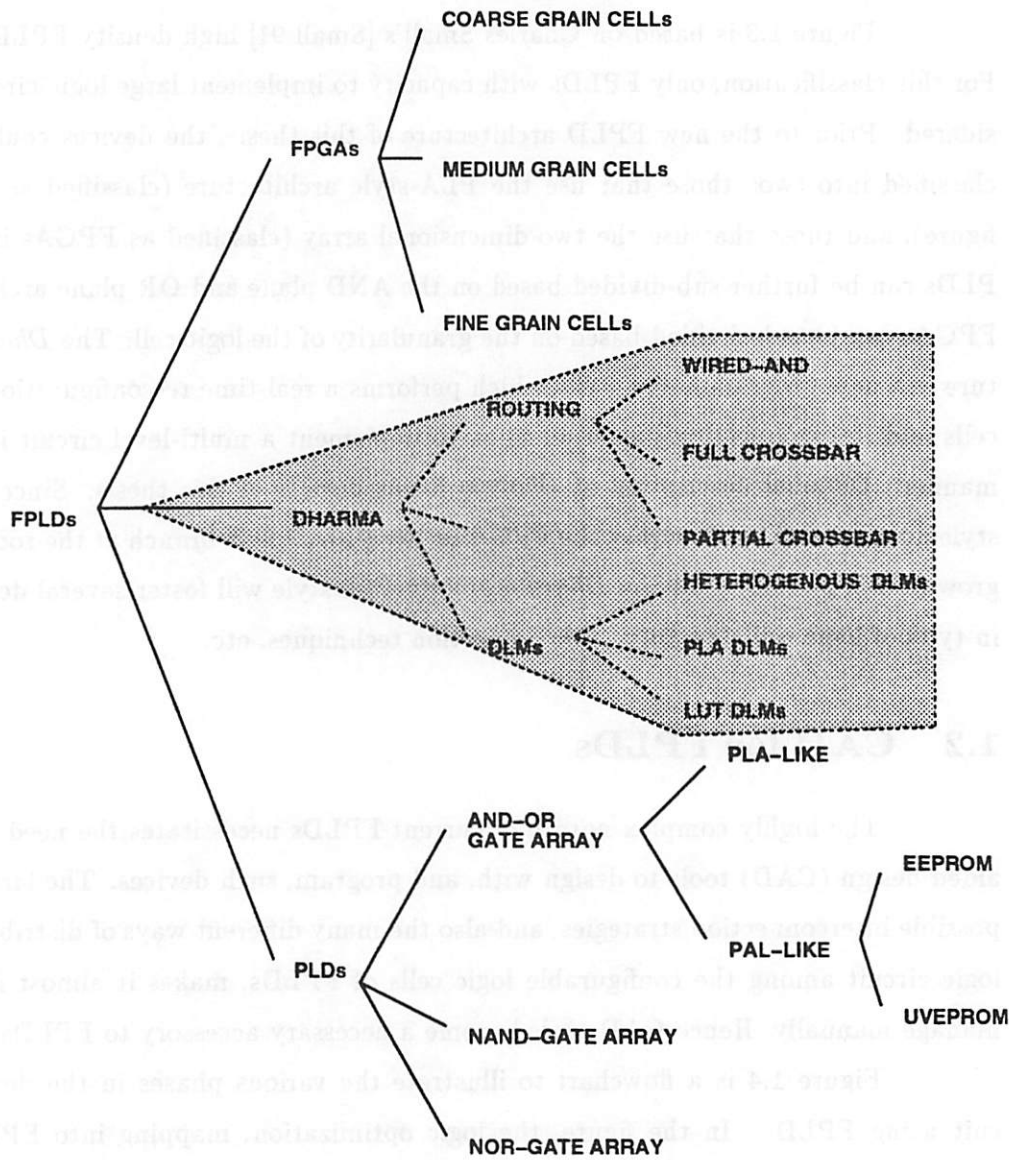


Figure 1.3: FPLD Classification

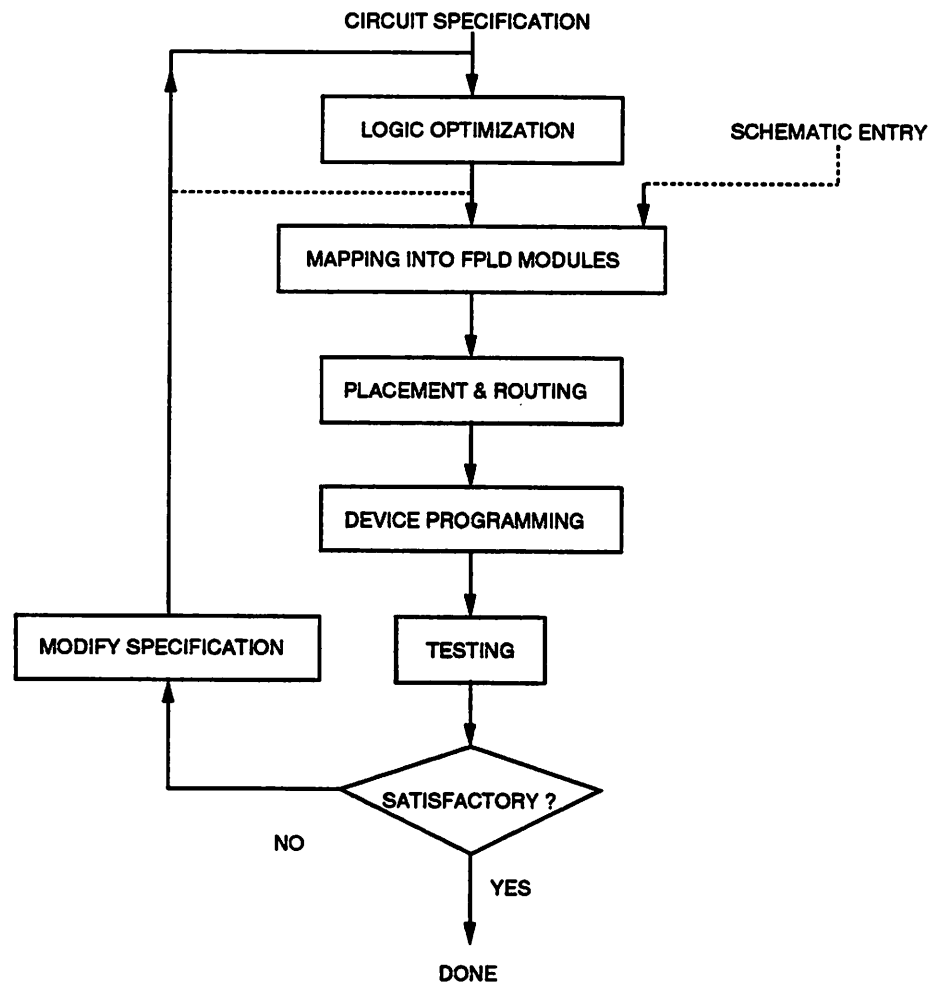


Figure 1.4: FPLD design flow



optimization is the process of manipulating the circuit structure and functionality of sub-circuits, while maintaining the specification, in a manner such that some parameters such as area of the overall circuit, and/or delay from inputs to outputs are optimized. A large body of work has been done in logic optimization for standard cell and gate array design styles [Brayton 90], and some FPLD optimization tools are based on these ideas. Some of these techniques, such as literal minimization (minimizing literals implies minimizing active cell area for standard cells and gate arrays) are not relevant to certain FPLDs such as LUT FPGAs, and in such cases, new optimization tools and techniques have emerged [Francis 92, Fujita 91, Murgai 90]. The next step in the design flow is mapping (also called packing or technology mapping). In this step, the architecture specific details are taken into consideration, and circuit structure is modified (maintaining the specification, of course) such that each gate or node in the circuit's network can be implemented by the logic modules of the FPLD on which the design is to be implemented. For PLDs, this step is usually simple and straight-forward. In the case of FPGAs, mapping is an area of active research [Bhat 92, Cong 92, Ercolani 91, Francis 91a, Francis 91, Karplus 91, Karplus 91a, Murgai 91a, Murgai 91, Murgai 90, Schlag 92, Trimberger 92, Woo 91]. Mapping is followed by placement and routing. In the placement step, the nodes in the mapped circuit are assigned positions on the FPLD (two-dimensional placement in the case of FPGAs), and the routing step figures out the path taken by the interconnections among the nodes. Following this, the actual programming of the FPLD is done (eg., loading 1's and 0's into the memory associated with the device, in the case of re-programmable FPLDs; or blowing fuses, anti-fuses in the case of one-time programmable FPLDs). The device now implements the circuit specification.

The steps of logic synthesis, placement and routing, and device programming are together called a design iteration. Since the circuit specification could be faulty (as humans usually tend to make errors), it may be necessary to go through several design iterations before a satisfactory circuit has been realized. The testing step in Figure 1.4 consists of plugging in the programmed device in the application it is intended for, and running several tests to check proper functionality <sup>1</sup>. If the test results are unsatisfactory, the specification is modified and another design iteration is performed. A typical design iteration could take about a day, on a personal computer such as the IBM 386 PC.

---

<sup>1</sup>In the case of fuse based devices, a "programming test" may also be necessary to make sure that the required links have been fused.

Figure 1.4 outlines only a typical design flow; modifications are possible. For example, in [Murgai 90], the authors recommend keeping the optimization and mapping together, as a single step. In [Bhat 92], mapping is combined with placement and routing. Vendor tools in the market do not perform logic optimization; the circuit specification (usually a schematic entry) is directly input to the mapping step (as shown by dashed lines in Figure 1.4), and the design iteration consists of mapping, placement & routing, and device programming. This is done because logic optimization tends to destroy the original circuit structure, which will make it almost impossible to pin-point design errors on the schematic.

### 1.3 Programmable logic system

The downside to the necessity of CAD tools in an FPLD based design is that the effective usage of the programmable device in a silicon-efficient and performance-optimized manner becomes the responsibility of the CAD tools. And unless CAD algorithms can use the special features of the FPLDs that they are assigned for designing with, the full potential of these devices cannot be effectively utilized. Because of the major role played by the CAD software in an FPLD based design, it is more appropriate to think of a programmable logic system (PLS), consisting of the FPLDs and their associated design tools.

Desirable features of a PLS are:

#### 1. CAD tool features

- (a) *Fully automated*: In view of the rapid time-to-market requirement, manual intervention is not desirable.
- (b) *Fast*: In the absence of foundry fabrication time, and since programming the device takes the order of milliseconds, the time required to synthesize the design for the FPLD's architecture defines the design cycle time, and the CAD tools run-time is the real bottleneck in an FPLD design.
- (c) *Performance-driven*: The circuit to be implemented on the FPLD can have timing constraints, and the CAD tools must be able to honor these constraints.

#### 2. Device features

- (a) *Efficient silicon usage*: The programmable nature of the device requires routing and logic resources to be available in sufficient quantity so as to allow all possible circuits to be implementable on the programmable device. But excessive resources could result in silicon wastage. Less resources could result in unroutable circuits or excessively lengthy CAD tool run-times.
- (b) *Low interconnect delays*: Programmable switches on the interconnection networks introduce significant delays, and can cause interconnection delays to exceed logic delays.

## 1.4 FPLD advantages

Along with increased complexity, there has also been a corresponding increase in the application areas for FPLDs. Application Specific Integrated Circuit (ASIC) users are now considering programmable logic devices as viable alternatives to foundry-based design styles such as gate arrays (now renamed as mask programmed gate arrays, MPGAs, to distinguish them from the FPGAs), sea-of-gates and standard cells. FPLDs offer two advantages for the ASIC user. The short design cycle of a PLS shortens the time-to-market, which is a critical factor in an ASIC's success. Also, since device costs are low, any design changes or design error fixes can be accommodated with negligible overhead (zero overhead for re-programmable devices). Gate arrays and sea-of-gate design styles require expensive mask fabrication, and changes and error corrections increase costs further. The non-recurring engineering (NRE) costs, which include mask fabrication and design change iteration costs, are absent in the case of FPLDs. Figure 1.5 shows the cost advantage of an FPGA [Xilinx 89] over foundry-based design styles. In addition to the ASIC market, re-programmable FPLDs have created new application areas such as hardware emulation [Quickturn 93], hardware acceleration [Hastie 90] and reconfigurable fast computers [Waugh 91].

## 1.5 Focus

This thesis focuses on the usage of FPLDs in the ASIC market, with the goal of extending their usage to applications which dictate timing performance. Complex circuits are now implemented on FPLDs instead of on gate arrays and sea-of-gates. However, in addition to being able to just *hold* the design within a programmable part, it is also necessary

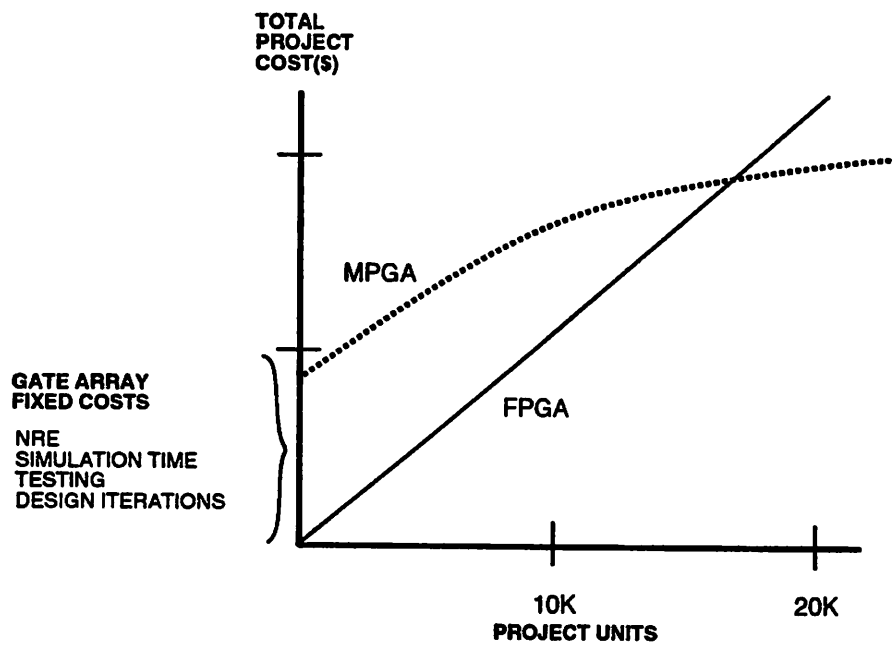


Figure 1.5: FPGA cost advantage over MPGA

---

to be able to perform the circuit functions fast. Only if the FPLD can meet the timing specifications of the logic circuit being implemented on the FPLD, can the FPLD be a viable alternative design style for ASIC users.

In this work, the emphasis is on the overall PLS. Both CAD and architecture aspects are considered. It may be possible to use existing FPLDs in high-performance applications by developing CAD tools that take timing performance into account when performing their design tasks. To explore this possibility, we choose a popular FPLD architecture, look-up table (LUT) FPGAs, and investigate the timing performance improvement by changing the structure of the mapped solution (of course, the logic function equality is maintained). In other words, we concentrate only on the technology mapping phase of logic synthesis, and propose a new performance-oriented mapping algorithm for FPGAs. For this part of the research, we restrict ourselves to combinational circuits only. This is justifiable since combinational circuits are easier to handle, and the ideas and algorithms can be easily extended to sequential circuits as well. The other possibility for overall PLS enhancement is to study architectural aspects. A new FPLD architecture, which is silicon efficient, performance-oriented and CAD friendly is proposed. As seen in Figure 1.3, this new architecture is different from existing FPLD devices and portends a new branch in the FPLD family tree.

We concentrate on single devices - i.e., we consider only those circuits that can be accommodated within a single programmable part. Circuits which cannot fit into a single part, will need to be partitioned across several parts, and this will require recourse to logic partitioning and physical partitioning techniques. Although these techniques have been extensively discussed in the literature [Donath 88, Yeh91] for standard cell and gate array based design styles, FPLD based partitioning techniques are still in their infancy, and merit further research.

## 1.6 Thesis overview

Definitions and glossary of more commonly used terms can be found in Chapter 2. The rest of this thesis is modeled after the PLS, in that it has two components. Part I has the software related work and Part II has the device architecture related work.

Part I deals with performance-oriented synthesis for LUT based FPGAs. By synthesis, we mean the mapping phase of Figure 1.4. The goal is to improve the performance of

a LUT FPGA, by modifying the mapped solution. Two aspects of the mapped solution are considered: ease of routing, and timing performance. The first sub-division of Part I deals with a routable mapping algorithm. Here, we combine the mapping step with the placement and routing step, using simulated-annealing. This approach yields mapping solutions that have lesser un-routed nets. The second sub-division of Part I presents a library-based performance-oriented mapping algorithm. This approach uses a concept of a 2-input LUT as a building block for the LUT library patterns and results in a very small library, several orders of magnitude smaller than previously documented LUT libraries [Francis 92]. This small library is used along with conventional library-based technology mapping techniques [Brayton 90] to give mapped solutions that have improved timing performance. The special nature of the library patterns is also exploited to yield faster pattern matching during the mapping process. The mapping algorithm also provides area-delay curve solutions, giving the user a family of solutions rather than a single *best-area* or *best-delay* solution.

Part II deals with the new *Dharma* architecture. We present the architecture, discuss CAD aspects for this new architecture and outline new research avenues. The central idea of *Dharma* is reconfiguration of the interconnections and the logic modules during run time (hence the architecture is *dynamic*). A multi-level circuit is implemented in a level by level manner, with inter-level connections accomplished using a crossbar interconnection network. The crossbar structure provides full routability, and the run-time reconfiguration allows reuse of the same crossbar to implement different net connections, making the architecture silicon efficient and practical. The net and logic delays are both predictable. This means that the synthesis step can make accurate estimate of the final (after device programming) timing behavior.

The final chapter concludes this thesis with a summary and future directions.



## Chapter 2

# Terminology

In this chapter, terms commonly used in the rest of the thesis are described. The chapter is meant to be used more as a reference, and hence the terms are arranged alphabetically. Terms and definitions specific to a chapter are defined in their respective chapters.

### Boolean function

$B = \{0, 1\}$  is a Boolean space, and a variable that takes values 0 or 1 is a *binary variable*. A *Boolean function*,  $F$  maps an  $n$ -dimensional Boolean space to an  $m$ -dimensional Boolean space, i.e.,

$$F : \{0, 1\}^n \rightarrow \{0, 1\}^m$$

A Boolean function is also called a *logic function* or simply *function*.

The AND function is represented by a '\*' or a space, the OR function is represented by a '+' and logical inversion is represented by a single quote after the variable, or an exclamation sign '!' before the variable.

### Boolean Network

A *Boolean network* is a *directed acyclic graph (DAG)* used to represent a multi-level logic function. Each node  $i$  in the DAG is associated with a variable  $y_i$  and a representation  $f_i$  of a logic function. Figure 2.1 illustrates a Boolean network implementing the following multi-level logic function.

$$x = q + r'$$



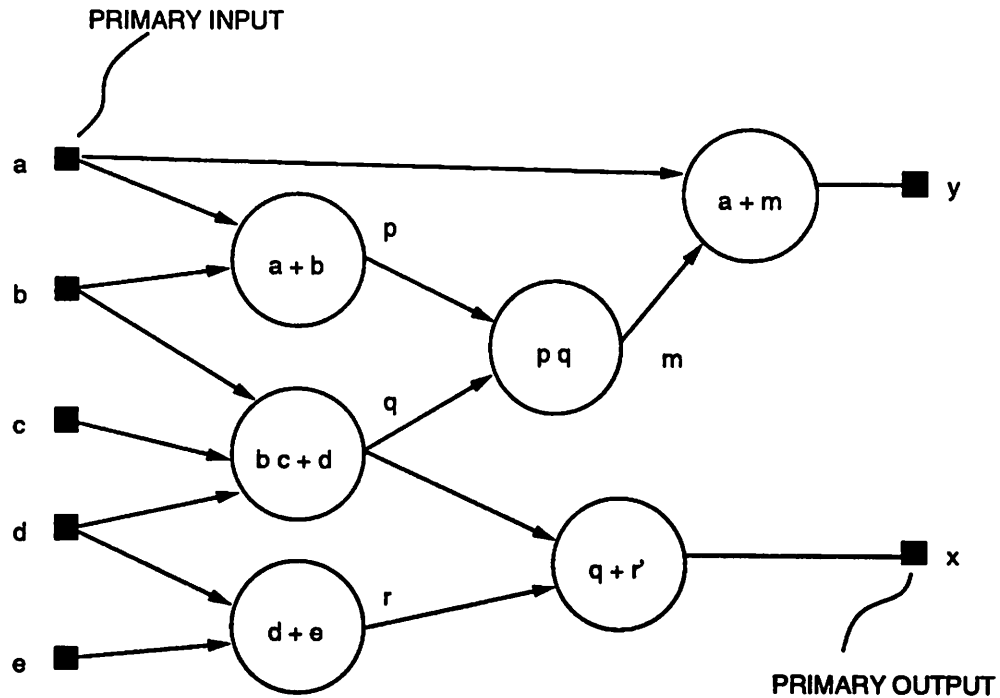


Figure 2.1: A Boolean Network is a DAG to represent a multi-level logic function.

---

$$y = a + m$$

$$m = p q$$

$$p = a + b$$

$$q = b c + d$$

$$r = d + e$$

In Figure 2.1,  $a, b, c, d, e$  are the PIs and  $x, y$  are the POs of the example multi-level logic function.

The *fanins* of a node  $n$  of a Boolean network is the set of nodes  $i$  whose output is directly connected to an input of  $n$ . In Figure 2.1,  $b, c, d$  are the fanins of node  $q$ . PIs do not have a fanin.

The *fanouts* of a node  $n$  is the set of nodes  $o$  that have an input directly connected to the output of  $n$ .  $m, x$  are the fanouts of  $q$ . POs do not have a fanout.

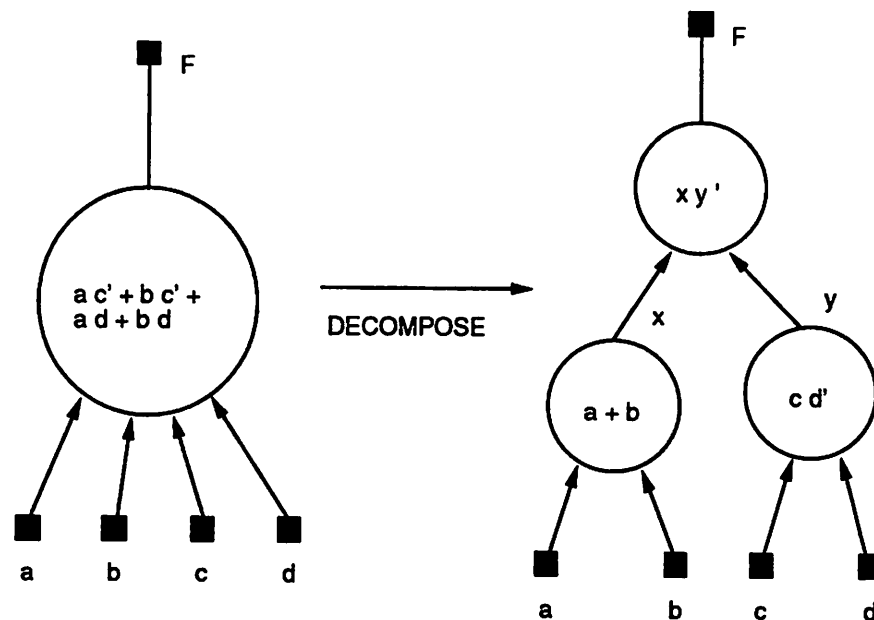


Figure 2.2: Decomposition is the process of re-expressing a single function as a collection of new functions.

---

## DAG

DAG stands for Directed Acyclic Graph. A Boolean Network is a DAG.

## Decomposition

*Decomposition* of a function is the process of re-expressing a single function as a collection of new functions. For example, re-expressing

$$F = ac' + bc' + ad + bd$$

as

$$F = x y'$$

$$x = a + b$$

$$y = c d'$$

is decomposition (see Figure 2.2).

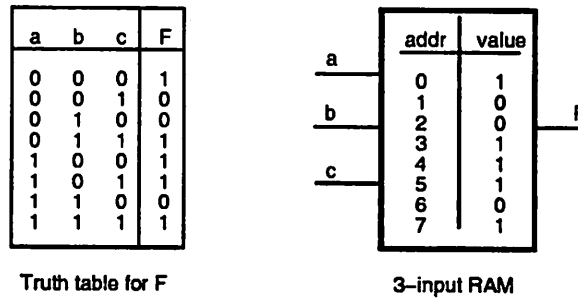


Figure 2.3: A look-up table (LUT) implementation of a 3-input Boolean function

Decomposition can be used to create a multi-level representation of a given logic function, specific to an application. For example, to represent the logic function as a DAG in which every node has  $\leq m$  inputs, where  $m$  is a specified integer constant; or to represent the logic function as a DAG in which every node's function is either a NAND or INV; etc.

### Logic module

A *logic module* is a piece of circuitry used to implement logic functions. In the case of LUT based FPLDs, like the Xilinx FPGAs, and the AT&T ORCA, logic modules are constituted from LUTs.

### Look-up table (LUT)

A *look-up table (LUT)* is a random access memory (RAM) module, used to implement a logic function. Inputs to the function are presented as address value to the RAM. The function's value corresponding to a particular input combination is stored in the memory location addressed by that particular input combination. Thus, when presented with an input combination, the RAM "looks up" the value of the function it is implementing, and presents this value at the RAM output.

In Figure 2.3, a three input function  $F$  is specified by means of a truth table. On the right side of Figure 2.3, a 3-input RAM module, with values stored to implement  $F$  is shown.

An  $m$ -input, 1-output LUT has  $2^m$  locations, each of which can be individually set to a 0 or 1, and therefore the LUT can be used to implement any function of  $m$  inputs.

An  $m$ -input,  $n$ -output LUT can implement any multiple-output Boolean function with  $\leq n$  outputs and  $\leq m$  inputs.

### Multi-level logic

*Multi-level logic* refers to any multiple-output Boolean function represented by a set of interconnected functions. If  $F$  is the multiple-output Boolean function represented as a multi-level logic, the inputs to  $F$  are called *primary inputs (PIs)* and the outputs of  $F$  are called *primary outputs (POs)*. See Boolean Network for an example.

### Packing

*Packing* is technology mapping in the context of FPGAs.

### Performance

Generally, three metrics are used to measure the performance of an implementation: the silicon area used, the timing delay, and the power consumed. However, in this thesis, we use the term performance interchangeably with *timing* performance.

### Placement

Given a netlist, i.e., a list of modules and the manner in which they are connected to each other, *placement* is the process of assigning a location to each and every module on a two-dimensional plane. Placement tries to place the modules in a manner such that after the subsequent routing step, the modules and interconnections use up minimal possible resources and/or the worst-case timing delay is minimized.

### Primary input (PI)

See **Multi-level logic**.

### Primary output(PO)

See **Multi-level logic**.

### Routing

Given a netlist of placed modules, the *routing* process determines the path for the interconnections. Routing is usually performed in two steps: *global routing* and *detailed*

*routing.*

### Technology mapping

*Technology mapping* or *mapping* is the process of representing a given Boolean network in terms of logic functions specific to a given technology. The software which performs this task is called the *mapper*. When the given technology is FPGA, this process is also called *packing*, and the corresponding software is called a *packer*.

The mapping process takes as input a Boolean network, called the *input* network and gives as output another Boolean network, called the *mapped* network. Each node in the mapped network can be implemented by the gates or logic modules of the technology for which the mapper is designed. For example, in the case of an  $m$ -input, 1-output LUT mapper, the mapped network has nodes with  $\leq m$  inputs each.

### Timing delay

The timing delay is a measure of the time required for signals to propagate through the physical implementation of a given circuit or Boolean network. A circuit's delay has two components: delay through the logic modules (gate delay), and delay through the interconnection (net delay).

We use a linear delay model to model the delay through a logic module. In this model, the delay from an input pin  $i$  of a logic module, to an output pin  $o$  is given by a linear equation,

$$I_{i,o} + R_{i,o}\Gamma$$

$I_{i,o}$  models the intrinsic delay through the module, from pin  $i$  to output  $o$ ,  $\Gamma$  represents the capacitive loading at output  $o$ , and the coefficient  $R_{i,o}$  represents the delay per unit capacitive load at  $o$ . The delay through the interconnection can also be incorporated into the above model. A net is approximated as a lumped capacitance,  $\Gamma_i$ .  $\Gamma_i$  is then added to the output load of the module which is the source of the net.

Consider the situation depicted in Figure 2.4. Let the signals *arrive* (i.e, settle to correct value) at pins  $a$  and  $b$  of logic module  $L$  at time instants  $t_a$  and  $t_b$ , respectively.  $L$  fans out to pins  $p$  and  $q$  of logic modules  $M$  and  $N$ , respectively. Let the arrival time of the output of  $L$ ,  $x$ , at pins  $p$  and  $q$  be denoted by  $t_p$  and  $t_q$ , respectively. In the worst-case

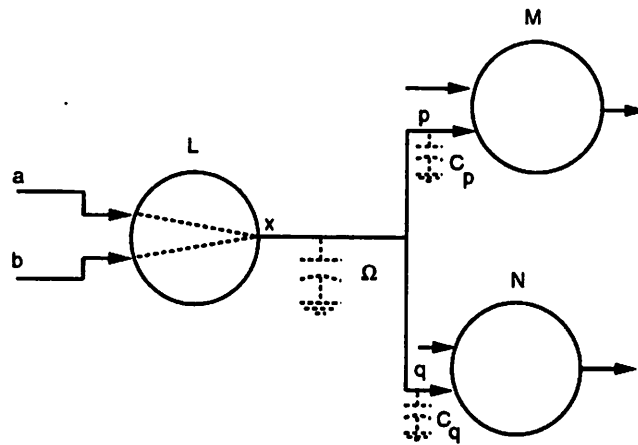


Figure 2.4: Example to illustrate worst-case delay analysis

---

delay analysis,

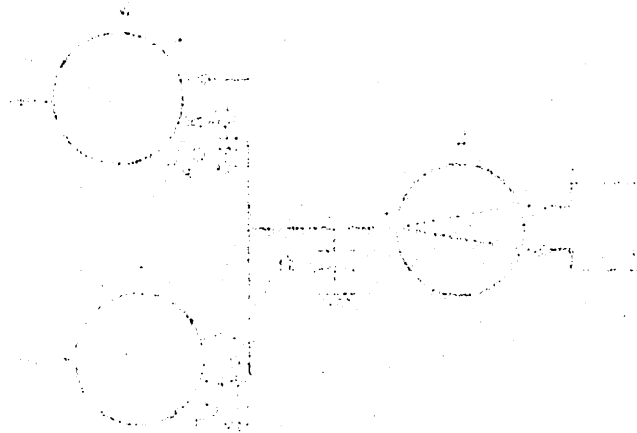
$$t_p = t_q = \max\{(t_a + I_{a,x} + R_{a,x}\Gamma), (t_b + I_{b,x} + R_{b,x}\Gamma)\}$$

where,

$$\Gamma = \Omega + C_p + C_q$$

$\Omega$  represents the capacitive loading due to the interconnection, and  $C_p$  and  $C_q$  are the parasitic capacitances at the pins  $p$  and  $q$ , respectively.

The capacitive loading due to the interconnection can be modeled as being proportional to the number of sinks (input pins) on the net.



... ..

... ..

... ..

... ..

$$I = \frac{1}{2} \pi \epsilon_0 \omega^2 E_0^2$$

... ..

... ..

... ..

... ..

## **Part I**

# **Performance-oriented Synthesis**



## Chapter 3

# Overview

In this part of the thesis, the focus is on improving the CAD for existing FPLDs. As already mentioned, a PLS has a software component and a device component. The FPLD vendors have always introduced a device first, and then tried to develop the software to support the device. As a result, there are several FPLDs in the market, whose full potential has not been realized because their CAD tools have not yet matured.

The question we are attempting to answer is as follows. *Given an FPLD, how best can we design the CAD tools so as to get the maximum usage and performance out of this device?* This is a very general question, and because of the large number of different FPLDs, and different phases in the design flow (see Figure 1.4) it is almost impossible to have a single answer to this question. We therefore concentrate only on a single type of FPLD and only on certain phases of the design flow.

The most popular (in terms of number of units sold) FPLD is the LUT based FPGA (devices such as the Xilinx XC3000 family, XC4000 family and the AT&T ORCA family). We choose these kinds of devices as our “given FPLD”. In the design flow, we concentrate on the mapping phase. Ideally, one could also look at the placement and routing phases. However, these require intimate knowledge of the device architecture, which is usually proprietary information. Also, placement and routing must be fine-tuned to specific architectures. Our purpose is to address a class of devices, hence it is not in the interest of this research to get bogged down by the idiosyncrasies of a specific device.

Two different aspects of the mapped solution are of interest. One is the timing behavior of the solution, and the other is the ease of routing.

### 3.1 Performance-oriented mapping

In applications where timing performance (henceforth called performance) is critical, the circuit specification is accompanied by timing constraints on the various output pins, in relation to the input pins. If the mapped solution is unable to meet these constraints, the device cannot be used for that particular application. For the purposes of our research, we impose the timing constraints in a slightly different manner. Given a circuit, our aim is to synthesize a mapped solution whose worst-case timing delay is minimal. In our experiments, all inputs are assumed to be present at the same time and the time delay at the latest arriving output signal is minimized. The reasons for this modification are as follows.

1. It is easier to benchmark different performance-improvement techniques, since the technique which yields the lowest worst-case delay is the better technique.
2. Our technique does not require timing constraints to be a part of the circuit specifications. Standard benchmark circuits do not have accompanying timing constraints.
3. Applications which supply timing specifications can still use the outcome of our research. If the mapped solution's timing delays are less than the specification's timing requirements, then the solution can be used for that particular application. If not, it means that the specifications have to be relaxed, since the solution provided is the best possible.

The delay through a circuit has two components: delay through the logic, and delay through the interconnection. With regard to LUT based FPGAs, prior efforts [Cong 92, Francis 91a, Murgai 91a] on performance-optimization have been concentrated on minimizing the delay through the logic. However, because of the limited amount of interconnection resources, LUT based FPGAs are characterized by dominant interconnect delays, which are difficult to estimate. Hence, mere logic delay minimization does not suffice. The delay of the final circuit (after placement & routing) is the relevant delay, and should be the objective being optimized.

A second consideration is that we are interested in addressing a *class* of devices, all based on LUTs. So, the mapper should be useful across the entire class of devices, with minor alterations to suit any particular architecture within this class.

A third consideration is that a family of mapped solutions is desirable. FPGA devices come in families, devices within a family differing in pin count and number of logic modules (larger devices are more costly). Circuit delay and circuit area (measured as the number of logic modules required to implement the circuit) bear an inverse relation: a circuit with better timing can only be obtained at the expense of more circuit area. However, if the mapper can provide a host of solutions, differing in area and delay, the user gets to choose a solution which is cost-effective and performance-optimized.

The above three requirements (accounting for net delay, using a single mapper across different architectures, and providing area-delay solutions) are not new to technology-mappers. Technology mappers used for standard-cell and gate array design styles have all the above features. Our idea is to extend these mappers to LUT FPGAs. However, these mappers use the concept of library-based mapping [Detjens 87, Keutzer 87]. Simple extension of the library-based approach to LUTs results in prohibitively large, impractical library of LUTs [Francis 92, Trimberger 92a]. We introduce the concept of a 2-input LUT as a building block (or primitive cell) for the library pattern, and demonstrate that this reduces library size by orders of magnitude, thereby making the library-based approach feasible and practical.

Using the new library patterns, and the old ideas of library-based mapping, our performance-oriented, library-based mapping algorithm can be set to optimize for depth (just to compare with previous work), delay, and to provide a family of solutions. Moreover, by simple alteration, the algorithm can be used for different LUT based architectures. However, our approach suffers the very bane of library-based mappers: sensitivity to initial decomposition. This disadvantage makes the area-minimization solutions inferior compared to other area-minimization based LUT mapping algorithms.

## 3.2 Routable mapping

Although it may seem that ease-of-routing has got nothing to do with performance-oriented synthesis, the mapping solution becomes useless if the placement and routing tools cannot complete the routing. In many instances, countless hours spent to manually route the circuit end up in vain because the mapping solution is truly impossible to route within the interconnection resource constraints of the device. Therefore, the first sub-division of this part of the thesis addresses this basic feasibility problem, and provides a technique to

alleviate the problem of unroutable circuits.

In our approach, we concentrate only on the objective of making the circuit routable, without regard to timing performance or area. Starting with a mapped solution to be placed and routed on a given FPGA device, we assume that the unused logic modules in the device are at our disposal, and we then re-distribute the logic, trading off logic block count for routability. Our algorithm integrates synthesis and physical design, much like [Pedram 91a, Pedram 91], but we use simulated annealing.

To make the integration of logic synthesis and physical design possible, we need to be able to treat the logic and routing resources in a unified manner. We model the FPGA architecture as an array of *qblocks* to be able to do this abstraction. This modeling forms the cornerstone of the routable mapping algorithm, which is called **RFR**, Re-synthesis For Routability.

### 3.3 LUTs and logic modules

In the next two chapters, we use the terms LUTs and logic modules interchangeably. In commercially available FPGA devices, logic modules are composed of interconnected LUTs. A logic module can have several outputs, and its constituent LUTs can share inputs. The manner in which the LUTs are interconnected, the number of inputs to the LUTs, and the number of outputs for the logic module varies from one family to another, and also from one vendor to another. Since our objective is to be able to address a class of devices, we restrict our mapping to consider only a single-output LUT. A network of single-output LUTs can be modified to fit into a specific architecture, using *merging* [Murgai 90] or library-based mapping (described in Chapter 5), and hence our restriction does not have any undue shortcomings. However, the library-based mapping approach of Chapter 5 cannot be extended to multi-output LUTs by any simple extension (therefore, *merging* will be required in such cases). The routable mapping technique does not depend on the LUT outputs or the logic module structure, and can actually be used for other types of FPGAs as well (non-LUT based).

## Chapter 4

# Routable Technology Mapping

This chapter concentrates on the routing feasibility of the mapped solution. As in the rest of this thesis, we have limited our attention to LUT based FPGAs. However, the ideas presented here are general enough that they can be extended to other types of FPGAs as well <sup>1</sup>.

Our approach to the *routable mapper* is based on a philosophy of integrating physical design and logic synthesis. Our ideas are along the lines of [Pedram 91a, Pedram 91], where technology mapping and logic restructuring/decomposition are performed along with placement. Here, we go one step further, and include routing during the mapping process, using simulated annealing to tie all three (mapping, placement and routing) together.

We first motivate the need for taking routing into account. Terms used are then described. A key to make possible the integration of routing with mapping is the modeling of the FPGA architecture, and we introduce the concept of a *qblock*. Our routable mapper, **RFR** (Resynthesis For Routability) is then described. Experimental results, demonstrating the effectiveness of the approach, are presented next. The last section outlines shortcomings of our approach and discusses possible avenues to further this work.

### 4.1 Motivation

Routability is a primary concern with current FPGAs. This concern involves two factors: the routing needs to be 100% complete and correct within the given resources,

---

<sup>1</sup>Among currently available FPGAs, routability is a problem only for LUT based FPGAs, like the Xilinx XC3000. The 100% routability in other FPGAs is more because of greater availability of routing resources, and is not because of any routing driven mapping.

and it must not add so much delay that critical paths are delayed beyond the applications requirements. When CAD tools fail to provide a completely routed circuit, one has to manually intervene. Designers spend hours (and sometimes days) trying to complete the routing, by manually changing the routing and the distribution of logic among the FPGA logic modules. Manual intervention prolongs the design cycle time, and the FPGA 'rapid time-to-market' advantage is lost. Further, manual intervention does not always prove successful.

#### 4.1.1 Cause of incomplete routing

There are only two reasons why a given circuit, to be realized on a given FPGA, results in a situation where some of the nets cannot be routed.

1. Existing CAD tools are not good enough.
2. Insufficient routing resources on the FPGA.

If the CAD tools are the best possible, and yet there are circuits which cannot be routed, then it is clear that the problem lies with the device itself, and its architecture needs to be altered to incorporate additional routing tracks and programmable connections. Our hypothesis is that the CAD tools are not good enough, and we proceed to improve them to enhance the possibility of obtaining fully routed circuits.

#### 4.1.2 Problem with existing CAD tools

Let us examine the development of CAD tools for FPGAs. In Figure 1.4, we showed the design flow for FPGAs. This flow is very similar to the traditional design flow for standard cell and gate array design styles. In Figure 1.4, we see that mapping is followed by placement, and routing is done last. Thus, the synthesis and physical design tools have developed independent of each other. The assumption is that, whatever be the mapping solution, the placement and routing steps can provide correct solutions, be it in meeting timing specifications or in completely routing all nets.

This assumption is valid for standard cell and gate array designs. In such cases, an accurate estimate of the wire-length can be made [Kuh 90] using standard measures such as perimeter of bounding box, length of spanning tree, etc. A good router can then yield a routing solution that is close to these estimates. The problem of unrouted nets is

virtually absent, since the router has the flexibility of moving cell-rows apart to make room for additional routing tracks. Hence, the traditional design flow, of mapping followed by placement and routing, works well for these design styles.

The same is not true of LUT FPGAs, however. The router does not have the flexibility to move logic modules to make room for extra routing. It has to deal with a fixed amount of tracks and programmable connections. Even with the best possible placement and routing tools, it may still be impossible to complete the routing, because the routing was not accounted for, during the synthesis step.

As an example, consider a hypothetical FPGA,  $\mathcal{P}$ . Let each logic module in  $\mathcal{P}$  be a 3-input, 1-output LUT. Consider the implementation of the following multiple-output function,  $F$ .

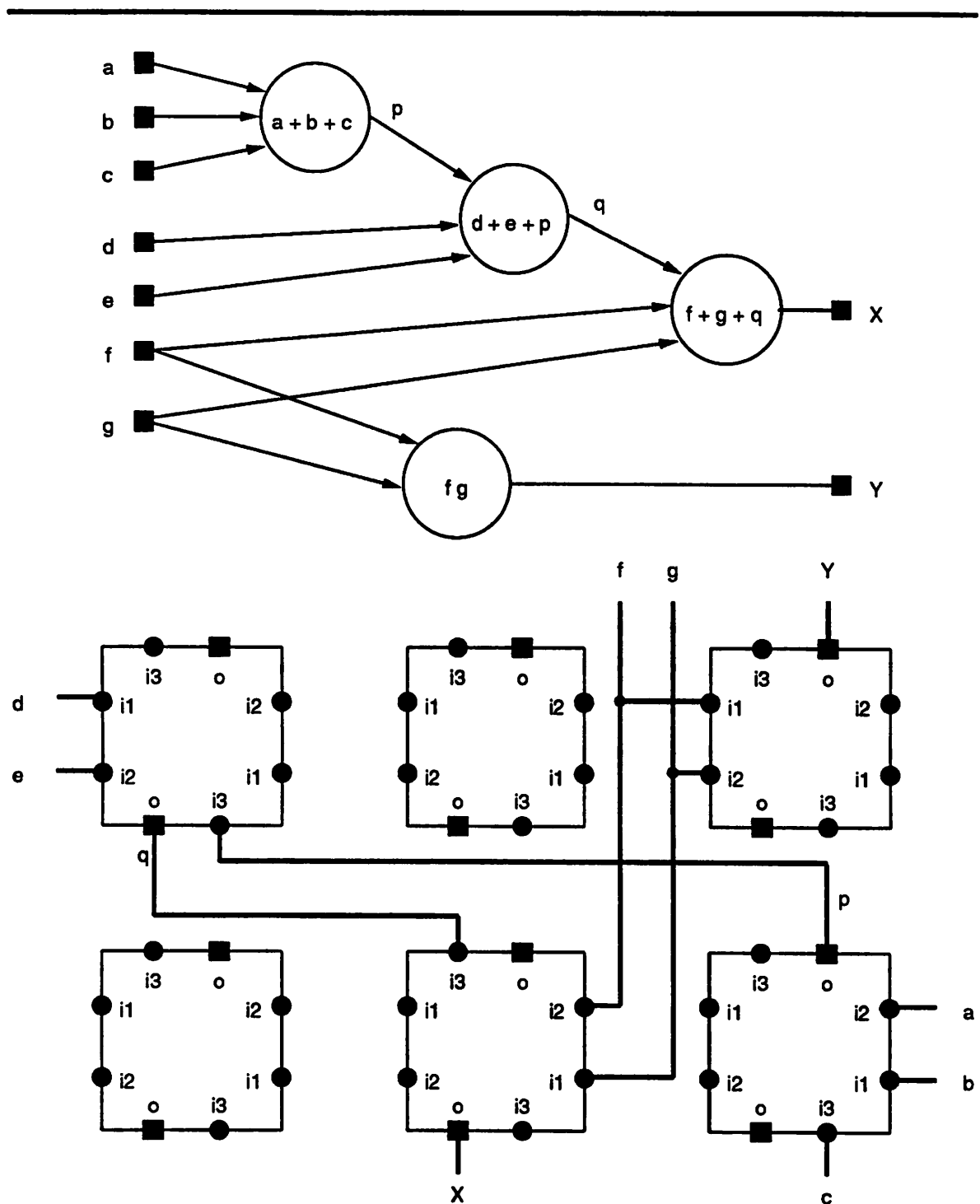
$$\begin{aligned} X &= a + b + c + d + e + f + g \\ Y &= f g \end{aligned}$$

Figure 4.1 shows one possible mapping,  $\mathcal{M}_1$  of  $F$ , and  $\mathcal{P}$  after mapping, placement and routing. Figure 4.2 shows a second mapping,  $\mathcal{M}_2$  of  $F$ , and the corresponding  $\mathcal{P}$  after placement and routing. By comparing Figure 4.2 and 4.1, we see that  $\mathcal{M}_2$  uses lesser routing tracks compared to  $\mathcal{M}_1$ . Thus, this example shows that by changing the mapping solution we can change the routing characteristics.

### 4.1.3 Our strategy

We are motivated by the above example to improve the routability by changing the mapping solution. Previous techniques [Francis 91, Francis 90, Karplus 91a, Murgai 91, Murgai 90, Woo 91] for FPGA technology mapping have addressed the goal of minimizing the number of LUTs or configurable logic blocks (CLBs). In our approach, we diminished the relative weight of this constraint. Our motivation is to use the available resources (both routing and LUTs) for an FPGA of given size (where size is specified by number of LUTs) in an optimal manner. Current FPGA vendors have only a limited number of different sized FPGAs, and one can easily extend our algorithm to choose the smallest sized FPGA by stepping through the different sizes, arranged in ascending order, and picking the next higher size whenever the current choice is infeasible.

Our approach is based on performing mapping *along* with placement and routing. Each mapping step is followed by an incremental routing step. Thus, there is an *integration*

Figure 4.1:  $\mathcal{M}_1$  implemented on  $\mathcal{P}$



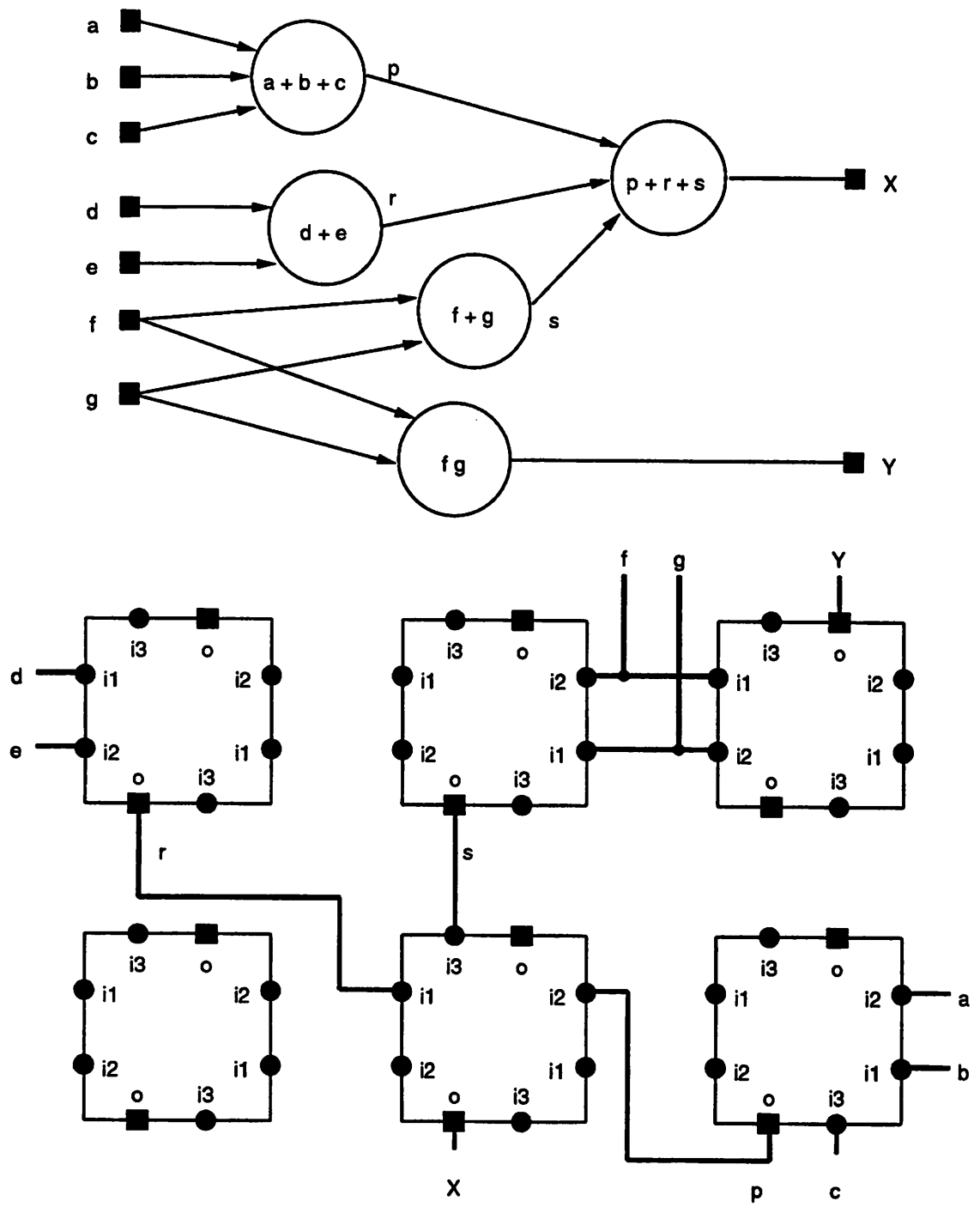


Figure 4.2:  $M_2$  implemented on  $\mathcal{P}$

of physical design with logic synthesis, in our approach. This philosophy of integrating physical design and logic synthesis is relatively new. In [Pedram 91a, Pedram 91], the authors describe techniques to perform technology mapping, logic restructuring and decomposition along with placement. In [Abouzeid 90], the authors use an estimation of the routing factor to perform logic synthesis operations. In [Murgai 91], a placement based technique to perform timing optimized logic synthesis for FPGAs is used. Unlike our approach, however, none of these integrate routing with synthesis.

## 4.2 Terminology

A *qblock* is a single cell of the FPGA. The FPGA is constructed as a two-dimensional array of qblocks. Section 4.3 elaborates on the qblock.

*Placking* (PLacement and paCKING) refers to the process of performing placement, routing and packing in tandem.

*Iota of logic* (IOL) is a small chunk of logic that is moved, or swapped amongst LUTs by the **RFR** algorithm. It can be a combinational function with a small number (1, 2 or 3) of inputs or a sequential element.

A packing is said to be *feasible* if the LUT can realize the logic assigned to it. Feasibility can be checked by comparing the corresponding number of inputs and outputs of the LUT and the logic function. e.g., it is feasible to assign a 3-input, 1-output function to a 4-input, 1-output LUT, but it is not feasible to assign a 5-input, 1-output function to the same LUT.

*RST* is an acronym for a *Rectilinear Steiner Tree*. An RST for a set  $A$  of points in the plane is a tree composed of vertical and horizontal lines which interconnect all members of  $A$ . Figure 4.3 shows a set of points  $A = \{a, b, c, d, e\}$ , connected by an RST. An *optimal* RST for  $A$  is one in which the lines have the shortest possible total length. Typically, there will be many optimal RSTs for  $A$ . The problem of finding an optimal RST is NP-complete [Garey 77], and many heuristic algorithms have been proposed [Hanan 66, Aho 77, Cohoon 90, Yang 72, Lee 76, Hwang 79, Lee 88, Ho 90].

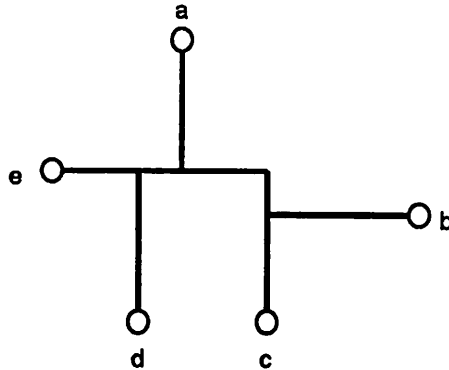


Figure 4.3:  $A = \{a, b, c, d, e\}$  connected by an RST

---

## 4.3 FPGA Model

We first need to abstract the FPGA architecture in a manner such that placing is possible. Here, we introduce the concept of a *qblock*, which can model any FPGA architecture by merely changing its parameters.

The FPGA architecture that we have assumed for our algorithm has four types of components:

1. LUTs
2. Latches
3. Configurable interconnection points and
4. Routing tracks.

### 4.3.1 FPGA model

In our abstraction, we model the FPGA as an array of identical *qblocks* (see Figure 4.4). The *qblocks* are symmetric, and abut each other in the horizontal and vertical directions. At the abutment, the horizontal tracks abut in the horizontal direction, and the vertical tracks abut in the vertical direction, as seen in Figure 4.4. Not shown in the figure are programmable switches to split the tracks into separate segments. In addition

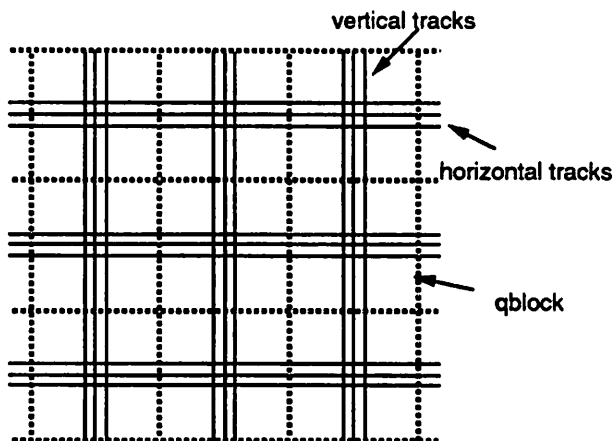


Figure 4.4: FPGA model. The FPGA is assumed to be an array of identical qblocks.

---

to these splittable tracks, there can also be continuous tracks (eg., long lines in the Xilinx architecture) and clock routing tracks.

### 4.3.2 qblock

Each qblock has the following resources.

