Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory

BY

Steven J.E. Wilton

A Thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy in the
Department of Electrical and Computer Engineering,
University of Toronto

Abstract

Architectures and Algorithms for Field-Programmable
Gate Arrays with Embedded Memory

Doctor of Philosophy, 1997
Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of Toronto

Recent dramatic improvements in integrated circuit fabrication technology have led to Field-Programmable Gate Arrays (FPGAs) capable of implementing entire digital systems, as opposed to the smaller logic circuits that have traditionally been targeted to FPGAs. Unlike the smaller circuits, these large systems often contain memory. Architectural support for the efficient implementation of memory in next-generation FPGAs is therefore crucial.

This dissertation examines the architecture of FPGAs with memory, as well as algorithms that map circuits into these devices. Three aspects are considered: the analysis of circuits that contain memory as well as the automated random generation of such circuits, the architecture and algorithms for stand-alone configurable memory devices, and architectures and algorithms for the embedding of memory arrays in an FPGA.

We first present statistics gathered from 171 circuits with memory. These statistics include the number of memories in each circuit and the width and depth of these memories. We identify common interconnect patterns between memory and logic. These statistics are then used to develop a circuit generator that stochastically generates realistic circuits with memory that can be used as benchmark circuits in architectural studies.

Next, we consider the architecture of a stand-alone configurable memory that is flexible enough to implement memory configurations with different numbers of memories, memory widths and depths. Instrumental in this work is the algorithms that map memory configurations to the device. These algorithms are used in an experimental framework to investigate the effect of various architectural parameters on the flexibility, chip area, and access time of the configurable memory.

Finally, the architecture of an FPGA containing both embedded memory arrays and

logic elements is considered, along with new automatic placement and routing algorithms that map circuits to the FPGA. We show that only 4 switches per memory block pin are required in the interconnection between the memory arrays and logic elements, and even lower in FPGAs with four or fewer memory arrays. In addition, we show that by providing direct connections between memory arrays, the FPGA density can be improved slightly, and the average memory access time can be improved by as much as 25%.

Acknowledgements

First and foremost, I'd like to thank my supervisors Jonathan Rose and Zvonko Vranesic for their technical advice, moral support, financial support, and the friendship that developed over my six years in Toronto. They have left a lasting impression that was invaluable during my time here, and will remain so once I return to BC and embark on my own research career.

I've also enjoyed working with all the members of the Computer and Electronics Groups. Although the names are far too numerous to list here, I'd like to thank everyone for making these six years so enjoyable.

I would also like to thank Jason Anderson, Vaughn Betz, Mike Hutton, and Mohammed Khalid for their valuable technical discussions during our weekly group meetings.

The FPGA community has been extremely supportive of this project. I'd like to thank employees at Altera, Xilinx, Lucent Technologies and Actel for giving me the opportunity to present my work and ensuring that my work remained industrially relevant. I'd especially like to thank Kerry Veenstra of Altera for supplying the results of a circuit survey and Igor Kostarnov from Hewlett-Packard Labs for providing the Scribbler circuit.

Financial support for this project was provided by the Natural Sciences and Engineering Research Council of Canada, MICRONET, the University of Toronto, and the Walter C. Sumner Memorial Foundation.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Requirements of an FPGA Memory Architecture 3

1.3 Coarse-Grained and Fine-Grained RAM 4

1.4 Research Approach and Dissertation Outline 5

2 Previous Work **7**

2.1 Implementing Logical Memories 7

2.2 Memory Resources in Gate Arrays 9

2.3 Fine-Grained Memory Resources in FPGAs 10

2.3.1 Xilinx 4000 Architecture 10

2.3.2 Lucent Technologies ORCA Architecture 13

2.3.3 Crosspoint Architecture 14

2.4 Coarse-Grained Memory Resources in FPGAs 15

2.4.1 Altera FLEX 10K Architecture 15

2.4.2 Actel 3200DX Architecture 17

2.4.3 Lattice ispLSI 6192 Architecture 17

2.4.4 Altera FLASHlogic 18

2.4.5 Kawasaki Steel FPGA 18

2.4.6 Plus Logic Architecture 18

2.5 Chip Express Laser-Programmable Gate Array 19

2.6 FiRM Architecture 19

2.7 Summary 21

3 Circuit Analysis and Generation **22**

3.1 Motivation . . . 22
3.2 Circuit Analysis . . . 24
3.2.1 Logical Memory Configurations . . . 24
3.2.2 Logical Memory Clustering . . . 26
3.2.3 Interconnect Patterns . . . 30
3.2.4 Summary of Circuit Analysis . . . 35
3.3 Stochastic Circuit Generator . . . 36
3.3.1 Choosing the Logical Memory Configuration . . . 36
3.3.2 Choosing Memory/Logic Interconnect Patterns and Number of Data-in and Data-out Subcircuits . . . 39
3.3.3 Generating Logic Subcircuits . . . 41
3.3.4 Connecting Logic and Memories . . . 44
3.3.5 Additional Constraints . . . 44
3.4 Summary . . . 45