1. Memory resources to implement LUTs
2. Latches for sequential logic
3. Routing resources
  - (a) Horizontal tracks
  - (b) Vertical tracks
  - (c) Corners to connect horizontal and vertical tracks

Figure 4.5 shows a pictorial representation of this abstraction. The size of the memory, the number of inputs and outputs to the memory, the number of shared inputs, the number of latches and the quantity of routing resources are parameters of the qblock and can be set according to the commercial FPGA architecture being modeled. All qblocks are identical, in that they have the same number of resources.

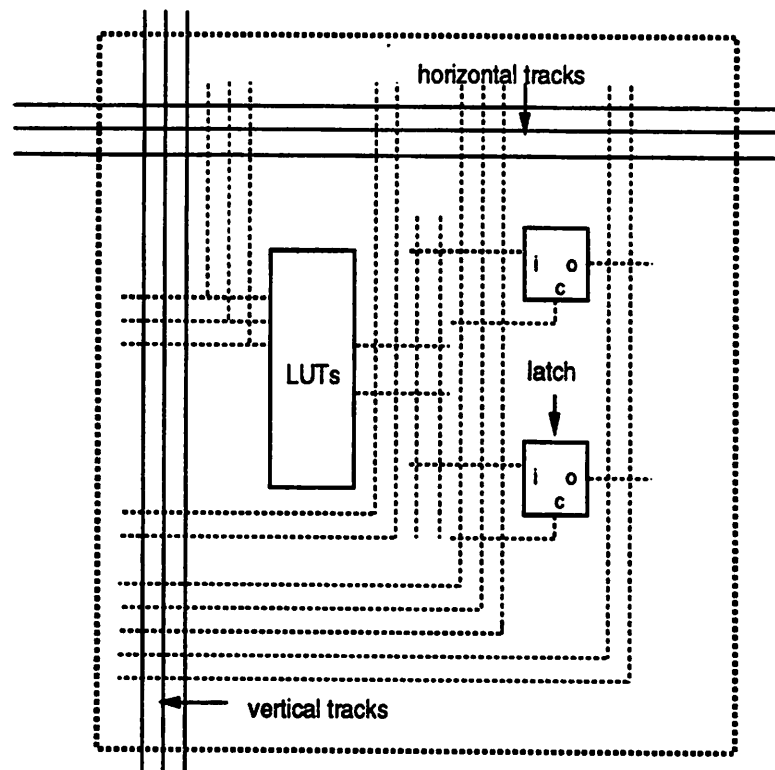


Figure 4.5: Internals of a qblock. The qblock has LUTs, latches and routing.

---

In Figure 4.5, the inputs and outputs of the LUTs and latches can be connected to any of the horizontal and vertical tracks, and this is illustrated by the extra routing lines (dotted). These extra lines are only for the purpose of illustration, the global router in our algorithm abstracts a qblock connection to lie directly on either the horizontal or vertical (or both) track(s). It is also assumed that the LUT outputs can be connected to latch inputs without going through the horizontal/vertical tracks.

### 4.3.3 Look-up table

The LUTs are configurable from the memory cells in a qblock. It is assumed that the memory cells are restructurable (by means of programmable switches), so that LUTs with varying number of inputs can be implemented. As an example, if there are 32 memory cells, then these can be structured as a 5-input, 1-output LUT, or a 4-input, 2-output LUT, or a 3-input, 4-output LUT, and so on. Additionally, two 4-input, 1-output LUTs (with no common inputs) and four 3-input, 1-output LUTs, and so on, are possible.

This model for the LUT is very general. We can restrict the number of inputs, number of outputs, number of shared inputs and the number of memory cells to model the LUTs of commercial LUT based FPGAs.

### 4.3.4 Routing resources

There are three kinds of routing resources: horizontal tracks, vertical tracks and corners. The number of such resources are fixed for each qblock, and all qblocks are assumed to have the same number of these resources. These are not supposed to represent the actual number of resources seen by the detailed router; they are for the purposes of global routing only. The algorithm is general enough to allow additional resources like connection points, long lines, etc.

## 4.4 Overview of our approach

We start from an initially optimized network, or from a mapped solution, eg., the output of Chortle-crf [Francis 91]. Each node in the input network is decomposed into IOLs. The IOLs are then re-organized in a manner such that the resulting network is easier to route, as compared to the input network. This re-organization is performed by swapping

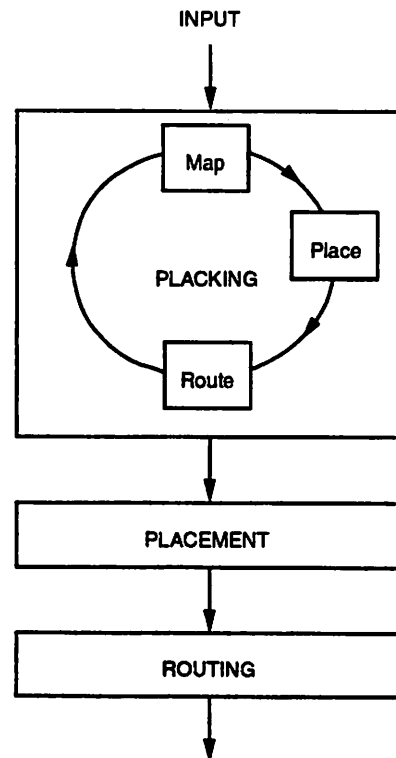


Figure 4.6: Overview of the RFR algorithm flow

---

IOLs, performing an incremental routing, and choosing the swap using a simulated annealing approach. After this re-organization, the new mapping solution is placed and routed using the vendor's commercial tools. Figure 4.6 shows the overall flow, and the placking step is explained in detail in the next section.

## 4.5 RFR algorithm

The algorithm for doing an integrated synthesis, placement and packing proceeds as follows.

1. Start from an initially optimized network [Murgai 90, Fujita 91].
2. Decompose the optimized network into a network of IOLs.

3. Pack the IOLs such that each cluster can be implemented using the latches and re-configurable LUTs within a qblock. i.e., obtain a feasible packing.
4. Place the feasible packs on the FPGA plane, using some placement program, so that each pack is assigned to one of the qblocks in the FPGA (this step is done because we require an initial solution before we begin simulated annealing).
5. Perform an RST global routing. The global routing populates the horizontal, vertical and corner routing resources of each qblock.
6. Randomly pick an IOL, and evaluate the cost of swapping it with another IOL, or moving it to another qblock. By swapping, we mean that the IOLs swap both their positions and their qblocks.
7. Either choose or discard the swap, using a simulated annealing cost function.
8. Repeat the swapping process till no further congested channels remain (in case the design cannot be implemented within this FPGA, we will have to terminate after a sufficient number of iterations).
9. At termination, we will have a packed, placed and globally routed FPGA, which is ready for detailed routing.

#### 4.5.1 Decomposition into IOLs

This is a crucial step, since the nodes in the decomposed network form the iota of logic that is swapped amongst clusters. The smaller the number of inputs, the greater is the possibility of performing feasible, cost reducing moves. We decompose into 2-input and 3-input nodes, allowing all possible functions, with a bias towards the more complex EXOR and EXNOR functions.

#### 4.5.2 Global routing

An RST based global routing is performed using the separable minimum spanning tree to Steiner tree conversion algorithm presented in [Ho 89]. The qblock resources are populated based on the path of the route. For example, if a route starts at qblock located at (1,4) in the FPGA array, and runs horizontally to (3,4), and then vertically to (4,4),



$I_k$	Other fanouts	Consequence of removal	
		$U_i$	$U_o$
$\ni L$	none $\in L$	Decrement	No change
$\ni L$	at least one $\in L$	No change	No change
$\in L$	all $\in L$	No change	Increment
$\in L$	at least one $\ni L$	No change	No change

Table 4.1: Updating  $U_i$  and  $U_o$ , considering input nets.

the route is assumed to occupy one horizontal track in each of qblocks at (1,4), (2,4) and (3,4); one vertical track in qblocks at (3,4) and (4,4) and one corner in the qblock at (3,4). The global routing is performed sequentially on a net-by-net basis; each net is processed independent of the routing performed on the other nets before it.

### 4.5.3 Checking logic feasibility

Swapping of IOLs  $I_1 \in A$  and  $I_2 \in B$ , where  $A, B$  are qblocks, is a 2-step process: (1) Remove  $I_1$  from  $A$  and insert it into  $B$ . (2) Remove  $I_2$  from  $B$  and insert it into  $A$ . At the end of the swap,  $A$  and  $B$  should be feasible, i.e, the LUTs and latches should satisfy the constraints of the target FPGA architecture.

Consider the case of combinational IOLs. For checking feasibility, we have to keep a running count of the number of “used-up” LUT inputs and outputs. When one (or both) of these exceeds the target architecture’s constraints, we have an infeasible LUT. Since we perform the swap in 2-steps, we need to consider only the consequences of removing or inserting a single IOL from or into an LUT.

Let  $I_r$  be an  $m$ -input, 1-output IOL being removed from LUT  $L$ . Let  $U_i, U_o$  be the number of used-up inputs and outputs of  $L$ . For each input net of  $I_r$ , one of four possibilities can occur, as tabulated in Table 4.1. Three possibilities arise at the output net, as listed in Table 4.2. In Table 4.1,  $I_k$  is the source of the input net, and in the second column, ‘Other fanouts’ means fanouts (of  $I_k$ ) other than  $I_r$ . ‘Increment’ means ‘increment by 1’ and ‘decrement’ means ‘decrement by 1’.

From Table 4.1 and 4.2, we see that, as a consequence of removing  $I_r$  from  $L$ ,  $U_i$  can increase by at most 1 and decrease by at most  $m$  and  $U_o$  can decrease by at most 1 and increase by at most  $m$ .

We can similarly list the possibilities for IOL insertion. These will be identical to

Fanouts	Consequence of removal	
	$U_i$	$U_o$
none $\in L$	No change	Decrement
all $\in L$	Increment	No change
some $\ni L$ , some $\in L$	Increment	Decrement

Table 4.2: Updating  $U_i$  and  $U_o$ , considering output net.

Tables 4.1 and 4.2, except that the consequence of insertion will be the reverse of removal, i.e., in Tables 4.1 and 4.2, we replace ‘increment’ by ‘decrement’ and vice versa. In this case,  $U_i$  can increase by at most  $m$  and decrease by at most 1 and  $U_o$  can increase by at most 1 and decrease by at most  $m$ .

For sequential IOLs, feasibility check involves checking whether the number of sequential elements exceeds the qblock resources and also checking for compatibility of control signals.

The above method is suitable for simple LUT feasibility checking. Commercial FPGAs, however, have complex interconnection of LUTs which share certain inputs, with some outputs being connected to sequential elements, etc. In such cases, it is more suitable to have a *feasibility check function* specific to an architecture. The function takes as input a logic mapping and produces a yes/no output depending on whether or not the mapping is feasible. One such checking function will be needed for each commercial architecture, and the user is required to select the appropriate one.

#### 4.5.4 Simulated annealing

The simulated annealing [Vecchi 83, Sechen 88] is straightforward. We start at a high temperature  $T$ , and evaluate  $P = \exp(-cost/T)$ , where cost is the cost of the swap. The swap is then chosen with probability  $P$ . The cost term consists of a weighted linear combination of two quantities: the wire-length and the congestion cost. The congestion cost is proportional to the difference between the congestion if the swap were implemented, and the present congestion. Congestion is quantified as the *number* of excess routing resources needed. At high temperatures, we can allow infeasible qblocks (i.e., the combinational function is allowed to have more than the allowed number of inputs/outputs, number of latches are allowed to exceed the logic cell quota, etc.). In such cases, the ‘amount of infeasibility’ is also combined into the cost function.

Let  $I, O, M, D$  be the number of LUT inputs, outputs, memory cells and latches respectively, as specified by the target architecture. Let  $U_i, U_o, U_m$  and  $U_l$  be the number of inputs, outputs, memory cells and latches used-up when a logic sub-network  $N$  is assigned to qblock  $Q$ . Then, the infeasibility in mapping  $N$  onto  $Q$  is quantified as  $\text{MAX}(X_o, X_i, X_m, X_l)$ , where  $X_o = \text{MAX}(0, (U_o - O))$  is the excess number of LUT outputs,  $X_i = \text{MAX}(0, (U_i - I))$  is the excess number of LUT inputs,  $X_m = \text{MAX}(0, (U_m - M))$  is the excess number of LUT memory bits and  $X_l = \text{MAX}(0, (U_l - D))$  is the excess number of latches.

Therefore, if  $\text{cost1} = \alpha W + \beta C$ , and  $\text{cost2} = \gamma I$ , where  $W, C$  and  $I$  are the wire-length, congestion and infeasibility costs, then  $\text{cost} = \text{cost1}$  if the swaps do not cause infeasibilities, and  $\text{cost} = \text{cost1} + \text{cost2}$  if there are infeasibilities.

#### 4.5.5 Re-synthesizing a packing solution

Instead of having a technology independent optimized network as input in step 1, we could start with an already packed FPGA and then make it routable. In this case, we decompose each pack into IOLs in step 2 and skip step 3. The rest of the algorithm is unaltered. Since we are swapping (or moving) IOLs around, the final logic network may have an altogether different structure, and the number of logic cells will usually change when compared to the input network.

## 4.6 Experimental results

A version of the algorithm has been implemented in the SIS [Brayton 87, Sentovich 92] framework. The algorithm can run in two modes -

1. Perform placement, routing and mapping simultaneously.
2. Perform placement and mapping simultaneously (no routing at each swap). Here the cost of the swap is measured using a bounding box wire-length.

The second mode can be used as a pre-processing step before running mode 1 (the actual algorithm of Section 4.5), since it is much faster.

The second mode also allows us to illustrate the effect of integrating physical design and logic synthesis. Specifically, we carried out an experiment to demonstrate how

the algorithm makes a trade-off between the number of qblocks and total wire-length, as the size of the FPGA is increased. Figure 4.7 illustrates our observation. We implemented a benchmark circuit (`planet.kiss2` in the MCNC benchmark set) on FPGAs of different sizes, using simulated-annealing placement and packing. Since the input/output pins are restricted to the boundary of the chip, larger FPGAs will require larger total wire-length for implementing a given design. **RFR** reduces this increase in wire-length by increasing the qblock count. In Figure 4.7, the x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the qblock count and wire-length, normalized to their value corresponding to an FPGA of size  $14 \times 14$ .

If the qblock count were kept fixed, the total wire-length would increase with FPGA size, since the separation between the input/output pins would increase. This is illustrated in Figure 4.8. We perform the placement using simulated-annealing. In Figure 4.8, the x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the wire-length, normalized to the value corresponding to an FPGA of size  $14 \times 14$ .

In Figure 4.9 we show the results obtained by running the algorithm in mode 1. We start with an initial random solution, and with the routing congestion as shown, and then run the algorithm to yield a packing with less, or no, routing congestion. Figures 4.9(a) and (c) show the routing congestion before running **RFR**, and Figures 4.9(c) and (d) show the congestion after the algorithm was run. The results are for the MCNC benchmark circuit `ex1.kiss2` on a  $10 \times 10$  FPGA. The congestion is shown by means of black rectangles within each qblock – the x-dimension of the black rectangle corresponds to the number of excess horizontal tracks and the y-dimension corresponds to the number of excess vertical tracks. In Figure 4.9(a) and (b), the FPGA has 8 horizontal and 8 vertical tracks per qblock. In this case, the algorithm is unable to remove all the congestion. We then increased the routing resources to 12 horizontal and 12 vertical tracks per qblock and Figure 4.9(c) and 4.9(d) show the congestion before and after running the algorithm, respectively. In this case, we get a congestion free solution. The total wire-length was 967 for 4.9(b) and 910 for 4.9(d).

The above experiments illustrate the effect of the algorithm for hypothetical FPGAs. We also interfaced **RFR** with Xilinx tools [Xilinx 89], to test the advantage of using the placking concept for a real, commercial FPGA. Our interface only handles combinational circuits.

We used the Xilinx XC3000 series chips as our target architecture. To test our algorithm, we needed examples with difficult routing characteristics. A few of the Chortle-

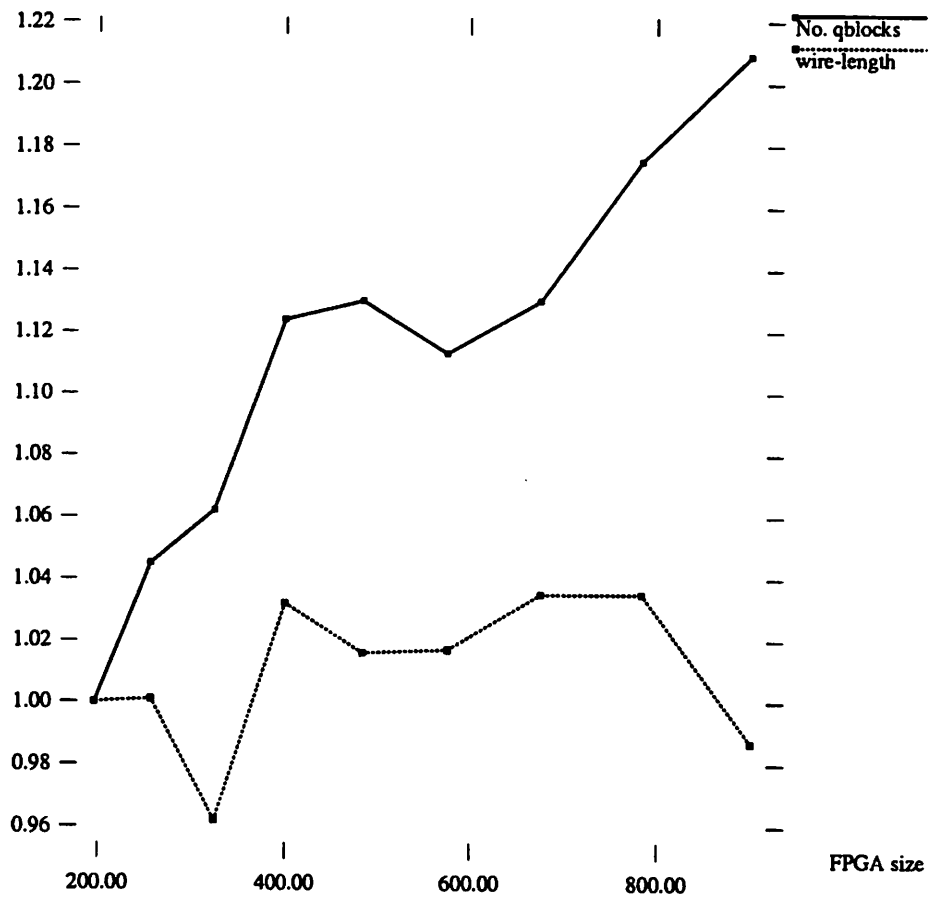


Figure 4.7: Effect of integrating physical design and logic synthesis. The algorithm trades-off qblock count for minimizing total wire-length. The x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the qblock count and wire-length, normalized to their value (of 177 and 2683 respectively) corresponding to an FPGA of size  $14 \times 14$ .

---

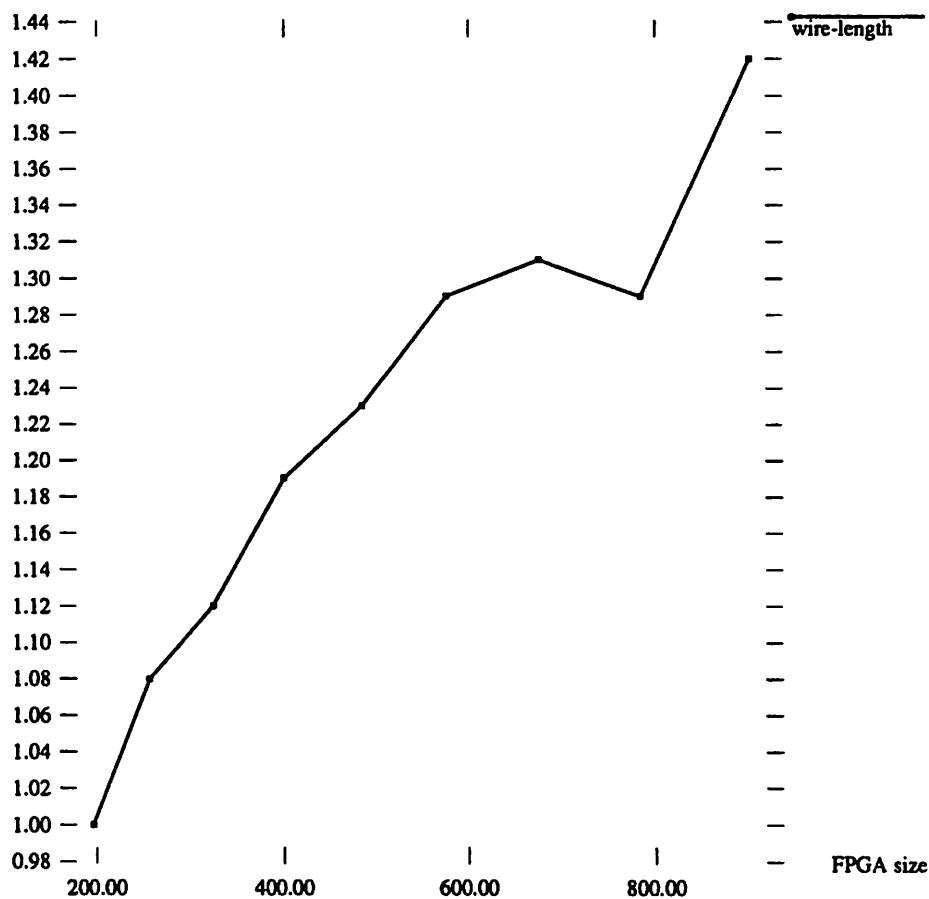


Figure 4.8: Illustrates how the wire-length would have increased with FPGA size, had the qblock count of the implementation been kept constant. The x-axis has the FPGA size in terms of number of qblocks, and the y-axis plots the wire-length, normalized to the value corresponding to an FPGA of size  $14 \times 14$ .

---

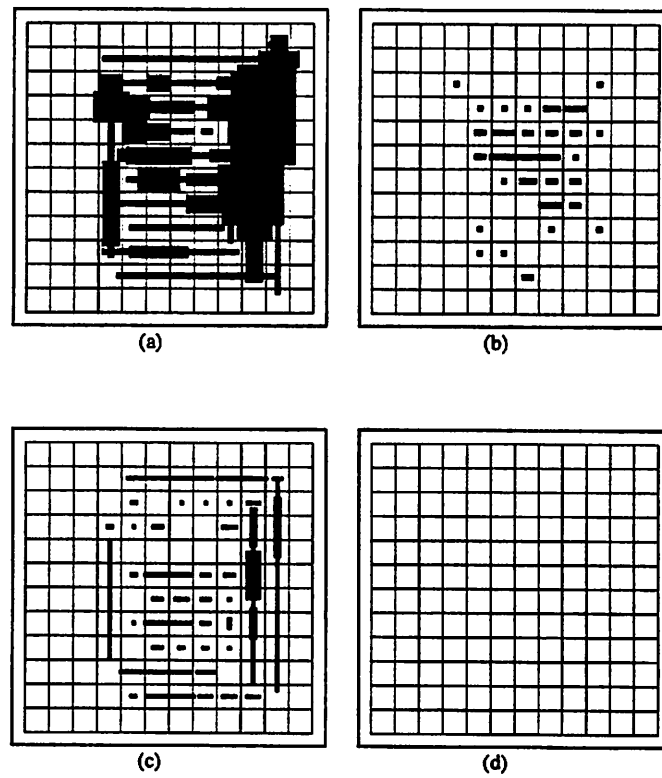


Figure 4.9: Routing congestion before [(a) and (c)] and after [(b) and (d)] running the algorithm in mode 1. (a) and (b) have 8 tracks, and (c) and (d) have 12 tracks. The total wire-length was 967 for (b) and 910 for (d).

---

mapped solutions [Francis 91] had such characteristics. i.e., when these examples were placed and routed on Xilinx 3000, using `apr` (Xilinx's place and route tool), some of the nets remained unrouted <sup>2</sup>.

Our experimental procedure consisted of (1) reading in the `Chortle` mapped examples (2) performing placking (as described in 4.5.5) (3) and finally doing place and route using `apr`. We used  $\alpha = 1.0$  and  $\beta = 4.0$ . Allowing infeasibilities did not help much. We used `Chortle`'s AND and OR nodes as our IOLs. Table 4.3 shows the results. The algorithm trades-off CLB count for routability. The CPU time shown is for the DEC 5000. The last column lists the Xilinx part used. `duke2` and `alu2` could also be fitted into 3030, but there was no improvement in routing, after `RFR`.

Example	Input circuit		After <code>RFR</code>			Xilinx Part (CLBs available)
	CLBs	Nets	CLBs	Nets	Time (s)	
<code>duke2</code>	92	15	108	8	3005	3042 (144)
<code>alu2</code>	93	7	101	2	3120	3042 (144)
<code>alu4</code>	152	51	171	17	4488	3064 (224)

Table 4.3: Results of running `RFR` for XC3000.

From the table, we see that there is a significant reduction in the number of unrouted nets (by a factor of 50%) or more, for all the 3 examples. The placking scheme is only partially effective, since the output of `RFR` still has unrouted nets. Since the IOLs are decomposed at the beginning, and the decomposition is kept fixed during the rest of the algorithm, it may be possible that the initial decomposition limits the number of different mapping solutions possible. Dynamically changing the decomposition, based on routing congestion information, during the course of the simulated annealing could make the algorithm more effective.

## 4.7 Conclusions

In this chapter, we presented heuristic techniques to alleviate the problem of obtaining routable FPGAs. These techniques are based on performing placement and packing in tandem, a process we call "placking." Global routing using RSTs is also done during placking, to measure routing congestion as the packing progresses. Simulated annealing

<sup>2</sup>The single output mapping solutions of `mis-pga` [Murgai 91] had very few unrouted nets for these examples.



has been used to bring together placement, routing and packing. Other techniques, like a greedy approach which chooses the best of all possible exchanges at each step, or one similar to Kernighan-Lin partitioning can be investigated.

We have not addressed timing issues directly (indirectly, by minimizing the wire-length, we decrease the delay in the route). An extension of this work would be to incorporate timing into the simulated annealing cost function. This would necessitate a timing analyzer to estimate the delay through an RST global route.

The final solution is dependent on the initial IOL decomposition. There are different ways to decompose the input Boolean network into IOLs, and we need to choose one that has favorable routing characteristics. Instead of keeping the decomposition fixed, one could dynamically vary it during the course of the annealing, using congestion and placement information.

The first step in the routing process is to generate a routing graph. This graph is a representation of the physical layout of the circuit, where nodes represent components and edges represent connections. The graph is then processed by a routing engine to find a path for each signal. The routing engine uses a variety of algorithms to find the best path, taking into account factors such as signal delay, congestion, and manufacturing constraints. The final output of the routing process is a set of routing files that can be used to fabricate the circuit.

The routing process is a complex task that requires a deep understanding of both the physical layout and the routing algorithms. In this chapter, we will explore the various steps involved in routing, from graph generation to final routing files.

## Chapter 5

# Performance-oriented Mapping

In this chapter, we present a mapping technique, for LUT FPGAs, which takes into account the delay in the nets during the mapping process. Hence the name performance-driven mapping. The algorithm is called **dpmap**, “dp” standing for dynamic programming (on which the algorithm is based). As outlined in Chapter 3, our approach extends the ideas of library-based mapping (LBM) used in standard cell and gate array designs. Since the LBM ideas form the core of **dpmap**, we first review the tree-covering approach used in standard cell mappers [Keutzer 87, Detjens 87], which use LBM. The important advantage of the tree-covering approach is that it can be used to optimize area, delay or their combination, just by changing the cost computation appropriately. Performance oriented technology mapping [Touati 90] and optimizing area under delay constraint [Chaudhary 92] are extensions of the library-based mapping approach. Why then, has the library-based approach not been used for LUT FPGA mapping? This question is answered in the following section. We then show that the problems associated with the LBM approach can be overcome by appropriate choice of a primitive cell, namely a 2-input LUT (TIL). We also show how the mapping process can be made faster by taking advantage of the special structure of the library patterns. This is followed by a description of our experimental procedure and results. Finally, we conclude with a list of shortcomings and possibilities to be investigated.

### 5.1 Motivation

Routing(net) delays are significant in LUT FPGAs and should be accounted for during the mapping phase. However, because of the inability of most LUT FPGA mappers

to handle net delays, current delay optimization is limited to depth optimization [Cong 92, Francis 91a], where the number of logic levels in the mapped network is minimized.

If a network has  $L$  levels of logic, the delay through the network  $t_D$ , using the delay analysis technique of Figure 2.4, is given by,

$$t_D = L \times t_{LM} + t_P + t_I,$$

where  $t_{LM}$  represents the delay through a logic module (this corresponds to  $I_{i,o}$  in Figure 2.4, assuming identical intrinsic delay  $\forall i$ , and  $\forall$  logic modules),  $t_P$  represents the delay through the input and output pads, and  $t_I$  represents the delay through the  $(L + 1)$  levels of interconnect. If  $t_I$  is ignored, then  $t_D = L \times t_{LM} + t_P$ , and minimizing  $L$ , minimizes  $t_D$ , since  $t_P$  is a constant quantity.

However, delay through the interconnection is significant, and at times dominates the logic delay. To demonstrate this point, we have tabulated the worst-case delays for some combinational benchmark circuits<sup>1</sup>. These delays are after place and route on the smallest Xilinx [Xilinx 89] XC3000 series chip that has enough number of CLBs to fit the design. Each logic module has 8ns delay (i.e.,  $t_{LM} = 8\text{ns}$ ), and input and output pads together have 33ns delay (i.e.,  $t_P = 33\text{ns}$ ). Two tables list our results, Table 5.1 and Table 5.2. All circuits in Table 5.1 have 3 levels of logic each, and all circuits in Table 5.2 have 4 levels of logic each.

Consider the ‘Delay’ column of Table 5.1. Although each circuit has 3 levels of logic, the delay varies by a factor of about 60%. The delay through the logic modules,  $t_{LM}$ , and the delay through the input and output pads is the same for each of these circuits.  $8 \times 3 + 33 = 57\text{ns}$ . But, the delay through the interconnect,  $t_I$ , is different and is the cause of this wide variation. We define a quantity, the routing delay per level,  $t_{RL}$ , given by

$$\begin{aligned} t_{RL} &= \frac{t_I}{(L + 1)} \\ &= \frac{t_D - t_P - L \times t_{LM}}{(L + 1)} \end{aligned}$$

For example, for the benchmark circuit 9symm1,  $t_D = 72.8\text{ns}$ ,  $L = 3$ ,  $t_P = 33\text{ns}$ , and  $t_{LM} = 8\text{ns}$ , giving

$$t_{RL} = \frac{72.8 - 33 - 3 \times 8}{(3 + 1)}$$

---

<sup>1</sup>We have used MCNC [MCNC] benchmark circuits. In the tables, examples beginning with the letter ‘c’ except for clip and count, are combinational parts of sequential benchmark examples. For example, cex2 is the combinational part of sequential benchmark ex2.

Circuit	Delay (in ns)	Routing delay per level (ns)
9sym	69.3	3.1
clion9	70.2	3.3
cs8	70.6	3.4
9symml	72.8	4.0
rd84	77.1	5.0
misex2	93.3	9.1
cex2	95.3	9.6
cdk16	102.8	11.5
b9	107.2	12.6

Table 5.1: Worst-case delays for some benchmark examples. All the circuits have 3 levels of logic each. Each circuit is placed and routed on the smallest Xilinx 3000 series chip that has enough number of CLBs to fit the design. The circuits have been arranged in ascending order of delays.

Circuit	Delay (in ns)	Routing delay per level (ns)
cex7	86.3	4.3
cex5	94.0	5.8
cex4	99.5	6.9
cs420	102.0	7.4
f51m	109.3	8.8
clip	120.5	11.1
count	133.1	13.6
apex7	168.6	20.7

Table 5.2: Worst-case delays for examples with 4 levels of logic each. Compared to Table 5.1, some circuits have lower delay than 3-level circuits.

$$\begin{aligned}
 &= \frac{15.8}{4} \\
 &= 3.95 \text{ ns}
 \end{aligned}$$

In the second column of Table 5.1 and 5.2, we tabulate the  $t_{RL}$  for each circuit. If  $t_{RL}$  were the same for circuits with same number of levels, then minimizing the number of levels would indeed minimize the total delay,  $t_D$ , and this is the implicit assumption made by depth minimizing algorithms. However, as seen in the second column, this is not the case;  $t_{RL}$  varies widely.

As a result of this variation in  $t_{RL}$ , we see that some circuits with 4 levels of logic (cex7, cex5, cex4 and cs420) have lesser delay than circuits with 3 levels of logic (cdk16, b9).

The conclusions from the above experiment are that the interconnection delay cannot be assumed to be a constant (hence, it cannot be ignored during the mapping), and that this delay could well be the dominating delay (eg., for apex7 in Table 5.2,  $L \times t_{LM} + t_P = 65$  ns, whereas  $t_I = 168.6 - 65 = 103.6$  ns).

We proceed to account for the interconnect delay during the mapping phase, in a manner similar to the performance-driven technology mapping approach [Touati 90] used for standard cell design styles. This approach uses library-based mapping, which offers us three main advantages.

Firstly, we wish to have a single mapper that can be used across an entire class of devices based on LUTs. There are many such FPGAs in the market (eg., the Xilinx 2000, 3000, 3100 and 4000 series chips, and the AT&T ORCA chips) and more may be coming soon. Implementing a mapper from scratch for each architecture is not the desired approach. Handling several technologies in the case of standard-cell and gate array design styles has been achieved by using the library-based mapping approach. We would like to use a similar approach for LUT FPGAs.

Secondly, as already explained above, interconnect delay needs to be taken into account during mapping. The library-based approach provides a simple mechanism to account for net delay estimates during the mapping phase.

Thirdly, FPGA devices come in families, devices within a family differing in pin count and number of logic modules (larger devices cost more). The user must be given the flexibility to choose amongst these devices. Hence, the mapper should provide the user with the flexibility to choose from several different solutions, with different areas and

delays, instead of providing a single *best-area* [Francis 91, Murgai 91] or *best-delay* [Cong 92, Francis 91a, Murgai 91] solution. In other words, the mapper should provide a mechanism for area-delay trade-off. Again, the library-based mapping approach provides this facility.

## 5.2 Terminology

*LBM* stands for Library-Based Mapping. LBM is the process of implementing a given Boolean network (*input network*) using a given set of gates, called the *gate library*.

*TIL* is the acronym for a Two-Input LUT, having one output. Figure 5.1 shows the symbolic representation of a TIL.



Figure 5.1: Symbolic representation of a 2-input LUT (TIL).

---

*TINN* stands for a Two-Input Nodes Network. It is a network in which every internal node has exactly 2 inputs. There is no restriction on the function being implemented by a node.

A *primitive cell* or *base cell* is a simple function or structure, from which every other function or structure can be constructed. For example, in the case of standard cell mapping, the input network, and the library gates are represented as a network of INVs and 2-input NANDs. The INVs and 2-input NANDs are primitive cells.

A library *pattern* is a representation of a library gate in terms of primitive cells.

The *subject graph* or *subject network* is the representation of the input network in terms of primitive cells.

## 5.3 Review of library-based mapping

In the case of standard cells and gate arrays, technology mapping is the process of implementing a given Boolean network (*input network*) using gates from a given library.

A successful technique for solving this is based on an approximation of the DAG covering problem, first proposed by [Keutzer 87]. The input network is decomposed into a forest of trees, and each tree is optimally covered using patterns representing the library gates. In this section, we review this basic tree-covering algorithm. The **dpmmap** algorithm uses this very algorithm, and hence this background is essential to understand the new approach.

### 5.3.1 Tree covering algorithm

The first step is to represent the input network in terms of *primitive* cells (eg., 2-input NANDs and INVs). This network, consisting only of 2-input NANDs and INVs, but logically equivalent to the input network, is called the *subject graph*. Each library gate is represented as a tree of primitive cells (if there are many such trees, each tree is stored). The gate representations will be called library *patterns*. Associated with each library pattern is a *cost*. The subject graph is partitioned into a forest of trees, and each tree is then optimally covered by the library patterns [Detjens 87]. The optimal covering is based on a dynamic programming algorithm.

In Figure 5.2, we show library patterns for an example library consisting of 4 gates, viz., 2-input NAND (*nand2*), an INV (*inv*), 3-input NAND (*nand3*) and a 3-input AND-OR-INVERT (*aoi*). These gates are represented in terms of the primitive cells, 2-input NAND and INV. For ease of explanation, we assume every gate has unit area cost.

The dynamic programming based covering algorithm proceeds from the leaves to the root (i.e., depth-first traversal). At each node, all the library patterns that match at this node are enumerated, and the cost of each match is evaluated. The matching pattern with the least cost is stored at the node. After processing the root node, in this manner, a simple back-tracing will yield the mapped solution.

Consider the mapping of Boolean function

$$Y = a'c' + b'c' + d + efg + h$$

using the example library of Figure 5.2. In Figure 5.3, on the left half, we show the subject network corresponding to  $Y$ , and on the right half, we show the mapped network, where each shaded region corresponds to a gate in the example library. The numbers on the left correspond to the best cost stored, for a minimum-area cost mapping.

The cost computation is based on the metric being optimized. For example, for a



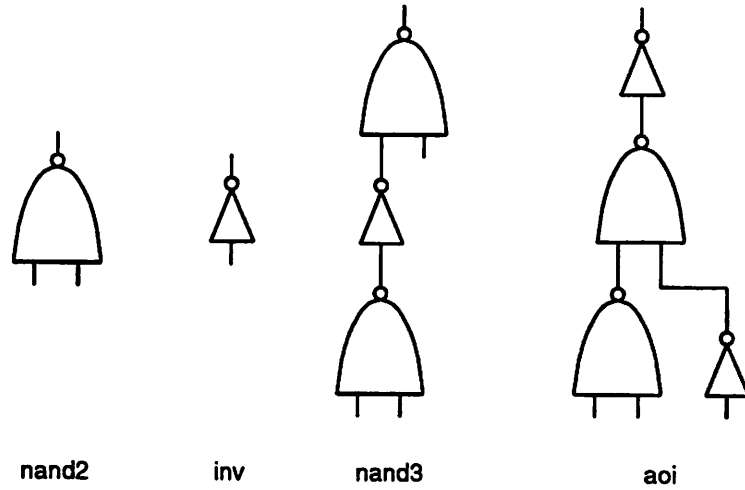


Figure 5.2: Example gate library with 4 gates. The library patterns are trees of INV and 2-input NANDs.

---

minimum-area solution, the cost of the match,  $m$ , at a node,  $n$ , is given by

$$cost(m) = area(m) + \sum_{v_i \in inputs(m)} cost(v_i).$$

Primary inputs have zero cost.

For a minimum-depth solution,

$$cost(m) = 1 + \max_{v_i \in inputs(m)} (cost(v_i)).$$

Primary inputs are at zero depth.

Using the delay model of Chapter 2, a minimum-delay solution can be obtained by computing the cost as

$$cost(m) = \gamma \times \#fanouts(n) + \max_{v_i \in inputs(m)} (cost(v_i) + delay_{v_i}(m)),$$

where  $\gamma$  is an estimate of the delay per fanout.  $delay_{v_i}(m)$  is the propagation delay from input  $v_i$  to the output, for gate  $m$ , and  $\#fanouts(n)$  is the number of fanouts of node  $n$ .

A combination of area and delay costs can be computed by using a linear combination, eg.,

$$cost(m) = w \times area\_cost(m) + (1 - w) \times delay\_cost(m)$$

where  $0 \leq w \leq 1$ .

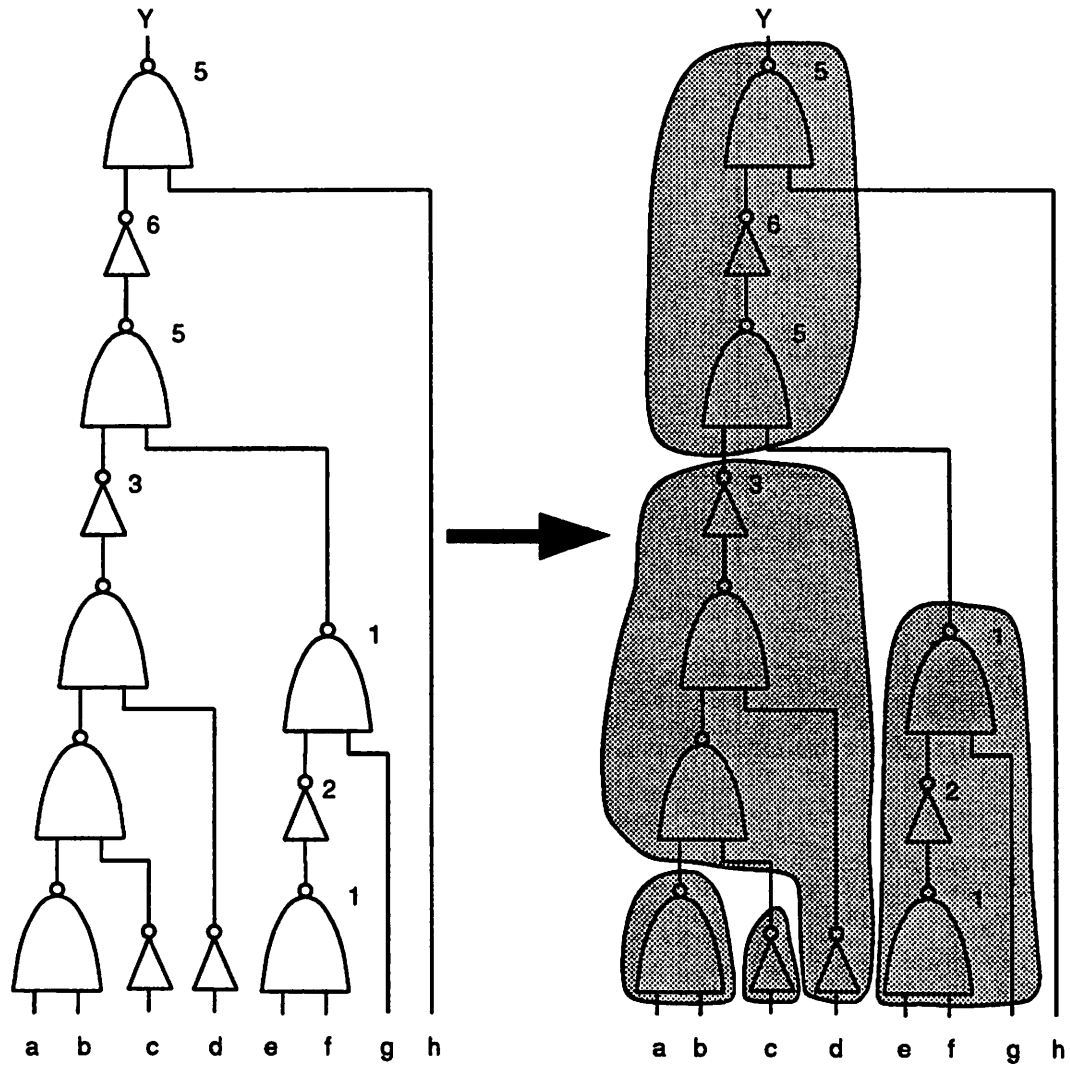


Figure 5.3: Tree covering for standard cells.

### 5.3.2 Partitioning DAGs

There are many ways in which the input network can be partitioned into a forest of trees. One approach is to partition at every node that has multiple fanouts (eg., see Figure 5.4). This could result in a lot of small trees, and the approach does not allow logic duplication. Another approach, single-cone partitioning, uses only primary outputs as roots of trees [Detjens 87]. This allows logic duplication (where the trees overlap). Usually, this approach yields better results for timing optimization.

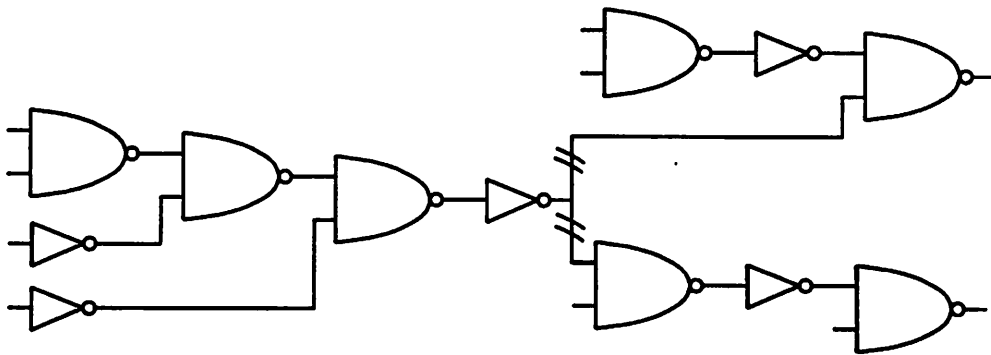


Figure 5.4: Partitioning a DAG by a break at every multi-fanout node

---

## 5.4 Why has this not been done before?

There have been attempts to extend the LBM approach to LUT mapping [Francis 92]. However, these did not meet with much success, since it was thought that the number of library patterns required to represent the LUT would be impractically large.

To quote from [Francis 92]: *The major obstacle to applying library-based technology mapping to LUT circuits is the large number of different functions that a  $K$ -input LUT can implement. The function implemented by a  $K$ -input LUT is determined by the values stored in its  $2^K$  memory bits. Since each bit can independently be either 0 or 1, there are  $2^{2^K}$  different Boolean functions of  $K$  variables. For values of  $K$  greater than 3 the library required to represent a  $K$ -input LUT becomes impractically large.*

A 4-input LUT can implement all the 65,536 functions of 4 variables. A 5-input LUT can implement all the 4,294,967,296 functions of 5 variables. The number of patterns

to be stored in the library can be reduced by noting that some patterns are equivalent after a permutation of inputs. Further reduction is possible by allowing functions differing by input or output inversions to share the same representation. In spite of this reduction, the number of functions is still impractically large. For the reader's convenience, the table of pattern count from [Francis 92] has been reproduced here as Table 5.3.

K	without permutations and inversions	with permutations	with permutations and inversions
2	16	12	4
3	256	80	14
4	65536	3984	232

Table 5.3: Previous library sizes, for  $K$ -input LUT mapping

## 5.5 TIL based LUT library

Previous attempts to use LBM for LUT mapping implicitly assumed that the library patterns would have to be represented in terms of 2-input NAND and INV primitive cells. However, the tree covering algorithm does not dictate the choice of primitive cells.

By choosing the primitive cells appropriately, the number of library patterns can be significantly reduced. This section shows how to use a 2-input LUT (TIL) as a primitive cell, and the advantages thereof.

### 5.5.1 TILs as primitive cells

A TIL can implement all functions of 2 variables. Hence it can *cover* any node with 2 inputs. Figure 5.1 introduces the symbol for a TIL.

A 3-input LUT can be decomposed into a tree of TILs. There are only 2 such trees: one with the root node having a left child as leaf and right child as a node with 2 children leaves; and the other with the root node having a left child as a node with 2 children leaves and the right child as leaf. Figure 5.5 shows these decompositions.

A 4-input LUT can similarly be decomposed into a tree of TILs. There are only 5 such trees. Figure 5.6 enumerates all the tree patterns that need to be stored in the mapping library, for a 4-input LUT based FPGA mapping. This library includes the 3-input and 2-input LUT patterns, since functions with 2 or 3 inputs can also be implemented in a

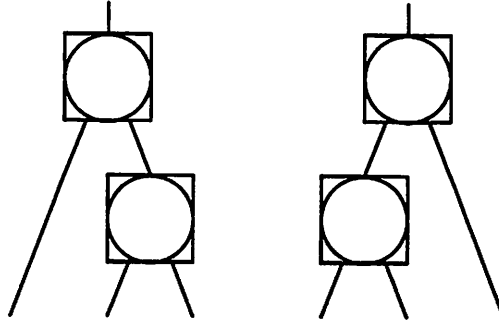


Figure 5.5: 3-input LUT as trees of 2-input LUTs

$K$	$K$ -input LUT Patterns	Total Patterns
2	1	1
3	2	3
4	5	8
5	14	22
6	42	64
7	132	196
8	429	625
9	1430	2055
10	4862	6917

Table 5.4: Number of patterns required to be stored for various values of  $K$ , where  $K$  is the number of LUT inputs.

4-input LUT. There are 8 patterns in all (For a  $K$ -input LUT mapping, patterns for all LUTs with  $\leq K$  inputs must be stored).

### 5.5.2 Small library

Table 5.4 tabulates the number of patterns per  $K$ -input LUT, and the total number of patterns required to be stored in the library, for values of  $K \leq 10$ . The number of patterns required for a  $K$ -input LUT is given by

$$N(K) = \sum_{i=1}^{K-1} N(i)N(K-i),$$

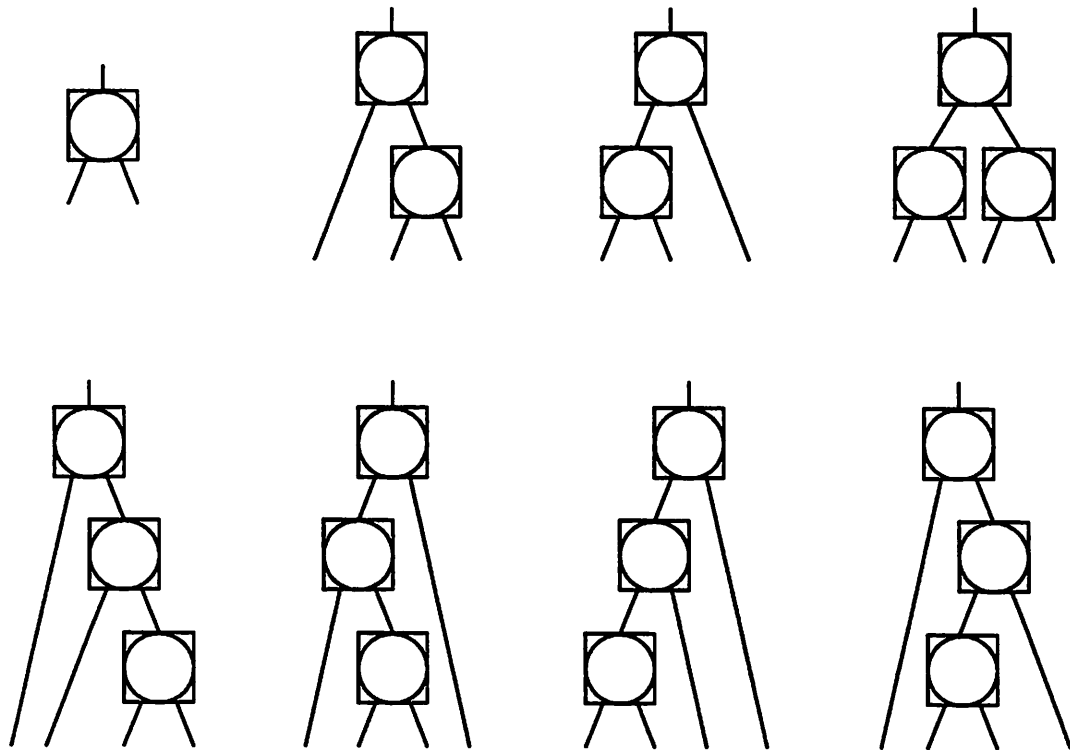


Figure 5.6: Library patterns for a 4-input LUT mapping

---

with  $N(1) = N(2) = 1$ . A  $K$ -input LUT library will have  $S(K)$  patterns, where

$$S(K) = \sum_{i=2}^K N(i).$$

Compared to Table 5.3, we note that this library is significantly smaller.

## 5.6 dpmap Algorithm

To use the concept of the TIL as primitive cell, the subject graph should be such that each node in the subject graph can be covered by any TIL. In other words, every node in the subject graph should have exactly two inputs. The node can have any function, the only restriction is to limit the number of inputs to 2. Such a graph is called a TINN (for Two-Input Node Network). Once the subject graph has been converted into a TINN, the tree covering algorithm can be used to cover the TINN by the LUT patterns (of the previous section). The complete mapping procedure therefore has the following main steps.

1. Convert the input network into a TINN.
2. Represent the LUT in the manner shown in the previous section.
3. Use the tree-covering approach to cover the TINN with the LUT patterns.

### 5.6.1 Input network $\rightarrow$ TINN

As a simple extension of the standard cell based technique, the input network can be converted into a TINN in the following manner (see Figure 5.7).

1. Decompose the input network into 2-input NANDs and INVs, using the same technique as for standard cell technology mappers.
2. Absorb each INV into its fanout or fanin gate, whichever is convenient (eg., if an INV fans out to multiple nodes, absorb it into its fanin).

An alternate technique would be to have a complete 2-input library, as used by conventional standard cell mappers. The input network is first mapped into the gates of this library, using a standard cell technology mapper.

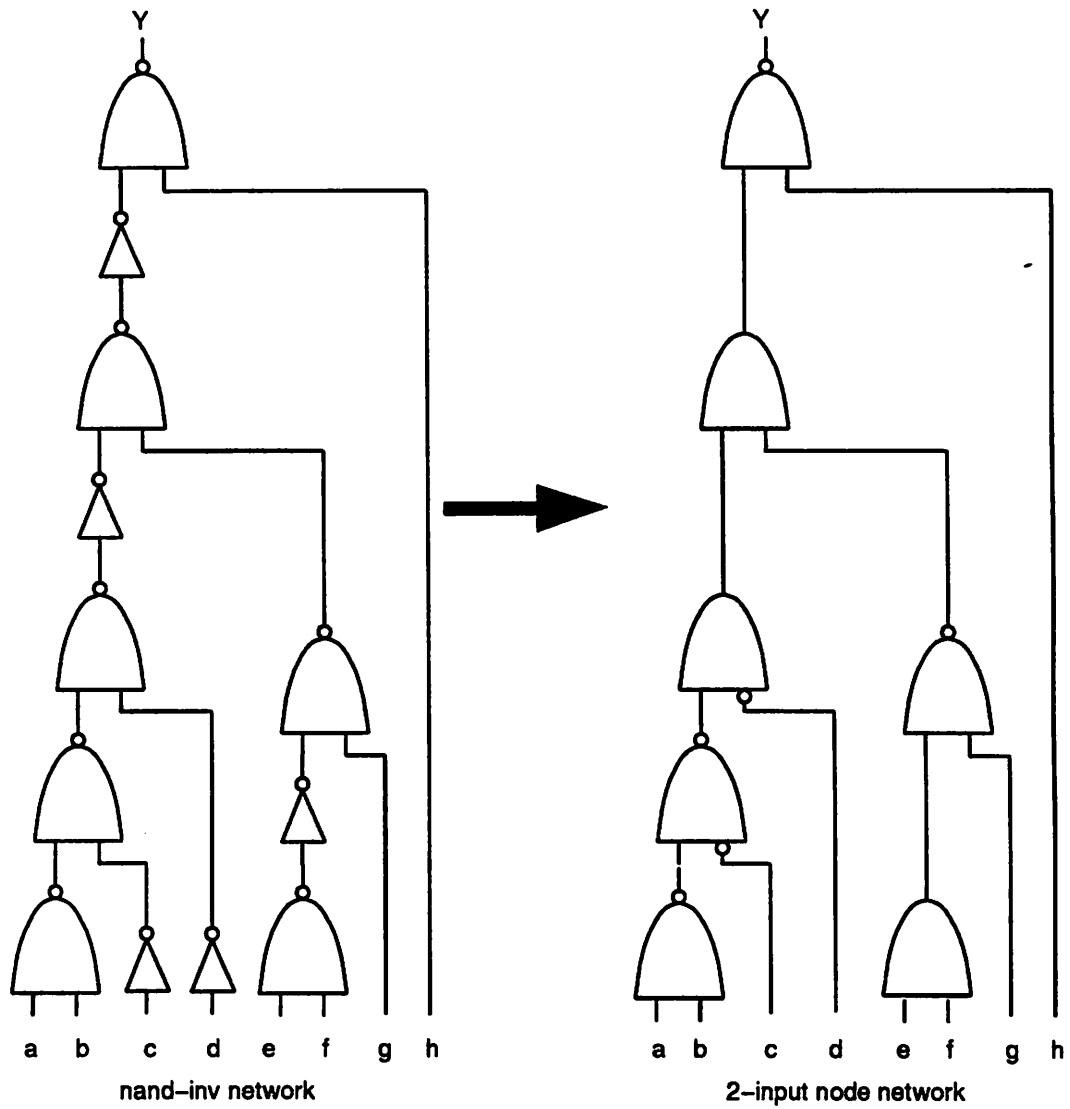


Figure 5.7: The network of Figure 5.3 is converted into a TINN by absorbing the inverters.

---



### 5.6.2 Tree covering

We set the cost for match  $m$ , at node  $n$ , as

$$cost(m) = w \times area\_cost(m) + (1 - w) \times delay\_cost(m).$$

The *area\_cost* is computed as

$$area\_cost(m) = 1 + \sum_{v_i \in inputs(m)} area\_cost(v_i),$$

where the '1' represents the cost of a single LUT. The *delay\_cost* is computed as

$$delay\_cost(m) = d_{LUT} + \gamma \times \#fanouts(n) + \max_{v_i \in inputs(m)} (delay\_cost(v_i)),$$

where  $\gamma$  represents the delay per fanout and  $d_{LUT}$  represents the delay through the LUT.  $\#fanouts(n)$  is the number of fanouts of  $n$ .

Using the above cost functions, four different mapping solutions are possible:

1. **Performance-driven**: Net delay is taken into consideration by setting an appropriate value for  $\gamma$ .
2. **Area-delay trade-off**: Stepping through different values of  $w$ , solutions with different areas and delays are obtained.
3. **Minimum depth**:  $w$  is set to 0,  $\gamma$  is set to 0 and  $d_{LUT}$  is set to 1. This gives a mapping solution with minimal topological depth.
4. **Minimum area**:  $w$  is set to 1. This gives a mapping solution with minimal LUT count.

Figure 5.8 shows how the network of 2-input nodes of Figure 5.7 is covered by the LUT patterns of Figure 5.6.

### 5.6.3 Handling reconvergent fanout

Consider the DAG of Figure 5.9. Nodes  $b$  and  $c$  have 2 fanouts each. The fanouts of  $b$  reconverge at node  $Y$ . Let us assume that we are using a 3-input LUT library (see Figure 5.5). The function at  $Y$  can be expressed as a 3-input function with inputs  $a$ ,  $b$  and  $c$ , thereby implying that a 3-input LUT pattern with inputs  $a$ ,  $b$  and  $c$  should match at node  $Y$ . However, using the library patterns of the Section 5.5, we will need a 5-input LUT pattern (see Figure 5.10) to cover the nodes of the 3-input LUT match at node  $Y$ .

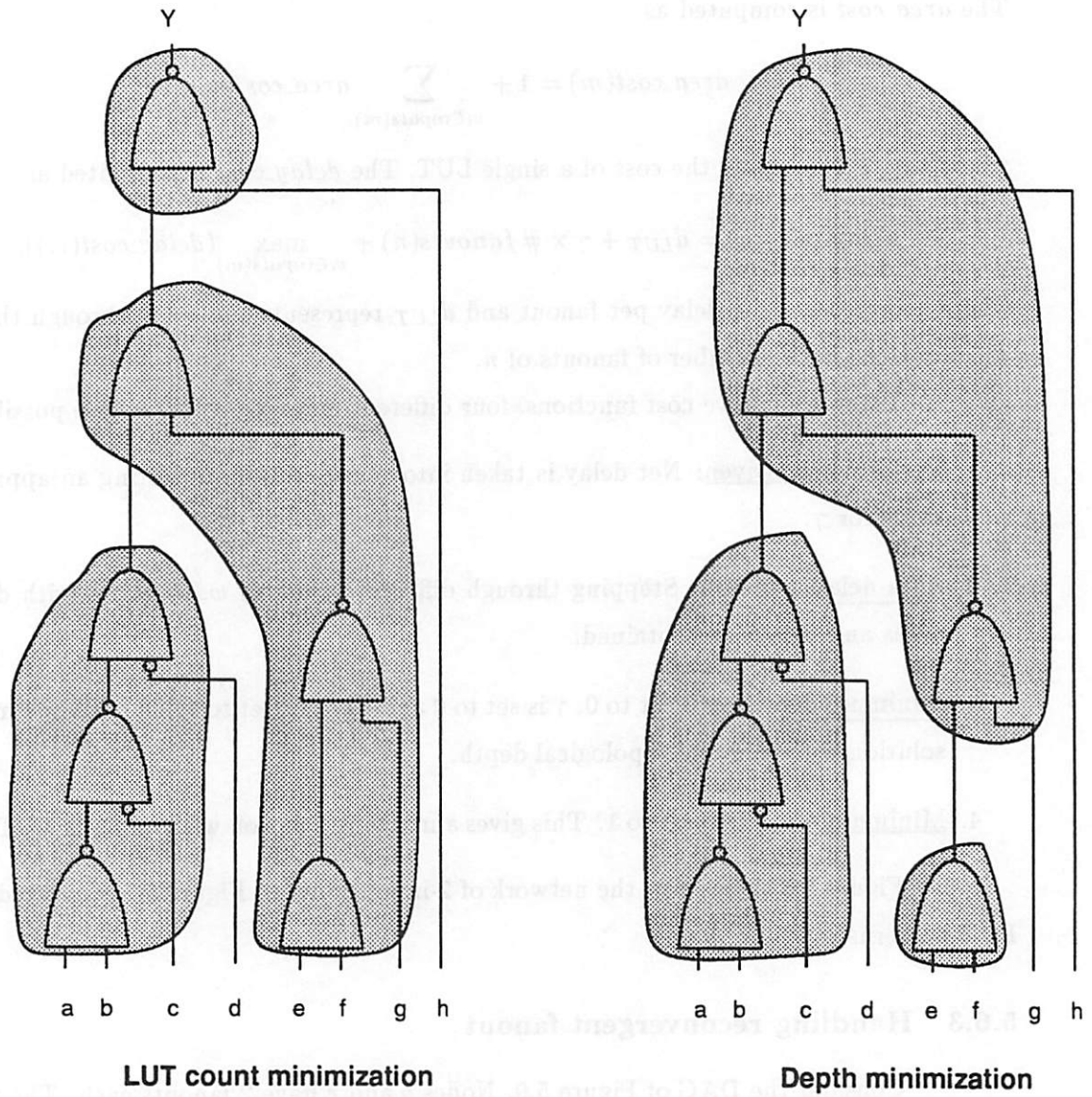


Figure 5.8: 4-input LUT mapping

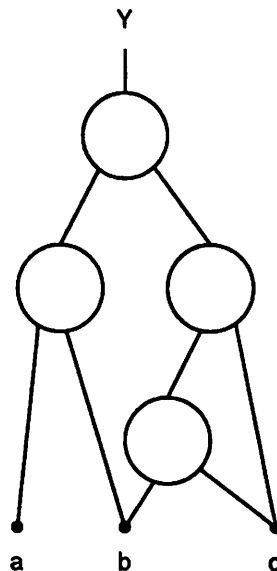


Figure 5.9: A 5-input LUT pattern is required to cover a 3-input LUT match, because of reconverging fanouts.

---

Unless we use the 5-input LUT pattern, this match at node  $Y$  will not be found. This example suggests a general strategy to handle reconverging fanouts. If we are performing a  $K$ -input LUT mapping, we must use a  $(K + s)$ -input LUT library during the matching process. Here,  $s (> 0)$  is called the *safety-factor*. For each  $(K + i)$ -input LUT match,  $m_{k+i}$ , where  $0 < i \leq s$ , the *actual* number of inputs,  $n_{act}$ , is counted using the formula

$$n_{act} = K + i - n_{fo}.$$

$n_{fo}$  is the number of fanins that fan out to more than one of the match's inputs. If  $n_{act} > K$ , the match is discarded. Otherwise, the match is treated as valid and its cost is computed.

Experiments show that values of  $s > 5$  do not alter the mapping solution, and hence  $s \sim 5$  seems a reasonable value for the safety-factor.

## 5.7 Fast tree matching

It can be observed that the LUT patterns are very regular. A LUT with  $K$  inputs has patterns constructed from patterns of LUTs with  $x$  inputs,  $x < K$ . For example, in

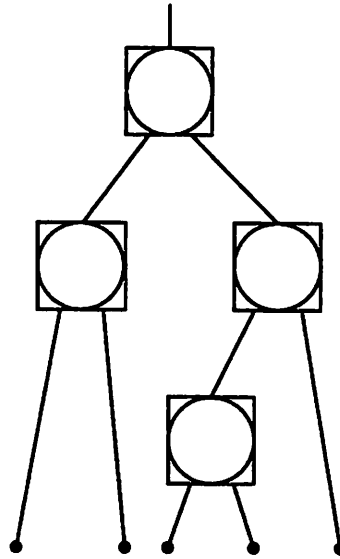


Figure 5.10: This 5-input LUT pattern can cover the nodes in Figure 5.9.

Figure 5.6, the 4-input LUT patterns are constructed from 2-input LUT patterns and 3-input LUT patterns. Using this observation, we can speed up the pattern matching process in the following manner.

### 5.7.1 Principle

At each tree node, an array of best costs is stored. The array index corresponds to the number of inputs in the matching pattern. At index  $i$ , we store the combination <sup>2</sup> of the fanin costs of the best match with  $i$  inputs. To find the best  $K$  input pattern match at node  $n$ , we combine the best  $x$  input pattern cost of the left child of  $n$  with the best  $K - x$  input pattern cost of the right child of  $n$ , for  $x = 1 \dots (K - 1)$ , and store the pattern with the best cost. The best cost amongst all patterns with  $i$  inputs,  $1 < i \leq K$  is stored at index 1. The cost at index 1 includes the cost of the matching pattern, whereas the costs at the other indices represents only the fanin costs. For example, consider the case of LUT count minimization. Let  $N$  be the node under consideration. If  $c(i)$  represents the cost at

<sup>2</sup>The manner in which the fanin costs are to be combined is dependent on what is being optimized. If it is LUT count, then the combination is an arithmetic sum of the fanin costs; if it is delay, then the combining operation chooses the maximum of the fanin delays.

index  $i$ , then the minimum fanin cost is given by

$$m = \min_{1 < i \leq K} \{c(i)\},$$

then

$$c(1) = m + A_1,$$

where  $A_1$  ( $= 1$ ) is the area of a LUT.

An  $i$ -input pattern match requires  $(i - 1)$  cost computations, and since a  $K$ -input LUT mapping requires  $(K - 1)$  pattern matches, at each node we require

$$\sum_{i=2}^K (i - 1) = \frac{K(K - 1)}{2}$$

cost computations. Thus the number of pattern matches are  $O(K^2)$ . This means that the effective library size is  $O(K^2)$  as a consequence of fast-matching.

### 5.7.2 Example

The fast matching technique is best illustrated by means of an example. Consider Figure 5.11 where we show the mapping procedure, for the TINN of Figure 5.7. The objective is to obtain a minimum area (LUT count) mapping solution. At each node, an array of best costs is shown. The array index is denoted by  $i$ , and runs left to right. The area cost (i.e., LUT count) is shown below each index. All the primary input nodes (i.e.,  $a$ ,  $b$ , etc.) have 0 area cost. For example, consider the array at the left of the 2-input NAND gate  $S$ . There is only one LUT pattern matching at this gate, and that is the 2-input LUT pattern. The fanins to this pattern are  $a$  and  $b$ , and each has a fanin cost of 0. Hence, the sum of the fanin costs (since we are minimizing area, the combination is a sum) is 0, and this is stored at index 2. Assuming all LUTs have the same area cost of 1, the best cost at  $S$ , stored at index 1, is  $= 0 + 1 = 1$ .

As a more detailed example, consider the computation of the best 4-input LUT pattern match at node  $N$  in Figure 5.11. The following costs have to be first computed.

1.  $C_{13}$ , corresponding to the left child's 1-input and right child's 3-input costs, as shown in Figure 5.12.
2.  $C_{22}$ , corresponding to the left child's 2-input and right child's 2-input costs, as shown in Figure 5.13.

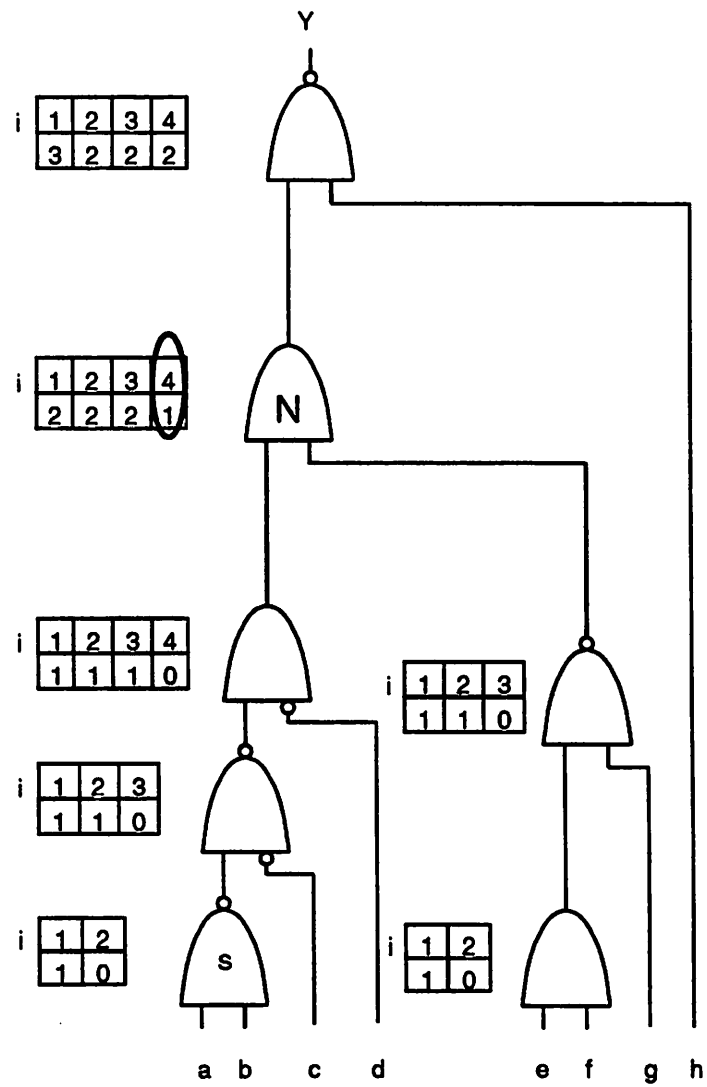


Figure 5.11: Example to illustrate fast tree-matching

fanin cost =  $1 + 0 = 1$   
 total cost =  $1 + 1 = 2$

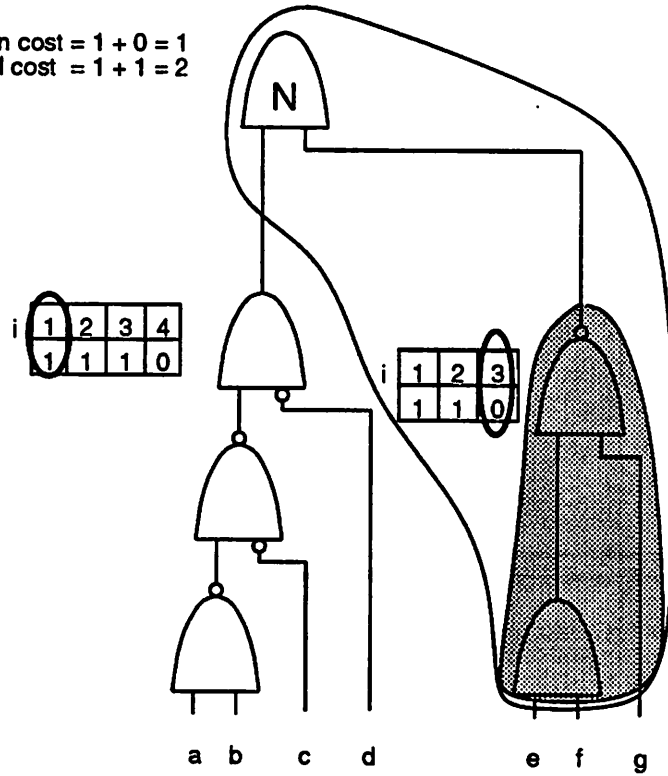
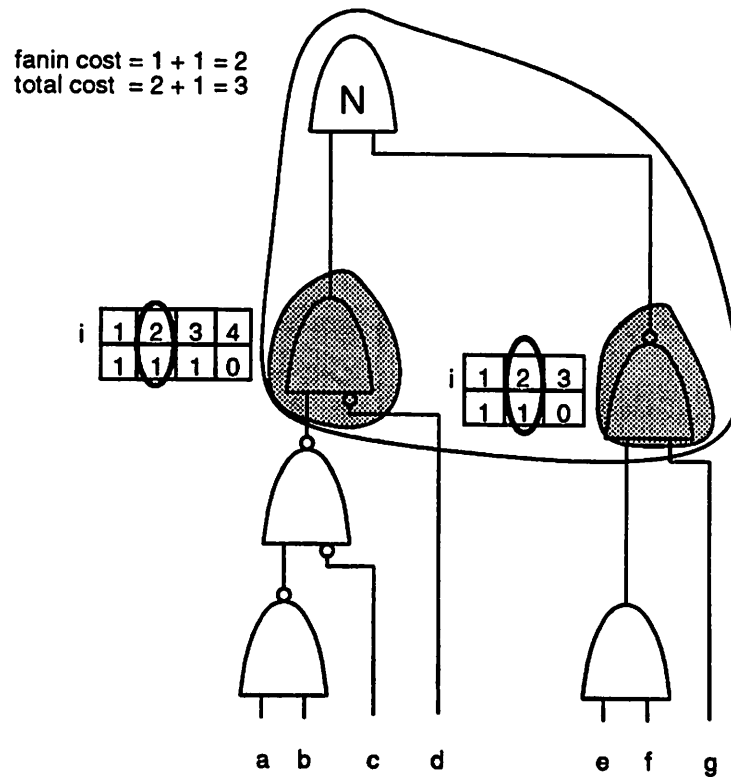


Figure 5.12: Computing cost  $C_{13}$ .

Figure 5.13: Computing cost  $C_{22}$ .

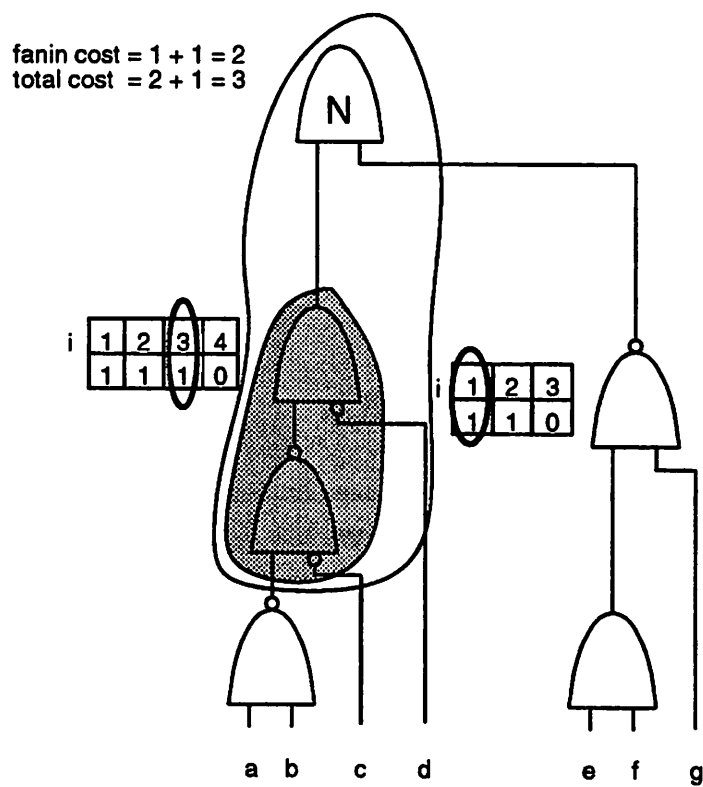
3.  $C_{31}$ , corresponding to the left child's 3-input and right child's 1-input costs, as shown in Figure 5.14.

Each cost computation involves just two memory look-ups and one addition, and is hence computationally inexpensive. The minimum of  $C_{13}$ ,  $C_{22}$  and  $C_{31}$  is stored as the fanin cost at index 4, for the array at node  $N$  in Figure 5.11.

### 5.7.3 Drawbacks

The fast tree matching approach gives optimal results when the subject graph is a tree, with leaves having only one fanout each. If the leaves have more than one fanout, then the approach can give sub-optimal results. Consider the mapping shown in Figure 5.15(a). The tree shown is a part of a larger circuit. The best costs, for area minimization, are shown in parentheses at each node. Nodes  $a$ ,  $b$ ,  $e$  and  $f$  are multi-fanout nodes, and we assume



Figure 5.14: Computing cost  $C_{31}$ .

that the DAG has been partitioned into trees by breaking at such multi-fanout nodes (see Figure 5.4).

In Figure 5.15(a), consider the 3-input LUT matching at node  $w$ . Two matches are possible, as shown by the ovals in the figure. However, the shaded oval has lower cost (5) compared to the unshaded oval (6). Hence the shaded oval is stored as the best 3-input match, at array index 3 at node  $w$ . However, this locally optimal solution (locally at  $w$ , that is), is not globally optimal. Consider node  $x$ , where the multiple fanouts of node  $b$  reconverge. Let us consider a 4-input LUT matching at node  $x$ . Three possible 4-input LUT matches are possible, as shown in Figures 5.15(b), 5.15(c) and 5.15(d). Figure 5.15(d) has the least area cost; but the fast matching approach cannot find this match. This is because the best match is obtained by combining the unshaded oval match at node  $w$  with the 2-input match at node  $p$ . However, fast match stores only a single match at each index. Since the unshaded oval was discarded in favor of the shaded oval at node  $w$ , the best 4-input LUT match at node  $x$  is lost.

## 5.8 Experimental results

The `dpmap` algorithm has been implemented in the SIS framework. Experimental results are reported in this section. We have compared our performance-driven mapping results with `mis_pga`. Area-delay results are also reported. In addition, for sake of comparison with prior work, we also ran the mapper with depth minimization and area minimization options.

For all our experiments, we started with a network optimized by standard methods [Saldanha 89]. We decomposed the input network into a network of 2-input nodes, using a 2-input library (the second approach described in the Section 5.6.1). The network was then partitioned into a forest of trees and the tree-covering algorithm was used on each such tree. We tried both techniques of partitioning a DAG into trees (as described in Section 5.3.2), and we have reported the better of the two results (since `dpmap` is fast, it is easy to try both techniques). We performed a 5-input LUT mapping (i.e.,  $K = 5$ ) and used fast tree matching. To account for reconvergent fanouts, we set the safety factor,  $s$ , to 5. Our MIS script is: `read_library 2ip.genlib, map, dpmap`. (`dpmap` performs the LUT mapping). `dpmap` is fast, and in general takes less than a minute of CPU time, on a DEC 5000.

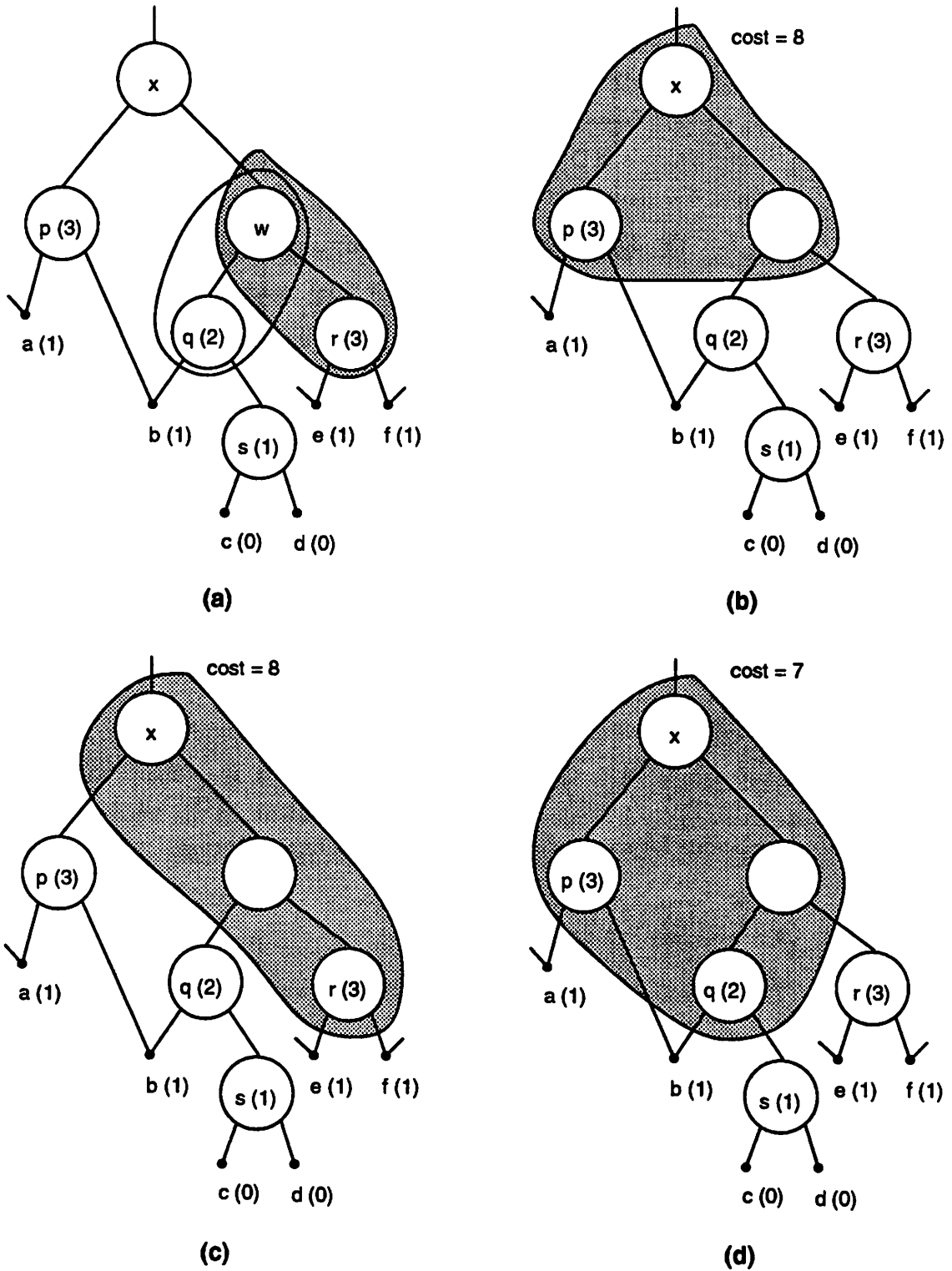


Figure 5.15: Example to illustrate the fast matching drawback

### 5.8.1 Performance-driven mapping

In this experiment, the net delays are incorporated into the cost computation by using the delay model of Chapter 2. The mapping solutions were placed and routed on the Xilinx XC3090 [Xilinx 89], using Xilinx’s `apr` software and a delay analysis was performed using Xilinx’s `XDelay` timing analyzer. Table 5.5 shows the delays for some MCNC examples (we have shown only examples with delays more than 100 ns, and those that could fit into the 3090). Our results are compared with `mis_pga`. It was found (by trial and error) that a  $\gamma$  of 1.7 ns/fanout gives a good estimate of the net delay, when the number of logic blocks is less than 60% of the maximum. The Xilinx part used had a CLB delay of 5.5 ns, and hence  $d_{LUT}$  was set to 5.5 ns.

Example	<code>dpmap</code> delay (ns)	<code>mis_pga</code> delay (in ns)
sao2	104.9	96.6
alu4	187.2	207.3
duke2	132.1	176.0
C499	118.8	216.8
apex2	104.0	124.0

Table 5.5: Performance-driven mapping results of `dpmap`. Delays are tabulated after place and route.

From the table, it can be observed that `dpmap` does significantly better, in all examples except `sao2`. A single value of  $\gamma$  has been used for all the benchmarks. This value may not be suitable for some examples, and `sao2` is one such case.

### 5.8.2 Area-delay trade-off

LBM provides a framework to do effective area-delay trade-offs. In Figure 5.16, we see that by varying the value of  $w$ , the relative weights of the area and delay costs (see Section 5.6.2), `dpmap` generates solutions with different number of LUTs and levels. The experiment was conducted with  $\gamma = 0$  and  $d_{LUT} = 1$ , as in the case of the minimum depth experiment.

### 5.8.3 Minimum depth

To put this work in perspective with published techniques, we compare the minimum depth mapping solution of `dpmap` with `flowmap`[Cong 92] in Table 5.6. `flowmap`

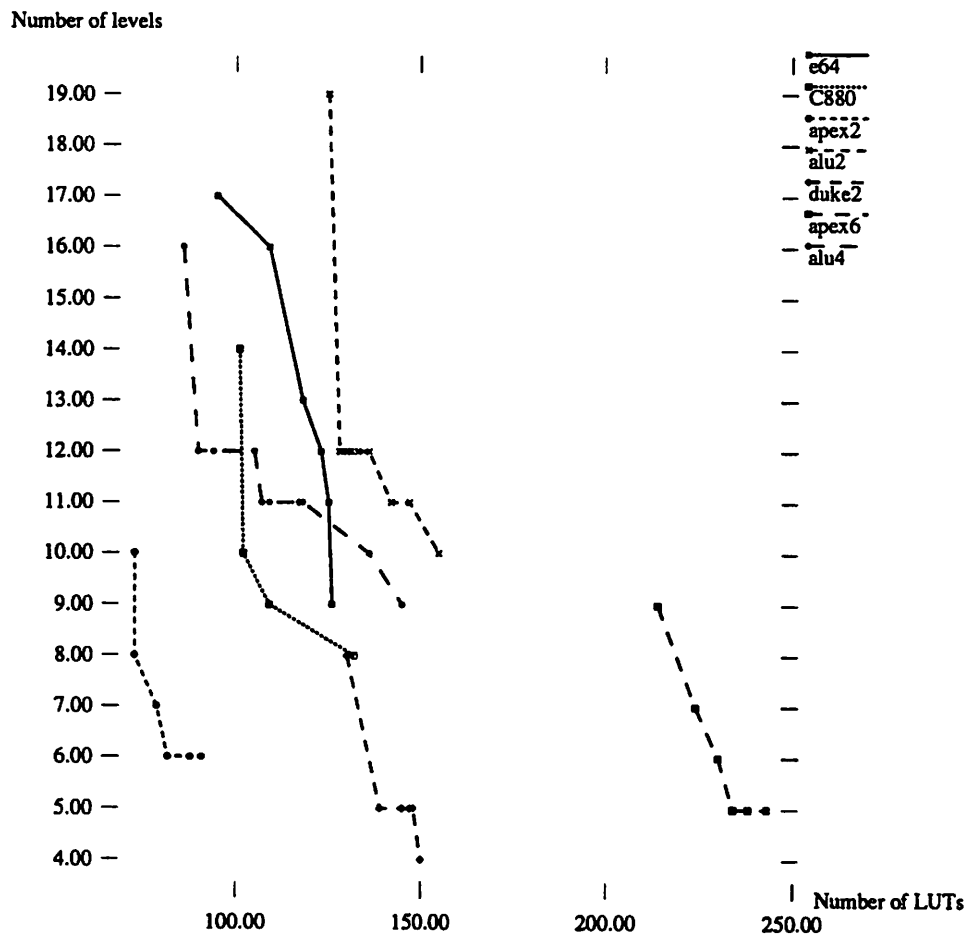


Figure 5.16: Area-delay trade-off.

Example	dpmap		flowmap			
	LUT	dpt	2ip lib		dmig	
			LUT	dpt	LUT	dpt
z4ml	8	3	7	2	5	2
misex1	14	3	18	3	19	3
vg2	34	3	33	3	52	4
count	40	5	38	5	43	5
9symml	51	4	63	4	96	5
9sym	69	5	88	5	148	5
apex7	78	4	91	4	100	4
rd84	38	4	54	4	70	4
e64	126	9	132	9	132	9
C880	132	8	152	8	146	8
apex2	91	6	112	6	192	5
alu2	155	10	175	10	168	9
duke2	150	4	203	4	203	4
C499	74	4	78	4	78	4
apex6	243	5	305	5	397	6
alu4	145	9	142	9	253	9
sao2	47	6	42	6	77	7
rd73	25	4	27	4	30	5
misex2	36	3	54	3	53	3
clip	36	5	40	4	52	4
b9	52	3	54	3	92	3

Table 5.6: Minimum-depth mapping solutions. LUTs columns list the number of 5-input LUTs and depth columns list the maximum topological levels.

gives the optimal depth mapping solution for a given input network. `flowmap` was run in the default mode. Before executing `flowmap`, the input network must be *feasible*, i.e., the number of inputs for every node should be less than the LUT inputs. Typically, the number of inputs is restricted to 2. The `flowmap` package has a command `dmig` to convert the input network to a feasible network. However, we found that our 2-input library based conversion yields better results. Both methods are listed in Table 5.6. From the table, we observe that `dpmap` produces the minimum depth solution (comparing with the *2ip lib* column of `flowmap`) in all but 2 cases (`z4ml` and `clip`). The LUT count is lesser by 11% in the case of `dpmap`.

Example	dpmap	mis_pga	chortle-crf	Xmap
z4ml	6	5	7	9
misex1	13	11	11	11
vg2	25	20	21	24
count	43	31	31	31
9symml	46	7	44	55
9sym	65	7	59	73
apex7	67	60	60	65
rd84	37	10	35	36
e64	95	80	80	80
C880	101	82	88	103
apex2	73	67	64	81
alu2	125	109	116	126
duke2	130	110	111	127
C499	81	68	89	75
apex6	214	182	198	231
alu4	86	55	70	98
sao2	33	28	27	37
rd73	21	6	16	21
misex2	34	28	28	28
clip	35	28	31	38
b9	45	39	41	48

Table 5.7: Minimum-area mapping solutions. The table lists the number of 5-input single-output LUTs

#### 5.8.4 Minimum area

Table 5.7 shows the number of 5-input LUTs needed to implement several benchmark examples. Results are compared with `mis_pga`[Murgai 91], `chortle-crf`[Francis 91] and `Xmap`[Karplus 91]. The numbers for these other algorithms are taken from [Murgai 91]. All use the same starting network.

`dpmap` always does worse than `mis_pga` (`mis_pga` has better area results because it performs logic optimization for LUTs along with mapping), and does worse than `chortle-crf` in all but one example (C499). But it is encouraging to note that our simple covering technique gives comparable results. Different decomposition techniques for the 2-input network generation, and logic optimization techniques for LUTs need to be tried to improve the results.

## 5.9 Conclusions

In this chapter, we showed how to extend the performance-driven mapping and area-delay trade-off capabilities of standard cell based mappers to LUTs. We introduced the 2-input LUT primitive cell, and used it as a building block to generate a small, practical library. Using a linear delay model, we accounted for net delays during the dynamic programming based mapping algorithm, and hence made the **dpmmap** LUT mapper a truly performance-driven mapping algorithm. The library-based approach gives the mapper the ability to be used across a *class* of LUT based FPGA architectures. The special structure of library patterns was exploited to reduce effective library size to  $O(K^2)$ , for  $K$ -input LUT mapping.

Experimental results indicate that the timing delays, after place and route are better than **mis\_pga** by about 17.5%. Area-delay trade-offs show that the solutions can span across as many as three Xilinx chips; thus giving the user the choice between area, speed and dollars. Depth minimization results are as good as the **flowmap** results in all but two examples; and **dpmmap** has 11% less LUTs.

**dpmmap** suffers from the disadvantage that the solution is dependent on the initial decomposition of the network into 2-input nodes, a problem inherent with LBM techniques. This explains the poor area-minimization results. A second disadvantage is that the LBM approach cannot handle multi-output gates. Most of the LUT based FPGAs have multi-output logic modules (eg., the XC3000 has 2-output LUTs). Mapping for such architectures has to proceed in two steps:

1. Map for single output LUTs
2. Combine the mapped LUTs, using techniques described in [Murgai 91].

This work can be extended to integrate mapping with placement (as in [Pedram 91]) for better net delay estimations. Since LBM is widely used for standard cells, there is a large body of researchers in this field, and any new developments can be easily incorporated into **dpmmap**. Libraries for other LUT based architectures, like the Xilinx XC4000 series and the AT&T ORCA need to be developed. The TIL need not be the only primitive cell; number of patterns built from 3-input LUTs are also small, and the mapping using these patterns needs to be investigated.



**Part II**

**Performance-oriented  
Architecture**

## Chapter 6

# Overview

In this part of the thesis, we focus on the device architecture of FPLDs. In Part I, the question we tried to answer was: *Given a class of devices (viz., LUT FPGAs), how best can we design the mapping software?* In Part II, our question becomes more general: *Given that we need to design a PLS, how best can we design the device?*

In Chapter 1, we listed the desirable features of a PLS, viz., automatic, performance-driven, fast CAD tools and a silicon-efficient, low-delay FPLD. But, CAD and the device are interlinked. We cannot design fully automated CAD if the device architecture is very complex and causes nets to remain unrouted. Nor can it be made performance-driven, if delays in the interconnect are hard to predict. Similarly, the silicon-efficiency of a device depends on how well the resources can be utilized by software aids. It is easy to design an architecture which results in intractable solutions for the software.

Keeping in mind that the fixed quantity of logic and interconnect resources on a device limits the freedom of the software, an architecture that is simple and easy to design with and at the same time utilizing silicon efficiently and having predictable interconnect delays is needed. We further motivate the need for a new architecture in Chapter 7.

Routing is the stumbling block for FPLDs. Device architects have come up with complex routing architectures, as can be seen in the XC4000 and the AT&T ORCA devices. These require intelligent and complex routing software. And to ensure that routing completion does not become a problem, abundant routing tracks are being provided in the newer devices, resulting in silicon wastage. We design our new architecture, *Dharma*, with the intent of keeping routing simple. The simplest way is to provide *full* connectivity, allowing any logic module to be connected to any other. Of course, if we bluntly apply this idea, the

device will end up having nothing but routing! To make this full connectivity possible, it should be possible to time-multiplex the routing hardware. In Chapter 8, we introduce the concept of *levelize-and-hold*, and show how interconnect resources can be time shared. The idea is to implement the multi-level combinational part of the circuit in a folded-pipeline manner, re-configuring logic and interconnection in real time. The *Dharma* architecture is then described in the same chapter.

To illustrate the *Dharma* architecture better, three example circuit implementations are shown in Chapter 9; one sequential and two combinational circuit examples are shown.

As brought out in Chapter 1, the idea of real time reconfiguration can foster families of FPLDs. Several variants of the basic architecture are possible, and some of these are described in Chapter 10.

Synthesis techniques for *Dharma* are then described in Chapter 11. We consider only physical synthesis. Given a net-list, we need to allocate the logic modules to different time slices, in a manner such that on-chip resources are not exceeded. We show that this problem can be formulated as an integer programming problem. We also show a modified Kernighan-Lin technique to solve the allocation problem, and present experimental results for this approach.

Chapter 12 presents analysis of the new architecture. Area and timing analyses are performed, and timing equations are presented. Several circuit examples are compared with the XC3000 performance, using the delay equation.

The chapter on “Unexplored terrain” discusses potential ways to further the architecture related work. *Dharma* is a CAD-oriented architecture, the motivation being to capitalize on software strengths, while avoiding software weaknesses. We point out two synthesis problems which, if solved, can result in improved *Dharma* devices.

## Chapter 7

# Why Another Architecture?

This part of the thesis presents a new architecture for an FPLD. Before we introduce the new architecture, we must examine the need for another architecture. As seen in the FPLD classification tree (ignoring the shaded portion), Figure 1.3, there are already many different kinds of FPLDs, and one may have the erroneous impression that there are enough devices out there already. In this chapter, we bring out the shortcomings of these devices, justify the need for a better device, and also list the features that are desirable in an FPLD. The next chapter will then present the new architecture that meets these desired features.

As described in Section 1.3, the programmable device forms a part of the PLS, and for it to become a viable alternative to other ASIC design styles, the PLS must have capabilities to meet high performance application requirements.

But, if we examine the history of FPLD evolution, we observe that these devices started off as a means for fast prototyping, essentially to verify the design's logic function correctness, with no regard to timing correctness. Over the years many different architecture styles have cropped up, with increased complexity, and more features, but without any real consideration for the overall PLS improvement or performance improvement.

Let us concentrate on FPGAs, because at the present time only they have the capability, albeit limited, to meet the high-performance ASIC market requirements. PLDs are not complex enough to implement large circuits.

Designing circuits on FPGAs has proceeded very much along the lines of MPGAs (see Figure 1.4):

1. Map the circuit gates into the logic blocks of the FPGA chip.
2. Place the logic blocks on the FPGA plane.
3. Route the connections using the FPGA programmable routing resources.

If performance (clock speed of the FPGA-implemented circuit) is made a criterion, at present there is no commercial FPGA chip and accompanying CAD tool which can guarantee a given clock speed, or perform the above 3 steps while taking into consideration both logic and routing delays. As a result, it is not yet possible to fully utilize a programmable chip in high-performance designs.

Three kinds of drawbacks arise in current FPGAs <sup>1</sup> which use these steps in the design process. We discuss them under the subheadings of Routing issues, Timing issues and CAD issues.

## 7.1 Routing issues

As outlined in Section 4.1, the programmable interconnection in current FPGAs presents two kinds of problems, routing completion (discussed in this section) and dominant timing delay (discussed in the next section).

Two approaches have come up to tackle the routing completion problem:

1. Architecture approach
2. CAD approach

### 7.1.1 Architecture approach

In the architecture approach, the belief is that the routing completion is not a real problem, and it arose simply because there were inadequate routing resources on the chip. Therefore, the solution is to increase the routing resources and routing flexibility. Consider the Xilinx FPGA chips. Designers complain of routing completion in the XC3000, but not in the case of the XC2000 or the XC4000. When XC3000 evolved out of the XC2000, logic modules were made more complex, and the number of modules per chip was increased, but the routing structure was left unchanged. However, the latest XC4000

---

<sup>1</sup>We concentrate on LUT FPGAs, since these are the most popular (in terms of number of units sold).

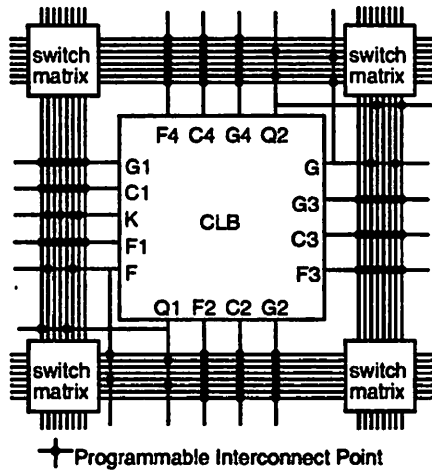
chips have significantly greater routing resources. Compared to the previous families, the XC4000 has twice as many horizontal and vertical *Long Lines* that can carry signals across the length or width of the chip; and the number of globally distributed clock signals has been increased from two to four. Compared to the XC3000 family, the XC4000 family has more than double the routing resources, and they are arranged in a more regular fashion [Alfke, Hsieh 90]. Figure 7.1 (from [Hsieh 90]) shows the routing flexibility and the four kinds of routing resources, viz., single lines, double lines, long lines and switch matrix, of the XC4000 family.

Similarly, the AT&T ORCA series of chips also have an abundance of routing resources [ORCA 93]. ORCA routing resources are made from metal segments called resource routing nodes (R-nodes). These R-nodes are connected together at configurable interconnect points (CIPs) to form user-defined nets. There are two types of R-nodes: internal and inter-PLC (programmable logic cell). The internal R-nodes are used to route signals within the PLC. Internal R-nodes consist of programmable function unit (PFU) input R-nodes, PFU output R-nodes, switching R-nodes, and bidirectional R-nodes. Inter-PLC R-nodes are used to route signals from one PLC to another. These are known as x1 R-nodes, x4 R-nodes, xL R-nodes, and direct R-nodes (see Figure 7.2). The x1 R-nodes are one PLC long, the x4 R-nodes are 4 PLCs long, and the xL R-nodes span the entire length or width of the ORCA chip. The direct R-nodes allow high-speed connections to directly adjacent PFUs on all four sides without using inter-PLC routing resources.

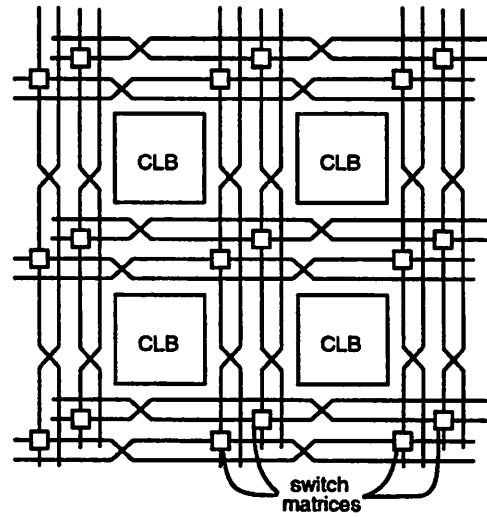
From Figures 7.1 and 7.2, we see that the architectural approach favors increase in the routing resources and routing flexibility. This eases the task of the router, and routing completion is usually possible. However, silicon resource has been wasted, and because of larger number of switches per track, the parasitic capacitance on the track is quite large, and hence causes poor timing performance.

### 7.1.2 CAD approach

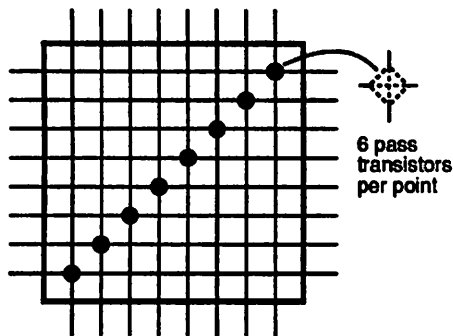
In the CAD approach, the CAD tool designer assumes that the problem with routing is because of ineffective CAD tools, and hopes to rectify the problem by modifying the CAD tool. One such CAD approach has been discussed in Chapter 4. Other approaches [Schlag 92, Trimberger 92] also use a similar principle, and try to modify the synthesis solution to alleviate the problem of incomplete routing. All these approaches have met with



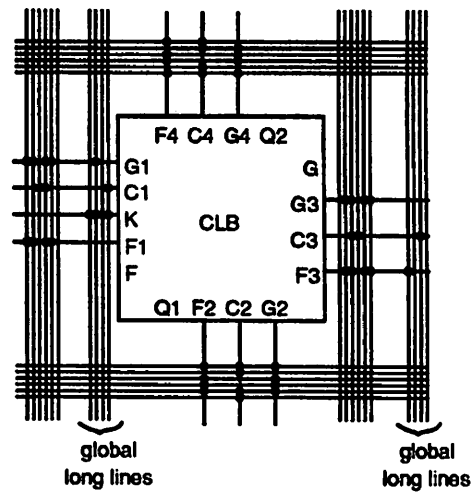
(a) CLB connections to single-length lines



(b) Double-length lines



(c) Switch Matrix connections



(d) Long Line resources

Figure 7.1: XC4000 family routing resources. This family has more than twice the routing resources of its predecessor, the XC3000 family.

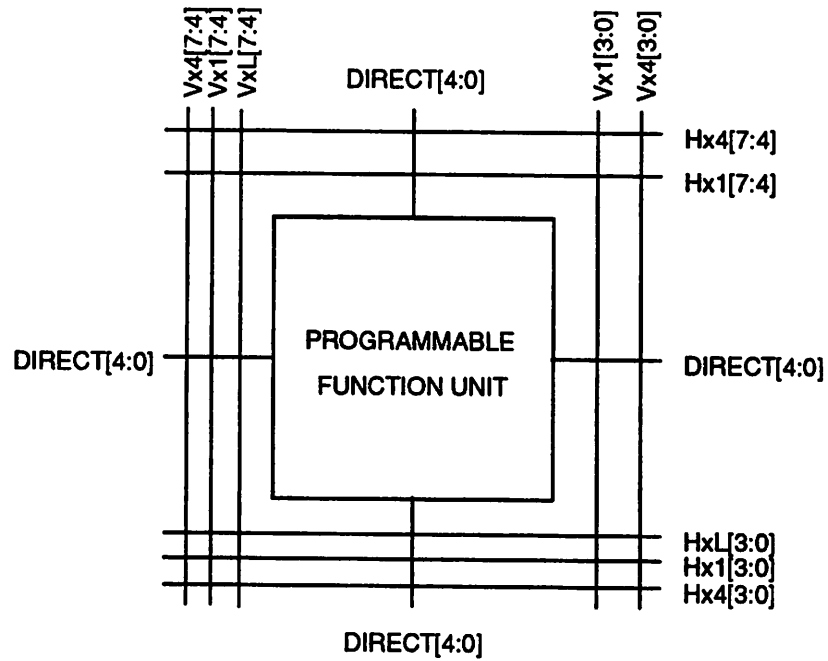


Figure 7.2: AT&T ORCA family routing resources. Figure shows the inter-PLC R-nodes.



limited success. There is no single quantity that can be effectively used as a routability measure, to determine a priori (i.e., before complete place and route) the routing characteristic of a synthesized solution, and having a variety of routing resources only increases this problem.

### 7.1.3 Summary

To sum up, routing completion is a problem in FPLDs. The architecture approach hopes to solve this problem by increasing the number of routing resources, but this could result in silicon wastage and increased parasitic capacitance. The CAD approach tries to modify the mapping solution to improve routability, but it has met with only limited success.

## 7.2 Timing issues

The second issue of concern with current LUT based FPGA devices is the propagation delay in the interconnections. This concern involves two factors.

1. Interconnection delay is the dominant delay.
2. Interconnection delay is difficult to estimate.

Let us study each in turn.

### 7.2.1 Dominant interconnect delay

We performed an experiment to study the interconnection (or routing) delay. We used the Xilinx XC3000 series chips as a test bed. Several MCNC combinational benchmark circuits were placed and routed on these chips. These circuits were mapped to Xilinx using the `mis_pga` mapping technique [Murgai 91a]<sup>2</sup>. The mapper generates an output circuit with 1-output LUTs, each having  $\leq 5$  inputs. We assign one such LUT to each CLB on the Xilinx chip. Each benchmark circuit is placed and routed on the smallest possible XC3000 series chip (i.e., the smallest chip which satisfied the condition,  $\# \text{ CLBs} \geq \# \text{ LUTs}$ ). However, if there were unrouted nets on such a chip, the next larger chip is used.

---

<sup>2</sup>It is possible that the results obtained could be different if a different mapping is used. However, the objective here is to just get a feel for the routing delay. We are not interested in obtaining the best possible answer.

Example	LUTs	Levels	CLB delay (ns), $C$	Routing delay (ns), $R$	$R/C$	Xilinx Part	CLBs available
5xp1	21	2	16	38.8	2.4	XC3020	64
C499	199	8	64	249.3	3.9	XC3064	224
C880	259	9	72	237.9	3.3	XC3090	320
alu2	121	6	48	106.4	2.2	XC3042	144
alu4	155	11	88	195.5	2.2	XC3064	224
apex7	96	4	32	103.6	3.2	XC3042	144
b9	49	3	24	50.2	2.1	XC3020	64
bw	28	1	8	34.6	4.3	XC3020	64
clip	54	4	32	55.5	1.7	XC3020	64
count	81	4	32	68.1	2.1	XC3030	100
duke2	164	6	48	114.2	2.4	XC3064	224
e64	213	5	40	130.5	3.3	XC3090	320
f51m	23	4	32	44.3	1.4	XC3020	64
misex2	37	3	24	36.3	1.5	XC3020	64
sao2	46	5	40	52.4	1.3	XC3020	64
vg2	100	8	64	54.9	0.9	XC3042	144
Average					2.4		

Table 7.1: Results from an experiment demonstrates that the routing delay is, on an average, 2.4 times the logic delay, for the XC3000 series.

The results of this experiment are tabulated in Table 7.1. The second column lists the number of 1-output LUTs in the examples, and the third column lists the number of levels of logic on the path with maximum delay. All Xilinx parts were of speed grade '70' (i.e., maximum speed of operation is 70MHz), and the CLB delay is 8ns for such parts. Hence, the CLB delay,  $C$ , in column 4, is obtained by multiplying the corresponding numbers in column 3 by 8. After place and route, using Xilinx's `apr` software, timing analysis was done using Xilinx's `XDelay` timing analyzer. Column 5, in Table 7.1 lists the worst-case routing delays,  $R$ , obtained from `XDelay`. We compute the ratio of routing to logic delays,  $R/C$ , and this is tabulated in column 6. Column 7 lists the part name of the Xilinx part, on which the example circuit was placed and routed and column 8 lists the number of available CLBs in that part.

From the  $R/C$  column, we see that the interconnect delay is significantly larger than the logic delay. Over the 16 examples listed in Table 7.1, the routing delay is, on the average, 2.4 times greater than the logic delay.

### 7.2.2 Widely varying interconnect delay

As shown in Section 5.1, the interconnect delay varies widely from circuit to circuit. For circuits with the same number of logic levels, the routing delay varies by a factor of about 60%. Our performance driven mapping algorithm, `dpmap`, tries to model the interconnection delay, using the equation  $d(N) \propto s(N)$ , where  $d(N)$  is the delay on net  $N$ , and  $s(N)$  is the number of sinks on net  $N$ . However, there is no single constant of proportionality that works well. For the XC3000 series, the value 1.7 was found to work well only when the chip was less than 60% populated. This wide variation is due to the nature of the programmable interconnect resources. If two points on the chip, close to each other, need to be connected, and if the interconnect resources in the intervening space have been used up, then the connection will have to be made in a round-about fashion, resulting in a very different net delay.

## 7.3 CAD issues

The 3 steps, on page 86 are the responsibility of the CAD tools accompanying the architecture. In case of failure to achieve satisfied routing the user may be required to change placement and routing manually which is both tedious and time consuming.

Steps 2 and 3 present two kinds of problems. Firstly, placement and routing tools perform badly for FPGAs, when timing specifications need to be considered. Although there have been new advances in timing-driven placement and routing techniques for standard cell and gate array design styles, these techniques have not yet been successful for FPGAs. This is partly because the synthesis procedure is unable to make an accurate estimate of net delays, and partly because the programmable routing structure makes it difficult for the placement tools to predict the router's behavior. Secondly, placement and routing tools are time consuming, and constitute the main bottleneck in the programmable logic system's design cycle time reduction.

In applications where the programmable device is used for prototyping, a different problem arises. Whenever small changes to the design are made (either specification modification or correcting errors), the steps have to be repeated. Not only does this process take time, but the new solution produced after place and route may be completely different from the previous solution. From performance standpoint, this new solution could very well invalidate the small change, and this means that additional design iterations will be required to obtain the desired result. This can be quite frustrating to the designer.

## 7.4 Desired features

Keeping the above three issues in mind, we list the features desired in any FPLD architecture.

1. **Guaranteed routability.** The programmable interconnect architecture should be such that a given net can easily be routed.
2. **Fast interconnections.** Guaranteed routability may imply that there should be an abundance of routing resources. However, for the FPLD to be useful in applications with performance constraints, the routing architecture should be such that the delay in the nets is as low as possible.
3. **CAD friendly.** With the overall PLS in mind, the architecture should be amenable from the CAD point of view. This means that the routing delays should be easy to estimate, and that the CAD tools should be able to perform their function in a fast manner.

In other words, the chip architect has to acknowledge that certain problems are too difficult to solve for the CAD tool, and hence design the programmable chip such that these problems do not occur. Research in the area of combining architecture and CAD is ongoing in the universities [Tseng 92, Brown 92]; but the efforts therein have been to modify existing FPGA architectures.

## 7.5 New architecture

We present an entirely different architecture. It has programmable routing elements and programmable logic elements like current FPGA chips. However, the architecture has been designed such that steps 2 and 3 on page 86 are reduced to almost trivial steps, and the CAD tool's responsibility now lies only in performing step 1. This means that the mapping solution in itself decides the performance of the circuit. The new architecture, named *Dharma* is presented in the next chapter.

## Chapter 8

# New Architecture

This chapter describes the new architecture, *Dharma*. The chapter is divided into three sections. In the first section, we explain the concept of levelize and hold, the central principle on which *Dharma* hinges. The next section explains the architecture itself. The last section describes the procedure involved in *Dharma*'s internal operation.

### 8.1 Principle of levelize and hold

In this section, we develop the central idea on which *Dharma* is based. Our idea is to time-share routing resources. We start by describing signal propagation in logic circuits, and then show how levelization and the addition of latches can make it possible to time-share routing.

#### 8.1.1 Signal propagation in logic circuits

In Figure 8.1 we show the general model of a sequential logic circuit. It consists of a combinational logic block and sequential elements(latches). The output of the latches are fed as input to the combinational logic block, and some outputs of the combinational logic block are fed back as inputs to the latches. In addition to the outputs from the latches, the combinational block also gets inputs directly; and some of its outputs need not feedback to the latches. The speed of this circuit is defined as the rate at which the latches can be clocked. For correct operation, if  $T_c$  is the time-period of the clock,  $D$  the delay through the combinational block and  $T_{sd}$  the latch set-up plus latch delay times,

$$T_c \geq D + T_{sd}.$$

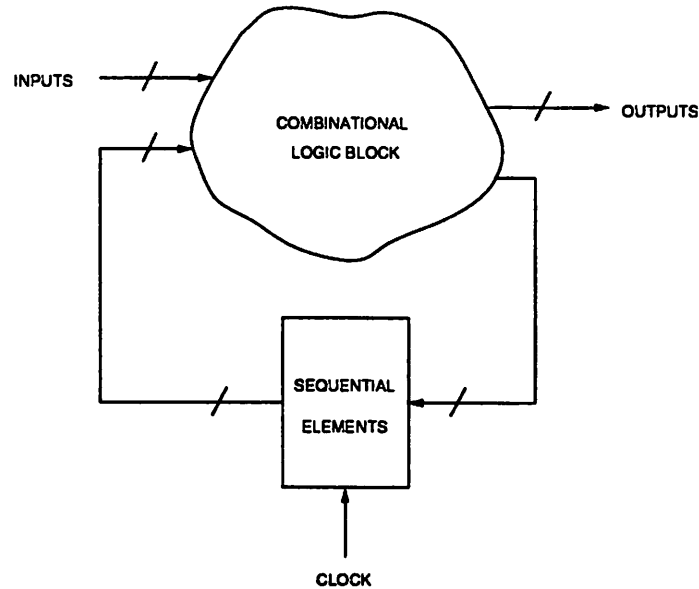


Figure 8.1: A general model of a sequential logic circuit.

---

What do we mean when we say that the combinational block has delay  $D$ ? Let the inputs to the combinational block assume their correct value at time instant  $t_i$ . Then, if the time instant at which *all* outputs have assumed their correct values (as determined by the function the combinational block is implementing) is  $t_o$ , then  $D = t_o - t_i$ . Some of the outputs may assume their correct values earlier than the other outputs, but in the definition of the delay  $D$ , we require *all* the outputs to have settled to their correct values.

How do signals flow in such a circuit? Assume that at some instant of time  $t_i$ , the latch outputs and other inputs to the combinational block are ready. After  $D$  units of time, the outputs of the combinational block are valid, and after another  $T_s$  (latch set-up time) units of time, the values can be latched. After  $T_d$  (latch propagation delay) units of time, the latch outputs are valid, and the signal flow cycle repeats.

Let us now observe how signals propagate through the combinational block. When the number of inputs and outputs are large, the combinational block is typically implemented as a multi-level circuit, as shown in Figure 8.2.

The nodes of the directed acyclic graph (DAG) in Figure 8.2 represent single output combinational gates, and the edges represent the interconnection between them.

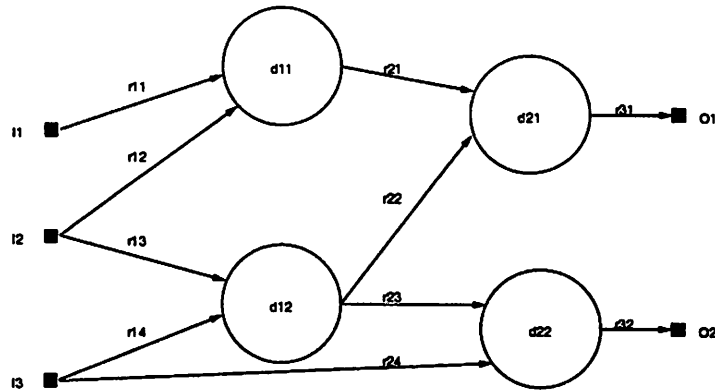


Figure 8.2: Multi-level implementation of combinational logic block.

The symbols within the nodes represent the delay through the gates and the interconnection delay is marked by symbols on the edges.

For ease of explanation, and without loss of generality, let us assume the gate delay to be the same for all gates, and let it be  $\delta$ . Similarly, let us assume the interconnect delay to be the same for every source-sink pair, and let it be  $\gamma$ . With these assumptions, we can immediately see that the delay of the entire combinational block,  $D$  is given by

$$D = L\delta + (L + 1)\gamma,$$

where  $L$  is the maximum number of levels in the DAG.

In Figure 8.3 we show how the signals propagate with time. In the first time interval  $\gamma$ , the signals propagate through the first level of interconnect, following which they propagate through the first level of logic in  $\delta$  time units. Hence, at the end of  $\gamma + \delta$  units of time, after inputs are valid, the signals have reached the outputs of the first level of logic. Similarly, the signals propagate through the other levels of logic and finally reach the outputs of the  $L$ th level after time  $L(\delta + \gamma)$ . One additional  $\gamma$  time interval is required to propagate through the interconnection to the output (or latch inputs).

Consider Figure 8.4, where we have introduced latches at all the inputs and at the outputs of all the nodes of Figure 8.2. For the present, let us assume that these latches are ideal, and have no set-up/hold times, nor any propagation delays. Let the inputs be latched at time instant  $t_i$ . After time interval  $\gamma + \delta$ , the outputs of the first level of logic are latched. At time instant  $t_i + 2(\gamma + \delta)$ , the outputs of the second level of logic are latched,



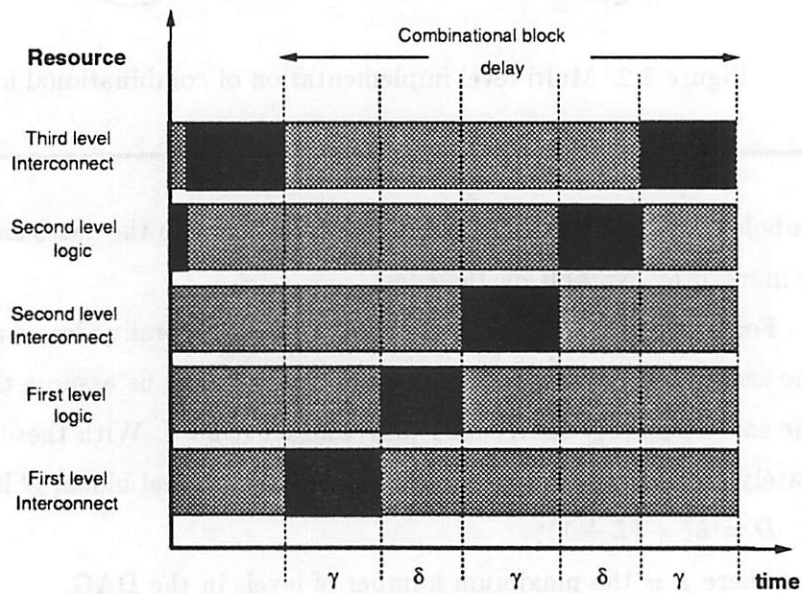


Figure 8.3: Signal propagation in Figure 8.2 and the identification of ‘idle’ resources. Lightly shaded regions correspond to time-intervals during which the logic or interconnect resource is not in use, when ideal latches are used to hold output logic levels, as illustrated in Figure 8.4.

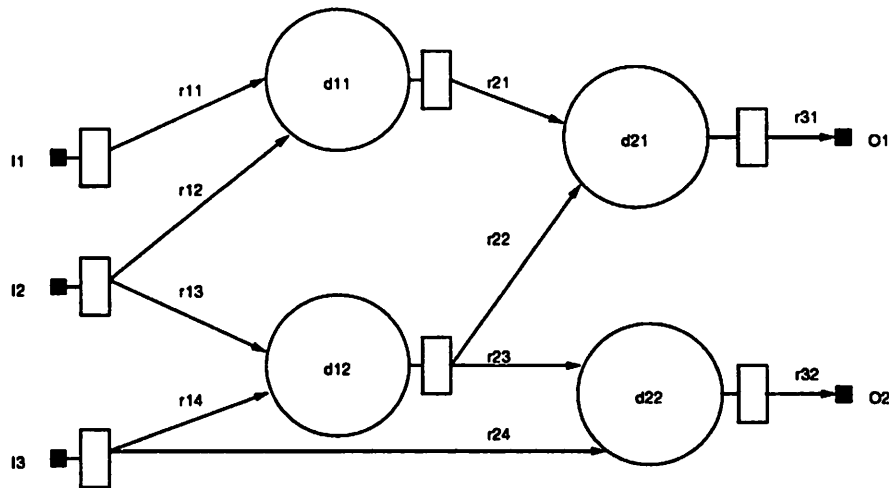


Figure 8.4: The DAG of Figure 8.2 is redrawn here with ideal latches at the outputs of every node and also at the inputs to the combinational logic block.

and so on. Since the latches make available the signal value to higher levels of logic, the interconnect and logic resources at level  $i$  perform useful work only during the time interval  $t_i + (i - 1)(\delta + \gamma)$  to  $t_i + i(\delta + \gamma)$  and are 'idle' (not producing anything new) for the rest of the time during a clock cycle  $t_i$  to  $(t_i + D)$ .

In Figure 8.3, we have used 2 different shades. Each resource, be it interconnect or logic, is shaded black when signals are propagating through it for the first time. At the end of the black shaded region, signals are available for use by the next resource (logic or interconnect, as the case may be). The light (grey) shade is used to show that the resource is, in some sense, 'idle', since it is not producing anything new. Our aim is to reuse the resource during its idle period.

### 8.1.2 Levelize-and-hold

From the above observation, we can develop a strategy for time-sharing resources. We first perform a topological levelization of the DAG representing the combinational block of the input circuit. Nodes at the same level cannot be time-shared since they evaluate their outputs in parallel. Nodes at different levels can be time-shared. i.e., nodes at level  $i + 1$  can use the same logic resource used by nodes at level  $i$ , since the nodes at level  $i + 1$  cannot

start their function evaluation until the nodes at level  $i$  have completed their evaluation. Similarly, the interconnect resource connecting level  $i$  to  $i + 1$  can be used to connect level  $i + 1$  signals to level  $i + 2$ .

Hence, *Dharma* needs to have one level of interconnect, one level of logic, and latches to hold the values of signals generated. We also need a means to change the interconnection and logic function, when the next level of logic and interconnect are to be implemented. In the next section we describe how this is achieved in *Dharma*.

### 8.1.3 Sequential elements

So far we only described how the combinational block of Figure 8.1 can be implemented by time-sharing logic and interconnection. Do the sequential elements of Figure 8.1 pose a problem? No, they do not, since they are required to latch the values of the outputs of the combinational block, and make them available as inputs when the next clock period begins. In our levelize-and-hold scheme, this function is already being performed by the latches of *hold*. Thus, using levelize-and-hold, the only difference between combinational and sequential circuits is that the last level is fed back to the first level in the latter, but not in the former.

## 8.2 *Dharma* Architecture

In this section, we describe the new architecture. We first give an overview using a high-level block diagram and then look at the detailed block diagram. The sub-blocks are then explained in turn.

### 8.2.1 High-level block diagram

The central idea of *Dharma* is the time-sharing of silicon area, especially the interconnection resources. This is made possible because of the topological levelization of the input circuit being implemented on *Dharma*. *Dharma* consists of four main blocks (see Figure 8.5):

1. A one-dimensional array of dynamic logic modules (DLM)
2. A one-dimensional array of pass-buffers

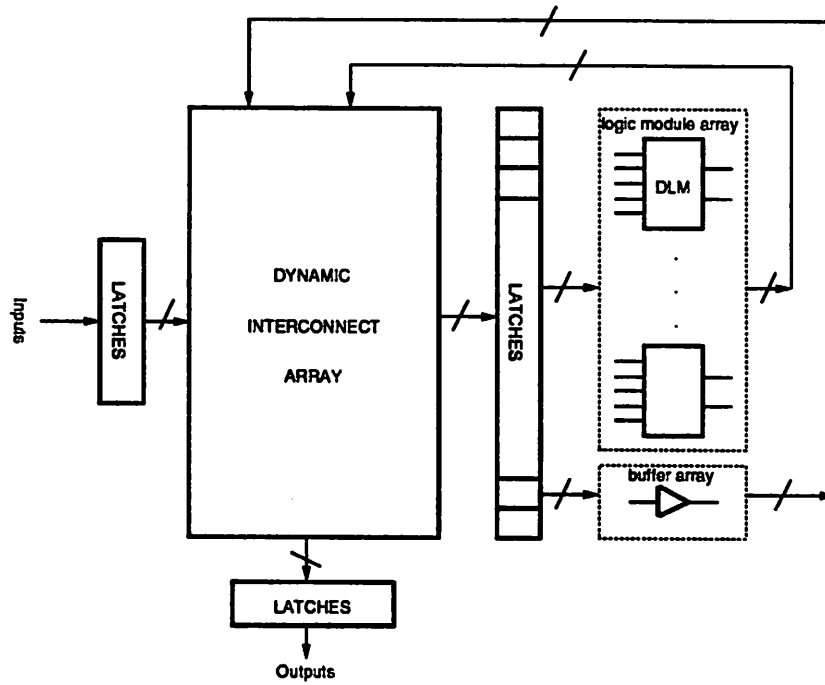


Figure 8.5: High level block diagram of *Dharma*

---

### 3. Dynamic Interconnection Array (DIA)

### 4. Latches

A circuit to be implemented on *Dharma* is first levelized. Let  $l$  be the number of levels. Then the implementation of this circuit on *Dharma* is a cyclic repetition of  $l$  internal cycles. In internal cycle  $i$ , the logic modules are configured to implement the functions in level  $i$  of the circuit, and the interconnection block is configured to implement the connections between level  $i$  and  $i + 1$ . We use the term *dynamic*, to describe the logic modules and the interconnection, because the function implemented by the logic modules and the interconnection structure is dynamically re-configured at every level. Note that programming of a circuit onto *Dharma* consists of a one-time compilation of the values to be loaded into the DLMs and DIAs. These values are then loaded onto the chip at power-on or 'boot time'.

A DLM generates an output which is a combinational function of its inputs. To allow the possibility of realizing any function, the DLM is an LUT (see Section 10.1 for other types of DLMs). All the DLMs can perform a look-up (based on the input configuration) in parallel. Hence, each level of logic of the input circuit can be evaluated in parallel by the DLM array (provided, of course, the number of functions being evaluated is less than the number of DLMs). The DIA connects the outputs of the DLMs and pass-buffers to the inputs of the next level of logic, through latches. To allow 100% routability, with predictable performance, the DIA block is designed such that all required connections are possible. The latches are used to store the signals between levels, and they also function as storage for state variables, in sequential circuit implementations. The pass-buffers are required to pass signals through levels; eg., if signal  $X$  is generated in level 2 and required in level 4,  $X$  will pass through level 3.

Based on the above four blocks, *Dharma* can be parameterized by the following variables (assigning different values to these variables will generate a chip-series).

1. Number of DLMs,  $C$ .
2. Number of inputs per logic DLM,  $K$ .
3. Number of pass-buffers,  $B$ .
4. Number of levels,  $L$ .

The number of internal latches would then be  $KC + B$ .

### 8.2.2 Detailed block diagram

Figure 8.6 is a detailed block diagram of the *Dharma* architecture.

In addition to the four main blocks illustrated in Figure 8.5, Figure 8.6 has the following:

1. *Level Generation Circuitry*, to count the level being implemented at the current clock instant.
2. *Internal Clocking Circuitry*, to generate the internal clock in synchronization with the external clock.
3. *Memory Write Circuitry*, to program the chip for a particular circuit.

The DIA block is split into two portions, an *L*-crossbar and a *B*-crossbar. The *L*-crossbar connects the outputs of the DLMs and the pass-buffers to the latches at the DLM inputs. The *B*-crossbar connects the pass-buffer outputs (after they are multiplexed with the primary inputs) to the latches at the pass-buffer inputs.

A DLM output which has to pass through to the next level is directly connected to a unique pass-buffer. This is a hard-wired connection. However, this connection is made to pass through a 2-to-1 multiplexer. The other input to the 2-to-1 multiplexer is from the *B*-crossbar. These 2-to-1 multiplexers are called *Buffer Muxes* and are shown in the lower right hand portion of Figure 8.6, to the left of the pass-buffer input latches. This interconnection scheme allows the pass-buffer to be used to pass other signals, when its corresponding DLM's output is not required at higher levels. In addition to the hard-wired connections, there are pass-buffers which are not associated with any DLM, and these are used to pass signals from pass-buffer outputs (the hard-wired connection is used to pass a signal the first time it is generated, i.e., by the DLM).

The chip's inputs (primary inputs) are latched into the *Primary Input Registers*. These latched signals enter the interconnection matrix through 2-to-1 multiplexers, called *PI Muxes*, shown atop the *L*-crossbar. The primary inputs are multiplexed with DLM outputs and pass-buffer outputs before entering the *L*-crossbar.

The outputs are latched into the *Primary Output Registers* (shown on the right side of Figure 8.6). The outputs can be from the DLMs directly, or can come through the

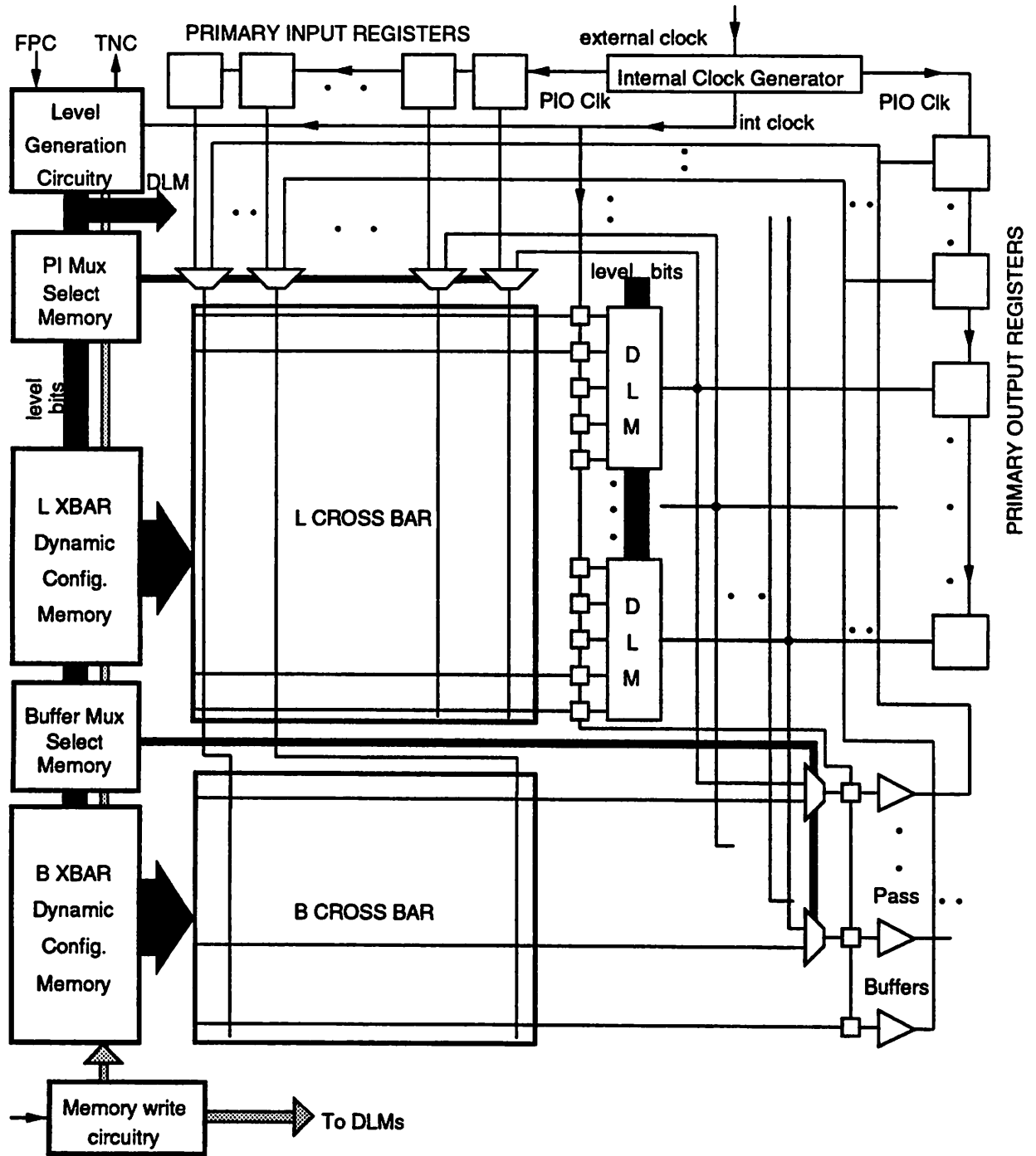


Figure 8.6: Detailed block diagram of *Dharma*

- [Woo 91] N. Woo, "A heuristic method for FPGA technology mapping based on edge visibility," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 248-251, 1991.
- [Xilinx 89] *Xilinx: The programmable gate array data book*, Xilinx Inc., 1989.
- [Yang 72] Y. Y. Yang and O. Wing, "Suboptimal algorithm for a wire routing problem," *IEEE Transactions on Circuits Theory*, pp. 508-511, Sept. 1972.
- [Yeh91] C. W. Yeh and C. K. Cheng, "A general purpose multiple way partitioning algorithm," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 421-426, 1991.



- [Sechen 88] C. Sechen, "Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing," *Proc. 25th Design Automation Conference*, pp. 73-80, 1988.
- [Sentovich 92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," *Proc. ICCD*, pp. 328-33, Oct. 1992.
- [Singh 92] S. Singh, J. Rose, P. Chow and D. Lewis, "The effect of logic block architecture on FPGA performance," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 3, pp. 281-287, Mar. 1992.
- [Small 91] C. S. Small, "High-density PLD architectures: family tree sorts out high-density PLDs," *EDN*, pp. 75-80, Sept. 1991.
- [Touati 90] H. J. Touati, *Performance-oriented technology mapping*, Ph.D. Thesis, U. C. Berkeley, UCB/ERL Memo No. M90/109, Nov. 1990.
- [Trimberger 92] S. Trimberger and M. Chene, "Placement-based partitioning for lookup-table-based FPGAs," *Proc. ICCD '92*, pp. 91-94, Oct. 1992.
- [Trimberger 92a] S. Trimberger, "A small complete mapping library for lookup-table-based FPGAs," *Proc. 2nd International Workshop on Field-Programmable Logic and Applications*, Aug. 1992.
- [Tseng 92] B. Tseng, J. Rose and S. Brown, "Improving FPGA routing architectures using architecture and CAD interactions," *Proc. ICCD '92*, pp. 99-104, Oct. 1992.
- [Vecchi 83] M. P. Vecchi and S. Kirkpatrick, "Global wiring by simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, no. 4, pp. 215-222, Oct. 1983.
- [Waugh 91] T. C. Waugh and A. Kopser, "Field programmable gate array key to reconfigurable array outperforming supercomputers," *Proc. CICC*, pp. 6.6.1-6.6.4, May 1991.

pass-buffers.

In Figure 8.6, the DLMs are 5-input, 1-output LUTs. Other kinds of DLMs are possible, and are explained in Chapter 10.

The *Level Generation Circuitry* keeps count of the level, and is controlled by the internal clock, the *FPC* signal, and the *Mode* bit (explained in Section 8.2.9). The current level is broadcast to the rest of the chip in the form of level bits.

The *L XBAR Dynamic Configuration Memory* block stores the configuration bits of the *L*-crossbar, and the *B XBAR Dynamic Configuration Memory* block stores the corresponding bits for the *B*-crossbar. These configuration bits are changed every level, and are addressed by the level bits.

The *PI Mux Select Memory* stores the select control for the *PI Muxes*. Each multiplexer requires 1 select bit, and this has to be changed at every level. Hence the *PI Mux Select Memory* is also addressed by the level bits. The *Buffer Mux Select Memory* block, likewise, stores the select control bits for the Buffer Muxes. These bits are also level dependent, and hence this block is also addressed by the level bits.

At power up, the *Memory write circuitry* loads in the values of the four memory blocks (*PI Mux Select*, *L XBAR Dynamic Configuration*, *Buffer Mux Select* and *B XBAR Dynamic Configuration*), the DLMs, and the level register in the *Level Generation Circuitry* (see Figure 8.11). These values are loaded in a serial manner from an external non-volatile memory (eg., an EPROM). The number of bits to be loaded into a chip is dependent on the on-chip resources (number of DLMs, number of pass-buffers, etc.). However, this number is invariant of the circuit being implemented on the chip, and hence can be preset at manufacture time. The memory write circuitry loads in the preset number of bits from external memory, and then stops. At completion, it signals the level generation circuitry to start the level count.

### 8.2.3 Clocking scheme

The *Internal Clock Generator* generates the internal clocks, labeled as *Int clock* and *PIO clock*. These clocks are synchronized with the external clock signal. The latches at the DLM inputs and the pass-buffer inputs are clocked by *Int clock*. *PIO clock* latches the chip inputs and outputs. There are two possible schemes of clocking.

1. EFP clocking scheme

## 2. S clocking schme

These clocking schemes are illustrated in Figure 8.7. The *Extended First Period* (EFP) scheme requires a non-uniform *Int clk*. The *Simple* or *Slow* (S) clocking scheme is simple, but results in a slower circuit. In the EFP scheme, the first period of the internal clock, following the rising edge of *Ext clk*, is longer compared to the rest of the periods. This first period is slowed down to accommodate the time required for the inputs to get latched into the input registers (see Section 12.2). Also, *Int clk* is  $180^\circ$  out of phase with *Ext clk*. *PIO clk* follows *Ext clk* exactly. However, the *Internal Clock Generator* must generate this explicitly (and not simply tie it to *Ext clk*) since there are applications (eg., a pure combinational circuit) which do not provide *Ext clk*. The S clocking scheme requires one additional *Int clk* cycle compared to the EFP scheme, for any circuit. Hence the “Slow” prefix. But, this scheme does not require the complex circuitry to generate extended first periods, and hence is very simple and straightforward. Here, *Int clk* is in phase with *Ext clk*.

### 8.2.4 Multi-chip operation

The *FPC* and *TNC* signals facilitate multi-chip operation. Such an operation may be desired when a circuit’s levels exceed the maximum levels on a single chip,  $L$ . Let  $l$  be the number of levels in the circuit, and let  $l > L$ . If the number of DLMS in a level  $d$ ,  $\#(d)$ , is such that  $\#(d) < C, \forall d$ , where  $C$  is the number of DLMS on a chip, then, although the circuit cannot be implemented on a single *Dharma* chip, its implementation can be spread across multiple *Dharma* chips. The number of such chips required would be  $\lceil l/L \rceil$ . These chips are connected in a chain-like fashion, with the first chip implementing levels  $(0 \dots (L - 1))$ , the second chip implementing levels  $(L \dots (2L - 1))$ , and so on. The outputs from the first chip will be connected as inputs to the second chip, the outputs from the second chip will be connected as inputs to the third chip, and so on, and finally the outputs from the last chip will be connected to the inputs of the first chip. Figure 8.8 shows this multi-chip interconnection scheme.

*FPC* stands for *From Previous Chip* and *TNC* stands for *To Next Chip* and it is obvious from Figure 8.8 why they are named thus.

The *Dharma* chips in Figure 8.8 have  $L = 4$  each, and the circuit being implemented has  $l = 10$ . Hence  $\lceil 10/4 \rceil = 3$  chips will be required, assuming that  $\#(d) < C$ ,

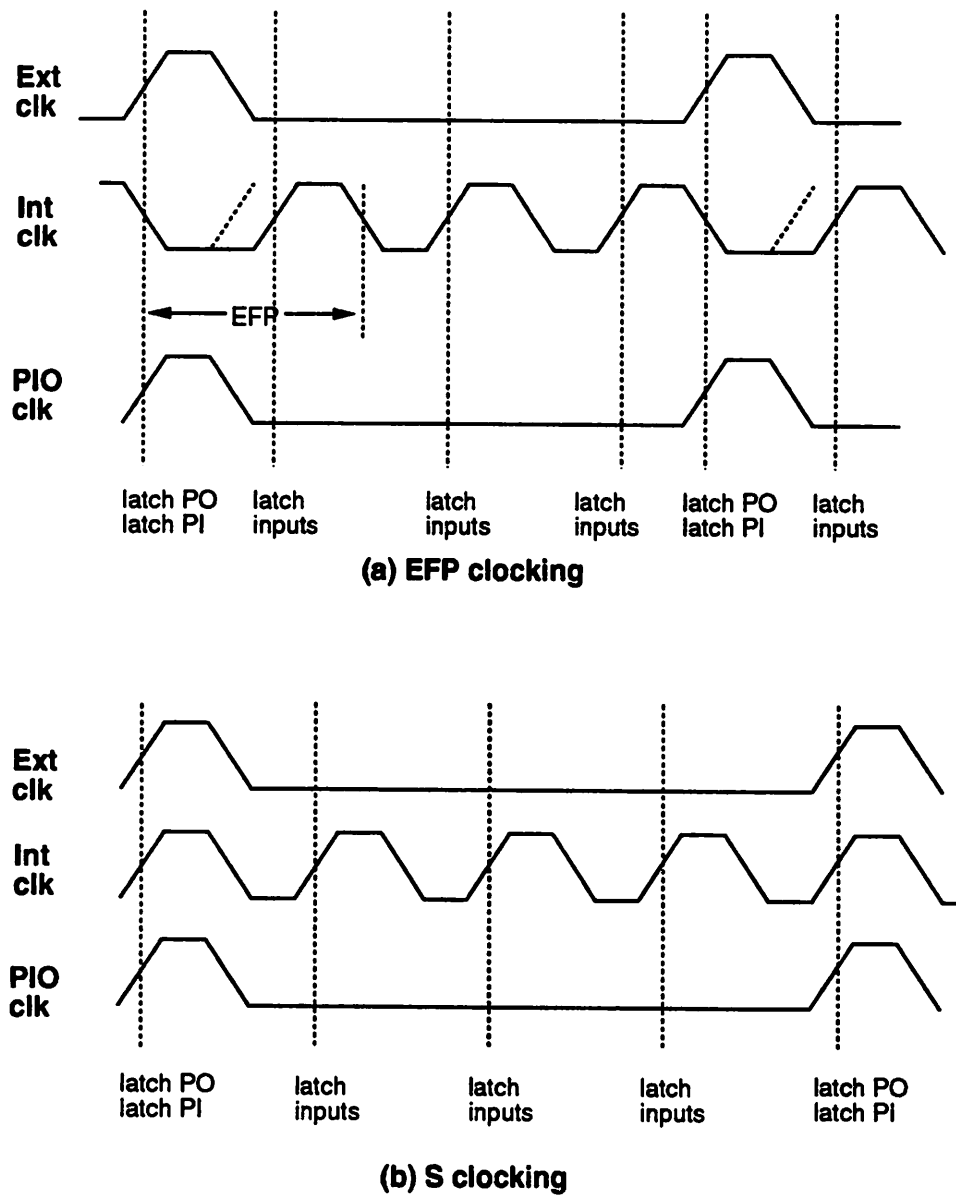


Figure 8.7: Clocking schemes

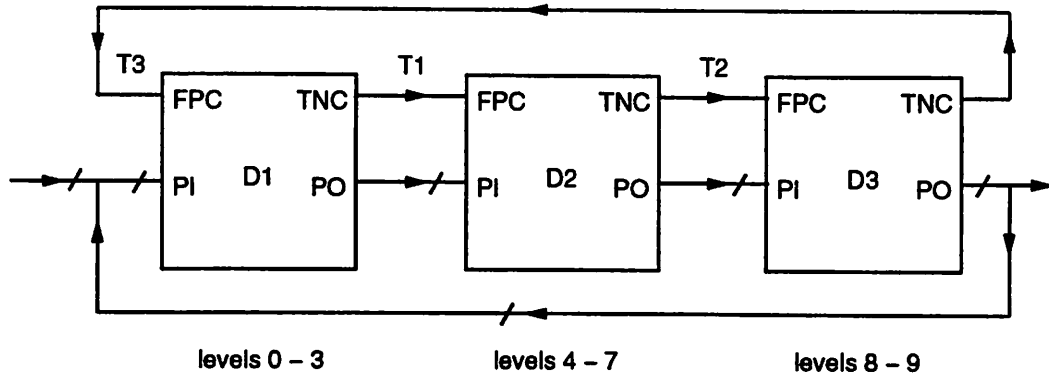


Figure 8.8: Multi-*Dharma* interconnection scheme. In this example,  $l = 10$ , and  $L = 4$ .

*Vd.* Chip *D1* implements levels 0 to 3, chip *D2* implements levels 4 to 7 and chip *D3* implements levels 8 and 9. The waveform sketches for the *TNC* and *FPC* signals are shown in Figure 8.12 and further explanation can be found in the subsection on *Level Generation Circuitry*.

A *Mode* bit is provided on each chip to select between single chip and multi-chip operation. See the *Level Generation Circuitry* subsection (Section 8.2.9) for more details.

### 8.2.5 Dynamic logic modules

The logic module has to be general enough that it can be programmed to perform any combinational function of its inputs. Among the known logic modules are PLAs, look-up tables and MUX based modules. Among these, only the look-up table based modules can perform all the combinational functions of its inputs. Xilinx series of chips have used look-up table based logic modules, in various configurations, and also with more than one output per module. From the logic synthesis point of view, logic modules with just one look-up table with a single output, have been easier to handle, and mapping for delay, has been shown to be optimal, in this case [Cong 92]. In *Dharma*, we have chosen the logic module to be a  $K$ -input, 1-output LUT. However, it should be noted that the architecture is general enough to allow other types of logic modules (see Chapter 10). The advantages of using more complex configurations would depend on how well the synthesis tools handle such logic modules.

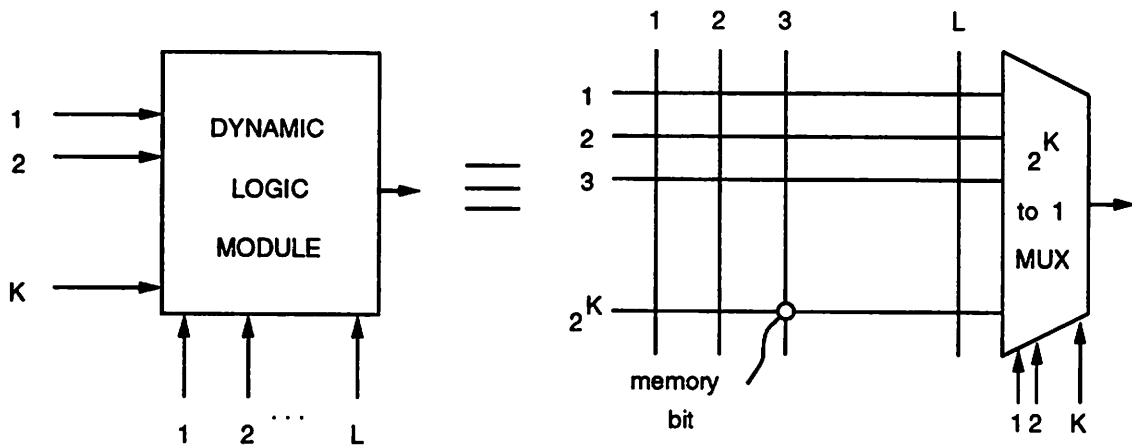


Figure 8.9: Dynamic logic module architecture

Each logic module may have to perform a different function at each level. A  $K$ -input, 1-output LUT requires  $2^K$  bits of memory to implement any function of  $K$ -inputs. If a single chip can be used to implement circuits of at most  $L$  levels, then each logic module will require  $L \times 2^K$  bits. A detailed diagram of the DLM is shown in Figure 8.9. There are many ways in which the level select lines and the  $K$  address lines can be combined together, and the figure shows one way. Which is the best one to use will depend on the area usage and memory read time, which would depend on the technology being used. Combining the level-selects and the  $K$  inputs together, is in essence making each DLM a  $(K + \log_2 L)$ -input, 1-output LUT, and this LUT should be designed such that area usage is kept down while giving timing performance as close to the  $K$ -input LUT as possible.

### 8.2.6 Pass-buffers

The pass-buffers are used to pass signals through levels, as already described in the previous section. The pass-buffer is only a pedagogical aid to deal with such signals. In the actual implementation, the pass-buffer can be replaced by a wire <sup>1</sup>.

<sup>1</sup>The pass-buffer should not be confused with the line drivers required to drive signals at the inputs of the crossbars. Such line drivers will be needed in the actual implementation of the *Dharma* chip, and are not shown in Figure 8.6 or 8.5.

### 8.2.7 Dynamic interconnection array

Figure 8.10 is a cut-out of the interconnection related circuitry, from Figure 8.6. The breakup into two crossbars is done so as to reduce the number of crosspoints.

Let us analyze how this reduction occurs. To keep the analysis general enough, let us assume that each DLM has  $m$  outputs, and  $K$  inputs. If there are  $C$  DLMs and  $B$  buffers, then  $(mC + B)$  output signals need to be connected to  $KC + B$  input latches. If a single crossbar were used to perform this routing, there would be  $(KC+B)(mC+B)$  crosspoint, assuming a full crossbar. Instead, by using two crossbars (the  $L$ -crossbar and  $B$ -crossbar), as shown, the number of crosspoints is reduced to  $(mC + B)KC + B^2$  (again, assuming full crossbars). Hence this reduces the number of crosspoints by  $mCB$ . If we assume that the number of buffers  $B$  is related to the number of DLMs  $C$ , by the relation  $B = \beta C$ , then the reduction in crosspoints,  $\mathcal{R}_{cp}$  is

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{mCB}{(KC + B)(mC + B)} \\ &= \frac{mC(\beta C)}{(KC + \beta C)(mC + \beta C)} \\ &= \frac{m\beta}{(K + \beta)(m + \beta)}\end{aligned}$$

For example, if  $K = 5$ ,  $m = 1$  and  $\beta = 2$  (i.e., the number of buffers is twice the number of DLMs), then,

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{2}{(5 + 2)(1 + 2)} \\ &= \frac{2}{21} \\ &= 9.5\%.\end{aligned}$$

If, instead  $m = 2$ , then,

$$\begin{aligned}\mathcal{R}_{cp} &= \frac{2 \times 2}{(5 + 2)(2 + 2)} \\ &= \frac{4}{28} \\ &= 14.3\%.\end{aligned}$$

The circuit inputs enter the interconnection matrix, multiplexed with the DLM and pass-buffer outputs. Input signals that are required at higher level in the circuit can be multiplexed with pass-buffer outputs (those pass-buffers that do not have a Buffer Mux at

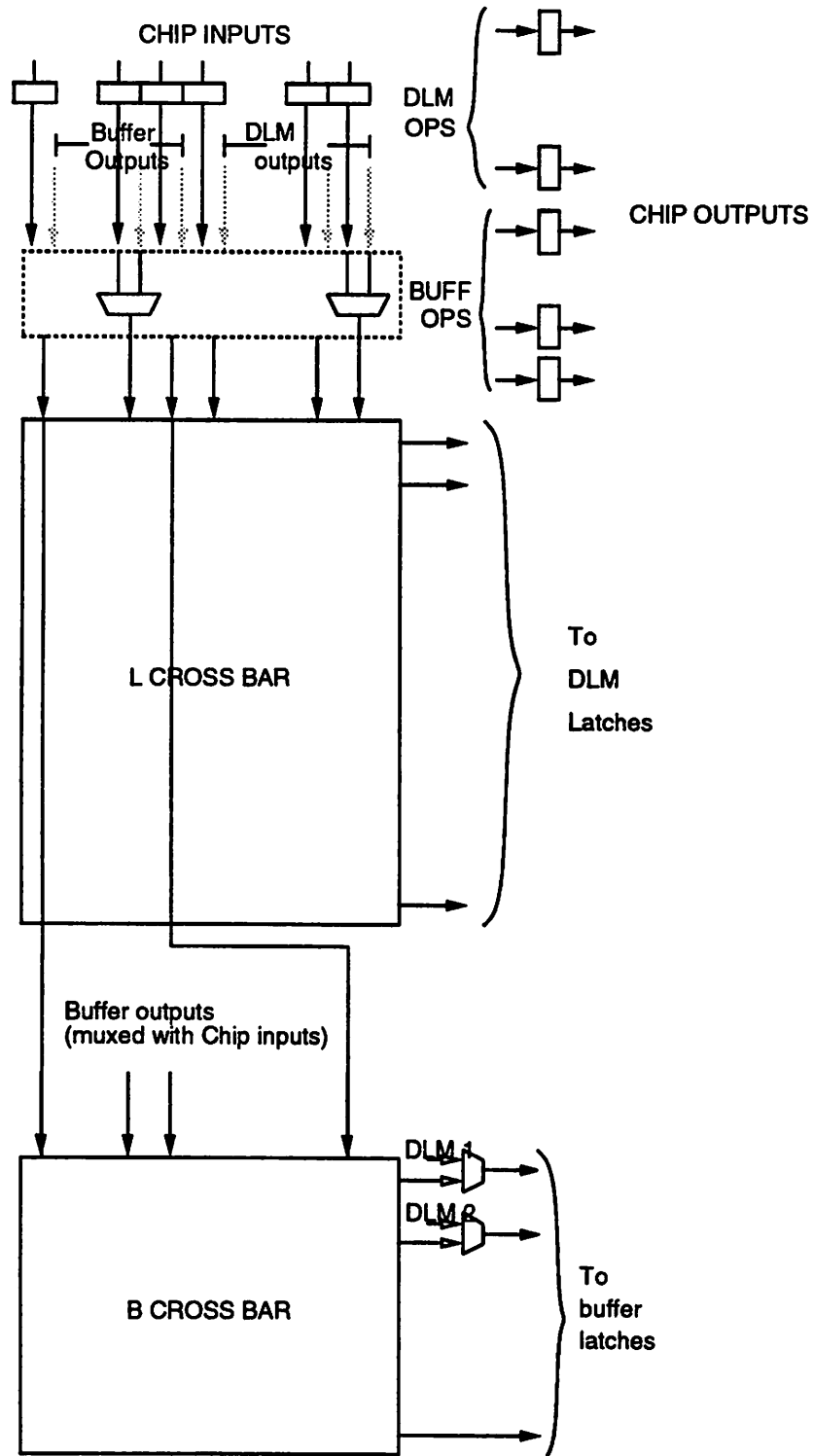


Figure 8.10: Dynamic interconnection block architecture



their input latch). In such cases, the input can be used directly (by setting the multiplexer select line to choose the input instead of the pass-buffer output) when that particular level is being evaluated. If the number of input signals required at higher levels exceeds the number of such pass-buffers, then some of the input signals will have to go through pass-buffers at lower levels.

There are many different ways in which to implement the crossbars, so as to reduce the number of switches and area occupied. Several such techniques are listed in Chapter 10. Whatever be the technique used, the idea is the same, to be able to connect the outputs of level  $i$  to the inputs of level  $i + 1$ , so that all nets are routed.

### 8.2.8 Latches

The number of internal latches on the chip equals the number of inputs to the DLM and pass-buffers, i.e.,  $KC + B$ . These latches serve 2 purposes:

1. To store the intermediate signals across levels
2. To store state variables of sequential circuits

The latches are clocked by the internal clock at a frequency of  $f_I$ . The latch value is unchanged during the level evaluation.

### 8.2.9 Level generation circuitry

An exploded view of the *Level Generation Circuitry* block of Figure 8.6 (upper left corner) is shown in Figure 8.11 <sup>2</sup>. It has the following components.

1. Number of circuit-levels register.
2. Level counter.
3. *Mode* bit, to choose between single chip and multi-chip operation.
4. Decision and control circuitry to control operation of the level counter and to generate the *TNC* signal.

---

<sup>2</sup>This diagram is slightly different from the one shown in [Bhat 93]. However, both perform the same function.

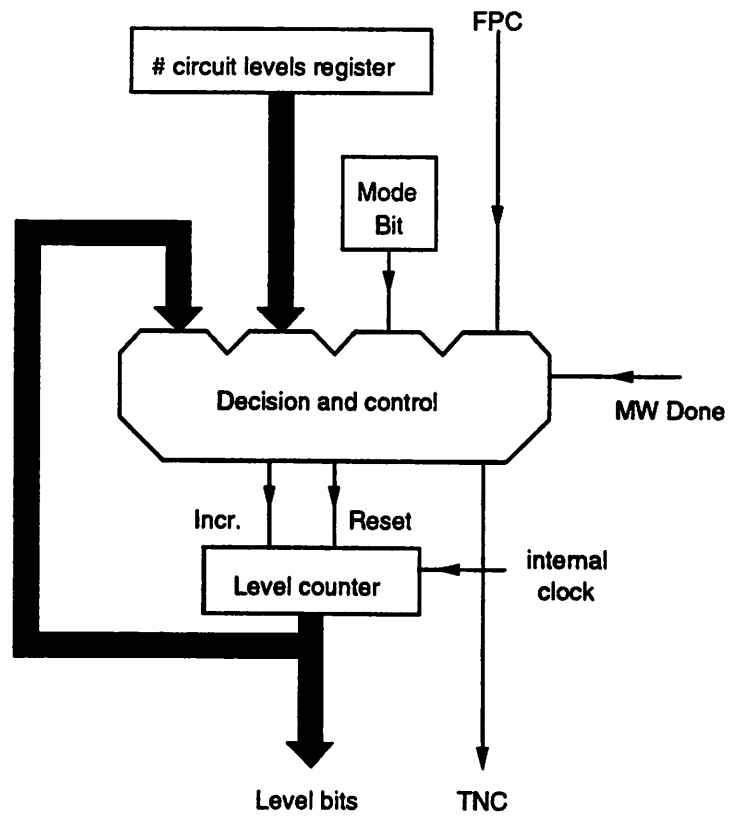


Figure 8.11: Block diagram of level generation circuitry.

---

The circuit-levels register stores the value of the number of levels the circuit has. This value must not exceed the maximum number of levels,  $L$ . The level counter's value is compared with this stored value and the counter is reset to zero when the two are equal. For example, if the circuit has 4 levels, then the level counter counts 0, 1, 2, 3, 0, 1, 2, 3, etc. This comparison operation is performed within the decision and control block. In Figure 8.8, chips  $D1$  and  $D2$  have the value 4 loaded in to their circuit-levels register, and chip  $D3$  has a value 2 in its circuit-levels register.

The level counter counts at every internal clock tick, when the *incr* signal is asserted. In single-chip operation, the *incr* signal is always asserted. In multi-chip operation, *incr* goes low when the count reaches a number which is one less than the value stored in the circuit-levels register.

The *Mode* bit selects between single chip and multi-chip operation. Although shown as a separate bit in Figure 8.11, it can be an external pin on the chip, which is pulled high or grounded, depending on how the chip must function.

Figure 8.12 shows the timing relationship amongst the *FPC* and *TNC* signals, for the example situation of Figure 8.8. The level counters in  $D1$  and  $D2$  count from 0 to 3, whereas the level counter in  $D3$  counts 0 and 1. The values of the level counters are shown symbolically by means of multi-level waveforms in Figure 8.12.  $D1$ 's level counter starts counting after its *FPC T3* goes high.  $D1$ 's *TNC* (same as  $D2$ 's *FPC*),  $T1$  goes high after its level counter reaches 3. This starts  $D2$ 's level count at the next clock.  $D1$ 's level count then stops and waits for its *FPC* to go high.  $T1$  is lowered after one clock cycle. Similarly,  $D2$ 's *TNC* (same as  $D3$ 's *FPC*),  $T2$  goes high after  $D2$ 's level counter reaches 3, and  $D3$ 's level counter starts counting at the next clock. Finally, when  $D3$ 's count reaches 1, its *TNC* (i.e.,  $T3$ ) goes high and the cycle repeats.

The decision circuitry senses the *FPC* signal when the *Mode* bit is asserted (multi-chip operation). If the signal is high, then the level counter is reset, and *incr* is asserted. The *FPC* signal is then ignored. When the count reaches a number which is one less than the value stored in the circuit-levels register, the decision circuitry asserts the *TNC* signal, and de-asserts *incr*. *TNC* is de-asserted in the next clock cycle. The decision circuitry again senses the *FPC* signal. *incr* remains de-asserted till *FPC* is asserted, after which the cycle repeats.

The *MW Done* (memory write done) signal is from the *Memory Write* block. On power up, the level counter is reset. The decision circuitry then waits for *MW Done*. When

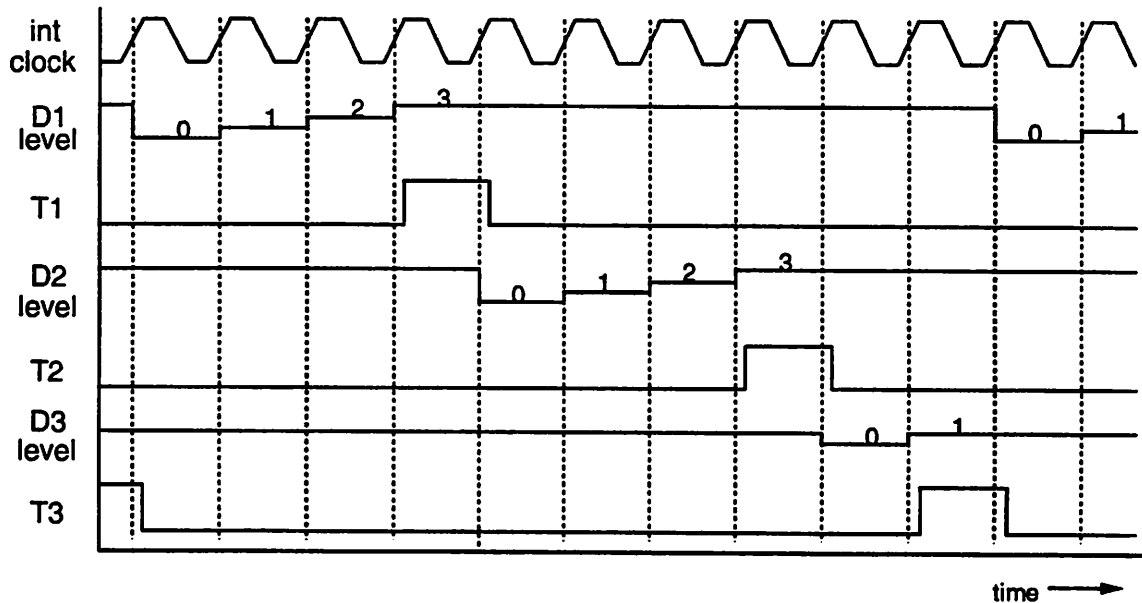


Figure 8.12: Timing waveforms for multi-chip operation.

this signal is asserted, *incr* is asserted, and the level counter starts counting from 0.

### 8.3 Operating procedure

To use *Dharma* to implement a given circuit design, the user must first convert the design specification into *Dharma*'s logic blocks, using a design flow similar to Figure 1.4, using software tools (see Chapter 11). The *Device Programming* software generates the bits that go into the various memory blocks and DLMs (see Figure 8.6). These bits are then stored in a non-volatile memory, such as an EPROM.

On power-up, the memory write circuitry of Figure 8.6 addresses this external memory (external to the *Dharma* chip, that is), and reads in the stored values therein. The bits are then loaded into the various memory blocks. This loading procedure will have to be repeated every time the power is switched on, since the memory bits on the *Dharma* are volatile<sup>3</sup>. After all the bits have been loaded (a fixed number for a given chip), the memory

<sup>3</sup>For this discussion, we have assumed that *Dharma* uses an SRAM technology. Depending on the DIA and DLM architecture, it is possible to have a non-volatile memory on the *Dharma* chip, in which case the loading procedure would be replaced by a one-time programming procedure to program the non-volatile

write circuitry's task is complete, and it remains idle for the rest of the time.

After the memory configuration bits are loaded, *Dharma* implements the circuit in a level by level fashion. Figure 8.13 illustrates the mechanism for a circuit with 3 levels. The figure shows the activities starting from power-up. In the figure, we assume an EFP clocking scheme (see Figure 8.7). After the memory bits are loaded, the *MW Done* signal is asserted. This signals the *Level Generation Circuitry* to start the level counter. The DIA is first configured to connect the Primary Inputs to the first level of logic (i.e., signals from Level 0 to Level 1). In the figure, this is shown on the row labeled *DIA Recnfg* as 'L0-L1', and we assume that this configuration takes half the internal clock cycle. At the negative edge of *Int clock*, the DIA is ready. The Primary inputs are latched at this instant. In the figure, this time instant is labeled *Latch PI*<sup>4</sup>.

During the negative half cycle, the primary input signals propagate through the DIA. At the next positive going edge, the signals are latched at the DLM and pass-buffer inputs (marked *Latch inputs* in the figure). As the DLMs evaluate the functions, the DIA is reconfigured to connect the next level, L1-L2. The signal propagation through the DLM and the reconfiguration of the DIA occur in parallel. The DLM is assumed to be a LUT, hence it does not have a reconfiguration time. However, if it were a different type of DLM which needed reconfiguration, such a reconfiguration could be done in parallel with the DLM propagation period. The operation continues in similar fashion, till power is turned off.

After the DLM has evaluated the uppermost level of logic (Level 3), the output signals can be latched into the Primary output registers, and the new primary input signals can be latched into the Primary input registers. The dynamic operation is transparent to the external world. Note that the rate at which the PI and PO registers can be latched (i.e., the rate of the external clock), is  $\frac{1}{(I+\delta)}$  times slower than the rate of *Int clock*, as explained in Section 12.2, equation 12.8.

The time instants marked  $\epsilon$  are referred in Chapter 9.

The DIA and DLM propagation times need not be equal. The duty cycle of the internal clock is set to the ratio  $\frac{\max(t_D, t_R)}{t_I}$ , where  $t_D$  is the sum of the DLM and latch propagation times,  $t_R$  is the sum of DIA reconfiguration time and latch hold time, and  $t_I$  is the sum of DIA propagation time and latch set-up time (see Section 12.2). A duty cycle

---

memory bits.

<sup>4</sup>In the label *Latch PI* and *Latch inputs*, the word *Latch* is used as a verb, and not as a noun.

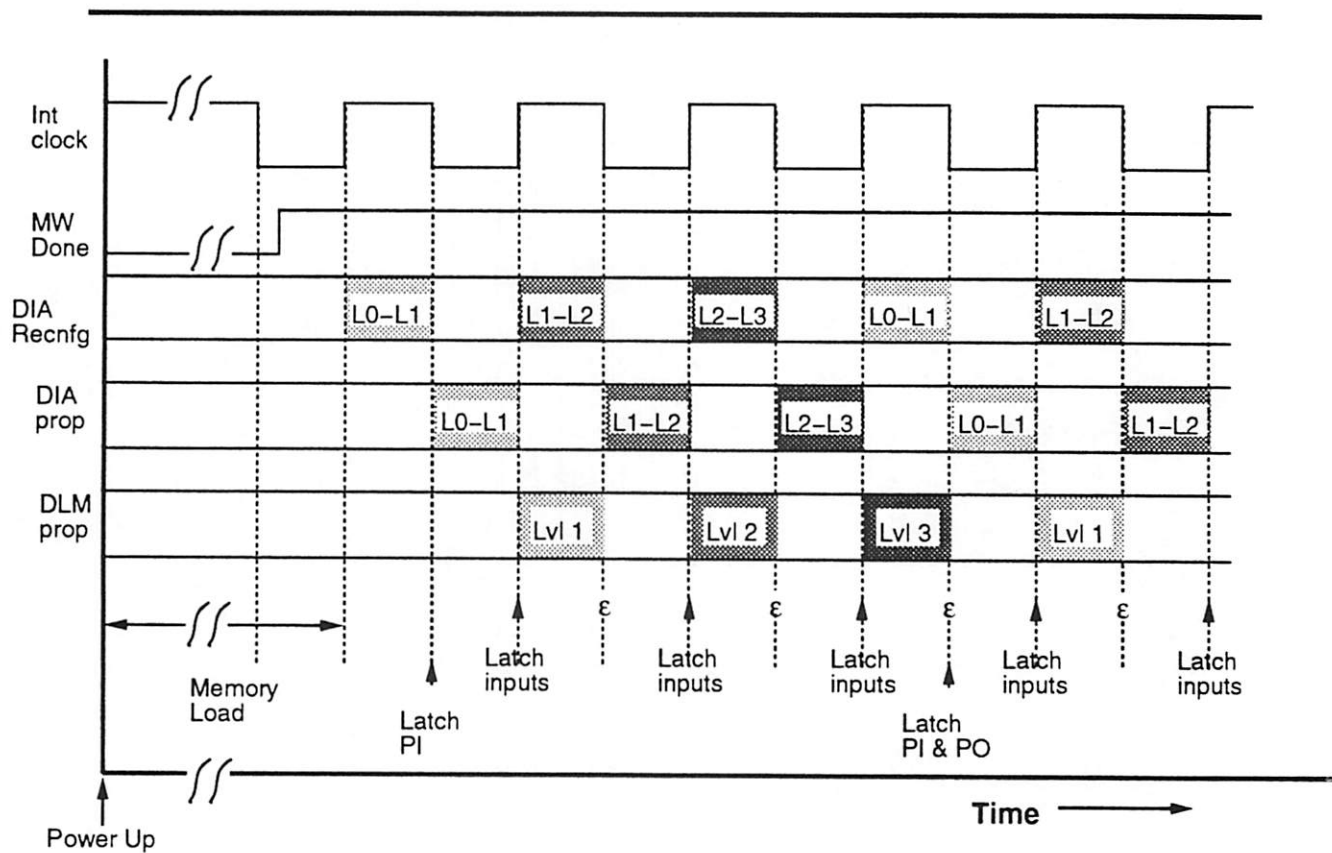


Figure 8.13: Timing diagram to illustrate *Dharma's* operation. The circuit design has 3 levels.

other than 50% does not change the activities described above in any way.

From Figure 8.13, we see that there is no necessity for any global control logic, to supervise the chip operations. All operations are synchronized with respect to the internal clock and the level bits.

## Chapter 9

# Examples to Illustrate Dharma

In this chapter, we illustrate *Dharma* by means of simple examples. We have chosen two combinational and one sequential example, with a small number of logic elements, and we demonstrate by means of figures, how the circuits are realized on a *Dharma* device. For the purposes of the illustration, we have chosen a simplified version of *Dharma*. This simplified architecture, called *Simple Dharma*, is discussed first. *Simple Dharma* retains the main concept of Figure 8.5, viz., levelizing and time sharing of resources. However, the number of DLMs and the size of the DIA is kept small, so as to help understand the concepts better. The first example is very simple, and is more of a warm up exercise. The second example is more involved. The third example demonstrates that sequential circuit realization is very much like the combinational one.

### 9.1 *Simple Dharma*

*Dharma* consists of DLMs, DIA, pass-buffers and latches. The number of inputs and outputs per DLM, the number of DLMs, the number of pass-buffers, the interconnection structure of the DIA, the number of levels that can be implemented, etc., can all be set to different values so as to yield a family of *Dharma* chips. For this simple illustration, we use a hypothetical, simple architecture. This architecture has the following features:

- All DLMs are identical.
- Each DLM has 3 inputs ( $K = 3$ ) and 2 outputs.
- There are 3 DLMs ( $C = 3$ ).



- There are 5 pass-buffers ( $B = 5$ ).
- The  $B$ -crossbar and  $L$ -crossbar are implemented as full crossbars.
- There are 5 chip inputs and 6 chip outputs.
- $L = 5$ , levels of logic can be implemented (i.e., each crossbar point has 5 bits of memory to choose the configuration, and each DLM has  $2^3 \times 2 \times 5 = 80$  bits to store the logic functions).

Figure 9.1 is a block diagram of this simple architecture. The crosspoints in the  $L$ -crossbar and  $B$ -crossbar are shown as shaded circles. The shaded circles represent the switch at the crosspoints and the memory bits required to configure the switch (to be either on or off). The chip inputs enter the interconnection array via multiplexers. In the figure, the inputs are shown to be multiplexed only with the pass-buffer outputs. As in Figure 8.6, inputs can also be multiplexed with DLM outputs, but for the purposes of this simple architecture, those multiplexers are unnecessary. Each multiplexer is individually selected. The multiplexer selection memory has 5 bits per multiplexer. The DLM outputs are connected to the pass-buffer inputs by means of multiplexers. Each of these multiplexers also has a separate select signal, controlled by 5 memory bits per multiplexer. To keep the diagram simple, the multiplexer selection memory is not shown. The DLM and pass-buffer inputs are latched; and these latches are clocked by an internal clock.

## 9.2 Combinational example 1

This is a very simple example, and is meant more as an introduction. Figure 9.2 shows the schematic diagram for this example circuit. There are 5 inputs,  $a$ ,  $b$ ,  $c$ ,  $d$  and  $e$ , and 2 outputs  $x$  and  $y$ .

In terms of boolean equations,

$$G = a + b + c$$

$$H = d e$$

$$x = G H' + G' H$$

$$y = c + H$$

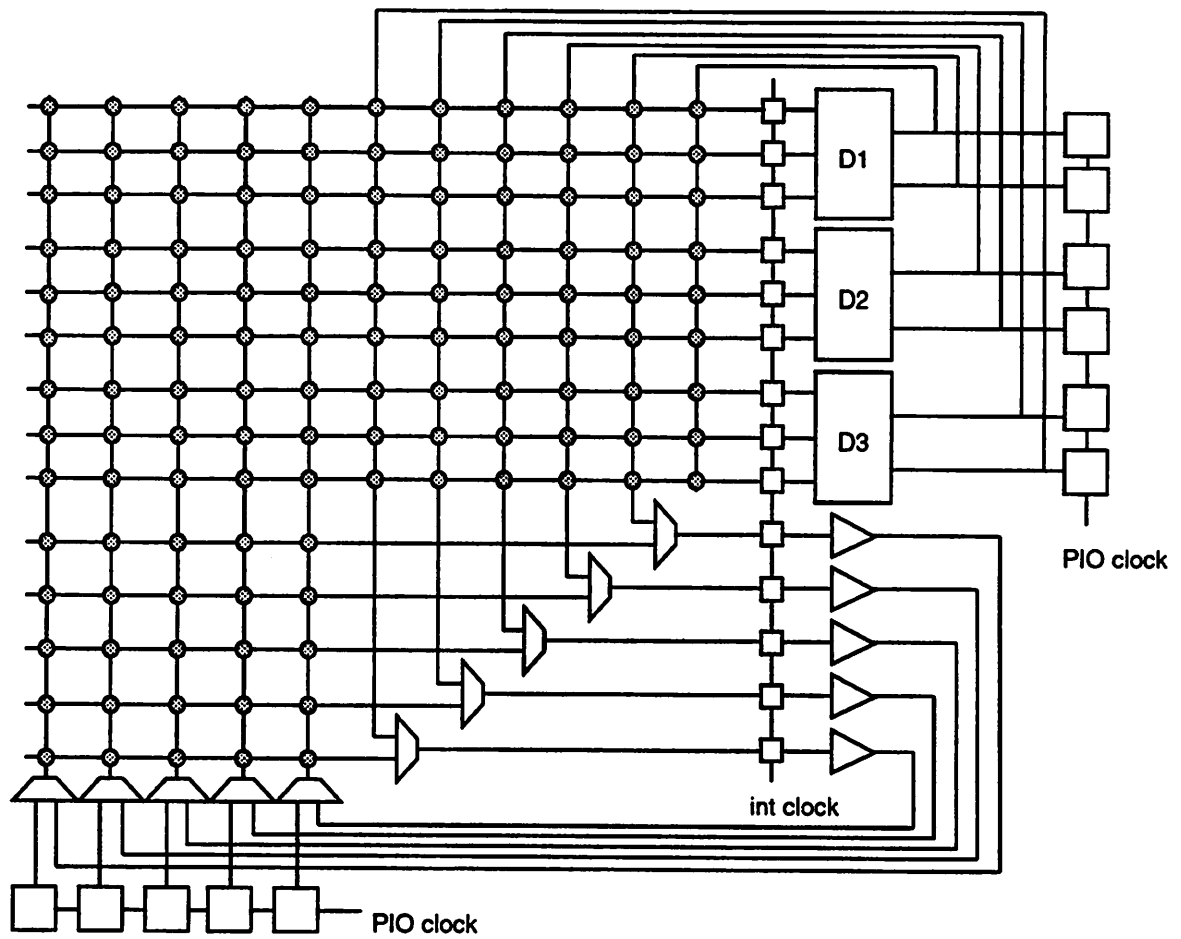


Figure 9.1: *Simple Dharma*

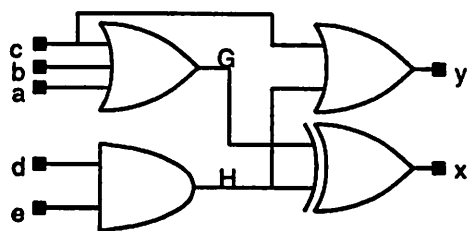


Figure 9.2: Schematic diagram for Example 1

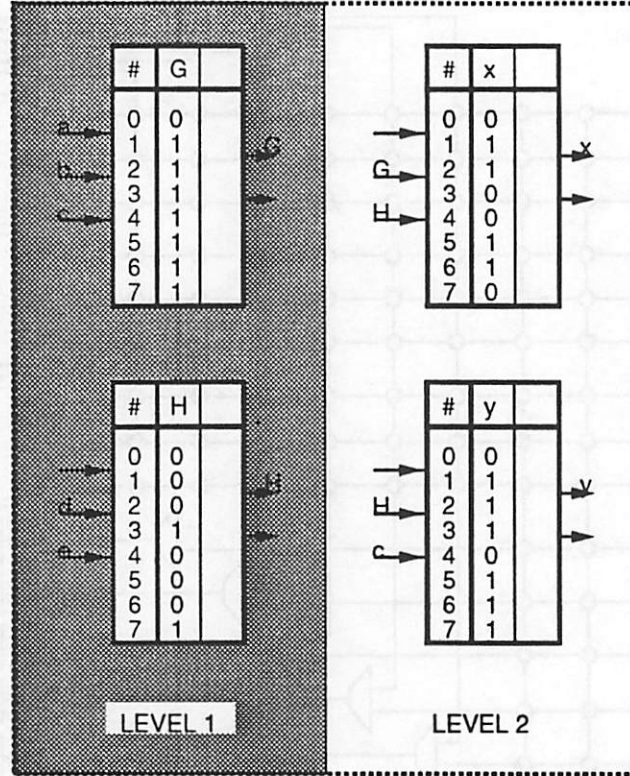


Figure 9.3: DLM values for Example 1

From Figure 9.2, we see that the circuit has 2 levels of logic. The realization of this example on *Simple Dharma* is explained in the following figures. Figure 9.3 shows the memory bits stored in the DLMs. In the first level inputs  $a$ ,  $b$ ,  $c$ ,  $d$  and  $e$  are connected to the DLMs, and outputs  $G$  and  $H$  are generated. In the second level,  $G$ ,  $H$  and  $c$  are connected to the DLMs to generate the outputs  $x$  and  $y$ .

Consider the bits stored to generate function  $G$ .  $G$  is the OR of its inputs  $a$ ,  $b$  and  $c$ . Hence it takes the value 1 whenever any input is 1, and takes the value 0 only when all the inputs are 0. The memory addressed when  $a$ ,  $b$  and  $c$  are all 0, is 0, and hence a 0 is stored at location 0. All the other memory locations have a 1 stored in them.

Next consider function  $H$ .  $H$  is the AND of its inputs  $d$  and  $e$ . Hence it takes the value 1 only when both inputs are 1. Since the DLM is a 3-input LUT, and  $H$  has only 2 inputs, half the memory locations are a duplicated. Signals  $d$  and  $e$  are connected to the

least significant bits of the memory address, and the most significant bit is not needed (an unused input can be connected to either Vcc or ground).  $d$  and  $e$ , both being 1 corresponds to memory addresses 3 (011) and 7 (111). Hence 1s are stored in these two locations. The rest of the memory bits are 0s.

The functions  $x$  and  $y$  are similarly programmed.

Figures 9.4 and 9.5 show the interconnection structure corresponding to Example 1. The signal values marked in these figures correspond to the time instants marked  $\epsilon$  in Figure 8.13. That is, the DLM has just generated signals at a particular level, and the DIA is configured to connect the next level. A black crosspoint represents a connection between the horizontal and vertical lines. A white crosspoint implies that the crossing lines are not connected. Multiplexer selection is shown by means of a line through the multiplexer symbol. This line connects the selected multiplexer input to the multiplexer output.

### 9.3 Combinational example 2

Figure 9.6 shows the schematic for Example 2. There are 5 inputs,  $a$ ,  $b$ ,  $c$ ,  $d$  and  $e$ , and 3 outputs,  $x$ ,  $y$  and  $z$ .

In terms of boolean equations,

$$G = a e$$

$$H = c d' + c' d$$

$$I = b d$$

$$F = a e' + a' e$$

$$J = I H' F$$

$$K = b c + b d + c d$$

$$x = G K + J$$

$$P = c d$$

$$y = H' F' b + H' F b' + H F' b' + H F b$$

$$M = F H b' + F H' b$$

$$N = K + G + M$$

$$O = b c' d + b c d'$$

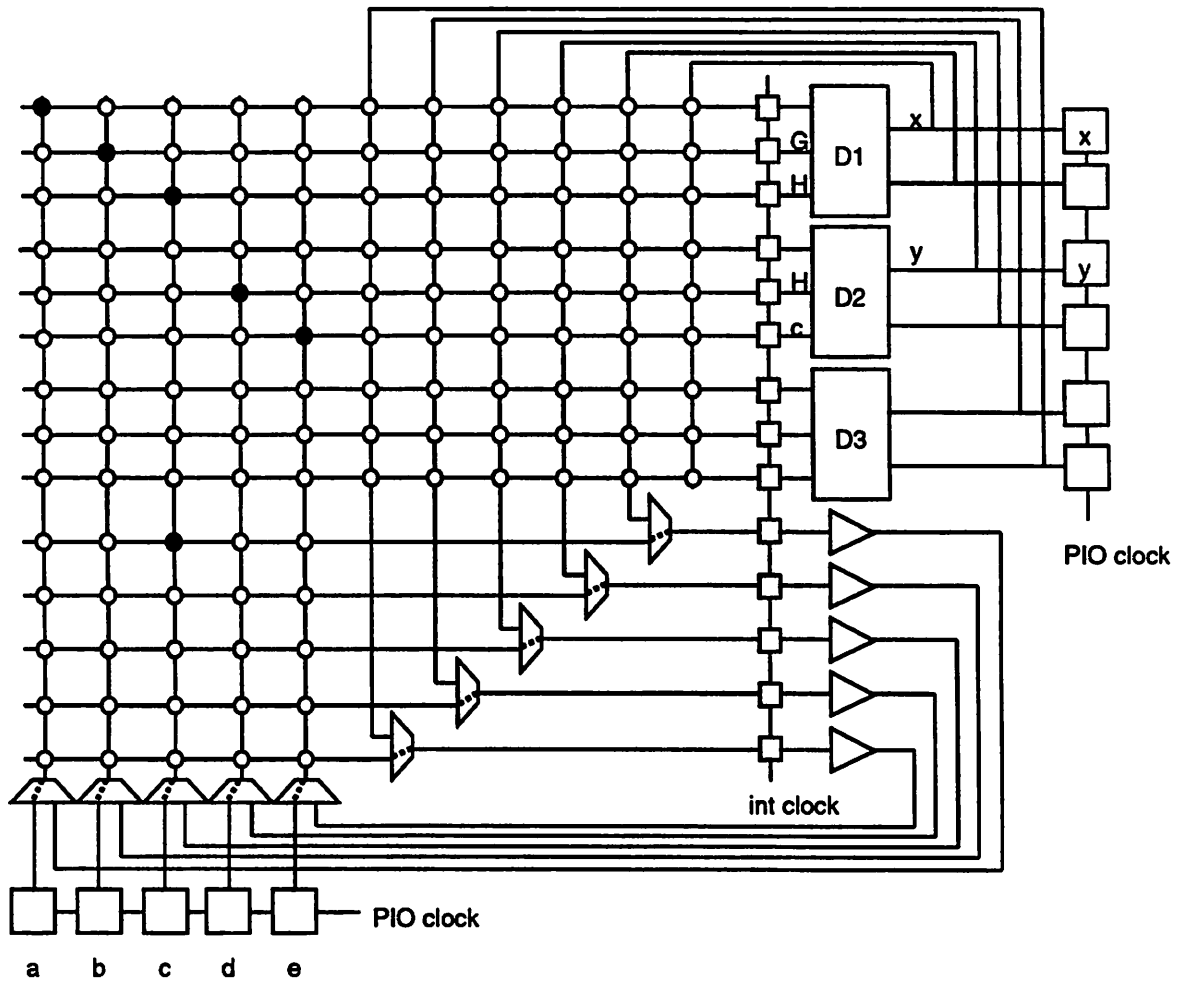


Figure 9.4: Example 1. DIA: Primary inputs → level 1; DLMs : Level 2.

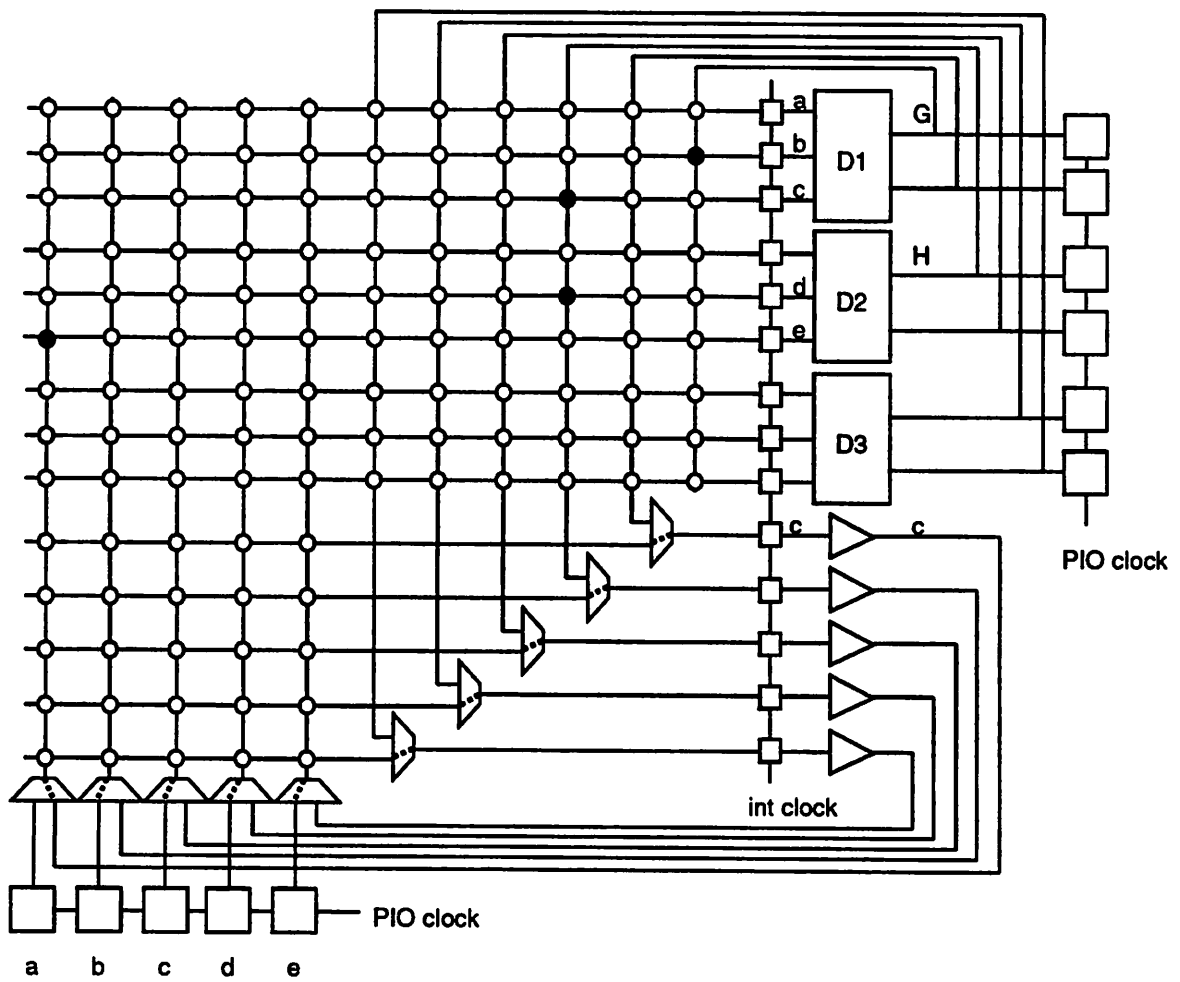


Figure 9.5: Example 1. DIA: Level 1 → Level 2; DLMs : Level 1.

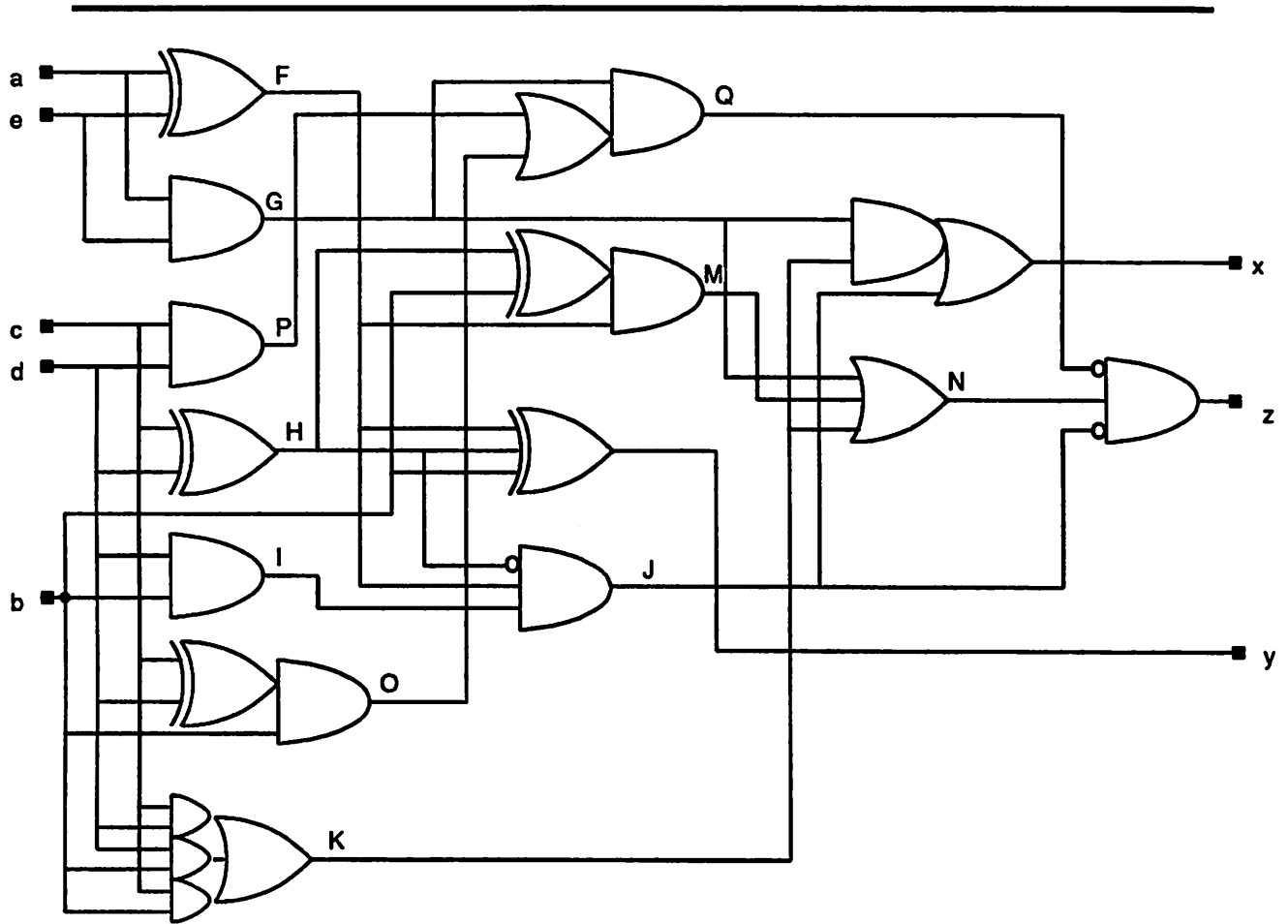


Figure 9.6: Schematic diagram for Example 2

$$Q = GP + GO$$

$$z = NJ'Q'$$

From Figure 9.6, we see that the circuit has 4 levels of logic. Note that topological levelization (schematic Figure 9.6 has been drawn topologically leveled – gates having the same topological level (see Chapter 11), from primary inputs, are vertically aligned) cannot be used to realize this circuit directly on *Simple Dharma*, since level 1 has 7 gates, and there are only 6 DLM outputs per level. In Figure 9.7, we show the level assignments to the various functions, after doing a temporal partitioning (Chapter 11). Figure 9.7 also shows the DLM memory bits required to perform the assigned functions.

In Figure 9.7, we note that  $x$  and  $y$  have been assigned to level 4 (topological level of  $x$  was 3, and that of  $y$  was 2). Also,  $K$ ,  $I$  and  $G$  have been moved to level 2, and  $Q$  and  $J$  have been moved to level 3. Functions that share common inputs have been assigned to the same DLM (eg.,  $H$  and  $O$ ; and  $K$  and  $I$ ).

Figures 9.8, 9.9, 9.10 and 9.11 show the interconnection configuration for the various levels. As in Example 1, the signal values correspond to the time instants marked  $\epsilon$  in Figure 8.13.

## 9.4 Sequential example

Figure 9.12(a) shows the circuit schematic of a simple sequential circuit (this is a bus arbitration circuit, and is a modified version of an example in [Alford 89]). The inputs are  $R1$  and  $R2$  and outputs are  $O1$  and  $O2$ . Signals  $S1$  and  $S2$  are the state variables. The (b) part of the figure shows only the combinational part of the finite state machine shown in (a).  $PS0$  and  $PS1$  are the *previous* (hence the prefix 'P') of the state variables  $S0$  and  $S1$  respectively. The circuit has a two-phase clock. This example therefore also serves to illustrate how multi-phase clocked circuits can be mapped to *Dharma*.

Figure 9.12(c) shows the phase relationship between the two clocks,  $\phi1$  and  $\phi2$ . The positive edge of  $\phi2$  must lag the corresponding edge of  $\phi1$  by  $\delta1$ .  $\delta1$  corresponds to the time required to generate signals  $S1$  and  $S2$ . Similarly, the next positive edge of  $\phi1$  should arrive only after a time delay of  $\delta2$ , where  $\delta2$  includes the latch propagation time and the propagation through the rightmost combinational gates in Figure 9.12(b). The input and output registers are clocked by the positive edge of  $\phi1$  and the state registers are clocked



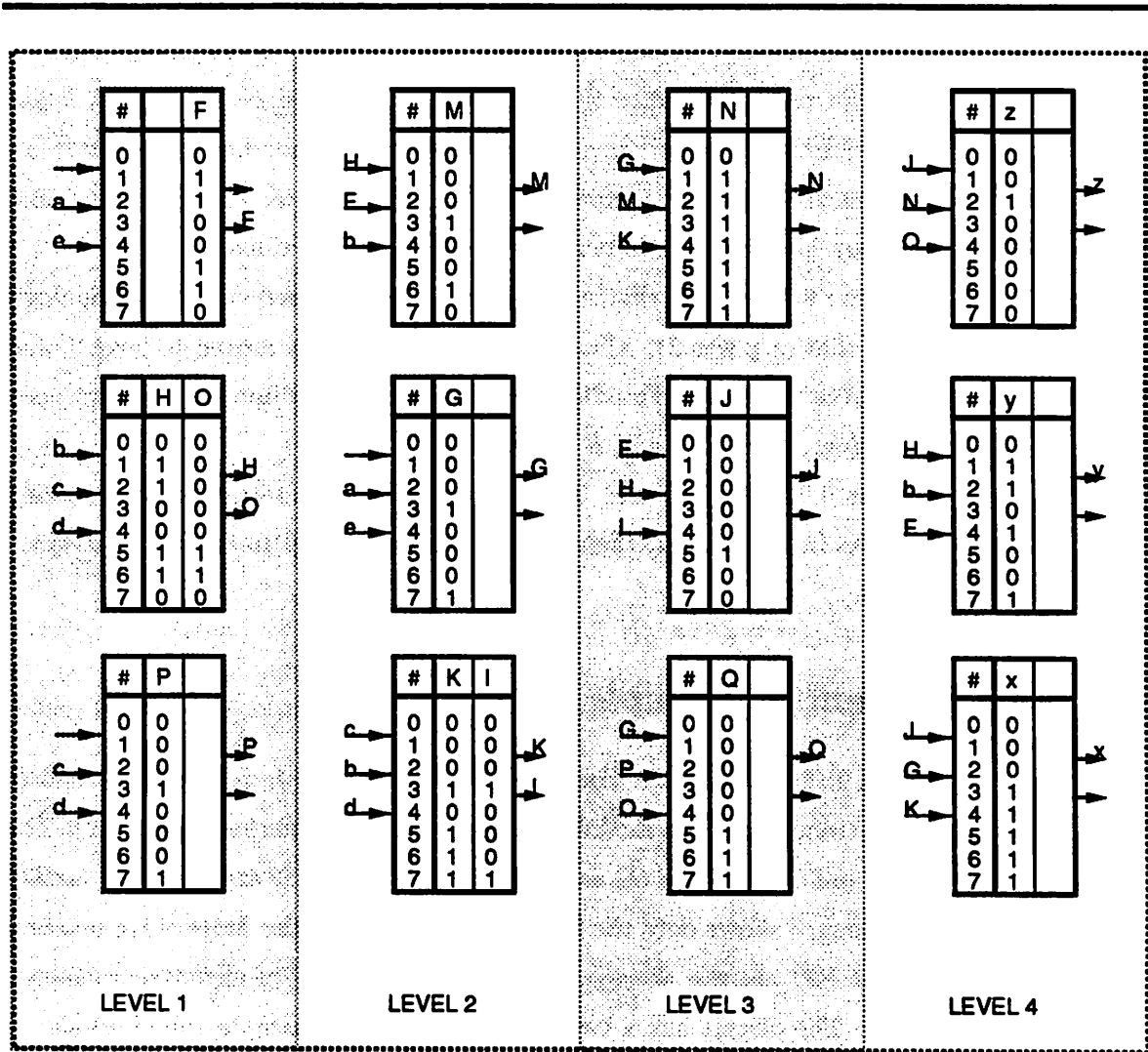


Figure 9.7: DLM values for Example 2

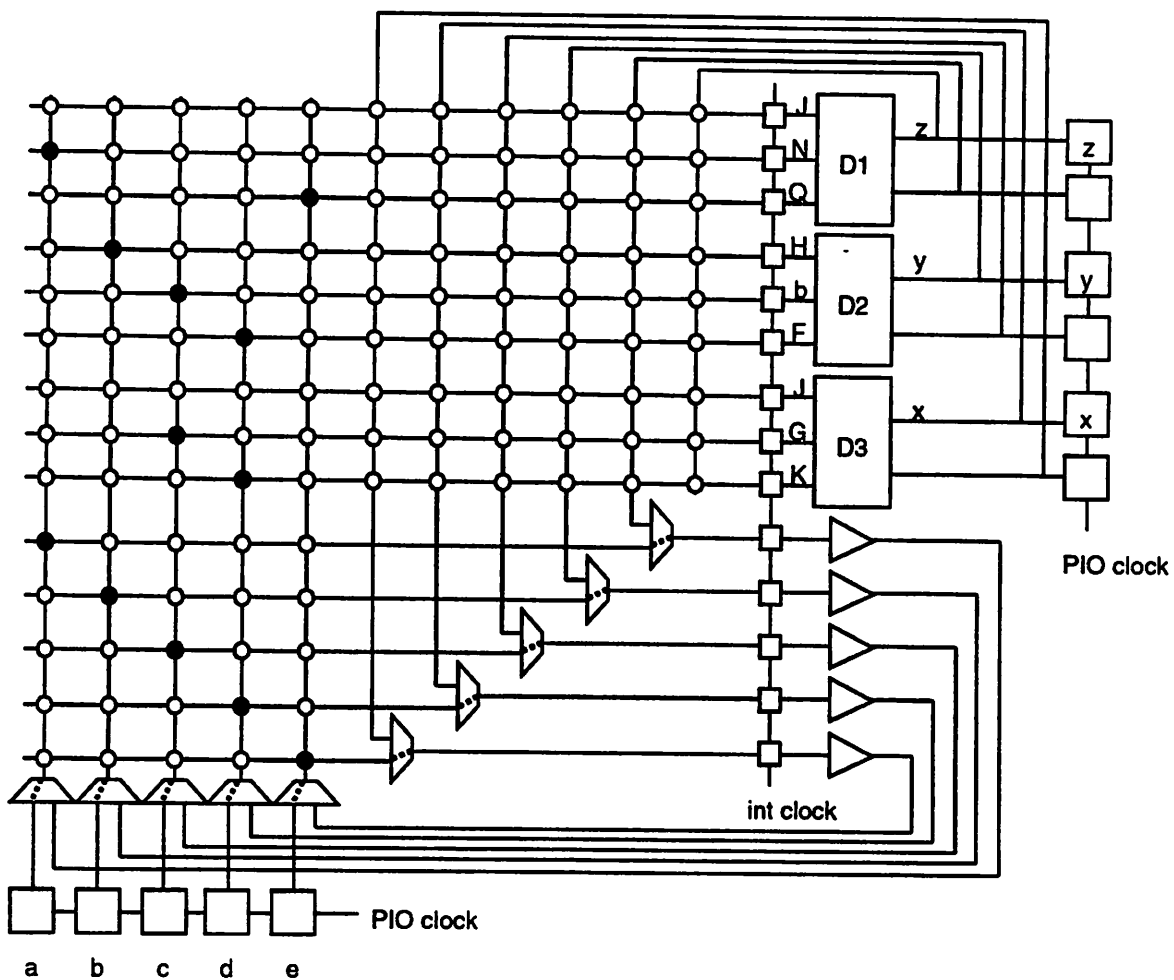


Figure 9.8: Example 2. DIA: Primary inputs → Level 1; DLMS : Level 4.

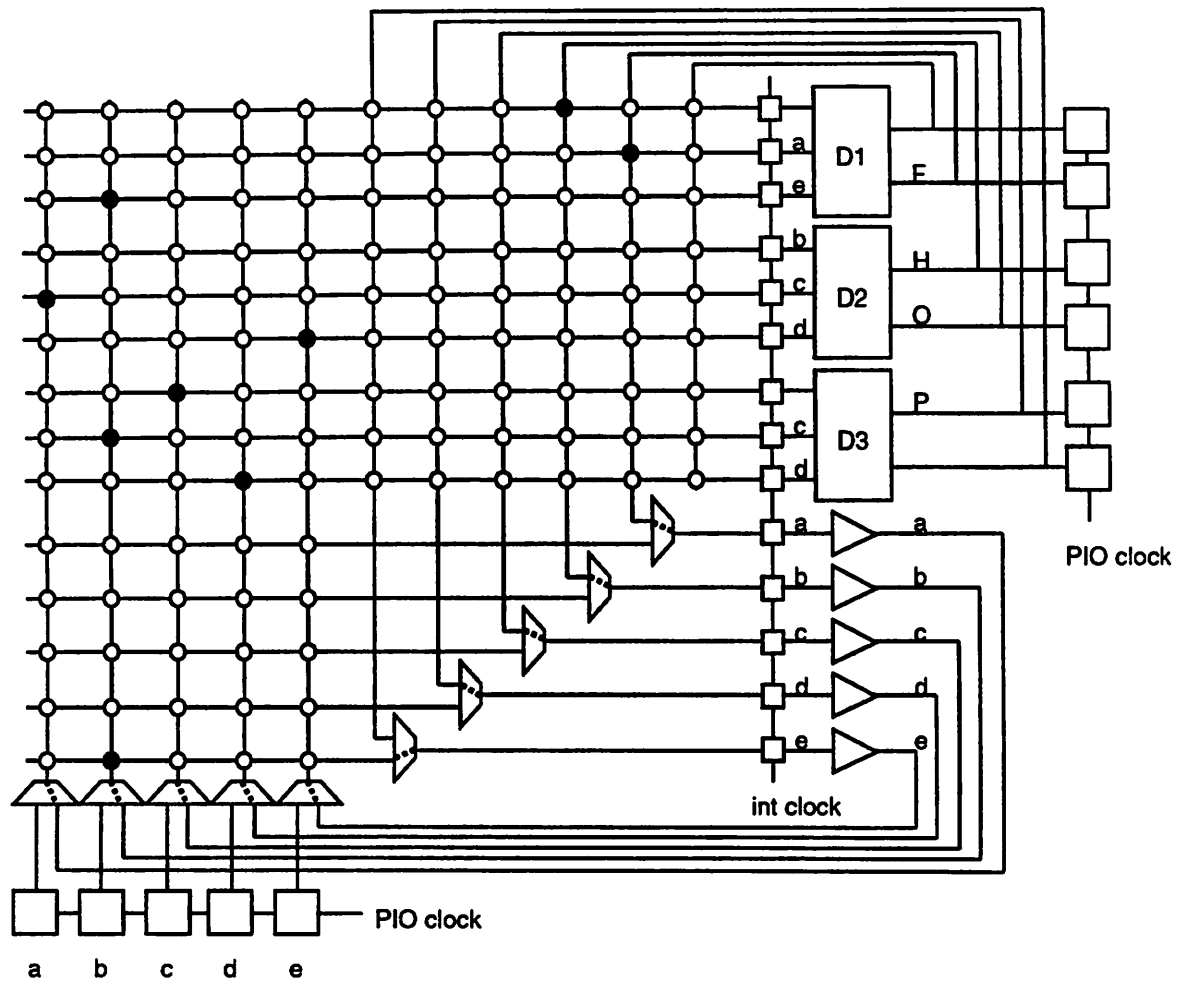


Figure 9.9: Example 2. DIA: Level 1 → Level 2; DLMs : Level 1.

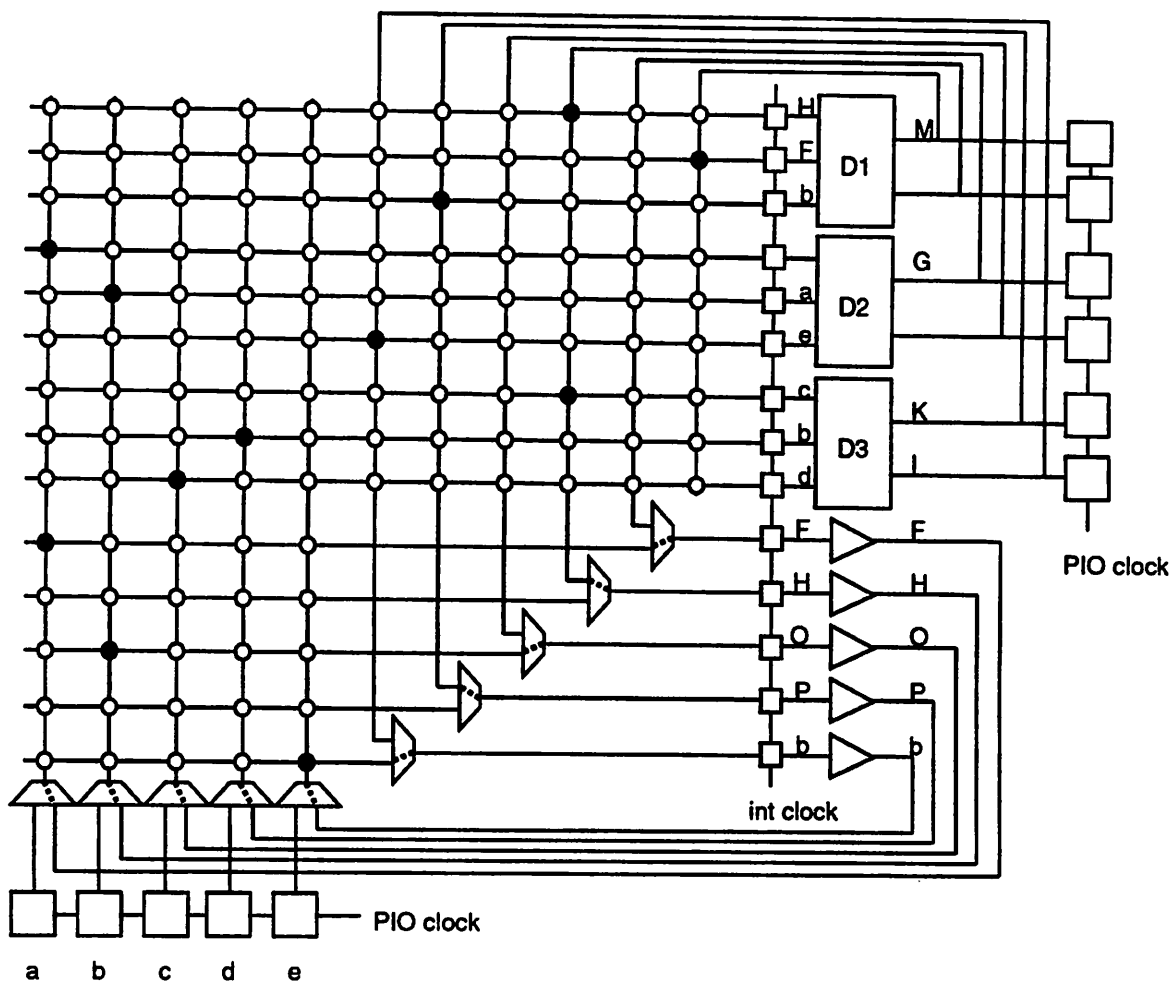


Figure 9.10: Example 2. DIA: Level 2 → Level 3; DLMS: Level 2.

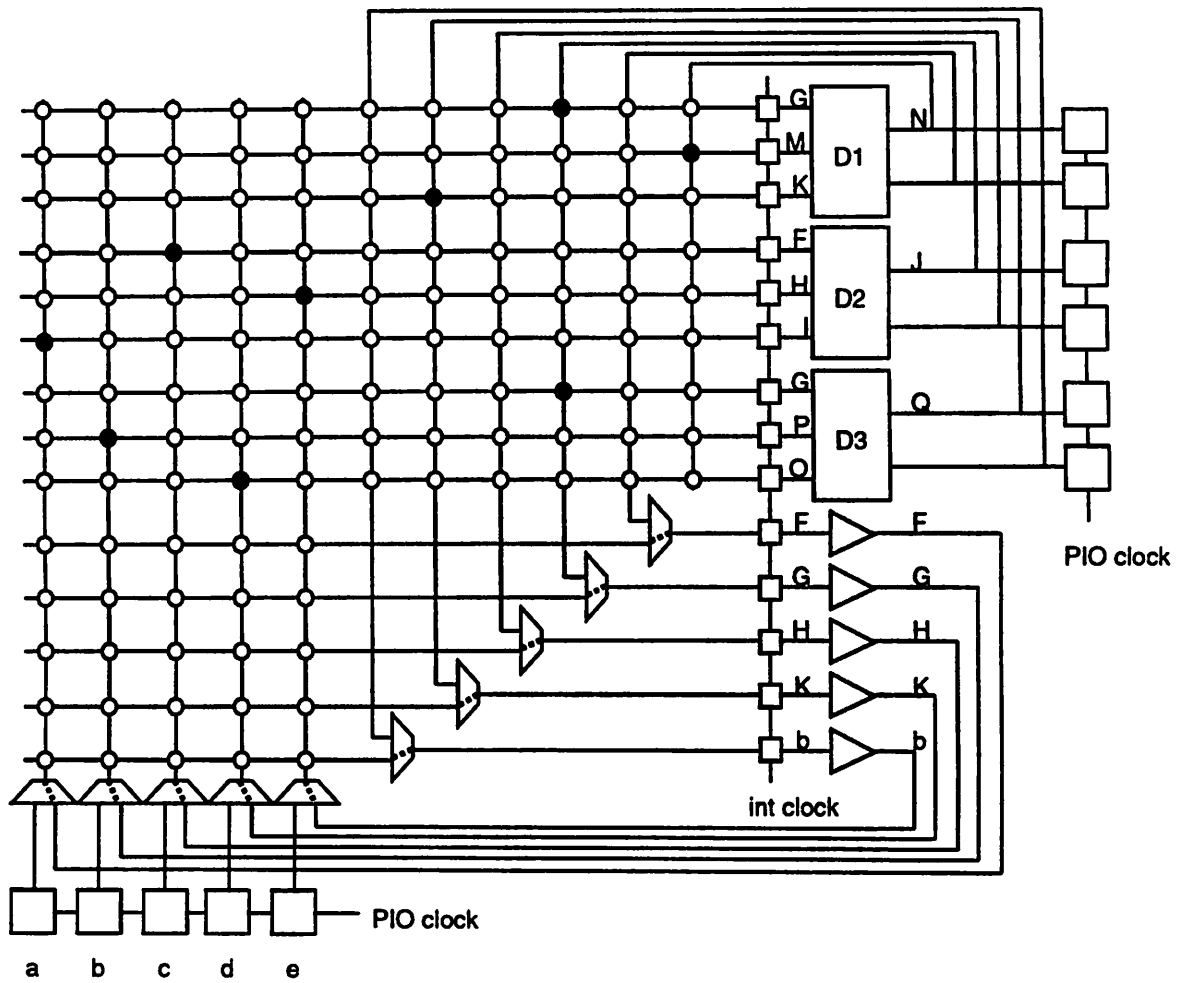
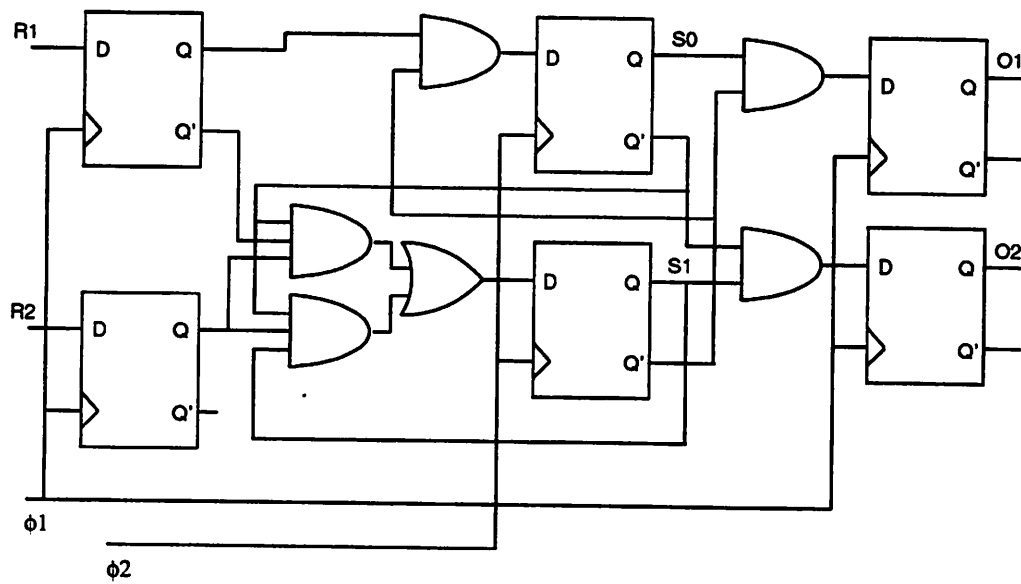
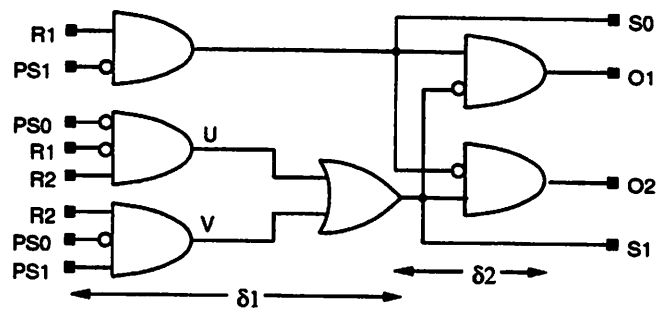


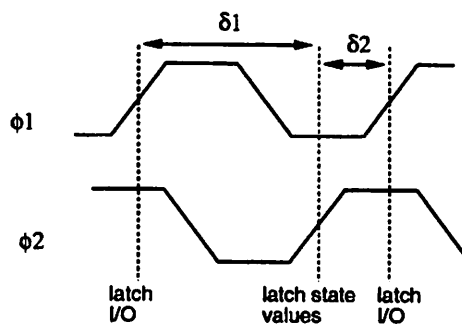
Figure 9.11: Example 2. DIA: Level 3 → Level 4; DLMs: Level 3.



(a) Circuit schematic



(b) Combinational part



(c) Clocking scheme

Figure 9.12: Sequential Example

by the positive edge of  $\phi_2$ .

Figure 9.13 shows the LUT values corresponding to Figure 9.12(b). The *Dharma* realization has 3 levels and requires only *one* phase of the clock,  $\phi_1$ . This clock can run at approximately one-third (see Section 12.2) the speed of the internal clock. Figures 9.14, 9.15 and 9.16 show the DIA configuration bits, and DLM signal values corresponding to time instants marked  $\epsilon$  in Figure 8.13. These are very similar to the combinational examples, except that the state variables are fed back into the DIA, at the last level, and become the  $PS_i$  ( $i = 0$  or  $1$ ) signals.

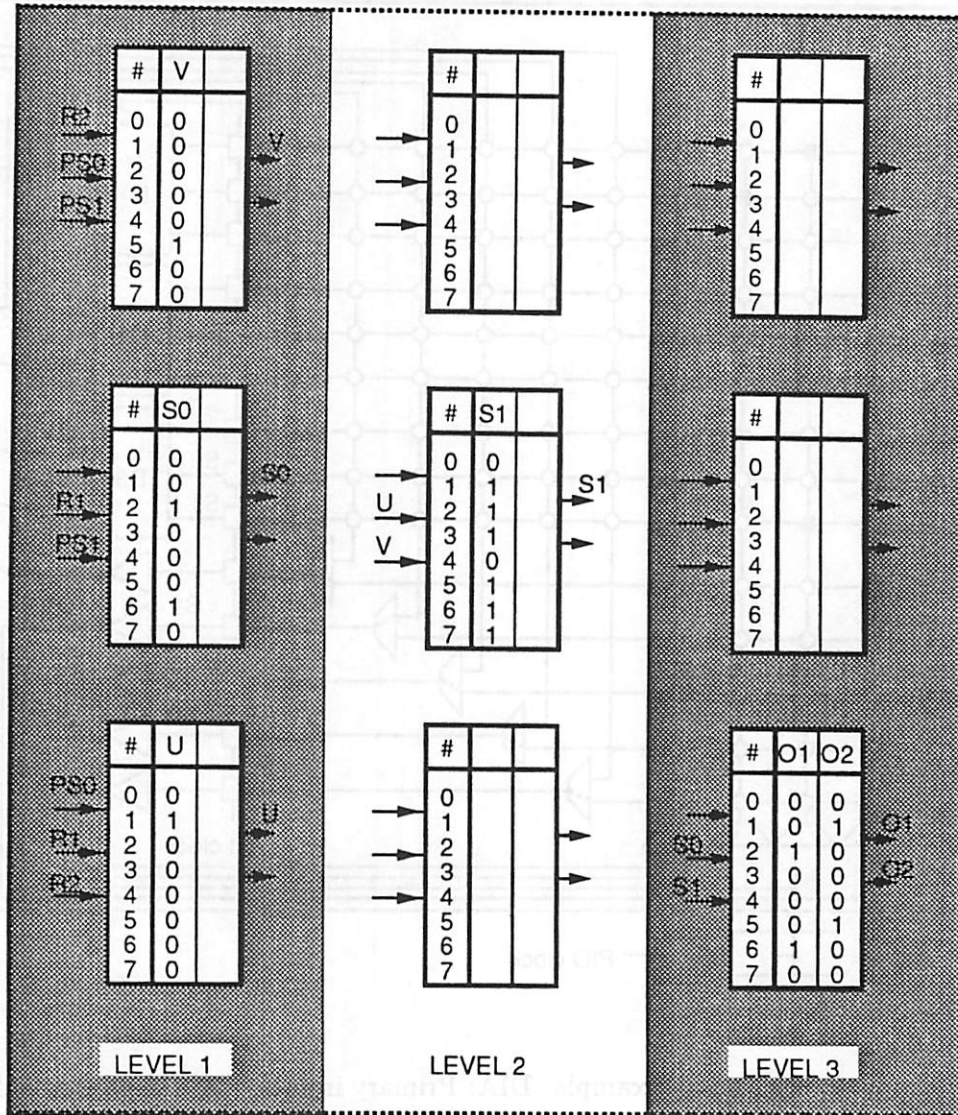


Figure 9.13: DLM values for the sequential example



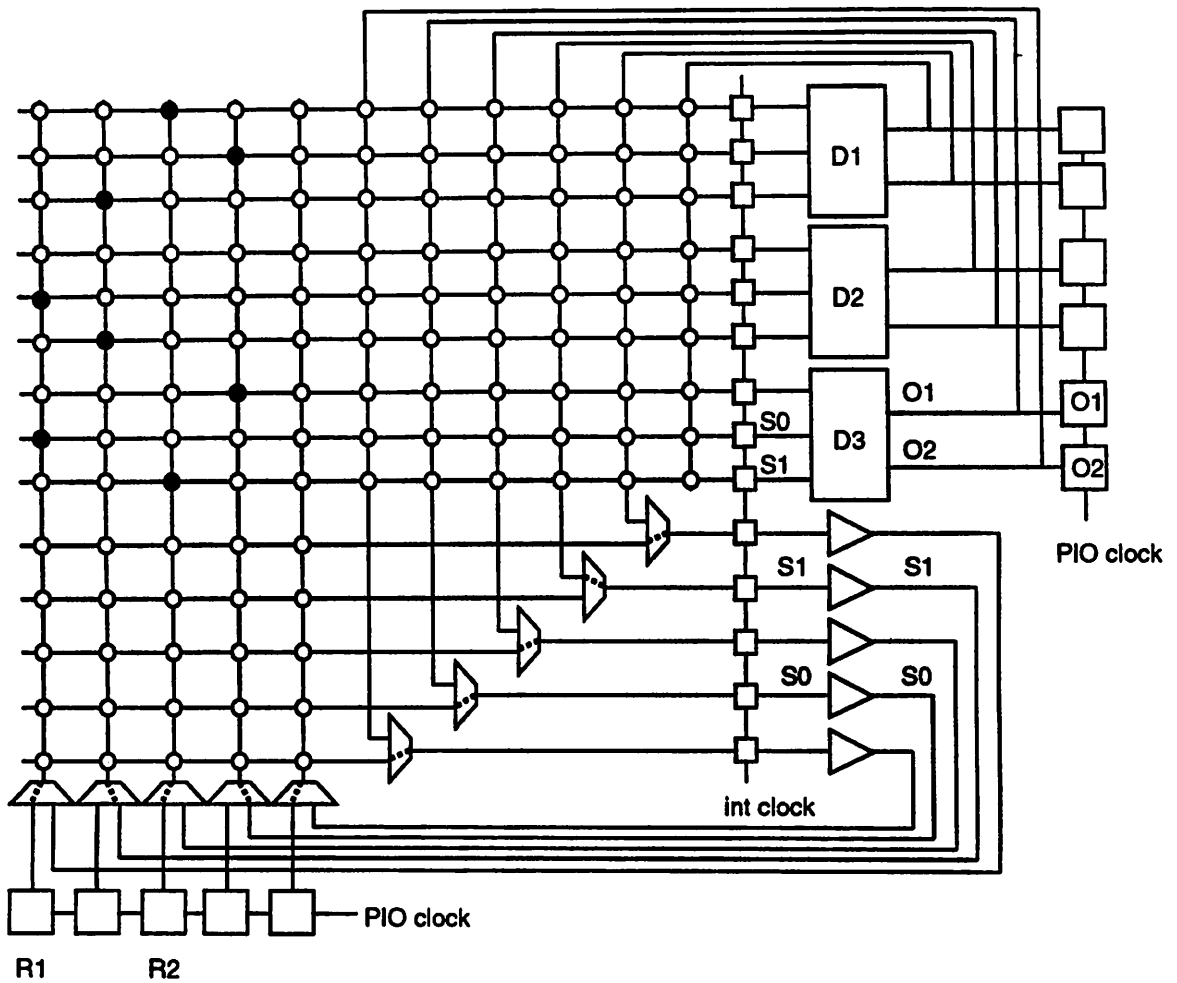


Figure 9.14: Sequential Example. DIA: Primary inputs, state variables → Level 1; DLMs: Level 3.

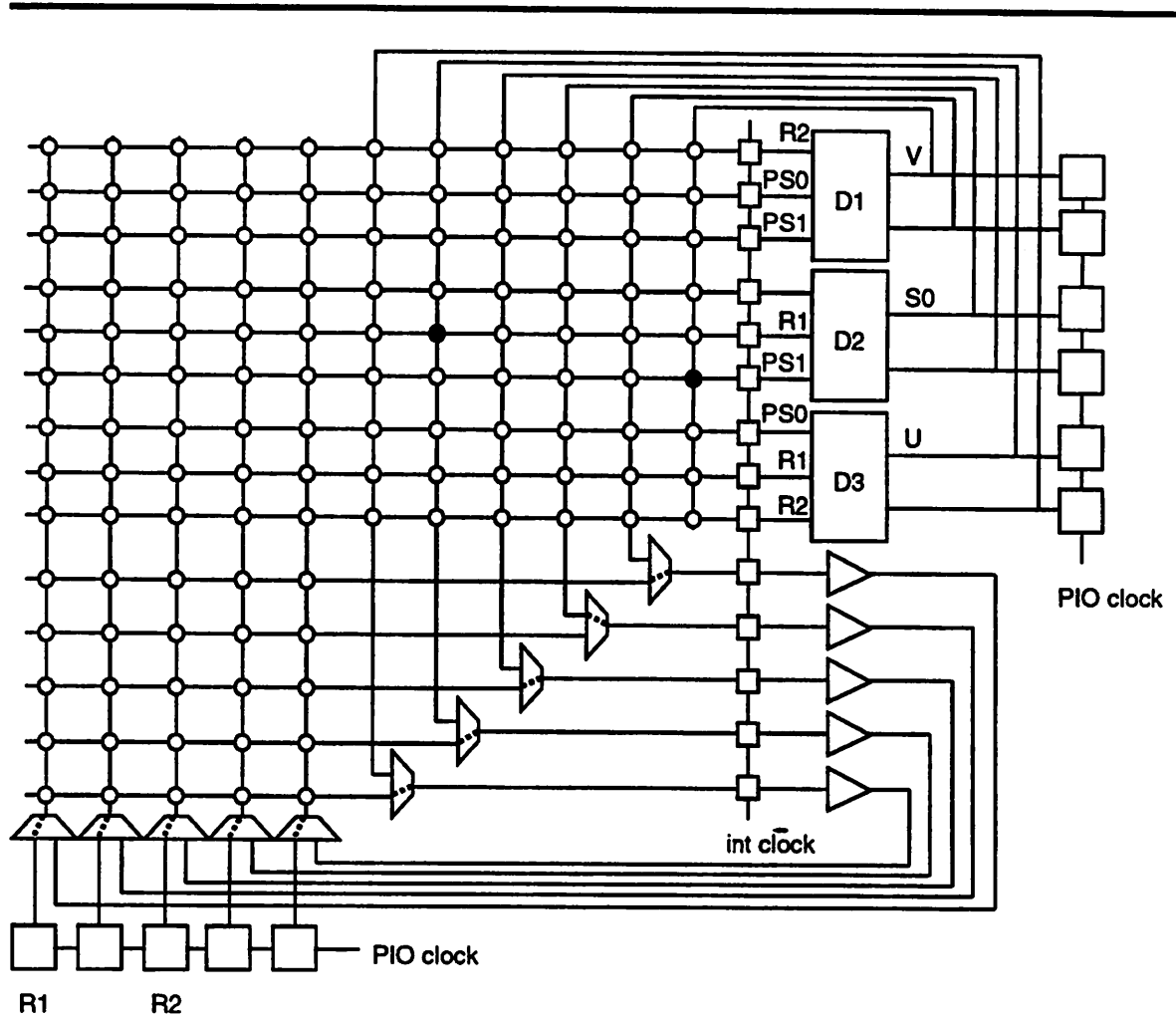


Figure 9.15: Sequential Example. DIA: Level 1 → Level 2; DLMs: Level 1.

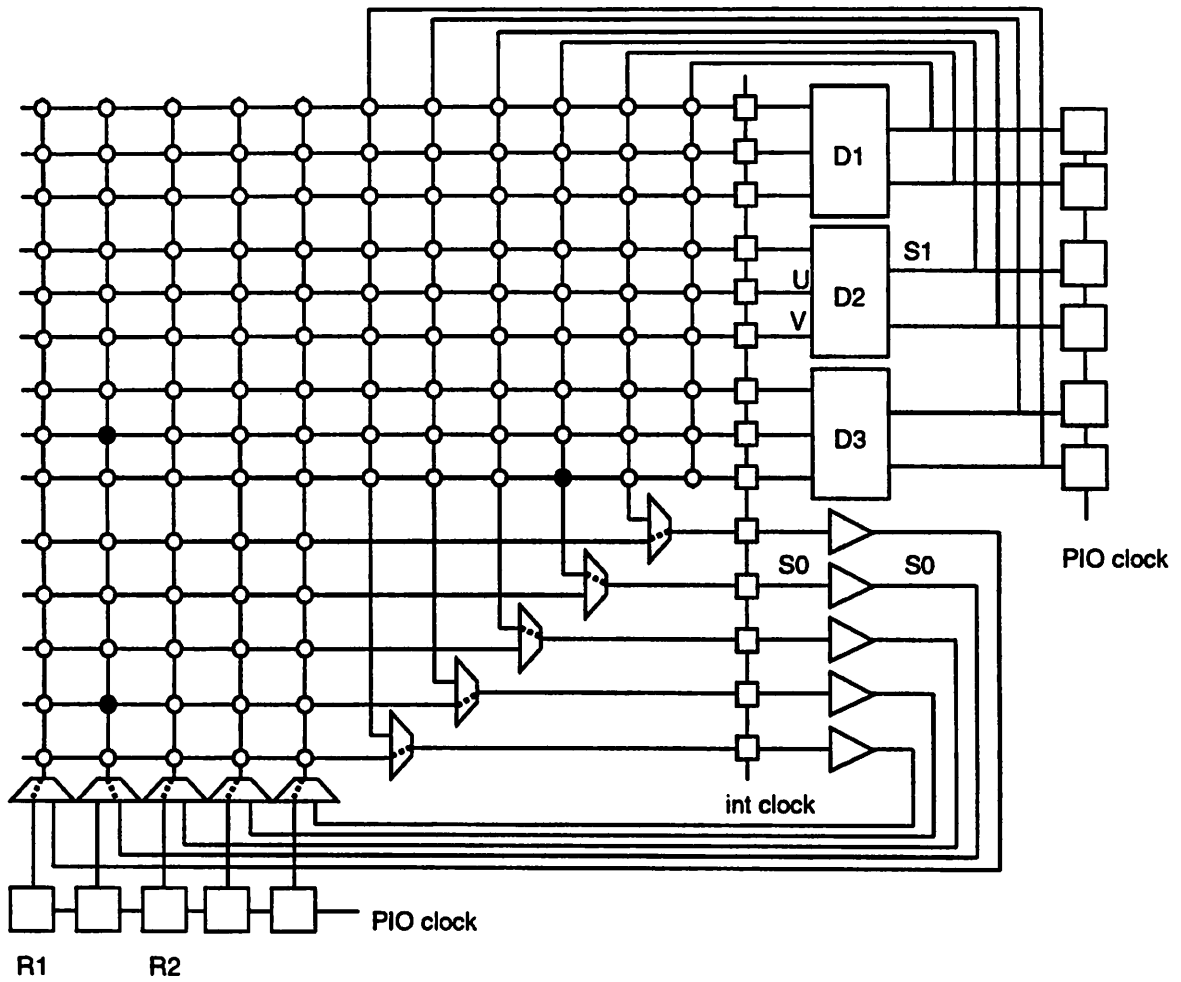


Figure 9.16: Sequential Example. DIA: Level 2 → Level 3; DLMs: Level 2.

## Chapter 10

# Variations and Modifications

In Chapter 8 we presented the *Dharma* architecture. The detailed block diagram (Figure 8.6) is just one possible way of realizing the dynamically reconfigurable folded pipeline of Figure 8.5. In fact, Figure 8.5 can be thought of as a generic architecture. Several variants to the specific architecture of Figure 8.6 are presented in this chapter. Variations and modifications of the DLM are grouped together in one section; similarly variations and modifications of the DIA are grouped together in another section. The DLM variations and the DIA variations are independent, unless stated otherwise. Therefore, each of the DLM variations can be used with each of the DIA variations; further, some variations within a group can be used along with the other variations in the same group. Hence, a large class of dynamically reconfigurable architectures is possible (see shaded region of Figure 1.3). The last section in this chapter describes how certain circuits with number of topological levels greater than the maximum number of levels allowed on a single *Dharma* chip, can be implemented on one chip, by means of modifications to the generic architecture.

### 10.1 Dynamic logic module

In Figure 8.6, we have shown the DLM to be a  $K$ -input, 1-output lookup table. *Dharma* is not necessarily limited to this kind of logic module. In general, any logic structure, which can be repetitively changed to implement different functions, can replace the  $K$ -input, 1-output LUT. In this section, we present complex LUT DLMs, multiplexer based DLMs, PLA based DLMs and heterogeneous DLMs.

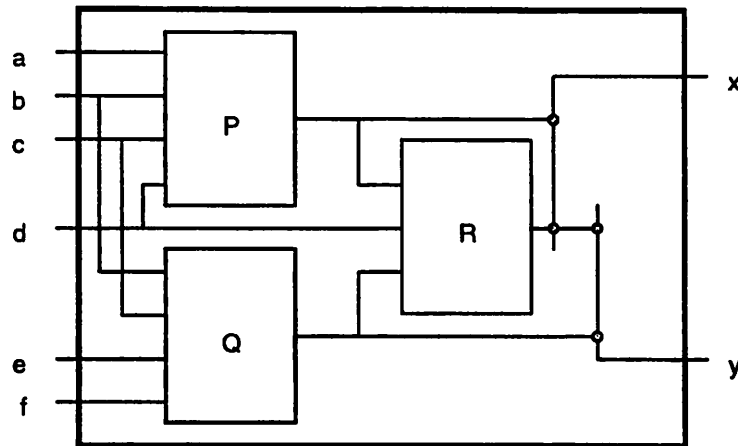


Figure 10.1: An example of a complex LUT DLM

### 10.1.1 Complex LUT DLMs

The  $K$ -input, 1-output LUT is one specific, simple manifestation of a broader class of reconfigurable logic modules based on the lookup table. A  $K$ -input  $m$ -output ( $m > 1$ ) LUT and interconnected LUTs are other such logic modules in this class; and can replace the  $K$ -input 1-output LUT in the *Dharma* architecture.

As an example, Figure 10.1 shows one possible complex DLM with 6 inputs ( $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$  and  $f$ ) and 2 outputs ( $x$  and  $y$ ).  $P$  and  $Q$  are 4-input 1-output LUTs and  $R$  is a 3-input, 1-output LUT.  $P$  and  $Q$  share two inputs ( $b$  and  $c$ ) and  $P$  and  $R$  share one input ( $d$ ). Any two of  $P$ ,  $Q$  and  $R$  can be selected as the DLM outputs by means of a mini-switching network on the output lines. The black dots represent crosspoints, which are similar to the crosspoints in a crossbar. Each output line has to be connected to at least *and* at most one of the LUT outputs. Hence exactly one switch on each output line is going to be on, and the other is going to be off. This implies that only one bit is required per output line, when the bit is set to '1', it turns on one of the crosspoints, and when it is set to '0', it turns on the other.  $P$  and  $Q$  require 16 bits each,  $R$  requires 8 bits and the crosspoints require 2 bits – giving a total of 40 bits per level of logic.

Several other LUT interconnections are possible, Figure 10.1 is meant to only give a feel for what is possible. Xilinx CLBs [Xilinx 89] and the ORCA PLC [ORCA 93] are

good examples of complex DLMs <sup>1</sup>.

### 10.1.2 Multiplexer based DLMs

The DLM is not restricted to use only LUTs. Like the Actel [Actel 91] logic module, the DLM can be constructed from multiplexers. Unlike the Actel logic modules, the programmable elements will be read/write memory bits in *Dharma*, and not anti-fuses.

### 10.1.3 PLA based DLMs

A two level AND-OR logic plane can be used for the DLM. That is, a mini-PLA or mini-PAL or any such other programmable AND-OR block can function as the reconfigurable logic module. Figure 10.2 shows a symbolic diagram of a 5-input, 2-output DLM with programmable AND plane and programmable OR plane. There are 5 product terms, and each output OR gate has these 5 terms as inputs. The inputs enter the AND plane in both true and complement form. The connections from the input lines to the AND gates, and the connections from the AND gate to the OR gate are programmable, and are controlled by means of stored configuration bits. An input can be present in either true or complement form (and not both) in any product term, or it need not be present at all. This implies that each intersection point requires a bit (i.e., we cannot use the bit saving scheme that was used in the mini-crossbar of Figure 10.1). For the example shown in Figure 10.2. the AND plane requires 50 bits, and the OR plane requires 10 bits, giving a total of 60 bits per level of logic.

Several other programmable AND-OR types of circuits (eg., fixed OR plane. invertible outputs, etc.) are possible, and Figure 10.2 serves only as an illustration of what we mean by a PLA based DLM.

### 10.1.4 Heterogeneous DLM array

The above discussion assumes that all DLMs in the logic module array (see Figure 8.5) are identical. This need not be the case. The array can be composed of different types of DLMs, with a couple of each type. The logic module array in Figure 10.3 has 5 DLMs, 2 of which are 3-input AND gates, 2 are 5-input 1-output LUTs and one is an

---

<sup>1</sup>Only the combinational part of the CLB or PLC can be used as a DLM. The latches in the CLB and PLC do not serve any purpose in the DLM.

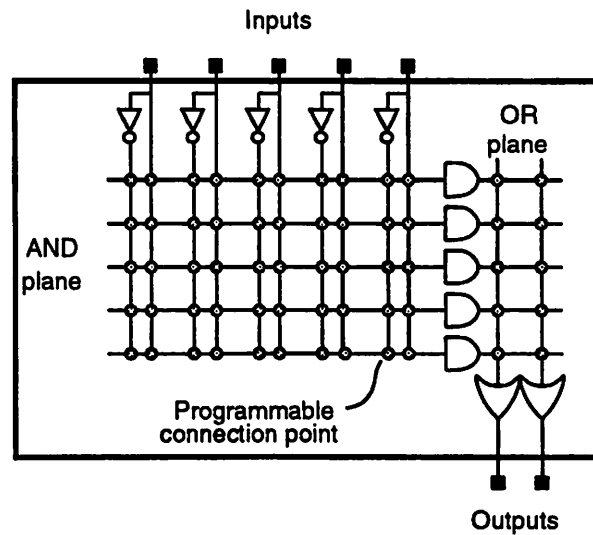


Figure 10.2: An example of a PLA based DLM

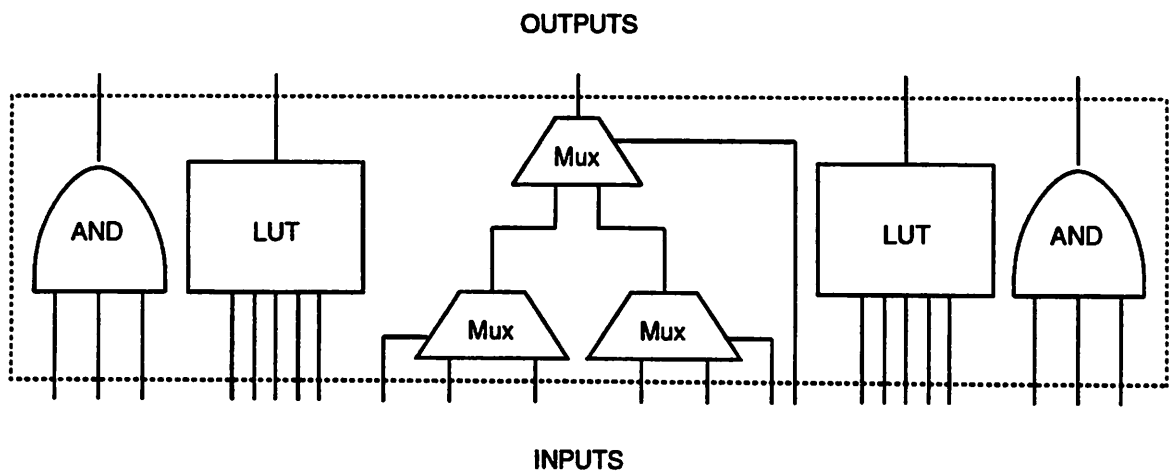


Figure 10.3: A heterogeneous DLM array

interconnection of 2-input multiplexers. The 3-input AND gate is not reconfigurable, and may seem out of place, but usually many combinational functions are simple ANDs or ORs, and by having these fixed functions in the DLM array, it is possible that there will be a saving in silicon area.

### 10.1.5 Technology dependent modifications

Modifications are also possible in how the actual implementation of the DLM is made. eg., a  $K$ -input, 1-output lookup table can be implemented using SRAM, DRAM, etc., technology. The dynamic nature of the DLMs is especially attractive to DRAM technology, since the refresh rate of the memory cells can be synchronized with the level addressing. Real-time reconfiguration does not imply that the memory has to have a 'write' capability. Read-only memories can also be used to store the configuration bits, the appropriate bits being chosen at each level by proper addressing using the level bits.

## 10.2 Dynamic interconnect array

The dynamic interconnect array has a large number of variations possible, both in terms of the overall architecture, and the actual implementation thereof. We have discussed below many of such variations.

### 10.2.1 Shift register at crosspoint

Each crosspoint of the crossbar has  $L$  bits of memory, a bit being used per level. These bits could be arranged in the manner of a shift register so that the change of level would just mean shifting one bit. Figure 10.4 shows a portion of the  $L$ -crossbar which uses this scheme. The connection at each crosspoint is achieved by means of an NMOS pass transistor, and each shift register has 4 bits (i.e.,  $L = 4$ ). Lines  $a$ ,  $b$  and  $c$  enter the crossbar from the PI muxes (see Figure 8.6), and the output lines  $x$  and  $y$  go to the latches at the DLM inputs. Connection between the horizontal and vertical lines is established whenever a '1' is present at the gate of the pass transistor.

The shift registers are clocked by the internal clock, and at the clock-edge all bits shift by one to the right. The rightmost bit loops back to the first bit. Figure 10.4(a) shows the bits at level  $d$ , and Figure 10.4(b) shows the bits shifted right by one, for the next



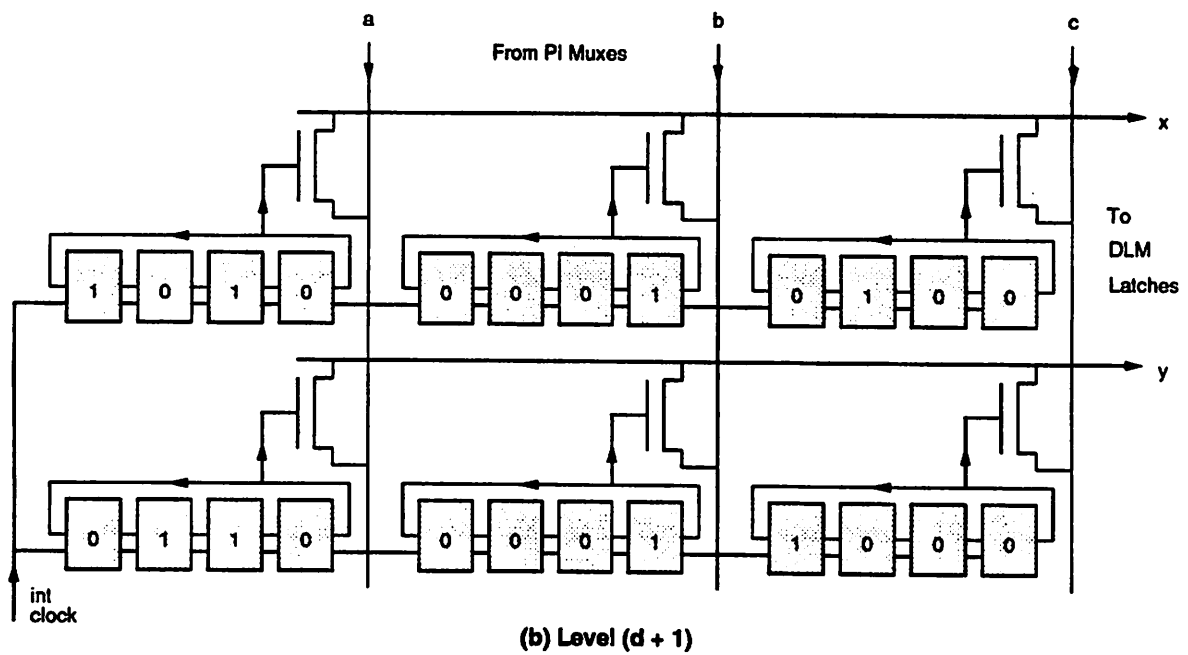
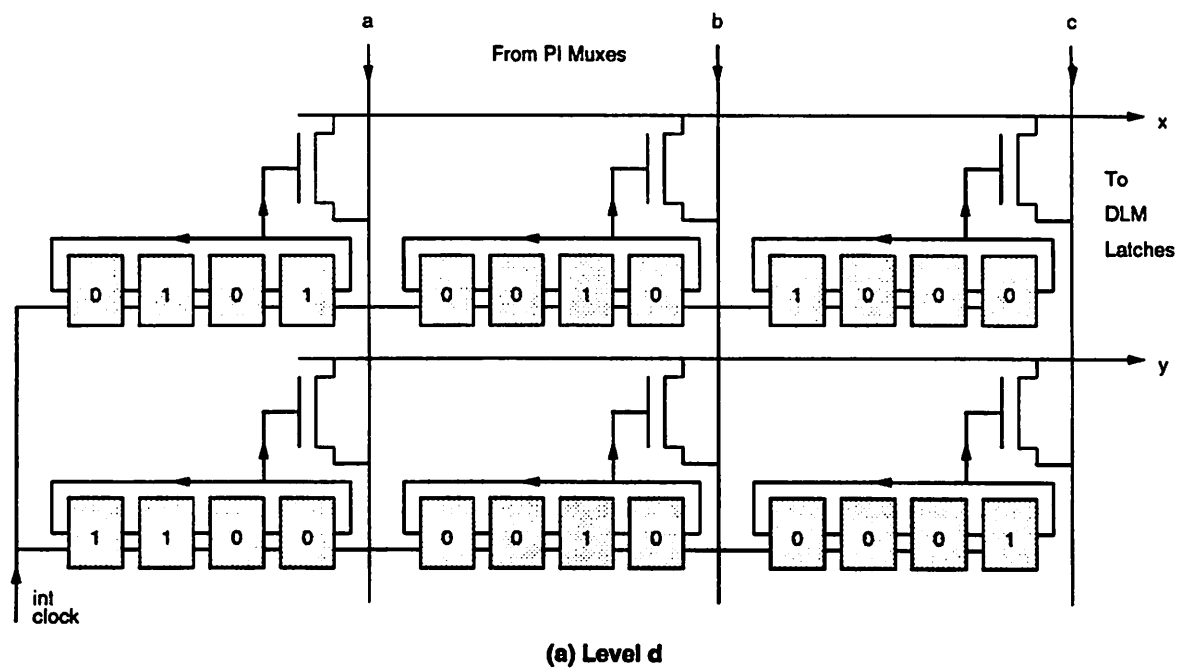


Figure 10.4: An example of  $L$ -crossbar which uses shift registers

level  $(d + 1)$ . In (a), vertical line  $a$  is connected to horizontal line  $x$ , and vertical line  $c$  is connected to horizontal line  $y$ . Similarly, in (b),  $b$  is connected to  $x$  and  $y$ <sup>2</sup>.

The disadvantage in such a shift register based DIA would be that the entire  $L$  bits will have to shift through before returning to the first level, and hence only circuits with  $l$  levels, where  $l$  is a perfect divisor of  $L$  can be implemented. However, this may not pose any real problem. The chip manufacturer can provide a series of chips with different  $L$ . The user then has to pick the chip with the appropriate  $L$  (i.e.,  $L$  such that it is equal to the circuit's levels  $l$  or is an exact multiple of  $l$ ). For example, a chip with  $L = 6$  can implement a circuit having  $l = 2, 3, \text{ or } 6$ . The  $l = 2$  and  $l = 3$  circuits, however, will be wasting a third, and a half of the resources, respectively.

### 10.2.2 Multiplexer based crossbar

Assume we are using a  $K$  input, 1-output LUT as a DLM. Let there be  $C$  such DLMs and  $B$  pass-buffers. Since there are  $(B + C)KC$  crosspoints in the  $L$ -crossbar, we will require  $(B + C)KCL$  memory bits. However, a signal fans out to at most one pin of the DLM. In other words, the  $(B + C)$  signals need to connect to only one of the  $K$  inputs per DLM. This means that the  $K$  bits can be reduced to  $\log_2 K$  bits per signal, reducing the total bits to  $(B + C)(\log_2 K)CL$ . However, this would warrant a decoder at every DLM, for each of the  $(B + C)$  lines. A more intelligent way of designing the crossbar is to recognize that at each LUT input we need, not a  $(B + C)$ -to- $K$  crossbar, but a  $(B + C)$ -to- $K$  multiplexer. A  $(B + C)$ -to-1 multiplexer requires  $\log_2(B + C)$  select lines, and we require  $K$  such multiplexers. Each select line needs a bit per level, therefore we need  $KCL \log_2(B + C)$  bits and  $KC$   $(B + C)$ -to-1 decoders. This would be more area efficient if

$KCL \log_2(B + C) + KC(B + C)\delta < KCL(B + C)$ , where  $\delta$  is the bit-equivalent of a decoder and gate.

$$\text{i.e., } \delta < L\left(1 - \frac{\log_2(C+B)}{(C+B)}\right).$$

Since the DLMs and interconnections both require decoders, there could perhaps be a sharing of decoders possible. This could require extra latches at the DLM outputs, and could also increase the routing delay.

---

<sup>2</sup>It is an error to have more than one vertical line connected to a horizontal line in the same level.

### 10.2.3 Partially hard-wired DIA

A further decrease in the number of crosspoints can be achieved if we allow certain hard-wired connections. Each DLM can have a hard-wired connection to  $K$  DLMs, and each DLM will have hard-wired inputs from  $K$  DLMs, preferably those that are adjacent to it. This would imply that the crossbar (or multiplexer, as the case may be) would then be reduced to a  $(B + C - K) \times KC$  crossbar, and there would be  $KC$  additional 2-input multiplexers (one at each DLM input), with associated  $KCL$  select bits. This interconnection scheme would decrease the number of fanout points from the DLM output to  $(C - K)K + K$ , instead of the earlier  $KC$ . However, the delay in the interconnect could increase, since the signal has to pass through an additional multiplexer (2-to-1 multiplexer).

Figure 10.5 shows the hard-wire connections from DLM  $D3$  to its adjacent DLMs.  $K = 5$  in this figure. The output of  $D3$  is *hard-wired* (through the 2-to-1 multiplexer) to input 5 of  $D1$ , input 4 of  $D2$ , input 3 of  $D3$ , input 2 of  $D4$  and input 1 of  $D5$ . The output of  $D3$  is connected to other DLMs (other than  $D1 \dots D5$ ), through the  $L$ -crossbar. Figure 10.6 shows a portion of the  $L$ -crossbar corresponding to the example of Figure 10.5. As a result of the hard-wire connections, several crosspoints are now unnecessary. For example, the output of DLM  $D3$  has no crosspoints at the input lines corresponding to DLMs  $D1 \dots D5$ . If  $D3$  needs to connect to  $D1$ , for example, then the hard-wire connection must be used.

### 10.2.4 Binomial concentrator based DIA

A different type of interconnection strategy can be adopted for LUT DLMs based on the following observation: the DLM inputs are all equivalent. Hence, the interconnection switch at the inputs is, by circuit switching terminology, a  $(B + C)$ -to- $K$  concentrator. In other words, we need to select any  $K$  of the  $(B + C)$  signals. This can be achieved by using a two-level sparsely connected crossbar, called a binomial concentrator [Masson 77]. The chief advantage of using such an interconnection strategy is that the number of crosspoints are reduced and as a result the number of bits used for storage are reduced. The disadvantage is that the signals now have to pass through two crosspoints in series. This could increase the interconnect delay (this is not clear at this point, simulation studies have to be performed to arrive at a definite answer – the reduced crosspoints implies that the signals have to be connected to lesser switch inputs, and this will reduce the parasitic capacitance, and could improve speed).

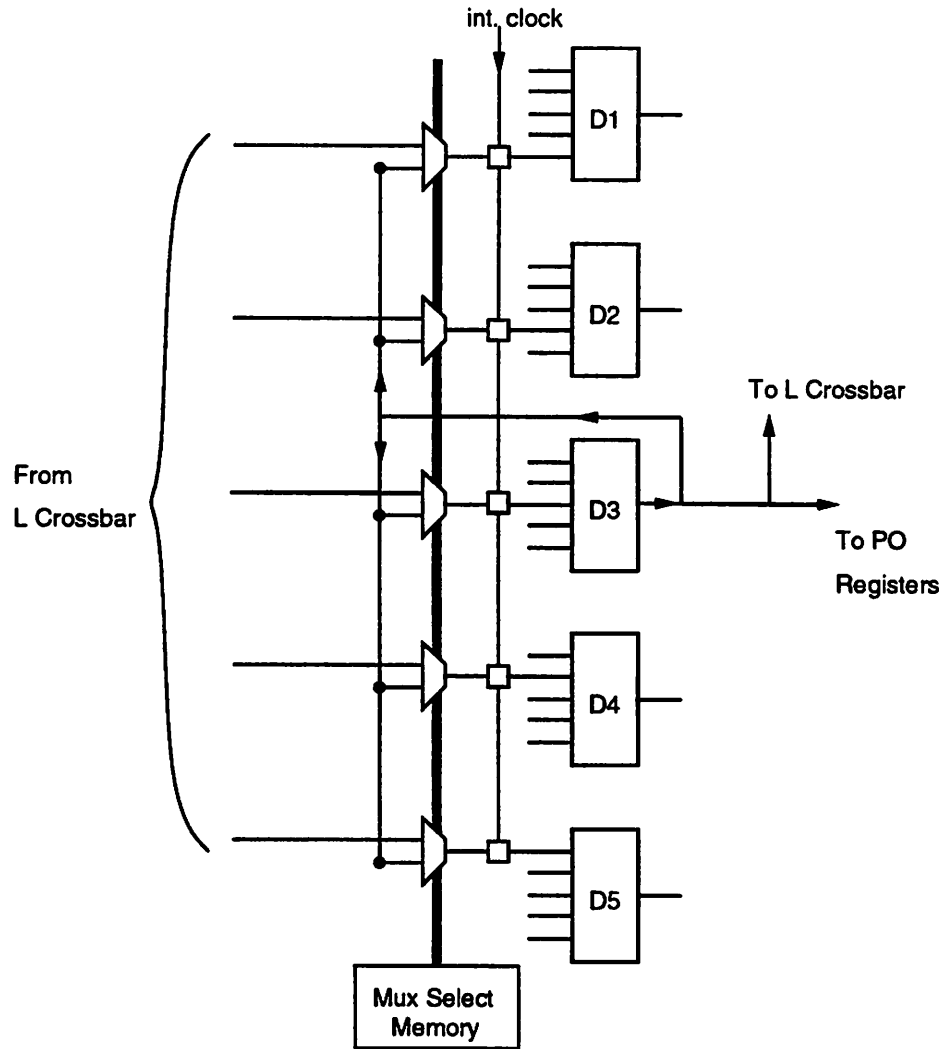


Figure 10.5: Hard-wire connections from a DLM to adjacent DLMs.

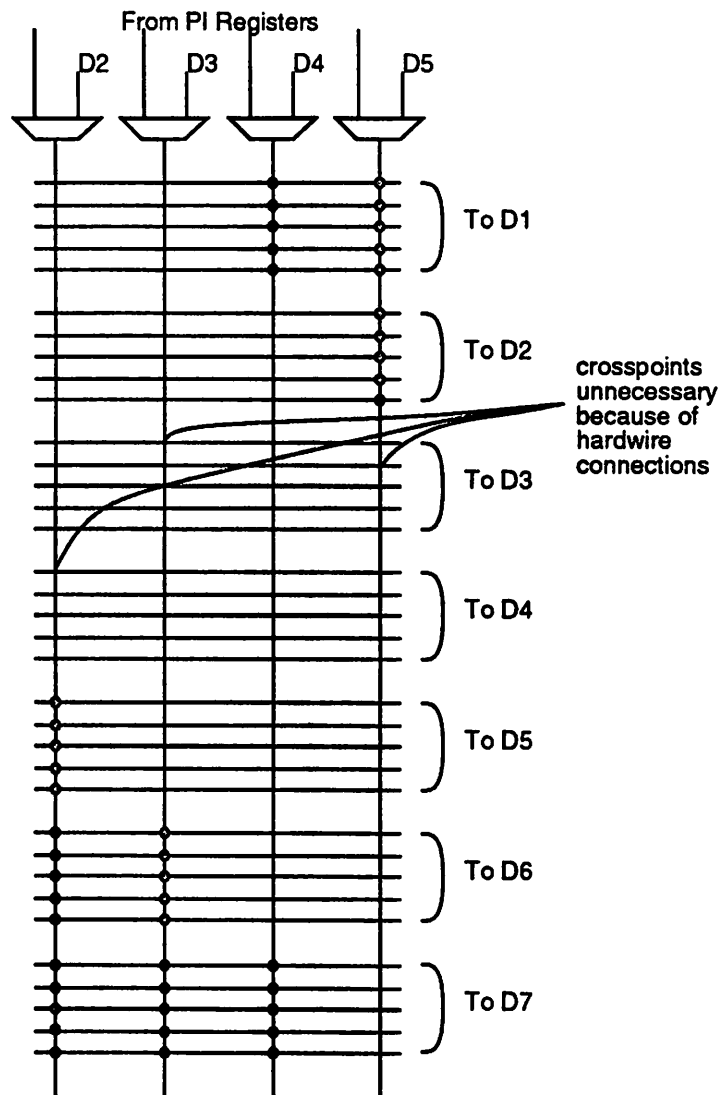


Figure 10.6: Hard-wire connections reduce *L*-crossbar crosspoints.

---

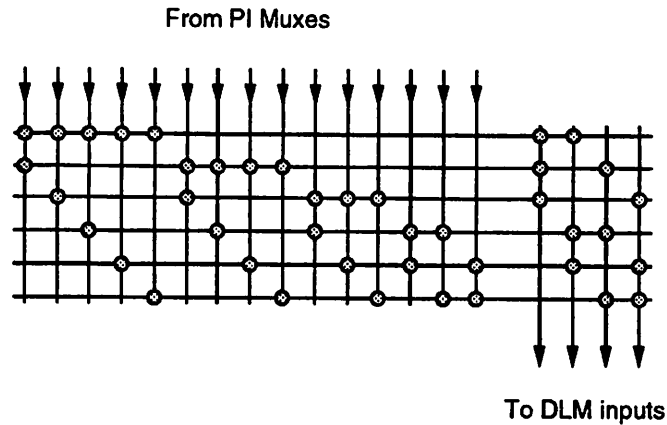


Figure 10.7: A 15-to-4 binomial concentrator

Figure 10.7 is a 15-to-4 binomial concentrator. This could be used as a part of the  $L$ -crossbar when  $K = 4$ . The crosspoints are distributed in such a manner that any four of the DLM and pass-buffer outputs can be connected to the DLM input lines.

### 10.2.5 DIA as a Copier

The above discussion has centered on using the crossbar to implement a real connection, i.e., the interconnect switch connects its inputs to its outputs (say through transistors). This type of connection is *required* in the case of analog signals, where the input signals can have different values. However, in our case we are dealing with digital signals, which take on only two values, either 0 or 1. So, the switch will still be functioning properly if its outputs are set to the value of the inputs they are supposed to be connected to. In essence, the switch is required to perform a one time copy of a selection of the inputs (those that are to be connected to the outputs) to the outputs. Can we take advantage of this fact to speed up the connection and reduce interconnect area? One answer to this question is delineated in the next paragraph.

Consider the case of just one switch output (DLM input). It can be connected to any one of the  $(B + C)$  signals (switch inputs) coming from the previous level. That is, its value should be set to the value of one of the  $(B + C)$  signals. This is equivalent to a memory read operation, where the memory consists of  $(B + C)$  bits, and one of them has

to be read onto the switch output. So, the selection can be performed by using precharge circuitry and sense amplifiers. The  $(B + C)$  signals correspond to one column of memory, and one of the cell select transistors is chosen by means of a decoder or local memory bit. There are both area and speed advantages in using this style of implementation. The cell select transistors are of minimum size as opposed to the large pass transistors that would have been required if a direct connection between switch input and output were required. And using the sense amp, the read operation can be performed fast. Further, the precharge time of the memory can be interleaved with the DLM propagation time, thus shortening the interconnect delay even further. This would mean that the interconnect delay would be of the same magnitude as an LUT based DLM (since the logic module delay is also a memory read operation). It could be even less, since the logic module is a  $2^K L$  bit memory (in the case of LUT based DLM) whereas the interconnect is a  $(B + C)$  bit memory<sup>3</sup>.

Figure 10.8 shows a section of the  $L$ -crossbar which uses this idea. The transistors shown in this figure can be much smaller (and are usually made of minimum dimensions) than those in Figure 10.4. The small squares sitting on the gate of the transistors represents the configuration bit at a given time instant (this can be presented at the gate using the shift register scheme, or the regular scheme of Figure 8.6). These are gated by the internal clock, so that they will be connected to the gate only after the vertical lines have a valid signal on them. All horizontal lines are precharged simultaneously. At the clock, the transistors with a '1' on their gate will try to equalize their respective horizontal line's voltage to that of their vertical line, and this causes a swing in the voltage on the horizontal line. The sense amplifiers detect the direction of the swing and set their output (going to the DLM inputs in Figure 10.8) to a '0' or '1' accordingly. This precharge/sense scheme requires that the voltage on the (highly capacitive) horizontal lines swing by only a tiny amount, which causes the propagation delay through the crossbar to be small. For the values shown in the figure,  $c$  gets connected to  $x$  and  $z$ , and  $b$  gets connected to  $y$ .

### 10.2.6 Sparsely connected $K$ -class DIA

So far, we only discussed fully connected switches, where it is always possible to connect any switch input to any switch output. These switches tend to be expensive in terms of area. It could be possible to decrease the area requirement of the DIA, by giving

---

<sup>3</sup>There are  $KC$  such  $(B + C)$  bit memories.

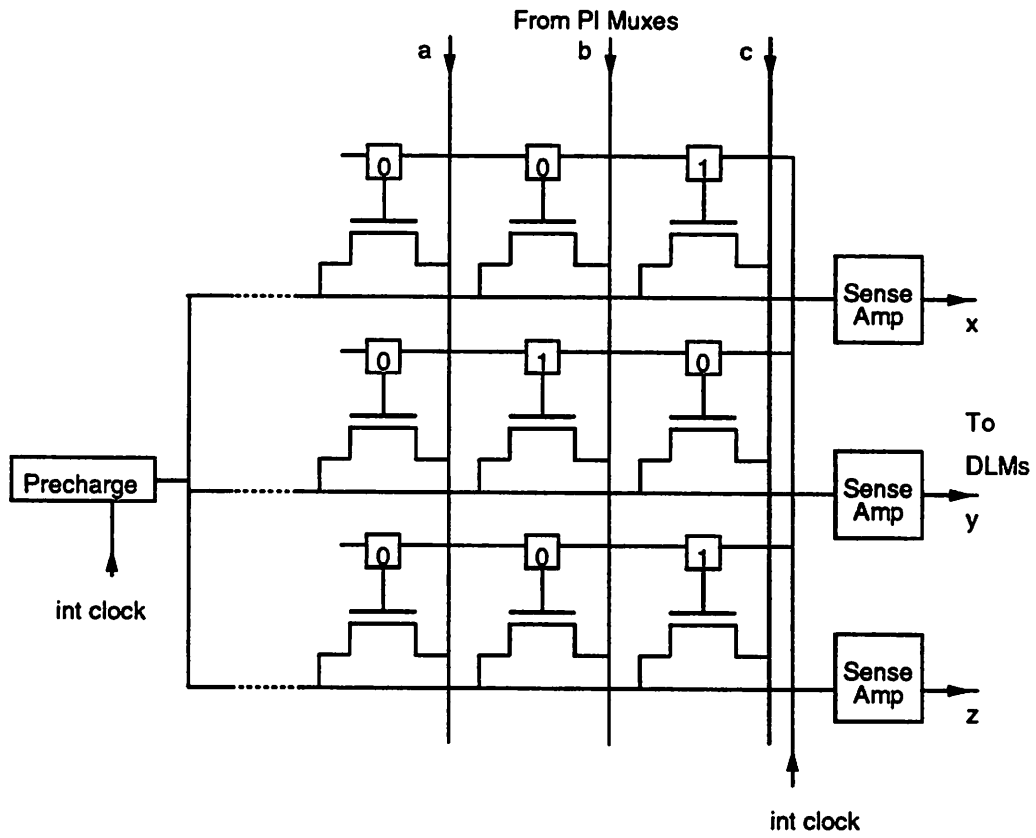


Figure 10.8: DIA as a *Copier*



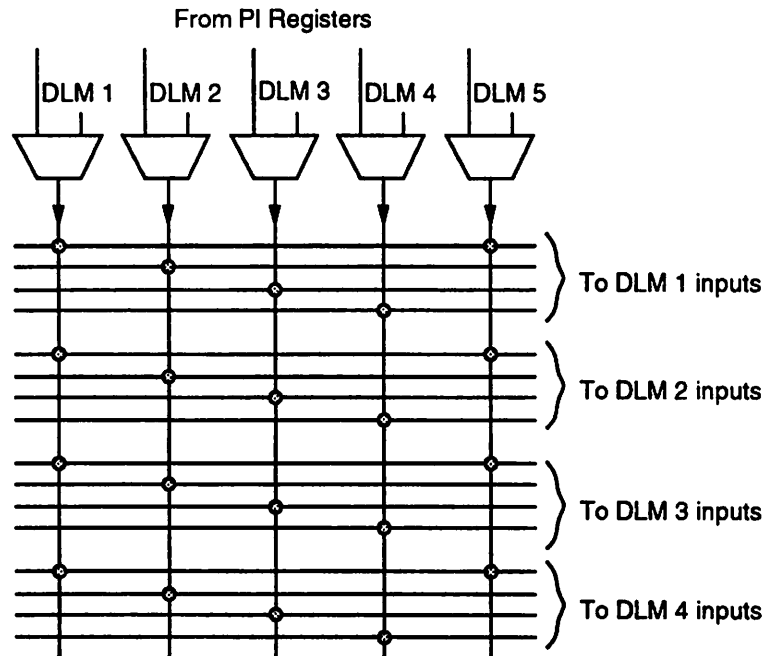


Figure 10.9: K-class DLMs reduce crosspoints

up the full connection flexibility. We discuss below a smart way of reducing the number of crosspoints, while maintaining a high probability of routing completion.

This scheme requires that the DLM have all inputs equivalent, as in the case of the LUT. Assume there are  $C$  DLMs, each with  $K$  inputs and 1-output. We divide the DLMs into  $K$  classes. Each class will have at most  $\lceil C/k \rceil$  and at least  $\lfloor C/K \rfloor$  modules. A module in class  $j$ ,  $j = 1 \dots K$ , is connected to the  $j^{\text{th}}$  input of each DLM (instead of to every input). This will decrease the number of crosspoints and memory bits by a factor of  $K$ . It has been found that for all the benchmark circuits tested, it is always possible to achieve complete connectivity using this connection strategy. Note that, we could still have some DLMs connected to every input, just in case it is not possible to complete the connections.

Figure 10.9 shows a portion of the  $L$ -crossbar which uses the sparsely connected  $K$ -class method. Here  $K = 4$ . The outputs of DLM 1 and 5 connect only to the first input lines of all DLMs; similarly output of DLM 2 connects only to the second input lines; and

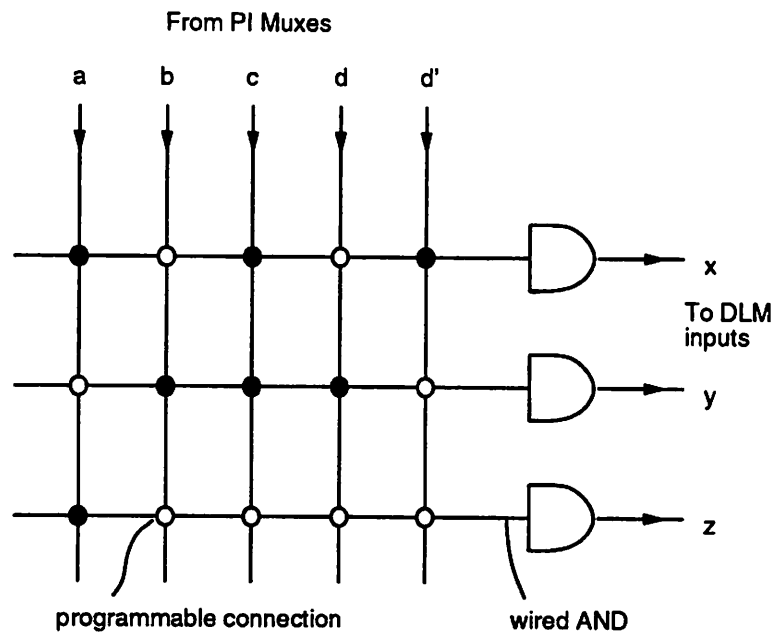


Figure 10.10: A DIA with wire-AND capability

so on.

### 10.2.7 Functional DIA <sup>4</sup>

In addition to performing a connection, it is possible to design the DIA such that a logic operation, such as AND or OR, can be performed together with the connection.

Figure 10.10 explains this concept. A portion of the  $L$ -crossbar is shown. The crosspoints are designed such that when more than one vertical line is connected to the horizontal line, an ANDing operation takes place. The connections (from vertical line to horizontal line) are programmable, as before. In the figure, a dark circle at the crosspoint represents a connection. Therefore,

$$x = acd'$$

$$y = bcd$$

$$z = a$$

<sup>4</sup>Dr. Kamal Chaudhary originated this idea.

Each wired AND gate can have  $\leq (B + C)$  inputs, thereby making this scheme very powerful. Since every logic function can be expressed in the sum-of-products form, the products (i.e., ANDing) can be formed in the DIA, and the sum (i.e., ORing) can be performed in the DLM. In other words, the DLMs can just be OR gates. As a result of the wired ANDing, it has been possible to reduce the number of levels in the circuit by 15-20% [Chaudhary 93].

To realize the full potential of this scheme, the signals may have to enter the crossbar in both true and complement form (eg.,  $d$  and  $d'$ ). This increases the size of the crossbar. Also, to achieve the wired AND, the number of transistors at each crosspoint will be doubled, compared to the normal case.

### 10.2.8 Technology dependent modifications

Like the DLM, the DIA is also easily suited to many different technologies. For example, if one were to use a DRAM technology, there would be considerable savings in area, without any complexity increase, since the DRAM refreshing could be synchronized with the level addressing. Like the DLM, a non-volatile memory technology (ROM) can also be used for the DIA. The shift register scheme can benefit in area if a technology such as a charge-coupled device (CCD) is used.

## 10.3 Level expander

Consider a feasible network  $\mathcal{N}$  (i.e., each node has  $\leq K$  inputs) with  $S$  nodes, and  $l$  levels. We wish to implement this circuit on a *Dharma* chip with  $C$  DLMs, each with  $K$  inputs, and  $L$  levels. If  $CL \leq S$  but  $l > L$ , then the chip has the resources (in terms of total logic modules, and interconnection configuration bits) to implement  $\mathcal{N}$ , but the architecture of Figure 8.6 needs to be modified to allow this. We call this modification the *Level Expander* circuitry. By using this modification, the number of levels that can be implemented in an  $L$  level chip can be raised to  $\Gamma$ ,  $\Gamma > L$ , without increasing the crossbar and DLM configuration memories (the PI Mux and Buffer Mux select memories will need to be enlarged).

### 10.3.1 Example

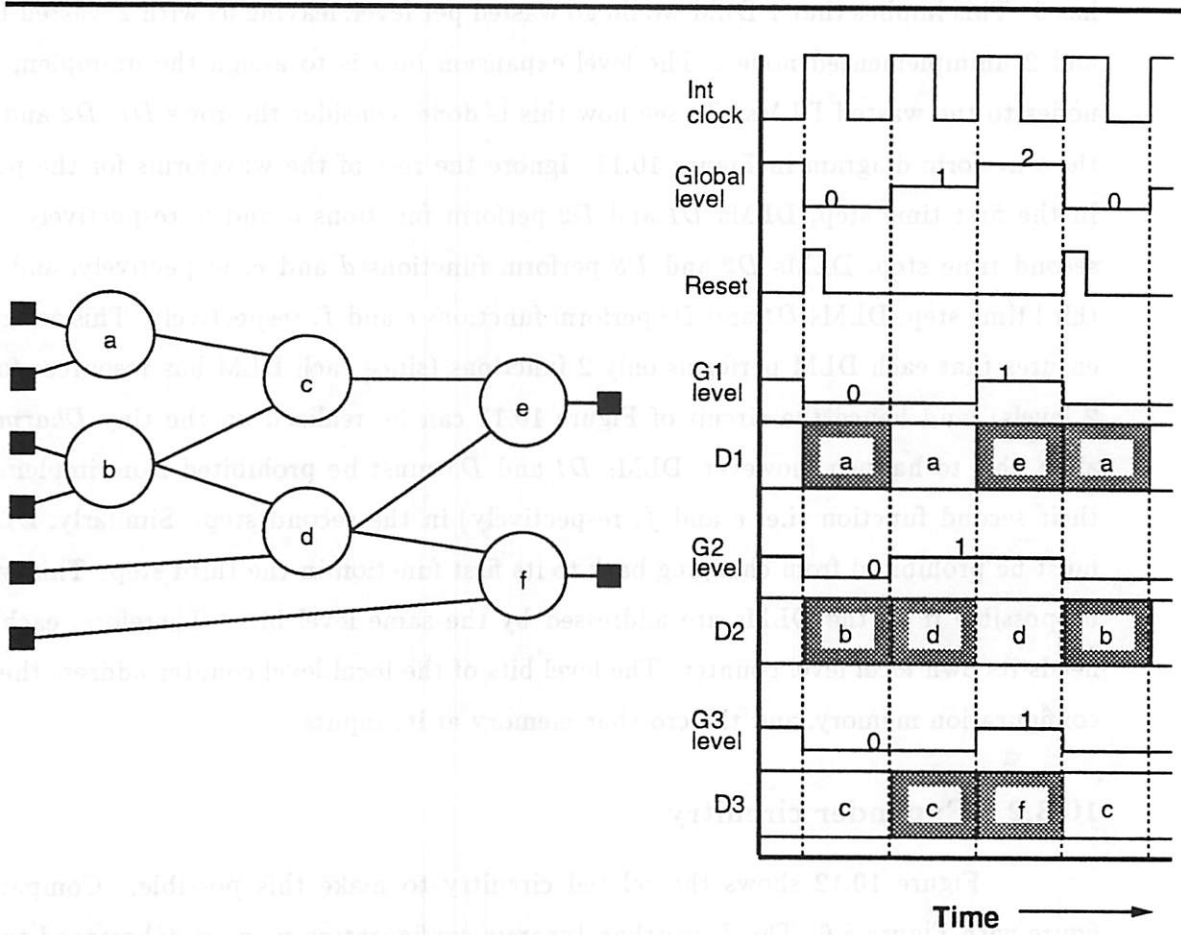


Figure 10.11: An example to illustrate level expansion

The principle of level expansion is best illustrated by means of a simple example. Let us consider a tiny *Dharma* which has 3 DLMS, and has  $L = 2$ . We wish to implement the DAG of Figure 10.11 on this tiny chip. The DAG has  $S = 6$  nodes, but  $l = 3$ . Therefore,  $CL = S = 6$ , and  $l > L$ . Without level expansion, it would not be possible to implement this DAG on the tiny *Dharma*. The circuit has only 2 nodes per level, although the chip has 3. This implies that 1 DLM would go wasted per level, leaving us with 2 wasted DLMS, and 2 unimplemented nodes. The level expansion idea is to assign the unimplementable nodes to the wasted DLMS. To see how this is done, consider the rows  $D1$ ,  $D2$  and  $D3$  of the waveform diagram in Figure 10.11. Ignore the rest of the waveforms for the present. In the first time step, DLMS  $D1$  and  $D2$  perform functions  $a$  and  $b$ , respectively. In the second time step, DLMS  $D2$  and  $D3$  perform functions  $d$  and  $c$ , respectively, and in the third time step, DLMS  $D1$  and  $D3$  perform functions  $e$  and  $f$ , respectively. This assignment ensures that each DLM performs only 2 functions (since each DLM has resources for only 2 levels), and hence the circuit of Figure 10.11 can be realized on the tiny *Dharma*. To allow this to happen, however, DLMS  $D1$  and  $D3$  must be prohibited from implementing their second function (i.e.  $e$  and  $f$ , respectively) in the second step. Similarly, DLM  $D2$  must be prohibited from changing back to its first function in the third step. This will not be possible if all the DLMS are addressed by the same level bits. Therefore, each DLM needs its own *local* level counter. The level bits of the local level counter address the DLM configuration memory, and the crossbar memory at its inputs.

### 10.3.2 Expander circuitry

Figure 10.12 shows the related circuitry to make this possible. Compare this figure with Figure 8.6. The  $L$ -crossbar dynamic configuration memory (shortened to *DCM* in Figure 10.12) has been partitioned into  $n$  parts, where  $n$  is the number of DLMS. Each partitioned memory ( $L1$ ,  $L2$ , ...) addresses the crosspoints at the inputs of a single DLM ( $D1$ ,  $D2$ , ..., respectively). Similarly, the  $B$ -crossbar DCM is also partitioned into  $m$  parts ( $B1$ ,  $B2$ , ...), where  $m$  is the number of pass-buffers. This partitioning scheme does not alter the number of memory bits, or routing resources. Each DLM and associated DCM are addressed by local level bits, generated by a *Local Level Generator*. Each buffer DCM is also similarly addressed. Hence there are a total of  $(n + m)$  local level generators. The *Level generation circuitry* block of Figure 8.6 now functions as a *global* level generator. The level

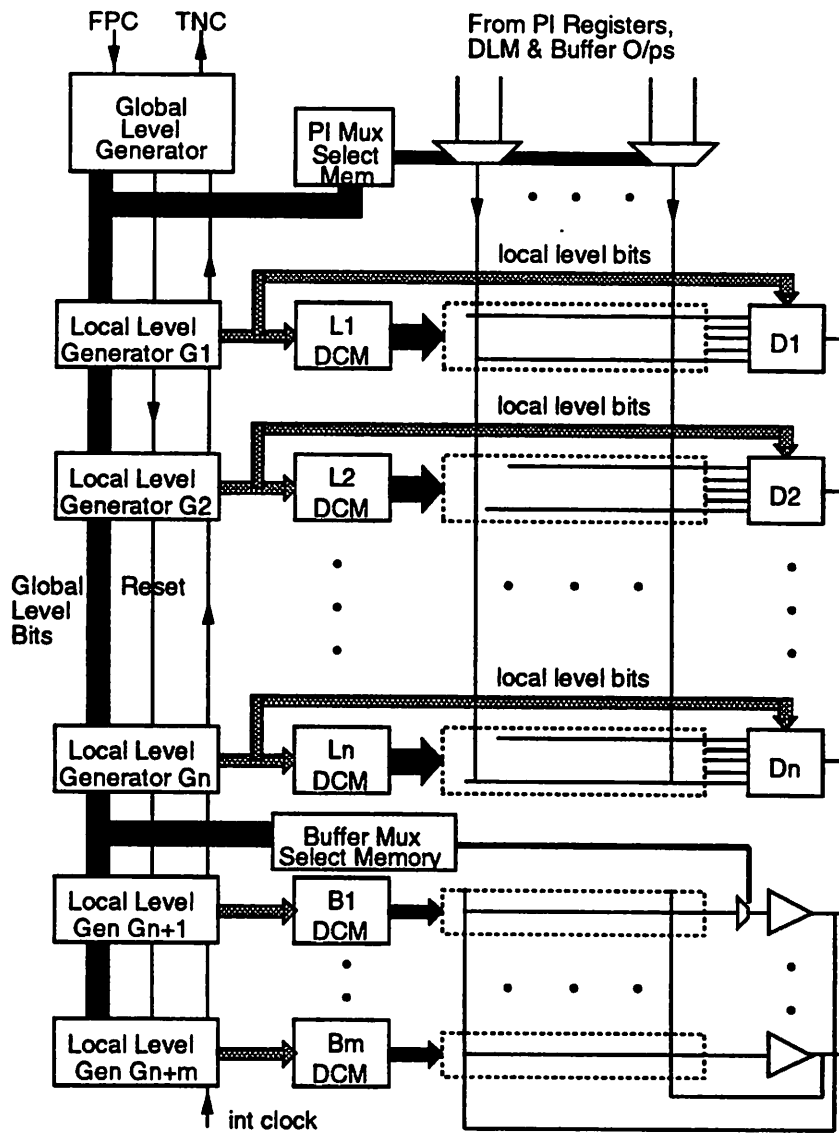


Figure 10.12: Modified *Dharma* with level expansion capability

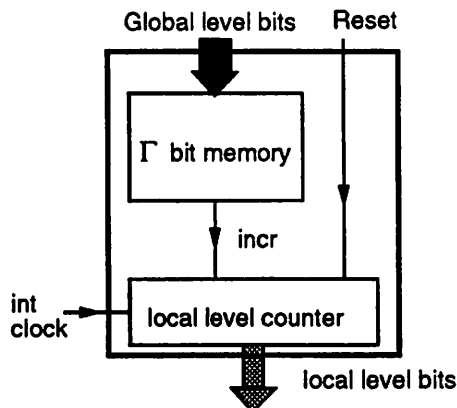


Figure 10.13: The *Local Level Generation* circuitry

bits generated by it are termed global level bits. Let  $\Gamma$  be the maximum number of circuit levels that can be implemented on the chip. Then, the global level counter (i.e., the level counter of Figure 8.11) is a divide-by- $\Gamma$  counter. The local level generators are addressed by the global level bits. The *Reset* signal of Figure 8.11 is now made globally available to all the local level generation blocks.

The multiplexer select memories (PI Mux select and Buffer Mux select) are addressed directly by the global level bits, and these have to be enlarged to  $\Gamma$  bits each. Since each mux is individually selected, the local level generation scheme will increase the number of bits used, rather than provide a saving, and hence the original addressing scheme is better in this case.

Not shown in Figure 10.12 are the input latches. These have to be clocked only when the DLM function is to change in the next clock, and hence their clock has to be gated by the *incr* bit of their respective local level generator.

### 10.3.3 Local level generator

All the local level generators are identical. Figure 10.13 shows a detailed diagram of the circuitry. Each local level counter is a divide-by- $L$  counter. Each local level generator has  $\Gamma$  bits of control memory that is addressed by the global level bits. One bit of memory is output every clock cycle (labeled *incr* in the figure). The local level counter is incremented

at the clock, when *incr* is asserted. Going back to Figure 10.11, the global level generation circuitry broadcasts a reset signal every 3 cycles, since the circuit being implemented has 3 levels. The local level counters in *G1* and *G3* hold their count at 0 in the second step, and the local level counter in *G2* holds its count at 1 in the third step. The *incr* bit is de-asserted in their respective control memories to make this pause in the count possible. Note that the function *c* computed by DLM *D3* in step 1 (which gives an erroneous value for *c*, since *a* and *b* are not yet ready) is ignored, since *c* is computed again in step 2. Although Figure 10.12 and Figure 10.13 may imply that the  $\Gamma$  bit control memories are distributed, in actual implementation, it can be single block of memory with  $(m + n)$  bit outputs going to the various local level generators.



The first part of the chapter deals with the general theory of variations and modifications. It is shown that the general theory of variations and modifications is a special case of the general theory of variations and modifications. The second part of the chapter deals with the general theory of variations and modifications. It is shown that the general theory of variations and modifications is a special case of the general theory of variations and modifications. The third part of the chapter deals with the general theory of variations and modifications. It is shown that the general theory of variations and modifications is a special case of the general theory of variations and modifications.

## Chapter 11

# Synthesis for Dharma

This chapter deals with mapping designs onto *Dharma*. We consider only the basic architecture of Chapter 8, and assume that the DLM is a  $K$ -input, 1-output LUT. The first section gives an introduction to what is required in a synthesis tool for *Dharma*. We look at one aspect of the synthesis, namely, the *temporal partitioning*<sup>1</sup> problem, where we assign nodes in the input network to levels (i.e., we assign a time slot to each). This problem can be formulated as an integer programming problem; the second section shows how that is so. However, we do not use the integer programming problem to do the level allocation – instead we solve it by using a heuristic algorithm; that is the subject of the third section. The last section presents results for the heuristic approach.

### 11.1 Introduction

Let us consider a *Dharma* chip with  $L$  levels and  $C$  DLMs. There are two aspects to the synthesis for *Dharma*. On one hand we need to generate a circuit such that the number of levels in the circuit  $l$  is minimized, since the worst case delay is proportional to  $l$  (see Chapter 12). On the other hand, the maximum number of nodes in any level,  $G$ , should be such that  $G \leq C$ , otherwise the circuit cannot be implemented on a single chip.

We tackle this problem in a two-step manner.

1. Given an input circuit specification, transform it into a DAG such that the number of inputs at every node  $\leq K$ , and the maximum number of topological levels  $l$  is minimized.

---

<sup>1</sup>Credit goes to Dr. Rajiv Shah for coining this term.

2. Allocate levels to the nodes obtained in Step 1, in a manner such that  $G \leq C$  (*temporal partitioning*).

Techniques for doing Step 1 are already known, for the case when the DLM is a LUT (see [Cong 92, Murgai 91a, Francis 91a] and Chapter 5). Hence we concentrate only on step 2. Note that it may not always be possible to satisfy the condition that  $G < C$  in Step 2. If so, we allow  $l$  to be increased. In Chapter 8, it was suggested that the input DAG be topologically leveled to allocate nodes to DLMs. Topological levelization is one method of doing the level allocation, but there are other methods too. From the requirements of step 2, topological levelization may not give a satisfactory result, since the levelization process does not consider the  $G < C$  constraint. Hence, the need for a different levelization procedure.

Before discussing the details of the new leveling scheme, we first define what exactly we mean by topological levelization.

### 11.1.1 Topological levelization

There are two ways of doing topological levelization.

Starting from the primary inputs, the levelization proceeds in the following manner.

Primary inputs are assigned level 0.

Level of a node = MAX(level of its fanins) + 1

Figure 11.1(a) shows level allocation using this algorithm for a very simple DAG example (although not shown in the figure, the edges are directed, and are from left to right).

Levelization from the primary outputs is similar, and is described below (for a DAG with  $M$  levels). Starting from the primary outputs, the levelization goes as follows.

Primary outputs are assigned level  $M$ .

Level of a node = MIN(level of its fanouts) - 1

Figure 11.1(b) shows the level allocation using this algorithm for the same DAG of Figure 11.1(a). Note that it is possible, and is usually the case, that the two level assignments will be different. Only nodes on the maximum path (i.e., a path from input to output with maximum length, length measured in terms of number of nodes) will have the same level assigned by both schemes. In the figure, nodes  $c$  and  $d$  are assigned to different levels by the two schemes. We call these nodes *slack nodes*, since there is some slack allowed in their level assignment. Slack nodes cannot lie on maximum paths.

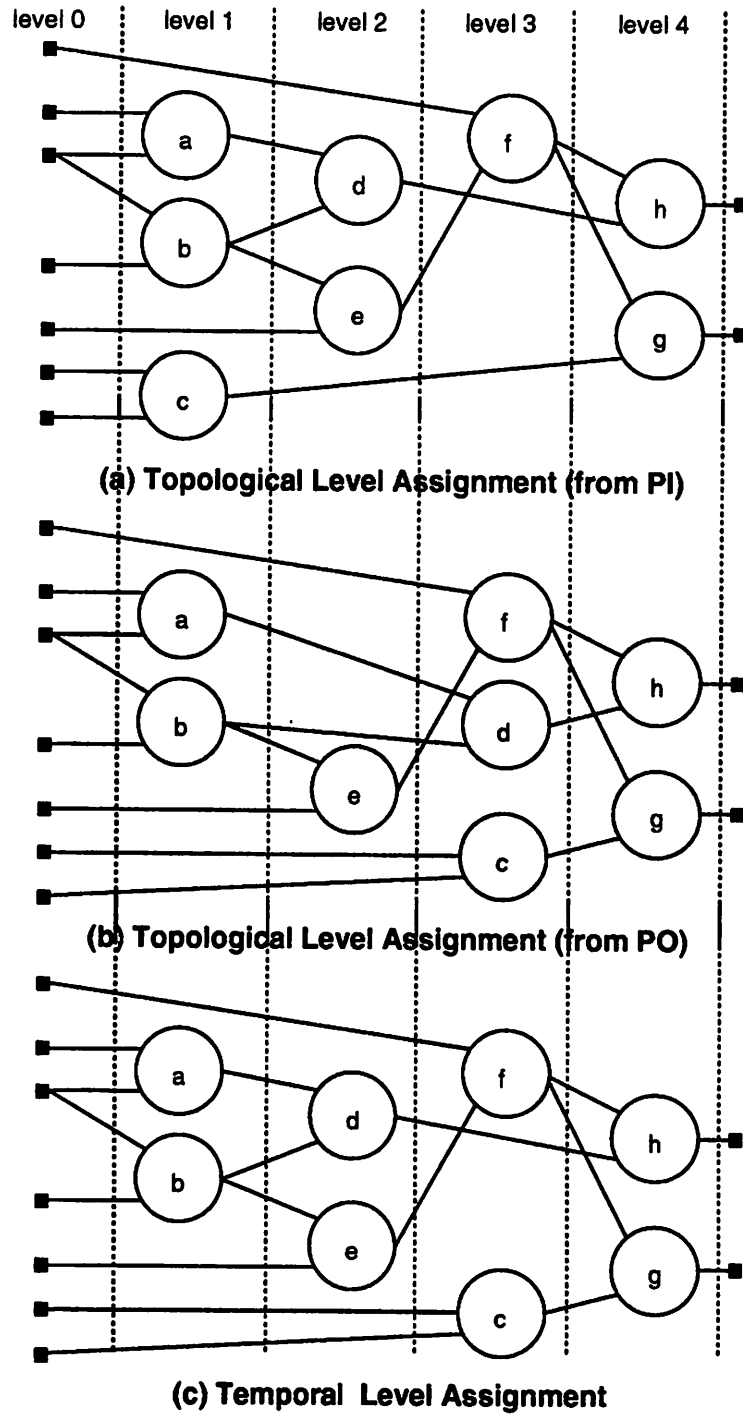


Figure 11.1: Different levelization techniques

### 11.1.2 Temporal partitioning

We need to determine which node functions can be evaluated in parallel, and at what time instant. That is, the nodes have to be partitioned in *time*. Hence the name temporal partitioning. More simply, each node has to be allocated a level, which could be different from its topological level. This cannot be solved by simple topological levelization always, as will be apparent from the discussion below.

Assume that the logic network is a feasible network (i.e., each node in the network has  $\leq K$  inputs). Let the combinational part of the feasible network be topologically levelized. Let  $l$  be the number of levels,  $S$  the total number of modules and  $G$  the maximum number of modules in any level. Note that  $G$  is dependent on the levelization scheme used. Let  $t_l(n)$  be the topologically assigned level of node  $n$ , and let  $a_l(n)$  be its allocated level, which will be assigned by the temporal partitioner.

If  $l \leq L$  and if the topological levelization has  $G \leq C$ , then there is no need for temporal partitioning, since each node can be assigned a level that is the same as its topologically assigned level, i.e.,  $a_l(n) = t_l(n), \forall n$ .

If, however  $l \leq L$ , and topological levelization has  $G > C$ , with  $S \leq CL$ , it may still be possible to implement this circuit on *Dharma* by a judicious choice of levels for the *slack nodes*. These nodes can be assigned a new level (different from their topological level); i.e., they are *moved* in *time*.

Consider the DAG of Figure 11.1. Assume it is to be implemented on a tiny *Dharma* chip which has  $C = 2$  and  $L = 3$ . The DAG has  $S = 6$  and  $l = 3$ . The topological level assignment from primary inputs (Figure 11.1(a)) has  $G = 3$  (level 1 has three nodes  $a$ ,  $b$  and  $c$ ), and the topological level assignment from primary outputs also has  $G = 3$  (level 3 has three nodes  $c$ ,  $d$  and  $f$ ). This may suggest that it is not possible to implement this DAG on the tiny *Dharma*. However, the level assignment in Figure 11.1(c) has  $G = 2$ , and this implies that it is indeed possible to implement the DAG on our tiny *Dharma*. Note that the levelization of Figure 11.1(c) can be derived from Figure 11.1(a), by moving  $c$  alone to level 3, and leaving the rest of the nodes unaltered. Similarly, we can derive Figure 11.1(c) from Figure 11.1(b), by moving node  $d$  to level 2, and leaving the other nodes unchanged.

The process of identifying nodes to be moved, and moving them appropriately in an effort to keep  $G$  within  $C$ , is what we call temporal partitioning.

### 11.1.3 Handling other designs

If  $l > L$ , with  $S \leq CL$ , then it may still be possible to implement the circuit on *Dharma* by grouping together modules across several levels. Each such group can then be realized in more than one internal cycle, before switching over to the next group. This has been discussed at length in Section 10.3. Techniques to choose which nodes can share a level, in such cases, need to be developed, and are not discussed here.

Note that if  $S \leq CL$ , and  $G > C$ , even after temporal partitioning, then it may still be possible to realize the design within a single chip, at the expense of increasing the number of levels.

When we deal with designs that do not fit on a single chip,  $S > CL$ , we will have to split it across several chips. In such cases, if  $G < C$  (for some levelization scheme), then the multi-chip interconnection scheme of Section 8.2.4 can be used to realize the design. If, however, the design is such that  $G > C$ , in spite of temporal partitioning, then splitting across chips could necessitate duplication of logic.

## 11.2 Levelization as an integer linear program

This section described how the level allocation problem can be formulated as an integer linear program (ILP). The objective is set as minimizing the maximum of the number of modules per level,  $G$ , without violating the fanin and fanout constraints of any node. This objective may seem counter-intuitive, since the constraint imposed by *Dharma* is to have

$$\#(d) \leq C, \quad \forall d \tag{11.1}$$

$\#(d)$  is the number of modules in level  $d$ . However, if the ILP's solution is  $< C$ , then the level allocation solution will automatically satisfy 11.1 since  $\#(d) \leq G, \forall d$ , by the definition of  $G$ . If however, the ILP's solution is  $> C$ , then it implies that it is impossible to satisfy 11.1 without increasing the number of levels.

We consider the level assignment for *slack nodes* only, since the levels of other nodes cannot be changed.

### 11.2.1 Upper and lower level limits

Let  $l_i$  denote the level assigned to node  $i$ , using the topological level assignment scheme, starting at the primary inputs. Then,  $l_i$  specifies the lowermost level that node  $i$  can be assigned to. Similarly, let  $u_i$  denote the level assigned to node  $i$ , using the topological level assignment scheme, starting at the primary outputs.  $u_i$  specifies the uppermost level for node  $i$ .

### 11.2.2 Level variable

We define a variable  $x_{ij}$ , to be associated with node  $i$ . The  $j$  index corresponds to the assignment of node  $i$  to level  $j$ .  $x_{ij}$  is allowed to take only one of two values, 0, 1. If  $x_{is}$  takes the value 1, then it means that node  $i$  is assigned to level  $s$ . Since a node can be assigned to only one level, only one of the  $l$   $x_{ij}$ 's can take a value 1. Therefore, for each node  $i$ , we have the constraints that

$$\sum_{j=1}^l x_{ij} = 1$$

Also, since node  $i$  cannot be assigned a level below  $l_i$ ,

$$x_{ij} = 0, \quad 0 < j < l_i.$$

Similarly, node  $i$  cannot be assigned a level above  $u_i$ . Therefore,

$$x_{ij} = 0, \quad u_i < j \leq l.$$

Combining the above three equations, we obtain,

$$\sum_{j=l_i}^{u_i} x_{ij} = 1 \tag{11.2}$$

and

$$\sum_{j=1}^{l_i} x_{ij} + \sum_{j=u_i+1}^l x_{ij} = 0 \tag{11.3}$$

### 11.2.3 Level of a node

If  $a_l(i)$  denotes the level allocated to node  $i$ , then the following equation expresses  $a_l(i)$  in terms of  $x_{ij}$ .

$$a_l(i) = \sum_{j=1}^l j x_{ij} \tag{11.4}$$

This follows directly from the definition of  $x_{ij}$ .

### 11.2.4 Fanin constraints

The circuit specification imposes fanin and fanout constraints. If node  $i$  has a fanin  $k$ , then the level assigned to node  $i$  should always exceed the level assigned to  $k$  by at least 1. Similarly, if node  $i$  has fanout  $f$ , then the level assigned to node  $f$  should always exceed the level assigned to  $i$  by at least 1.

However, if we specify the fanin constraints for every node, the fanout constraints become redundant, since  $i$ 's fanout constraint will be covered by  $f$ 's fanin constraint. Hence, we need to look at only the fanin constraints <sup>2</sup>.

So, for node  $i$ , with fanin  $k$ ,

$$a_l(i) \geq a_l(k) + 1$$

Using equation 11.4, and re-arranging the terms,

$$\sum_{j=1}^l j(x_{ij} - x_{kj}) \geq 1.$$

Since  $x_{ij}$  and  $x_{kj}$  are both 0 for  $0 < j < l_k$  and  $u_i < j \leq l$ , because of the upper-limit and lower-limit constraints,

$$\sum_{j=l_k}^{u_i} j(x_{ij} - x_{kj}) \geq 1. \quad (11.5)$$

### 11.2.5 Nodes per level

At each level  $j$ , the number of nodes must be less than  $G$ . This is expressed as.

$$\sum_{i=1}^n x_{ij} \leq G, \quad (11.6)$$

where  $n$  is the number of nodes in the circuit.

### 11.2.6 Complete ILP

Using equations 11.2, 11.3, 11.5 and 11.6, the complete integer linear program is as follows.

---

<sup>2</sup>If the fanout node is not a *slack node*, the constraint imposed by it is the same as that imposed by the  $u_i$  constraint.



$$\begin{aligned}
& \text{minimize } G \\
& \text{Subject to:} \\
& \sum_{j=l_i}^{u_i} x_{ij} = 1, \quad \forall i \\
& \sum_{j=1}^{l_i} x_{ij} + \sum_{j=u_i+1}^l x_{ij} = 0, \quad \forall i \\
& \sum_{i=1}^n x_{ij} \leq G, \quad \forall j \\
& \sum_{j=l_k}^{u_i} j(x_{ij} - x_{kj}) \geq 1, \quad \forall k, k \in \text{fanin}(i) \\
& x_{ij} \geq 0 \\
& x_{ij} \leq 1
\end{aligned}$$

### 11.3 Temporal partitioning

In the previous section, we showed how to formulate the level allocation problem as an integer linear program. In this section, we present a simple heuristic for the level allocation problem. This heuristic is very fast and effective, and can handle other constraints (such as pass-buffer minimization) in addition to Constraint 11.1. Experimental results are presented in the next section.

#### 11.3.1 Uniform DLM distribution

As before, let there be  $C$  DLMs. Let the input circuit (level minimized) have  $l$  levels and  $S$  modules. Then, the best possible way to distribute the modules is to have at most  $\lceil \frac{S}{l} \rceil$  modules per level. If such a module distribution can be obtained, then there will be a maximum utilization of chip resources. It may not be possible to achieve this kind of a distribution for all circuits, because of the constraints imposed by the fanin, fanout and non-slack nodes; however, it is something that is to be aimed for.

We choose this distribution as our objective. That is, given an input circuit, allocate the modules to levels in a manner such that the number of modules in a level  $i$ ,  $\#(i)$ , is as close as possible to the number of modules in another level  $j$ , for all values of  $i$  and  $j$ . This is to say that we wish to have a uniform distribution of DLMs across the levels. Mathematically, our objective is to minimize the variance of the DLM distribution.

### 11.3.2 Modified Kernighan-Lin heuristic

We try to achieve a uniform distribution of DLMs, by using a modified version of the Kernighan-Lin heuristic [Kernighan 70]. Our approach resembles the scheduling techniques used in high-level and data-path synthesis [Park 91], although our constraints and objectives are slightly different.

Kernighan and Lin proposed an algorithm to solve the graph bi-partitioning problem, i.e., given a partition, find two partitions  $A$  and  $B$  such that the number of edge-crossings are minimum. The Kernighan-Lin (KL) heuristic is based on an iterative improvement scheme. Starting from an arbitrary initial partition  $A$  and  $B$ , the algorithm seeks to find a group of elements in  $A$  to be swapped with a group of elements in  $B$ , such that the resulting partitions are optimal. In each iteration of the KL algorithm, a set of movements are selected and carried out, and the next iteration is performed upon the result of the previous iteration. The movements that give the best possible gain improvement (even if negative) in a particular iteration, are the movements that are carried out in that iteration. Allowing negative gain improvements (in other words, an inferior solution) gives the algorithm capability to climb out of local minima. The cumulative gain improvement (sum of the gains in each iteration) is tracked. This gain starts at zero, at the first iteration, and returns to zero at the last iteration (when all elements have changed placed). In between, it will have reached a peak. The sequence of movements up to this peak are retained, and the rest are discarded. The resulting partition is then set as the initial partition, and the entire procedure is repeated until a stage is reached when the initial partition can no longer be improved (this typically takes 2 to 3 iterations in most cases).

In our case, we are performing an  $l$ -way partitioning, and the objective is to have uniformly sized partitions. We start from an initial level allocation that satisfies the fanin constraints. Such an allocation can be obtained using topological levelization. Assume we use topological levelization starting from primary outputs. All the slack nodes are identified and formed into an *active set*. Only members of the active set are considered for movement. A move consists of moving a node from its initial level to a new level, without violating fanin or fanout constraints. The cost function of the move is set to minimize the variance of the DLM distribution.

Figure 11.2 outlines our procedure in  $C$ -like pseudo-code. Variable  $ix$  is the move-index, and is incremented after every move. The variable  $\mathcal{G}$  keeps track of the peak cu-

```
repeat{
  ix = 1;
  G = 0; /* peak cumulative gain */
  I = 0; /* index corresponding to peak cumulative gain */
  cum_gain = 0; /* cumulative gain */
  Build an active_set;
  while (active_set not null) {
    Select node i with maximum move_gain;
    Move node i;
    Remove node i from active_set;
    cum_gain = cum_gain + move_gain;
    if cum_gain > G {
      G = cum_gain;
      I = ix;
    }
    Increment ix;
  }
  Only treat the first I moves, from 1 ...I,
  as valid, and undo the rest of the moves.
} until (G ≤ 0)
```

Figure 11.2: Temporal partitioning procedure

---

mulative gain, and  $\mathcal{I}$  stores the move-index corresponding to this gain. To start with, an *active\_set* is created. This comprises of the slack nodes. While there are nodes in this set, a node whose move that maximizes the gain is chosen. It is moved, and then removed from the *active\_set*. The cumulative gain is incremented by the gain of this move. If the resulting gain surpasses its previous peak value,  $\mathcal{G}$  and  $\mathcal{I}$  are updated. After all the nodes have been moved, the inner **while** loop terminates. Only the sequence of moves from index 1 to  $\mathcal{I}$  are valid, and the other moves are discarded. The procedure within the outer **repeat** loop is repeated until the moves do not cause any further improvements.

### 11.3.3 Move selection

The criterion for move selection is to minimize the variation of the DLM distribution. Let the current distribution be such that there are  $d_i$  DLMs allocated to level  $i$ , for  $i = 1 \dots l$ . Let us consider the cost of moving DLM  $m$  from its current level *cur* to a new level *new*. Let the number of modules currently in level *cur* be  $c$ , i.e.,  $d_{cur} = c$ . Similarly, let  $d_{new} = n$ . If this move is made, then  $d_{cur} = c - 1$  and  $d_{new} = n + 1$ . Let  $\mu$  represent the mean value of the DLM distribution, i.e.,  $\mu = S/l$ . If  $v_{cur}$  and  $v_{new}$  represent the current and new variances, then,

$$\begin{aligned}
 v_{cur} &= \sum_{i=1}^l [d_i - \mu]^2 \\
 &= [d_{cur} - \mu]^2 + [d_{new} - \mu]^2 + \sum_{i=1, i \neq new, cur}^l [d_i - \mu]^2 \\
 &= [c - \mu]^2 + [n - \mu]^2 + \sum_{i=1, i \neq new, cur}^l [d_i - \mu]^2 \tag{11.7}
 \end{aligned}$$

Similarly,

$$v_{new} = [(c - 1) - \mu]^2 + [(n + 1) - \mu]^2 + \sum_{i=1, i \neq new, cur}^l [d_i - \mu]^2 \tag{11.8}$$

$v_{cur}$  and  $v_{new}$  are non-negative. Hence, if we define  $\mathcal{D}_v$  as

$$\mathcal{D}_v = v_{cur} - v_{new}, \tag{11.9}$$

then,  $\mathcal{D}_v > 0$  whenever  $v_{new}$  is less than  $v_{cur}$ . Since we wish to minimize the variance, we choose a move that maximizes  $\mathcal{D}_v$ . Using equations 11.7 and 11.8 in 11.9,

$$\mathcal{D}_v = [c - \mu]^2 + [n - \mu]^2 - [(c - 1) - \mu]^2 - [(n + 1) - \mu]^2$$

$$\begin{aligned}
&= c^2 - 2c\mu + n^2 - 2n\mu - (c-1)^2 + 2(c-1)\mu - (n+1)^2 + 2(n+1)\mu \\
&= c^2 + n^2 - 2\mu[c+n - (c-1) - (n+1)] - c^2 + 2c + 1 - n^2 - 2n - 1 \\
&= 2c - 2n - 2 \\
&= 2(c - n - 1) \\
&= 2(d_{cur} - d_{new} - 1)
\end{aligned}$$

From this result, we observe that the variance minimizing move is independent of  $\mu$ , and is dependent only on  $d_{cur}$  and  $d_{new}$ . Figure 11.3 outlines our move procedure in a *C*-like pseudo-code. This procedure is called to perform the function of the first line within the inner while loop of Figure 11.2. The procedure loops through all nodes in *active\_set*. For each node  $n$ , the gain is computed using the above formula (the multiply by 2 can be dropped, since we are interested in computing only the maximum, both for the *move\_gain* and the *cum\_gain* (Figure 11.2)). All the possible levels to which the node can be moved to, are considered, and the gain is computed for each level. Since a node's level has to be greater than all its fanins, levels below  $(i_{max}(n) + 1)$  are not considered.  $i_{max}(n)$  is the level of the fanin of  $n$  with the maximum level, i.e.,

$$i_{max}(n) = \max_{f \in fanin(n)} \{a_l(f)\}.$$

Similarly, a node's level has to be lesser than all its fanouts. Hence levels above  $(o_{min}(n) - 1)$  are not considered, where

$$o_{min}(n) = \min_{f \in fanout(n)} \{a_l(f)\}.$$

The procedure returns the node with the maximum move gain, and its new level position.

## 11.4 Experimental results

Table 11.1 tabulates the results of using procedures 11.2 and 11.3 on a set of MCNC benchmark examples. The examples were first mapped to 5 input LUTs, with minimum levels, using *mis\_pga*[Murgai 91a]<sup>3</sup>.

The second column lists the total number of modules,  $C$ , in the example, and the third column lists the number of logic levels,  $l$ , along the longest path. The  $[C/l]$  column is

---

<sup>3</sup>Any other mapper can be used, like [Cong 92] or *dpmap* of Chapter 5. These experiments were run before *dpmap* was discovered. Since the aim is to study the level distribution, it was felt unnecessary to re-run using other mappers.

---

```

/* Input is the active_set */
/* Outputs are the node to move, its new level and the move's gain */
Move_select() {
     $\mathcal{M}_g = 0$ ; /* peak move_gain */
    foreach node  $n \in active\_set$ {
         $cur = a_l(n)$ ; /*  $n$ 's current level */
        /*  $i_{max}(n)$  is the max fanin level */
        /*  $o_{min}(n)$  is the min fanout level */
        for ( $new = (i_{max}(n) + 1) \dots (o_{max}(n) - 1), new \neq cur$ ) {
            /* consider all possible new levels */
             $move\_gain = d_{cur} - d_{new} - 1$ ;
            if ( $move\_gain > \mathcal{M}_g$ ) {
                 $\mathcal{M}_g = move\_gain$ ;
                 $\mathcal{D} = n$ ; /* store module to move */
                 $\mathcal{N} = new$ ; /* store new level */
            }
        }
    }
    Return  $\mathcal{M}_g$ ,  $\mathcal{D}$  and  $\mathcal{N}$ .
}

```

Figure 11.3: Move selection procedure

to give an idea about the number of modules that would be in each level, if the distribution were perfectly uniform. We start from a topologically leveled graph, leveling from the primary outputs. The columns under the 'Before' heading correspond to the distribution statistics for this type of levelization. 'Min' is the minimum of the DLMs in any level, and 'Max' is the maximum number of DLMs in any level.  $\sigma$  is the standard deviation of the distribution. The columns under 'After' correspond to similar statistics after temporal partitioning. The last column records the time taken, in CPU seconds, on a DEC 5100, to run the algorithm.

The results are tabulated in ascending order of standard deviation values of the DLM distributions after temporal partitioning. The algorithm is quite successful, and gives phenomenal improvements in several examples (*des*, *rot*, *apex6*, *C5315*). Figure 11.4 shows the results for the *C5315* example, in graphical form.

A few of the examples, however, like *C499* and *alu2*, cannot be partitioned successfully. In such cases, the number of levels may have to be increased to get a more uniform distribution (at the expense of slowing down the circuit), or logic synthesis techniques may be required to synthesize the circuit in a manner that is more suitable for *Dharma* (see Chapter 13).

## 11.5 Conclusions

In this chapter, we first described an integer linear programming approach to the level allocation problem, and then developed a heuristic for level allocation. The heuristic is called temporal partitioning. Temporal partitioning uses a cost function similar to the Kernighan-Lin heuristic. Experimental results show that the heuristic can yield a fairly uniform DLM distribution for several MCNC benchmark examples.

The move selection procedure (see Figure 11.3) can be modified to take into consideration number of buffers per level, in addition to the number of DLMs per level. Any other types of architectural constraints can also be easily incorporated. This flexibility is the main advantage of the temporal partitioning heuristic.

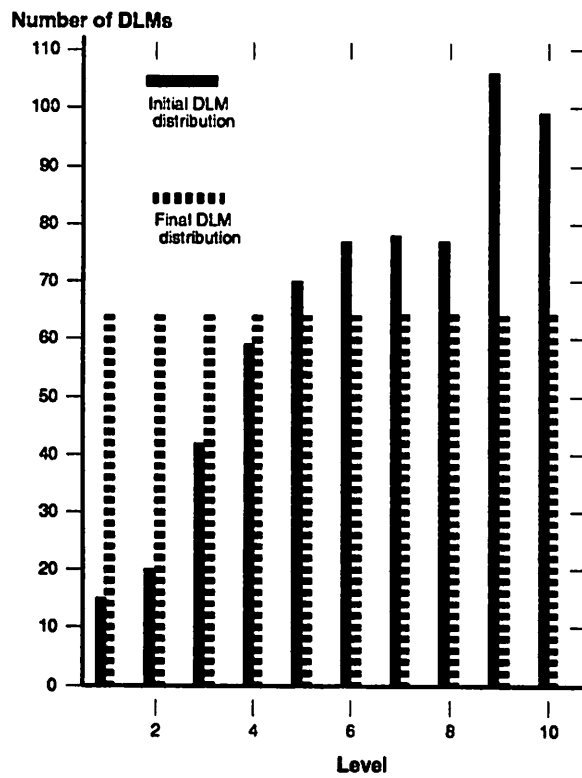


Figure 11.4: DLM distribution before and after temporal partitioning, for the MCNC benchmark example C5315.

---



Example	Modules $C$	Levels $l$	$\lceil C/l \rceil$	Before			After			Time CPU s
				Min	Max	$\sigma$	Min	Max	$\sigma$	
des	1397	11	127	6	244	69.6	127	127	0.0	135
rot	322	7	46	15	83	22.3	46	46	0.0	4
apex6	274	5	55	9	98	33.5	54	55	0.4	4
apex7	95	4	24	6	36	11.9	23	24	0.4	0
f51m	23	4	5	2	8	2.5	5	6	0.4	0
C5315	643	10	65	15	106	29.0	64	65	0.5	21
misex2	37	3	13	4	18	6.0	12	13	0.5	0
b9	49	3	17	13	18	2.1	15	16	0.5	0
rd84	13	3	5	2	7	2.1	4	5	0.5	0
9sym	7	3	3	1	5	1.9	1	3	0.9	0
duke2	164	6	28	1	48	16.2	25	28	1.1	1
C880	259	9	29	7	41	10.5	30	25	1.4	2
count	81	4	21	5	34	10.9	16	22	2.5	1
sao2	46	5	10	4	15	3.5	4	12	3.0	0
e64	213	5	43	5	67	24.1	40	46	3.0	4
clip	54	4	14	5	21	5.7	5	18	5.1	0
vg2	100	8	13	6	24	6.3	6	20	5.8	0
alu4	155	11	12	3	30	9.0	5	21	6.1	1
apex2	116	6	20	3	33	11.1	3	26	9.0	0
alu2	121	6	21	6	39	13.1	6	31	10.9	1
C499	199	8	25	4	51	14.8	15	42	11.1	1

Table 11.1: DLM distribution statistics before and after temporal partitioning

## Chapter 12

# Architecture Analysis

Since we are proposing a new architecture, we must analyze the pros and cons, and compare with existing architectures. We perform the analysis under the following sections:

1. Area analysis
2. Timing analysis
3. Experimental verification

### 12.1 Area analysis

We do a comparison of the area requirements of one particular implementation of the *Dharma* architecture, with existing FPGA devices. We show that the new *Dharma* architecture's area is comparable with the Xilinx and AT&T ORCA devices, and it is therefore feasible to implement *Dharma*.

#### 12.1.1 Assumptions

We proceed with the following assumptions for the *Dharma* chip:

1. DLMS are  $K$ -input, 1-output LUTs.
2. There are  $C$  DLMS and  $L$  maximum levels; assume  $K \approx \log_2 L$ .
3. There are  $B$  buffers; assume  $B = 2 \times C$ .

4. The DIA is implemented as a one-level crossbar, with two crossbars, the  $L$ -crossbar and the  $B$ -crossbar. The DLMs are divided into 2 classes, those that connect to even pins, and those that connect to odd pins of DLMs at the next level.
5. There are  $B$  input lines, which enter the  $L$ -crossbar and  $B$ -crossbar through 2-to-1 multiplexers.
6. The chip uses a global level counting scheme as in Figure 8.6.

If the DIA is implemented as a two-level crossbar (like the binomial concentrator) or like a copier (see Section 10.2), it is possible that area requirements will reduce. A sparsely connected  $K$ -class DIA will also reduce area requirements. A complex DLM, with interconnected LUTs could also reduce the area (but will increase software complexity).

### 12.1.2 Area components of *Dharma*

Following are the chief components to be considered for the area analysis:

1. DLM area
  - (a) Number of bits ( $2^K \times L$  bits per DLM)
  - (b) Decoder area (one  $K$ -input and one  $(\log_2 L)$ -input decoder per DLM)
  - (c) Multiplexer area (one mux per DLM)
  - (d) Output buffer area (one buffer per DLM, to drive crossbar)
2. DIA area
  - (a) L-crossbar
    - i. Number of bits ( $(B + C)$ -input,  $(K \times C)$ -output crossbar,  $L$  bits per crosspoint)
    - ii. Number of pass transistors (one per crosspoint)
    - iii. Number of lines (one per input and one per output)
  - (b) B-crossbar
    - i. Number of bits ( $B$ -input,  $B$ -output crossbar,  $L$  bits per crosspoint)
    - ii. Number of pass transistors (one per crosspoint)
    - iii. Number of lines (one per input and one per output)

- iv. Multiplexers (one per primary input and one per DLM output)
- 3. Latch area (a latch at each DLM and buffer input)
- 4. Memory configuration circuitry area
- 5. Buffer area ( $B$  buffers to drive crossbar)

### 12.1.3 The analysis

#### Configuration bits

			if $B = 2C$
DLM bits	=	$(2^K L) \times C$	$2^K LC$
L-crossbar	=	$\frac{(B+C)(KC)L}{2}$	$\frac{3C^2KL}{2}$
B-crossbar	=	$(B)(B) \times L$	$4C^2L$
PI Mux select bits	=	$B \times L$	$2CL$
Buffer Mux select bits	=	$C \times L$	$CL$
Total	=	$L\left\{\frac{KC^2}{2} + (2^K + 1)C + B^2 + B + \frac{KCB}{2}\right\}$	$LC\left\{\left(\frac{3K}{2} + 4\right)C + (2^K + 3)\right\}$

Table 12.1: Number of configuration bits

The  $\frac{1}{2}$  term in the L-crossbar arises from the odd-even classification of DLMs.

#### DLM decoders

Assuming an x-y decode scheme, each DLM will have 2 decoders; one  $K$ -input decoder to decode the actual DLM inputs, and one  $(\log_2 L)$ -input decoder to decode the level bits. Assuming  $K \approx \log_2 L$ <sup>1</sup>, there are  $2C$   $K$ -input decoders.

#### DLM Muxes

Under the x-y decode scheme, we will require a  $2^K$ -to-1 multiplexer per DLM, hence  $C$  such multiplexers are required.

<sup>1</sup>Typically,  $K < \log_2 L$ . So this is a conservative estimate.

**Pass transistors**

		if $B = 2C$
L-crossbar pass transistors	= $\frac{(B+C)(KC)}{2}$	$\frac{3C^2K}{2}$
B-crossbar pass transistors	= $\frac{B}{2}(B)$	$4C^2$
<b>Total</b>	= $\frac{KCB}{2} + B^2 + \frac{KC^2}{2}$	$(\frac{3K}{2} + 4)C^2$

Table 12.2: Number of pass transistors

**Lines**

		if $B = 2C$
L-crossbar lines	= $B + C + KC$	$(3 + K)C$
B-crossbar lines	= $B + B$	$4C$
<b>Total</b>	= $3B + (K + 1)C$	$(7 + K)C$

Table 12.3: Number of routing lines

We assume that each line spans the width or height of the chip (conservative estimate).

**2-to-1 multiplexers**

$C$  Buffer muxes and  $B$  PI muxes are required, giving a total of  $(B + C)$  ( $= 3C$ ) 2-to-1 multiplexers.

**Latches**

		if $B = 2C$
DLM input latches	= $KC$	$KC$
Buffer input latches	= $B$	$2C$
<b>Total</b>	= $B + KC$	$(K + 2)C$

Table 12.4: Number of latches

### Memory configuration circuitry

This circuitry performs the initial configuration of the bits into the configuration bits, in a serial fashion. Hence, the overhead associated with this circuitry can be assumed to be proportional to the number of bits. Let the constant of proportionality be  $\zeta$ . Hence,  $\text{area} = LC\zeta\left\{\left(\frac{3K}{2} + 4\right)C + 2^K + 3\right\}$ .

### Buffer area

$(B + C)$  ( $= 3C$ ) output buffers to drive crossbar lines.

#### 12.1.4 Comparison with Xilinx 2000 series

The XC2000 has 4-input LUTs and hence the DLMs are directly comparable to these LUTs. Each CLB has 2 outputs (these are different when the LUT is broken into two 3-input LUTs), one latch, three 2-to-1 multiplexers, and three 3-to-1 multiplexers.

The XC2000 has 3 kinds of interconnection resources: general purpose interconnect, long lines, and direct connections.

For the purposes of comparing interconnect resource area, we assume that both the *Dharma* and the XC2000 chip have the same die size. The interconnect lines are measured in terms of chip width and height. Specifically, we assume a square die (width = height), and normalize the interconnection line length to the chip width. In the analysis, it will turn out that *Dharma* will have comparable area to the Xilinx chip (*Dharma* will have smaller area), and hence the assumption of same die size is valid (the assumption is conservative, and favors Xilinx).

The XC2000 has 5 vertical general purpose interconnects<sup>2</sup> and 2 vertical long lines per column; 4 horizontal general purpose interconnects and 1 horizontal long line per row; and 1 direct interconnect in horizontal and vertical direction per CLB (connecting adjacent CLBs). If the chip has  $P$  CLBs, there are  $5 + 2 + 4 + 1 + 1 + 1 = 14\sqrt{P}$  chip-width lines.

Each line can be connected to the CLB inputs/outputs and other lines by means of programmable interconnect points (PIPs). The Xilinx data book does not specify the number of pass-transistors, but as a lower bound, we assume that each PIP has one pass-

---

<sup>2</sup>Each general purpose interconnect line is segmented, segments being connected by means of switch matrices. One column(row) of such segments ( $\sqrt{P}$  segments) is treated as one chip-width line in this comparison.

Area component	XC2064	<i>Dharma</i>	Remarks
Number of bits	11254	$8 \times 8 \left\{ \left( \frac{3 \times 4}{2} + 4 \right) 8 + 2^4 + 3 \right\}$ = 6336	Xilinx data book
Decoders	64, 4-input	16	8 (of 16) are 3-input for <i>Dharma</i>
DLM Mux	None	8, 16-to-1	(mux + decoder) area for <i>Dharma</i> < XC2064 decoder area
Pass transistors	$74 \times 64$ = 4736	$\left( \frac{3 \times 4}{2} + 4 \right) 8^2$ = 640	
Lines	$14 \times 8$ = 112	$(7 + 4) 8$ = 88	
2-to-1 mux	$3 \times 64$ = 192	$3 \times 8$ = 24	
3-to-1 mux	192	None	
Latches	64	$(4 + 2) 8$ = 48	
Memory config	11254ζ	6336ζ	same overhead / bit
Buffer	$2 \times 64$ = 128	$3 \times 8$ = 32	

Table 12.5: *Dharma*'s area comparison with XC2064

transistor<sup>3</sup>. From the XACT software's plot of the Xilinx chip, we count 50 PIPs surrounding each CLB. There are 2 switch matrices per CLB, and each switch matrix is  $2 \times 2$ . Assuming 2 switch points per matrix, with 6 pass transistors per point, we have 24 pass-transistors per CLB, in the switch matrix. Thus we have a total of  $74P$  pass-transistors. We assume 2 output buffers per CLB, resulting in  $2P$  buffers.

### XC2064

The XC2064 has 64 CLBs, therefore  $P = 64$ . We compare it with a *Dharma* chip with  $C = 8$ ;  $K = 4$ ;  $L = 8$  (note:  $K < \log_2 L$ ).

Table 12.5 shows the comparison. The *Dharma* resources use less area. It is reasonable to assume that the DLM mux and decoder areas for *Dharma* will sum up to be less than the decoder areas of the XC2064.

<sup>3</sup>This may be a poor lower bound, since PIPs may have more than one pass transistor to allow connections in different directions. However, we choose to err on the safer side.

Area component	XC2018	<i>Dharma</i>	Remarks
Number of bits	17238	$10 \times 10 \left\{ \left( \frac{3 \times 4}{2} + 4 \right) 10 + 2^4 + 3 \right\}$ = 11900.	Xilinx data book
Decoders	100, 4-input	16, 4-input	(mux + decoder) area for <i>Dharma</i> < XC2018 decoder area
DLM Mux	None	10, 16-to-1	
Pass transistors	$74 \times 100$ = 7400	$\left( \frac{3 \times 4}{2} + 4 \right) 10^2$ = 1000	
Lines	$14 \times 10$ = 140	$(7 + 4) 10$ = 110	
2-to-1 mux	$3 \times 100$ = 300	$3 \times 10$ = 30	
3-to-1 mux	300	None	
Latches	100	$(4 + 2) 10$ = 60	
Memory config	17238ζ	11900ζ	same overhead / bit
Buffer	$2 \times 100$ = 200	$3 \times 10$ = 30	

Table 12.6: *Dharma*'s area comparison with XC2018**XC2018**

The XC2018 has 100 CLBs, therefore  $P = 100$ . We compare it with a *Dharma* chip with  $C = 10$ ;  $K = 4$ ;  $L = 10$  (note:  $CL = P$  and  $K < \log_2 L$ ). Table 12.6 shows the comparison. It is reasonable to assume that the DLM Mux and decoder areas for *Dharma* will sum up to be less than the decoder areas of the XC2018. Again, the *Dharma* resources use less total area.

**12.1.5 Comparison with Xilinx 3000 series**

The Xilinx 3000 series chips use an interconnect pattern similar to the 2000 series. The CLBs are different, however. Each CLB has 2 latches, one 5-input LUT that can be configured to implement two 4-input functions, seven 2-to-1 muxes, two 3-to-1 muxes and one 2-input OR gate.

As in the 2000 series, we assume that XC3000 chip and *Dharma* have similar die sizes, and normalize line lengths to chip-width. There are 5 general purpose vertical lines and 3 vertical long lines per column, and 5 general purpose horizontal lines and one horizontal



Area component	XC3090	<i>Dharma</i>	Remarks
Number of bits	62668	$16 \times 20 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 20 + 2^5 + 3 \right\}$ = 84800.	Xilinx data book
Decoders	320, 5-input	40, 5-input	
DLM Mux	None	20, 32-to-1	
Pass transistors	$100 \times 320$ = 32000	$\left( \frac{3 \times 5}{2} + 4 \right) 20^2$ = 4600	
Lines	$9 \times 20 + 7 \times 16$ = 292	$(7 + 5) 20$ = 240	
2-to-1 mux	$7 \times 320$ = 2240	$3 \times 20$ = 60	
3-to-1 mux	$2 \times 320$ = 640	None	
Latches	640	$(5 + 2) \times 20$ = 140	
Memory config	62668 $\zeta$	84800 $\zeta$	same overhead / bit
Buffer	$2 \times 320$ = 640	$3 \times 20$ = 60	

Table 12.7: *Dharma*'s area comparison with XC3090

long line per column. A direct interconnect from a CLB output connects to adjacent CLBs in the horizontal and vertical directions. Hence there are a total of  $(5+3+1)r + (5+1+1)c = 9r + 7c$  chip-width lines, where  $r$  is the number of rows and  $c$  is the number of columns.

Each line can be connected to the CLB inputs/outputs and other lines by means of PIPs. We use the number of PIPs as the lower bound on the number of pass-transistors, although each PIP may have more than one pass transistor to implement connections in various directions. From the data book and the XACT software's detailed diagram of the XC3000 series, we calculated the number of pass-transistors to be  $100P$ , where  $30P$  pass-transistors are used in the  $5 \times 5$  switch matrix (5 switch points, with 6 pass transistors per point), and  $70P$  PIPs surround CLBs. There are  $2P$  output buffers to drive the lines.

### XC3090

The XC3090 has 320 CLBs, each with a 5-input LUT. Therefore,  $P = 320$ . There are 20 rows and 16 columns, which implies  $r = 20$  and  $c = 16$ . We use *Dharma* parameters as  $C = 20$ ;  $K = 5$ ;  $L = 16$  (note:  $CL = P$  and  $K < \log_2 L$ ).

Table 12.7 shows the comparison. Since it is difficult to make a comparison directly,

Component	XC3090	<i>Dharma</i>
Bits	62668	84800
Decoders	$320 \times 64$ = 20480	$40 \times 64$ = 2560
DLM mux	0	$20 \times 32$ = 640
Pass transistor	16000	2300
2-to-1 mux	2240	60
3-to-1 mux	$640 \times 4$ = 2560	0
Latches	$640 \times 3$ = 1920	$140 \times 3$ = 420
Memory config	62668	84800
Buffer	$640 \times 2$ = 1280	$60 \times 2$ = 120
Total	169816	175700

Table 12.8: Gate equivalent area comparison with XC3090

Area component	XC3090	<i>Dharma</i>
Number of bits	62668	$16 \times 20 \left\{ \left( \frac{3 \times 5}{5} + 4 \right) 20 + 2^5 + 3 \right\}$ = 56000.
Pass transistors	32000	$\left( \frac{3 \times 5}{5} + 4 \right) 20^2$ = 2800

Table 12.9: *K*-class DIA fares better

we use gate equivalent analysis. We assume that each bit is equivalent to 1 gate, each DLM mux is equivalent to 32 gates (wired OR, uses column select from column decoder), each decoder is equivalent to 64 gates, each latch is 3 gate equivalents, each memory configuration overhead is 1 gate equivalent per bit, each buffer is 2 gate equivalent, each 2-to-1 mux is 1 gate-equivalents, each 3-to-1 mux is 4 gate-equivalents, and each pass transistor is 0.5 gate equivalent. We then have the gate-equivalent totals as in Table 12.8.

Thus, the *Dharma* chip is bigger by a factor of 3.5%.

If we use a *K*-class division, for the DLM connections to the next level DLM inputs (i.e., each DLM output connects to only one input of the next level DLM), the number of bits and pass transistors changes as tabulated in Table 12.9.

Therefore, the gate equivalents decrease by  $2(84800 - 56000) + 0.5(4600 - 2800) =$

Size	XC4000	<i>Dharma</i>
64	35956	$C = 8; L = 8; K = 5$ $8 \times 8 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 8 + 2^5 + 3 \right\}$ $= 8128$
100	51788	$C = 10; L = 10; K = 5$ $10 \times 10 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 10 + 2^5 + 3 \right\}$ $= 15000$
196	92092	$C = 20; L = 10; K = 5$ $10 \times 20 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 20 + 2^5 + 3 \right\}$ $= 53000$
324	143916	$C = 20; L = 16; K = 5$ $16 \times 20 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 20 + 2^5 + 3 \right\}$ $= 84800$
400	174148	$C = 25; L = 16; K = 5$ $16 \times 25 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 25 + 2^5 + 3 \right\}$ $= 12900$

Table 12.10: *Dharma*'s configuration bits comparison with XC4000

58500, and becomes 117200, 31% less than the XC3090.

### 12.1.6 Comparison with Xilinx 4000 series

A direct one-to-one comparison with the XC4000 series is not possible because of the significant difference in the CLB structure. We cannot compare one CLB with one DLM, unless we change the DLM structure itself. However, from the analysis so far, it is clear that the dominating factor is the number of bits, and we hence tabulate, in Table 12.10, the number of bits for the XC4000 series chips and the *Dharma* chips (odd-even class interconnection) with  $CL = P$ , where  $P$  is the number of CLBs in the XC4000 series chips. We assume a 5-input LUT for the DLM.

From the table, we observe that *Dharma* uses far fewer configuration bits.

### 12.1.7 Comparison with the AT&T ORCA

The comments on comparison with the XC4000 apply to the AT&T ORCA chips as well. We just list the number of bits in Table 12.11. Again, *Dharma* uses far fewer configuration bits.

Size	ORCA	<i>Dharma</i>
100	42816	$C = 10; L = 10; K = 5$ $10 \times 10 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 10 + 2^5 + 3 \right\}$ $= 15000$
196	81048	$C = 20; L = 10; K = 5$ $10 \times 20 \left\{ \left( \frac{3 \times 5}{2} + 4 \right) 20 + 2^5 + 3 \right\}$ $= 53000$

Table 12.11: *Dharma*'s configuration bits comparison with AT&T ORCA

## 12.2 Timing analysis

In this section, timing equations for *Dharma* are derived. The equations are dependent on the clocking scheme used, and hence there are two equations, *EFP timing equation* and *S timing equation*. Each equation gives the total time required for signals to propagate from the inputs to the outputs, for an  $l$  level circuit, when the circuit is implemented on a *Dharma* chip.

### 12.2.1 Notation

The following lists the symbols used in the timing equations.

$D_{EFP}$	Delay from input to output for EFP clocking.
$D_S$	Delay from input to output for S clocking.
$f_E$	<i>Ext clk</i> frequency.
$f_I$	<i>Int clk</i> frequency.
$l$	Number of levels in the circuit being implemented.
$T_d$	Clock period of <i>Int clk</i> .
$T_h$	Time duration for which <i>Int clk</i> is high.
$T_l$	Time duration for which <i>Int clk</i> is low.
$t_{dlm}$	Worst-case propagation time through DLM.
$t_{dia}$	Worst-case propagation time through DIA.
$t_{lrcfg}$	Reconfiguration time for DIA.
$t_{Lrcfg}$	Reconfiguration time for DLM (= 0 for LUT DLM).
$t_{ld}$	Latch/Register propagation time.
$t_{lh}$	Latch/Register hold time.
$t_{ls}$	Latch/Register set-up time.
$t_{mux}$	Worst-case propagation time through a 2-input multiplexer.
$t_{xbL}$	Worst-case propagation time through the L crossbar.
$t_{xbB}$	Worst-case propagation time through the B crossbar.

### 12.2.2 EFP timing equation

Consider Figures 8.7(a), 8.13 and 12.1.

In Figure 12.1, the path from input register to output register has been unfolded. The unfolded path has been drawn in a snake like fashion. All possible connections (DLM to DLM, DLM to buffer, buffer to buffer, and buffer to DLM) have been shown in the figure. The vertical dotted line, labeled **H** corresponds to the time instant at which *Int clk* goes high, and the vertical dotted line labeled **L** corresponds to the time instant at which *Int clk* goes low. The various components have been drawn such that they are aligned to these two time instances. The jog in the vertical line **L** corresponds to the extension in  $T_l$ , the low portion of *Int clk*, to account for the delay through the primary input registers.

In Figure 8.13, the reconfiguration time of the DIA is time-shared with the propagation time through the DLM, and this occurs when *Int clk* is high. The DIA can be reconfigured after a delay of  $t_{lh}$ , after latching the inputs, for proper latch operation.

From Figure 12.1 and 8.13,

$$T_h \geq t_{Ircfg} + t_{lh}$$

$$T_h \geq t_{dlm} + t_{ld}$$

Combining these two equations,

$$T_h \geq \max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} \quad (12.1)$$

From Figure 12.1, the worst-case delay through the DIA is given by

$$t_{dia} = \max\{(t_{mux} + t_{xbL}), (2 \times t_{mux} + t_{xbB})\}. \quad (12.2)$$

Since the B crossbar is much smaller than the L crossbar, it is likely that  $t_{dia} = t_{mux} + t_{xbL}$ .

After signals have propagated through the interconnect, an additional duration of  $t_{ls}$  is required before *Int clk* can go high. Therefore,

$$T_l \geq t_{dia} + t_{ls} \quad (12.3)$$

For the first period, however, *Int clk* has to remain low for an extra duration of  $t_{ld}$  (propagation through the primary input register).

Combining equations 12.1 and 12.3,

$$\begin{aligned} T_d &= T_h + T_l \\ &\geq \max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls} \end{aligned} \quad (12.4)$$

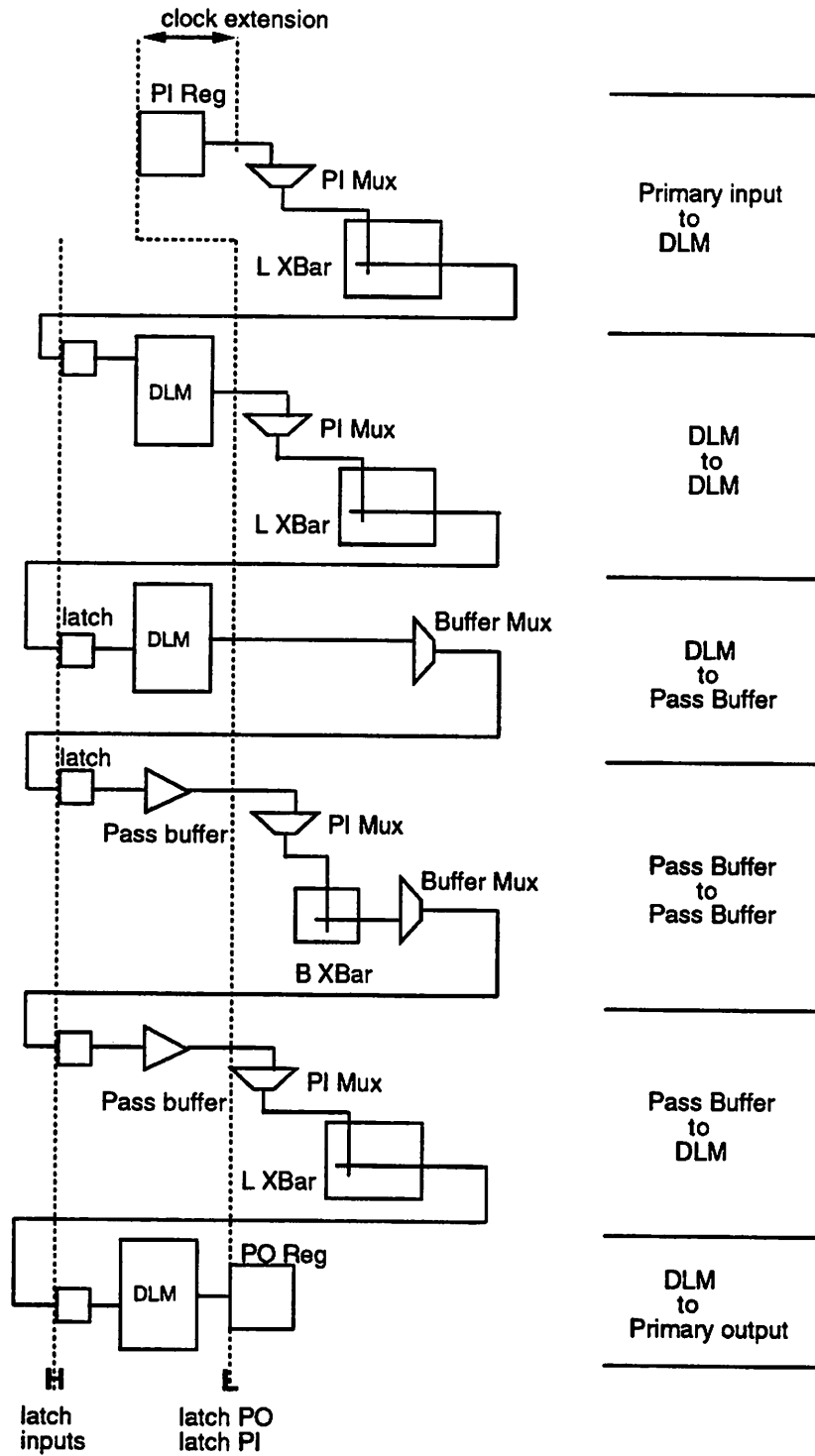


Figure 12.1: Input to output signal path for EFP clocking scheme

For an  $l$  level circuit, there are  $l$  durations of  $T_h$ , and  $l$  durations of  $T_l$ , in the EFP scheme. The first  $T_l$  is extended by  $t_{ld}$ . Hence, the delay from input to output,

$$\begin{aligned} D_{EFP} &= l \times T_h + l \times T_l + t_{ld} \\ &= l \times T_d + t_{ld} \end{aligned} \quad (12.5)$$

Substituting equation 12.4 in 12.5, and assuming that *Int clk* operates at the maximum rate,

$$D_{EFP} = l \times [\max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls}] + t_{ld} \quad (12.6)$$

Equation 12.6 is the EFP timing equation. The delay through the implemented circuit varies linearly as the number of logic levels,  $l$ . This is a key feature of the *Dharma* architecture.

If  $(t_{Ircfg} + t_{lh}) < (t_{dlm} + t_{ld})$  (as will usually be the case, especially if the DLM is an interconnection of LUTs), then

$$D_{EFP} = l \times [t_{dlm} + t_{dia} + t_{ld} + t_{ls}] + t_{ld} \quad (12.7)$$

Equation 12.5 gives the minimum clock period of *Ext clk*. If  $f_I$  is *Int clk* frequency, and  $f_E$  is the *Ext clk* frequency,

$$\begin{aligned} f_E &= \frac{1}{D_{EFP}} \\ &= \frac{1}{l \times T_d + t_{ld}} \\ &= \frac{1}{T_d(l + \delta)} \\ f_E &= \frac{f_I}{l + \delta} \end{aligned} \quad (12.8)$$

where  $\delta = \frac{t_{ld}}{T_d}$ . Thus, *Ext clk* is slower than the *Int clk* frequency by a factor of  $\frac{1}{l+\delta}$  for EFP clocking.

### 12.2.3 S timing equation

Figure 12.2 shows the unfolded signal path for the S clocking scheme. Comparing with Figure 12.1, in Figure 12.2 the PI and PO registers are clocked in phase with the input latches, and hence are placed vertically aligned to the **H** dotted line. As a result of this shift

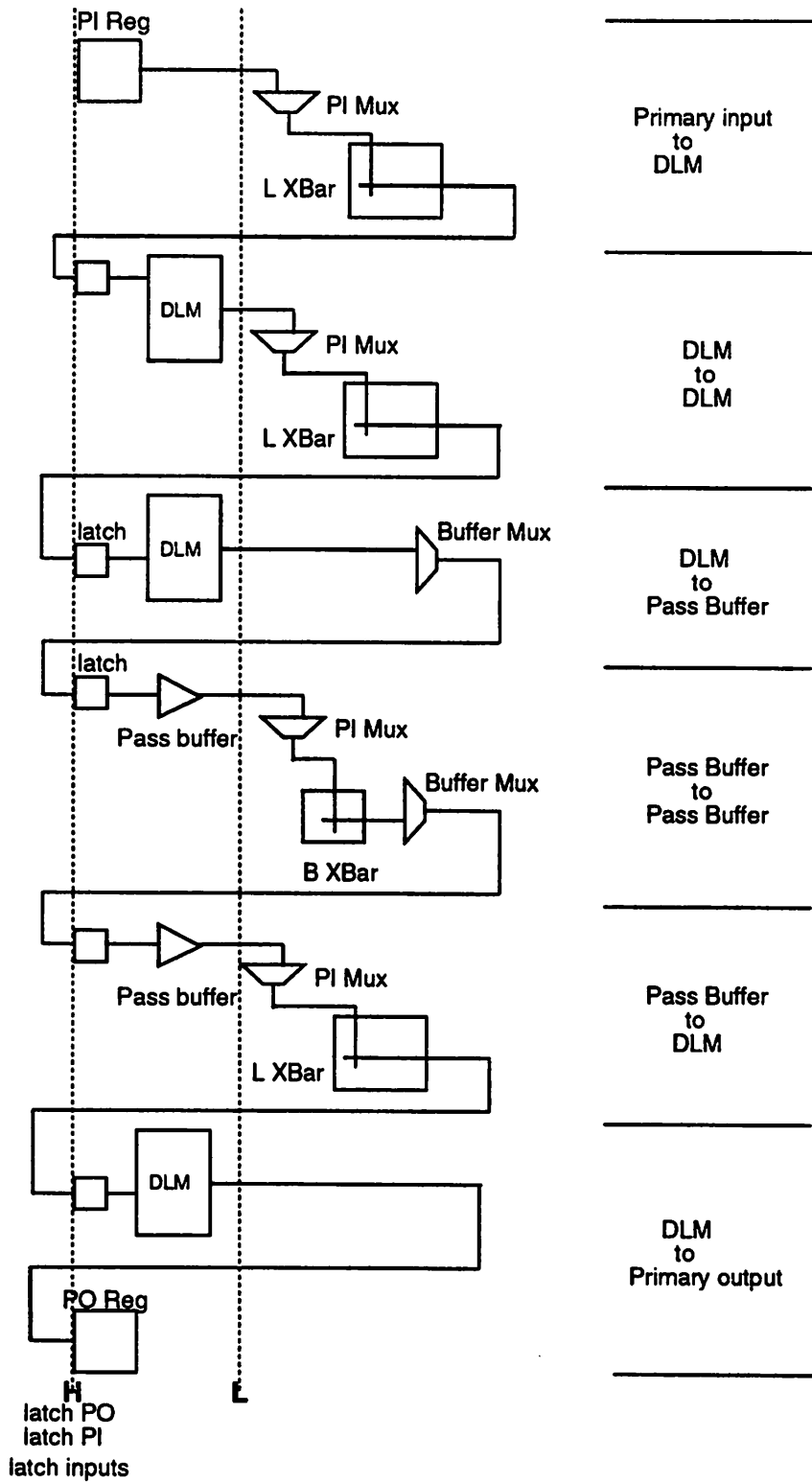


Figure 12.2: Input to output signal path for S clocking scheme



in time, the propagation time through the primary input register ( $t_{ld}$ ) is now in phase with the propagation time through the input latches, and hence does not cause an extension of the clock period. However, this phase alignment increases the number of *Int clk* cycles to  $(l + 1)$ , as shown in Figure 8.7(b). The rest of Figure 12.2 is similar to Figure 12.1. The derivation of  $T_d$  remains unchanged, and is exactly as given in equation 12.4. The S timing equation, for an  $l$  level circuit implemented on *Dharma* is given by

$$D_S = (l + 1) \times T_d \quad (12.9)$$

Substituting equation 12.4,

$$D_S = (l + 1) \times [\max\{(t_{Ircfg} + t_{lh}), (t_{dlm} + t_{ld})\} + t_{dia} + t_{ls}] \quad (12.10)$$

If  $(t_{Ircfg} + t_{lh}) < (t_{dlm} + t_{ld})$ , then

$$D_S = (l + 1) \times [t_{dlm} + t_{dia} + t_{ld} + t_{ls}] \quad (12.11)$$

From equation 12.9,

$$f_E = \frac{f_I}{l + 1} \quad (12.12)$$

So, *Ext clk* is  $\frac{1}{l+1}$  times slower than *Int clk* in the S clocking scheme.

Comparing equation 12.11 with 12.7, the circuit will be slower by  $t_{dlm} + t_{dia} + t_{ls}$ . Hence the “slow” prefix for this clocking scheme.

#### 12.2.4 Remarks

The time required by a  $K$ -input LUT CLB for logic function evaluation will be less than the time required by a DLM of equal functional capability, since the DLM is essentially a  $(K + \log_2 L)$ -input LUT. However, since there will be fewer decoders (by a factor of  $L$ ), they can be made larger to provide the same speed as that of a  $K$ -input LUT CLB.

The interconnect delay (see equation 12.2) is a constant predictable delay, whose value depends on the crossbar implementation (several types are possible, as described in Section 10.2). Typical values of 4 to 5 ns delay for  $t_{dia}$  is expected [Horng 92]. The sum of the latch set-up and propagation times is of the order of 3 to 4 ns. Thus, the latch and interconnect together introduce a latency of about 7 to 9 ns. While the Xilinx style architecture boasts of a much smaller interconnection delay, experiments over actual circuits show that routing delay could be as high as 48 ns.

### 12.3 Experimental verification

We performed a comparison of *Dharma*'s timing behavior with that of the Xilinx 3000 series chips. Our experiment is conducted in the following manner: Several MCNC combinational benchmark circuits were mapped to Xilinx using the `mis_pga` mapping technique [Murgai 91a]. The mapper generates an output circuit with 1-output LUTs, each having  $\leq 5$  inputs. One such LUT is assigned to each CLB on the Xilinx chip. Each benchmark circuit is placed and routed (using the Xilinx place and route tool `apr`) on the smallest possible XC3000 series chip (i.e., the smallest chip which satisfied the condition,  $\# \text{ CLBs} \geq \# \text{ LUTs}$ ). However, if there were unrouted nets on such a chip, the next larger chip is used.

The results are tabulated in Table 12.12 (see Table 7.1 also). The second column lists the number of 1-output LUTs in the examples, and the third column lists the number of levels of logic on the path with maximum delay. The 'Xilinx delay' column lists the worst-case delay as computed by Xilinx's timing analyzer `XDelay`.

The temporal partitioning technique of Section 11.3 is used to levelize the LUTs for *Dharma*. Columns 'Max DLMS' and 'Max buffers' list the statistics after temporal partitioning. 'Max DLMS' is the maximum number of DLMS in any level, and 'Max buffers' is the maximum number of buffers in any level. A *Dharma* chip with appropriate  $C$  and  $B$  is selected to implement the circuit, based on these numbers. The delays for the *Dharma* implementation are calculated using equations 12.7 and 12.11 for the EFP and S clocking schemes respectively, and these are listed in the columns towards the right hand side of Table 12.12. The Xilinx delay numbers include IOB delays; hence additional IOB delays are added to the *Dharma* delay numbers as well. The percentage decrease in *Dharma* delays, as compared to the Xilinx delays are listed in the '% impr.' columns.

In computing the expected delay on *Dharma*, we make the following assumptions.

1. Xilinx CLB delay = *Dharma* DLM delay ( $t_{dlm}$ ) = 8 ns.
2. Xilinx IOB delay = *Dharma* IOB delay =  $6 + 27 = 33$  ns.
3. *Dharma* interconnect delay ( $t_{dia}$ ) = 5 ns
4. *Dharma* latch set-up delay ( $t_{ls}$ ) = 0.2 ns.
5. *Dharma* latch propagation delay ( $t_{ld}$ ) = 3.8 ns.

Example	LUTs	Levels	Max DLMs	Max buffers	Xilinx delay	<i>Dharma</i>			
						EFP clock		S clock	
						delay	% impr.	delay	% impr.
5xp1	21	2	11	7	87.8	71.0	-19.1	84.0	-4.3
C499	199	8	42	59	346.3	173.0	-50.0	186.0	-46.3
C880	259	9	30	110	342.9	190.0	-44.6	203.0	-40.8
alu2	121	6	31	24	187.4	139.0	-25.8	152.0	-18.9
alu4	155	11	21	42	316.5	224.0	-29.2	237.0	-25.1
apex7	96	4	24	53	168.6	105.0	-37.7	118.0	-30.0
b9	49	3	16	38	107.2	88.0	-17.9	101.0	-5.8
bw	28	1	28	0	75.6	54.0	-28.6	67.0	-11.4
clip	54	4	18	18	120.5	105.0	-12.9	118.0	-2.1
count	81	4	22	43	133.1	105.0	-21.1	118.0	-11.3
duke2	164	6	28	72	195.2	139.0	-28.8	152.0	-22.1
e64	213	5	46	90	203.5	122.0	-40.0	135.0	-33.7
f51m	23	4	6	13	109.3	105.0	-3.9	118.0	8.0
misex2	37	3	13	24	93.3	88.0	-5.7	101.0	8.3
sao2	46	5	12	14	125.4	122.0	-2.7	135.0	7.7
vg2	100	8	20	17	151.9	173.0	13.9	186.0	22.4
Average							-22.1		-12.8

Table 12.12: Table showing results of an experiment comparing *Dharma* with Xilinx's 3000 series. Xilinx results were obtained after place and route, using *apr*, followed by *Xdelay*'s timing analysis. *Dharma* results are from the timing equations 12.7 and 12.11.

From these estimated delays for *Dharma*, we observe an average improvement of 22.1% if the EFP clocking scheme is used, and an improvement of 12.8% if the S clocking scheme is used, for the 16 MCNC benchmark circuits.

Since faster DIAs are possible, there is a further scope for improving the *Dharma* delay numbers. Also, the mapping for *Dharma* has only been done using temporal partitioning; i.e., no real logic synthesis has been performed (see Chapter 13 for a possible approach for logic synthesis for *Dharma*). Logic synthesis geared for *Dharma* can further reduce the timing delay, and also reduce resource utilization (number of DLMS and buffers used).

## 12.4 Conclusions

In this chapter, the new *Dharma* architecture's silicon area usage and timing behavior were studied. We compared *Dharma*'s area usage with existing reconfigurable architectures, such as the Xilinx series of chips and the AT&T ORCA. We derived timing equations for *Dharma*, which provide timing delay and clock rates for circuits implemented on *Dharma*.

Our area analysis showed that the silicon area of the *Dharma* device is expected to be smaller compared to the XC 2000 series chips, for the same logic capability. *Dharma* has about the same area as the XC 3090 and *Dharma* has far fewer configurable memory bits (and switches) compared to the XC 4000 and AT&T ORCA FPGAs. Timing performance estimates of *Dharma* compared to the XC 3000 series show that circuits implemented on *Dharma* can be faster by about 13% if the S clocking scheme is used, and by about 22% if the EFP clocking scheme is used.

One may tend to argue that the *Dharma* numbers show only a few percent improvement, and it may therefore not be worth the effort to go for a *Dharma* style FPLD. However, it must be pointed out that in our analysis, we have taken a conservative approach in favor of Xilinx. It was necessary to be conservative, since our approach is based on estimates (a *Dharma* device has not yet been implemented). Estimates can sometimes be off by a few percent, and it is best to err on the safer side; so that errors in the estimate, if any, can only be to *Dharma*'s advantage. Thus, our analysis yields the worst-possible numbers for *Dharma*. Synthesis results are sub-optimal; synthesis tailored for *Dharma* can improve timing behavior. We have assumed an SRAM technology – since this is the one used by Xilinx and AT&T. However, as discussed in Chapter 10, the *Dharma* architec-

ture is amenable to other technologies such as DRAM and CCD, which can result in area savings and better timing performance (Xilinx and ORCA devices cannot readily benefit from these other technologies). DIA size can be reduced by using the full  $K$ -class DIA (see Section 10.2.6), and interconnect delay can be reduced by using the precharge and sense scheme (see Section 10.2.5).

In spite of the conservative approach, *Dharma* has better silicon area and timing performance. With a suitable choice of technology and architectural modifications (from amongst the several listed in Chapter 10), and with a set of CAD tools specifically developed for *Dharma*, it is expected that a *Dharma* based PLS will have clear-cut advantages.

## Chapter 13

# Unexplored Terrain

The purpose of this chapter is to point out potential avenues for furthering the work on the *Dharma* architecture. One is to build a prototype device. This may be straightforward for expert designers, and is not discussed here. Chapter 11 only discussed synthesis as a level allocation problem. In the first section here, we describe the logic synthesis requirements for *Dharma*. Next, we look at improvements possible in the overall architecture, if software techniques are available to solve certain special problems that arise. Our objective is overall PLS improvement, and architecture improvements that cannot be exploited by the accompanying software are not real improvements. We describe a synthesis problem which, if solved, can cause an improvement in the area and performance of the device.

### 13.1 Logic synthesis for *Dharma*

From a logic synthesis viewpoint, *Dharma* poses a unique problem. Given a circuit specification, the synthesizer should generate a network with the following objectives.

1. Number of levels must be minimized.
2. Number of modules in any level  $d$ ,  $M_d = \max(\#(d))$ , must be minimized.
3. Number of inter-level signals (signals that travel more than one level) must be minimized.

Minimizing the number of levels improves the performance since the delay through the *Dharma* realization is proportional to  $l$ , the number of circuit levels. Minimizing  $l$  also

minimizes the number of configuration bits, since these are also proportional to  $l$  (i.e., by minimizing  $l$ , one can use a smaller *Dharma* chip).

The circuit is implemented on a *Dharma* device which has  $C \geq M_d$ . Thus, minimizing  $M_d$  allows one to use a smaller chip. Similarly, the number of inter-level signals determines the number of on-chip pass-buffers required, and again minimizing this number, can allow a smaller chip to be used for the circuit realization.

Our approach in Chapter 11 partitioned the above objectives into two. Objective 1 was solved with no regard to the other two constraints. However, we conjecture that a logic synthesizer that considers all the three objectives together can give better results compared to those reported in Chapter 11.

One possible approach to this synthesis problem, is along the lines of [Kim 93]. Although the approach used in [Kim 93] is for wave-pipelined circuits, the objectives are somewhat similar. In the case of wave-pipelining, the logic synthesis solution should be such that the resulting circuit is *balanced*. That is, the shortest and longest paths at every node, from the primary inputs, must be equal. One way of achieving this balance is to add delay buffers to the shorter path so that it becomes as slow as the longer path. However, this is not the best way, as the simple case in Figure 13.1 (taken from [Kim 93]) illustrates. Figure 13.1(a) is one possible implementation of the function

$$y = ((a + b) c + d) e + d g.$$

To balance this circuit requires 10 delay buffers. However, the same function can be realized as in Figure 13.1(b). The second realization is balanced with only one delay buffer. The authors in [Kim 93] present a *balancing algorithm* which restructures a given circuit into a balanced circuit, keeping the area (area due to gates plus delay buffers) and delay minimum.

The balancing algorithm first decomposes the input circuit into a network of 2-input gates. Then the algorithm starts at the most unbalanced (largest difference between longest and shortest paths) output node. This node is collapsed, with a limit on the size of the collapsed node. Unbalanced fanin nodes are recursively balanced. The collapsing is followed by a balanced decomposition.

The balanced decomposition is the key step in the balancing algorithm (and it is this step that needs to be modified for *Dharma*). For a balanced decomposition, the kernel and co-kernel nodes must have equal logic depths, which is one less than the remainder node's logic depth. The cost function to choose a kernel for decomposition is set using

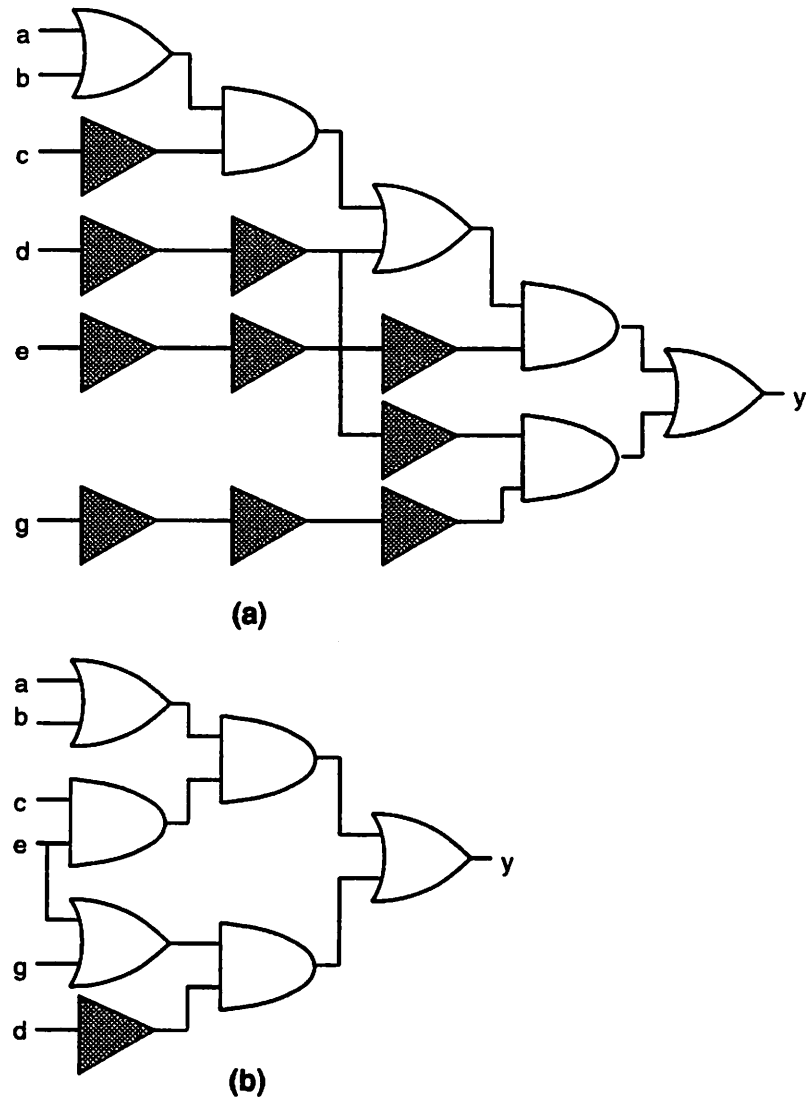


Figure 13.1: Unbalanced and balanced implementation of function  $y$ .

---



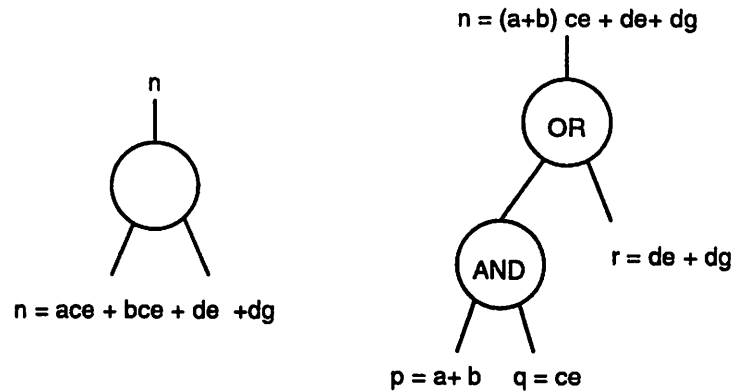


Figure 13.2: Example of balanced decomposition.

this requirement. In Figure 13.2, function  $n$  is being decomposed. A candidate kernel is  $p = (a + b)$ . The corresponding co-kernel is  $q = ce$  and the remainder is  $r = de + dg$ . Hence, if  $p$  and  $q$  have the same level (i.e., logic depth), and  $r$  is at the next level, then this decomposition results in balancing node  $n$ .

Therefore, the quality of a kernel  $p$  for the decomposition of node  $n$  is chosen as

$$\begin{aligned} \text{cost}(n, p) = & \| \text{delay}(p) - \text{delay}(q) \| + \\ & \| \text{delay}(r) - \text{MAX}(\text{delay}(p), \text{delay}(q)) - \text{gate\_delay} \| \end{aligned}$$

A kernel with the least cost is chosen for the decomposition.

For *Dharma*, balancing is relevant since it minimizes pass-buffer usage. In addition to balancing, in *Dharma*, the number of nodes per level must be minimized. This must be incorporated into the kernel selection cost function. One way to do this could be to compute

$$l\_cost(n, p) = \#(\text{level}(p)) + \#(\text{level}(q)) + \#(\text{level}(r)),$$

where  $\#(d)$  is the number of nodes at level  $d$ .  $l\_cost$  could then be added to  $\text{cost}(n, p)$  with a weighting factor. The estimation of delay is straightforward in the case of *Dharma*, because of the linear relationship to the logic level (this is not the case for standard cells and other design styles, for which the above algorithm was intended, and the authors describe heuristics to estimate delay in such cases). The number of inputs per node, and the type of kernel selection will also have to be modified for *Dharma*, based on the logic function

capabilities of the DLMs. For example, if the DLM is a  $K$ -input LUT, then LUT based decomposition strategies can be incorporated.

## 13.2 Repetitive structures

In Chapter 10, we discussed the sparsely connected  $K$ -class DIA. This technique reduces the number of cross-points by almost a factor of  $K$ . However, it necessitates a graph-coloring type of placement algorithm to perform the placement of the modules in such a manner that there are no unrouted nets.

Here we discuss a very sparse crossbar (VSC) DIA. Such a crossbar requires sophisticated synthesis techniques to make sure that the given circuit can be fully routed. Unless such techniques are developed, the VSC cannot be used. In this section, we motivate the principle behind the VSC, and outline synthesis requirements for the same.

### 13.2.1 Principle

If the combinational part of a circuit can be synthesized in a manner such that there is a repetition of interconnection patterns, then part of the reconfigurable crossbar can be made of *fixed* connections instead of programmable connections. Let  $\mathcal{P}_k$  be a matrix of 0's and 1's such that the rows of  $\mathcal{P}_k$  correspond to the horizontal lines of the DIA and the columns to the vertical lines. An element  $a_{ij} = 1$  implies that vertical line  $j$  connects to horizontal line  $i$  at level  $k$ , and a '0' implies that there is no connection. If we can synthesize a mapping such that the  $\mathcal{P}_i$ 's are as identical as possible, i.e.,

$$\|\mathcal{P}_2 - \mathcal{P}_1\| \approx 0, \|\mathcal{P}_3 - \mathcal{P}_2\| \approx 0, \|\mathcal{P}_4 - \mathcal{P}_3\| \approx 0, \dots,$$

then it means that certain configuration bits in the interconnection remain the same every level, and can hence be fixed instead of programmable. Which portion is to be kept fixed is dependent on the ability of the synthesis algorithm.

### 13.2.2 Example

As an example, consider the tiny *Dharma* architecture of Figure 13.3. Each DLM is a 2-input LUT, with 1 output. The  $L$ -crossbar has only 8 programmable points. There are four fixed connections. Let us assume that there are no buffers, for the purposes of this

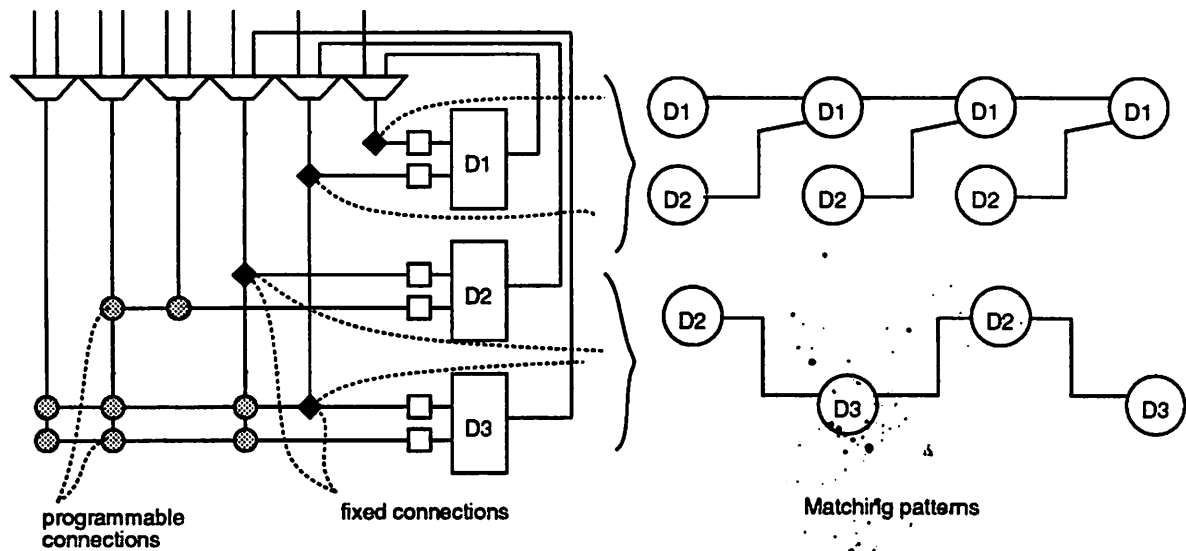


Figure 13.3: Tiny *Dharma* with VSC.

illustration. DLMs  $D1$  and  $D2$  are connected to DLM  $D1$  through fixed connection points (through the PI muxes). These connections can implement a portion of the circuit that has the repetitive pattern shown on the right side of Figure 13.3.

Similarly, DLM  $D3$  is connected to  $D2$ , and vice-versa, using fixed connections. Portions of the circuit that match the pattern on the right hand side of Figure 13.3 can be mapped on to  $D2$  and  $D3$ , using these connections.

The programmable connections allow other portions of the circuit to be realized (we cannot expect every part of the circuit to have repetitive patterns).

Let  $x$  and  $y$  be two functions, as specified below, which are to be realized on this tiny *Dharma* architecture.

$$x = acd + bcd + cdgh'i' + cdg'hi' + e + f$$

$$y = cde'f'gh'i'j + cde'f'g'hi'j$$

The task of the synthesizer is to transform this input specification in a manner such that the resulting network has the repetitive patterns shown in Figure 13.3. Consider

the following solution.

$$\begin{aligned}
 x &= u + v \\
 y &= vw \\
 p &= a + b \\
 q &= cd \\
 o &= gh' + g'h \\
 r &= pq \\
 s &= e + f \\
 t &= oq \\
 u &= r + s \\
 v &= i't \\
 w &= js'
 \end{aligned}$$

Figure 13.4(a) shows this network in graphical form. Figure 13.4(b) identifies the patterns of Figure 13.3 in this network, and this guides which function will be implemented in which DLM, and at what time instant.

### 13.2.3 Remarks

The example shows just a couple of repetitive patterns. Several more are possible, especially when the number of DLM inputs are more than 2. The synthesizer requirements are similar to those of a library-based mapper, which matches patterns in a given input specification. However, the pattern matching is more involved here, since patterns share common nodes. For example, the two patterns of Figure 13.3 share  $D2$ . Patterns that can be easily handled by the synthesizer must be chosen, and these set the location of the fixed connections.

Another possibility to investigate is to have not only repetitive connection patterns, but also repetitive function patterns. For example, if the function  $a + b'$  is used in every level, then the DLM realizing this function need have only configuration bits for one level.

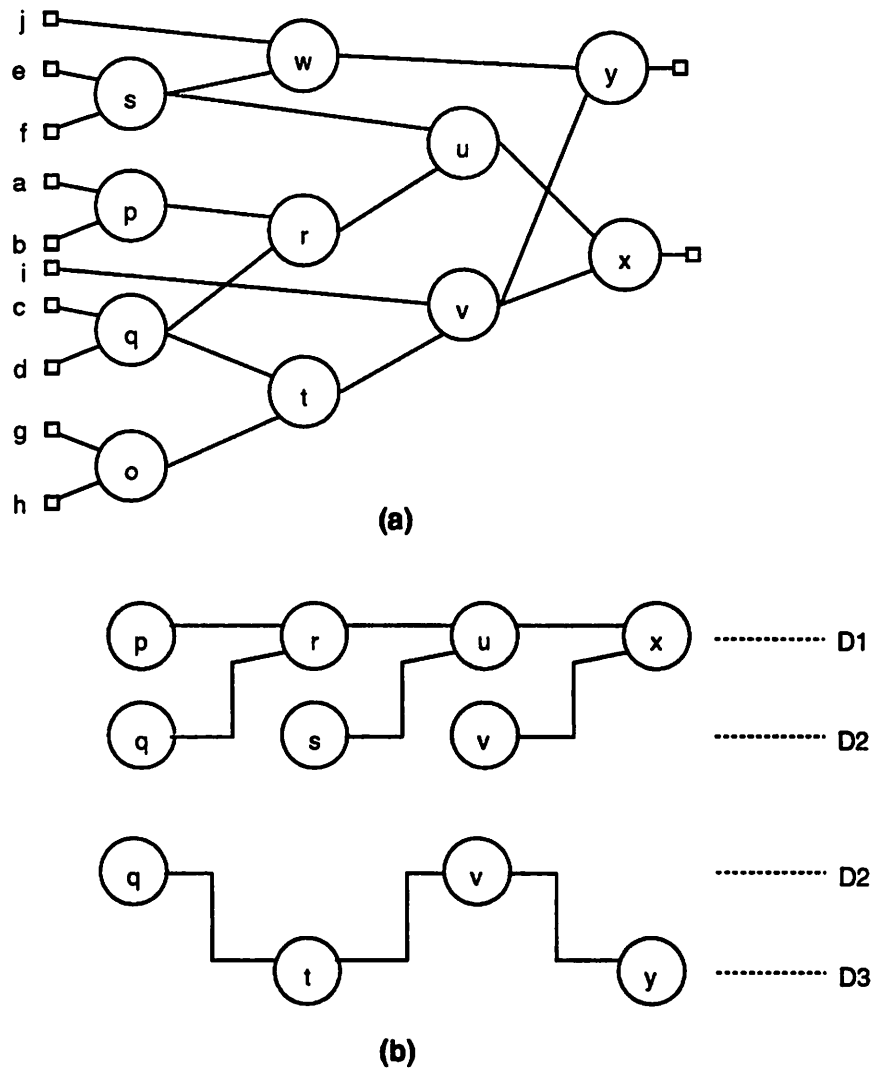


Figure 13.4: An example DAG with repetitive structures.

---

instead of for every level. A heterogenous DLM array (as in section 10.1.4) will be useful if a synthesizer with such a capability can be developed.

The first of these is the fact that the world is not a flat plane, but a curved surface. This is a fact that is often overlooked, but it is one that has a profound effect on the way we think about the world and our place in it. The second is the fact that the world is not a uniform expanse, but a complex and diverse one. There are many different cultures, languages, and ways of life, and each of these has its own unique contribution to make to the world. The third is the fact that the world is not a static entity, but a dynamic one that is constantly changing and evolving. This is a fact that is often overlooked, but it is one that has a profound effect on the way we think about the world and our place in it.

## Chapter 14

# Conclusions

This thesis addressed the use of field programmable devices in applications with timing performance constraints. FPLDs provide rapid *time-to-market* capabilities, and at a *low cost* – two key features for the success of ASICs. Existing FPLDs are used more for prototyping and for implementing circuits with non-critical timing specifications. With the increasing need for faster circuits, FPLDs and their accompanying software must be able to implement circuits with timing constraints on programmable devices; and this work addresses just such a need.

In Part I, we presented performance-driven synthesis techniques for existing LUT based FPGAs. Two issues were addressed independently - routing feasibility, and timing performance. We presented the **RFR** algorithm which performs synthesis (mapping), placement and routing in tandem within a simulated annealing algorithm. Experimental results showed how the approach reduced the number of unrouted nets. We showed how the performance-driven mapping and area-delay trade-off techniques of standard cell library-based mappers can be extended to LUTs. The concept of a two-input LUT primitive cell, which reduces the library size for LUT libraries by several orders of magnitude, was introduced. The special structure of the LUT library patterns was exploited to reduce effective library size even further, to  $O(K^2)$ , for a  $K$ -input LUT library. Experiments showed that this **dpmap** algorithm produces faster circuits, and provides useful area-delay trade-off features. **dpmap** can be extended in several directions. Libraries for the recently introduced LUT FPGAs need to be developed. A three-input LUT can also be used in addition to the 2-input LUT as a primitive cell. Integrating placement with library-based mapping can give better estimates of net delay (as compared to the linear delay model used in **dpmap**).



The initial decomposition into two-input nodes, which is the starting point for the **dpmmap** algorithm, is an open problem and merits further study.

In Part II, we introduced a new FPLD architecture, *Dharma*. *Dharma* is based on the concept of time-sharing of resources in an effort to have a fully routable, CAD friendly FPLD with predictable timing performance and silicon efficiency. Real-time reconfiguration of logic and routing resources is used to implement a circuit in a folded pipe-line fashion, pipe-lining at the gate level. We explained the operation of *Dharma* and discussed several possible variations of a basic *Dharma* architecture. We introduced a synthesis scheme for *Dharma* based on temporal partitioning and presented experimental results. Area and timing analyses of *Dharma* were done, and we showed that even a worst-case *Dharma* device has advantages, both in terms of area and timing. The *Dharma* architecture presents many interesting challenges and avenues for research. Firstly, a prototype device needs to be built. *Dharma* is a memory intensive design, amenable to many technologies. The DIA has to be made small and fast, and the *Copier* idea (see Section 10.2.5) may be helpful. Sparse crossbars, along with hard-wired connections give area and speed benefits, but require intelligent synthesis tools to ensure routability. A family of *Dharma* devices, each differing in DIA and DLM architecture can be fabricated, as shown in Figure 1.3. Secondly, as outlined in Chapter 13, development of logic synthesis algorithms for *Dharma* can give better timing performance and reduce resource utilization. Thirdly, the *Dharma* idea can be extended to the system level, instead of just at the device level, to allow large sections of a system to be reconfigured while other sections are performing useful tasks.

FPLDs have so far been used mainly to implement “glue logic”. Rapid time-to-market, and low cost are only two of the features of programmable devices that have been exploited so far. The third feature of field programmable devices is their ability to be reconfigured in real-time. The *Dharma* architecture exploited this feature to its advantage. But *Dharma* has only scratched the surface of potential uses for real-time reconfiguration. Properly put to use, real-time reconfiguration (*dynamic/virtual logic*, or *logic recycling*) can foster new computing paradigms, and create a revolution in logic design.

# Bibliography

- [Actel 91] *ACT family field programmable gate array databook*, Actel Corporation, March 1991.
- [Abouzeid 90] P. Abouzeid, K. Sakouti, G. Saucier and F. Poirot, "Multilevel synthesis minimizing the routing factor," *Proc. 27th ACM/IEEE Design Automation Conference*, pp. 365-368, 1990.
- [Aho 77] A. V. Aho, M. R. Garey and F. K. Hwang, "Rectilinear Steiner trees: efficient special-case algorithms," *Network*, vol. 7, pp. 37-58, 1977.
- [Alfke] Peter Alfke, Xilinx, Inc. Personal Communication.
- [Alford 89] R. C. Alford, *Programmable logic designer's guide*, Howard W. Sams and Co., 1989.
- [Altera 85] *Altera: EPLD handbook*, Altera Corporation, 1985.
- [Bhat 93] N. B. Bhat, K. Chaudhary and E. S. Kuh, *Performance-oriented fully routable dynamic architecture for a field programmable logic device*, Technical report, Electronics Research Laboratory, U. C. Berkeley, UCB/ERL Memo No. M93/42, June 1993.
- [Bhat 92a] N. B. Bhat and D. D. Hill, "Routable technology mapping for LUT FPGAs," *Proc. ICCD '92*, pp. 95-98, Oct. 1992.
- [Bhat 92] N. B. Bhat and D. D. Hill, "Routable technology mapping for FPGAs," *Proc. FPGA '92*, pp. 143-147, Feb. 1992.

- [Brayton 90] R. K. Brayton, G. D. Hachtel and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. of the IEEE*, vol. 78, no. 2, pp. 264-300, Feb. 1990.
- [Brayton 87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS: a multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1062-1081, Nov. 1987.
- [Brown 92] S. D. Brown, R. J. Francis, J. Rose and Z. G. Vranesic, *Field-programmable gate arrays*, Kluwer Academic Publishers, 1992.
- [Chaudhary 93] K. Chaudhary, Personal Communication.
- [Chaudhary 92] K. Chaudhary and M. Pedram, "A near optimal algorithm for technology mapping minimizing area under delay constraints," *Proc. 29th Design Automation Conference*, pp. 492-498, 1992.
- [Chen 92] K. C. Chen, "Logic minimization of lookup-table based FPGAs," *Proc. FPGA '92*, pp. 71-76, Feb. 1992.
- [Cohoon 90] J. P. Cohoon, D. S. Richard and J. S. Salowe, "An optimal Steiner tree algorithm for a net whose terminal lie on the perimeter of a rectangle," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 4, pp. 398-407, April 1990.
- [Cong 92] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA design," *Proc. Int. Conf. CAD (ICCAD-92)*, pp. 48-53, Nov. 1992.
- [Detjens 87] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "Technology mapping in MIS," *Proc. Int. Conf. CAD (ICCAD-87)*, pp. 116-119, Nov. 1987.
- [Donath 88] W. E. Donath, *Logic partitioning in Physical design automation of VLSI systems*, B. Preas and M. Lorenzetti, editors. The Benjamin/Cummings Publisher Company, Menlo Park, California 94025, 1988.

- [Ercolani 91] S. Ercolani and G. De Micheli, "Technology mapping for electrically programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 234-239, 1991.
- [Filo 91] D. Filo, J. C. Yang, F. Mailhot and G. De Micheli, "Technology mapping for a two-output RAM-based field programmable gate array," *Proc. EDAC*, pp. 534-538, Feb. 1991.
- [Fujita 91] M. Fujita and Y. Matsunaga, "Multi-level minimization based on minimal support and its application to the minimization of look-up table type FPGAs," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 560-563, Nov. 1991.
- [Francis 92] R. J. Francis, "A tutorial on logic synthesis for lookup-table based FPGAs," *Proc. Int. Conf. CAD (ICCAD-92)*, pp. 40-47, Nov. 1992.
- [Francis 91a] R. Francis, J. Rose and Z. Vranesic, "Technology mapping of lookup table-based FPGAs for performance," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 568-571, Nov. 1991.
- [Francis 91] R. Francis, J. Rose and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 227-233, 1991.
- [Francis 90] R. Francis, J. Rose and K. Chung, "Chortle: a technology mapping program for lookup table-based field programmable gate arrays," *Proc. 27th Design Automation Conference*, pp. 613-619, 1990.
- [Garey 77] M. R. Garey and D. S. Johnson, "The rectilinear Steiner tree problem is NP-complete," *SIAM Journal of Applied Mathematics*, vol. 32, no. 4, pp. 37-58, 1977.
- [Hanan 66] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM Journal of Applied Mathematics*, vol. 14, no. 2, pp. 255-265, 1966.
- [Hastie 90] N. Hastie and R. Cliff, "The implementation of hardware subroutines on field programmable gate arrays," *Proc. CICC*, pp. 31.4.1-31.4.4, May 1990.

- [Ho 89] J. Ho, G. Vijayan and C. K. Wong, "A new approach to the rectilinear steiner tree problem," *Proc. 26th ACM/IEEE Design Automation Conference*, pp. 161-166, 1989.
- [Ho 90] J. Ho, G. Vijayan and C. K. Wong, "New algorithms for the rectilinear Steiner tree problem," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 2, pp. 185-193, Feb. 1990.
- [Hodges 88] D. Hodges and H. G. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw Hill, Second Edition, 1988.
- [Horng 92] C-H Horng, Vice-President, Research and Development, I-Cube Design Systems, Inc. Personal Communication. I-Cube markets large (80 by 80) single-chip crossbar switches.
- [Hsieh 90] H-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey and R. Kanazawa, "Third-generation architecture boosts speed and density of field-programmable gate arrays," *Proc. Custom Integrated Circuits Conference (CICC 90)*, pp. 31.2.1-31.2.7, 1990.
- [Hwang 79] F. K. Hwang, "An  $O(n \log n)$  algorithm for suboptimal rectilinear Steiner trees," *IEEE Transactions on Circuits and Systems*, vol. CAS-26, no. 1, pp. 75-77, Jan. 1979.
- [Karplus 91a] K. Karplus, "Xmap: a technology mapper for table-lookup field-programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 240-243, 1991.
- [Karplus 91] K. Karplus, "Amap: a technology mapper for selector-based field-programmable gate arrays," *Proc. 28th Design Automation Conference*, pp. 244-247, 1991.
- [Kernighan 70] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291-308, 1970.

- [Keutzer 87] K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," *Proc. 24-th Design Automation Conference*, pp. 341-347, 1987.
- [Kim 93] T-S. Kim, W. Burleson and M. Ciesielski, "Logic restructuring for wave-pipelined circuits," *Proc. International Workshop on Logic Synthesis*, Tahoe City, CA, pp. 4b1-4b14, May 1993.
- [Kouloheris 91] J. L. Kouloheris and A. El Gamal, "FPGA performance versus cell granularity," *Proc. CICC*, pp. 6.2.1-6.2.4, May 1991.
- [Kuh 90] E. S. Kuh and T. Ohtsuki, "Recent advances in VLSI layout," *Proc. of the IEEE*, vol. 78, no. 2, pp. 237-263, Feb. 1990.
- [Lee 76] J. H. Lee, N. K. Base and F. K. Hwang, "Use of Steiner's problem in suboptimal routing in rectilinear metric," *IEEE Transactions on Circuit and System*, vol. CAS-23, no. 7, pp. 470-476, July 1976.
- [Lee 88] K-W. Lee and C. Sechen, "A new global router for row-based layout," *Proc. Int. Conf. CAD (ICCAD-88)*, pp. 180-183, Nov. 1988.
- [Masson 77] G. M. Masson, "Binomial switching networks for concentration and distribution," *IEEE Transactions on Communication*, COM-25, no.9, pp. 873-883, Sept. 1977.
- [MCNC] S. Yang, "Logic synthesis and optimization benchmarks user guide - version 3.0," Microelectronic Center of North Carolina, Jan. 1991.
- [Murgai 91a] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Performance directed synthesis for table look up programmable gate arrays," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 572-575, Nov. 1991.
- [Murgai 91] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 564-567, Nov. 1991.

- [Murgai 90] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," *Proc. 27th Design Automation Conference*, pp. 620-625, 1990.
- [ORCA 93] *Optimized Reconfigurable Cell Array (ORCA) Series Field-Programmable Gate Arrays*, AT&T Microelectronics, Advance Data Sheet, Feb 1993.
- [Park 91] I-C. Park, C-M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," *Proc. 28th Design Automation Conference*, pp. 680-685, 1991.
- [Pedram 91a] M. Pedram and N. Bhat, "Layout driven logic decomposition/restructuring," *Proc. Int. Conf. CAD (ICCAD-91)*, pp. 134-137, Nov. 1991.
- [Pedram 91] M. Pedram and N. Bhat, "Layout driven technology mapping," *Proc. 28th Design Automation Conference*, pp. 99-105, 1991.
- [Quickturn 93] *A new approach to design verification and system integration: computer aided prototyping*, Quickturn Systems Inc., 1993.
- [Saldanha 89] A. Saldanha, A. Wang, R. Brayton and A. Sangiovanni-Vincentelli, "Multi-level logic simplification using don't cares and filters," *Proc. 26th Design Automation Conference*, 1989.
- [Sawkar 92] P. Sawkar, D. Thomas, "Area and delay mapping for table look-up based field programmable gate arrays," *Proc. 29th Design Automation Conference*, pp. 368-373, 1992.
- [Schlag 92] M. Schlag, J. Kong and P. K. Chan, "Routability-driven technology mapping for lookup-table-based FPGAs," *Proc. ICCD*, pp. 86-90, Oct. 1992.
- [Schlichtmann 92] U. Schlichtmann, F. Brglez and M. Hermann, "Characterization of Boolean functions for rapid matching in EPGA technology mapping," *Proc. 29th Design Automation Conference*, pp. 374-379, 1992.