4 Stand-alone Configurable Memories: Architecture and Algorithms 46
4.1 Architectural Framework . . . 47
4.1.1 L1 Data Mapping Block . . . 48
4.1.2 L2 Data Mapping Block and Address Mapping Block . . . 50
4.1.3 Wide Mapping Blocks . . . 53
4.2 Algorithms . . . 54
4.2.1 Trivial Checks . . . 55
4.2.2 Logical-to-Physical Mapping . . . 56
4.2.3 Place and Route . . . 67
4.2.4 Mapping Results . . . 71
4.3 Summary . . . 73

5 Evaluation of Stand-Alone Architectures 74
5.1 Methodology . . . 75
5.2 Area Model . . . 76
5.2.1 Array Area Model . . . 76
5.2.2 Routing Area Model . . . 78
5.2.3 Total Area . . . 80

5.2.4 Technology and layout parameters 80
5.2.5 Area Measurements 80
5.3 Access Time Model 81
5.3.1 Array Delay 82
5.3.2 Routing Delay 83
5.3.3 Delay Results 83
5.4 Architectural Study 84
5.4.1 Number of Basic Arrays 84
5.4.2 L1 Mapping Block Capability 87
5.4.3 Data Bus Granularity 89
5.4.4 L2 Switch Patterns 91
5.5 External Pin Assignment Flexibility 93
5.6 Conclusions 97

6 Embedded Arrays: Architecture and Algorithms 99
6.1 Basic Architecture 100
6.1.1 Memory Resources 100
6.1.2 Logic Resources 102
6.1.3 Memory/Logic Interconnect 104
6.2 Implementation Tools 106
6.2.1 Placement Program 108
6.2.2 Routing Program 111
6.2.3 Validation of Placement and Routing Tools 118
6.3 Summary 122

7 The Memory/Logic Interface 124
7.1 Memory/Logic Interface Flexibility 125
7.1.1 Methodology 126
7.1.2 Effect of F_m on FPGA with 16 Memory Blocks 133
7.1.3 Effect of F_m on FPGAs with Fewer Memory Blocks 139
7.1.4 Effect of Connection Block Flexibility 144
7.1.5 Memory at Edge of FPGA 144
7.1.6 Summary of Flexibility Results 147

7.2 Enhancements to Support Memory-to-Memory Connections 148
7.2.1 Pins on Both Sides of Arrays 148
7.2.2 Dedicated Memory-to-Memory Switches 152
7.3 Scribbler Circuit 162
7.4 Conclusions 167

8 Conclusions and Future Work **168**
8.1 Dissertation Summary 168
8.2 Future Work 169
8.2.1 Architecture Studies 169
8.2.2 Algorithm Studies 170
8.2.3 Circuit Analysis Studies 171
8.2.4 Long-Term Research 172

List of Tables

1.1 Example systems. 3

2.1 Gate arrays with memory. 9
2.2 Xilinx CLBs required to implement logical memories (from [1]). 12
2.3 Summary of memory resources in commercial FPGAs. 21

3.1 Information available for example circuits. 25
3.2 Additional logical memory statistics. 26
3.3 Additional cluster statistics. 30
3.4 Data-in and data-out interconnection type statistics. 35
3.5 Probability distribution functions used to select the number of clusters and logical memories per cluster. 36
3.6 Probability distribution functions used to select depth and width of each memory. 38
3.7 Probability distribution for choosing interconnect type. 39
3.8 Probability distribution for choosing number of data-in and data-out subcircuits. 40
3.9 MCNC circuits used as logic subcircuits (* = only used as data sink subcircuit). 42
3.10 Default values for β and γ. 44
3.11 SCIRC constraints. 45

4.1 Architectural parameters. 48
4.2 Symbol values for examples of Figure 4.10 and 4.11 60
4.3 Array and data bus requirements for 128x16. 63
4.4 Array and data bus requirements for 896x3. 63
4.5 Breakdown of how many organizations remain after elimination process. . . 64

4.6 Symbol values for examples of Figure 4.15 66
4.7 Mapping results for FiRM architecture. 72
4.8 Mapping results: $B = 8192$, $N = 8$, $M = Q = 4$, $W_{\text{eff}} = \{1, 2, 4, 8\}$. 72

5.1 Technology parameters for the area model. 80
5.2 Area model predictions (results in mbe's). 81
5.3 Model predictions ($B = 8$Kbits, $N = 8, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$). . . 83
5.4 L2 switch pattern results ($B = 8K, N = 8, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$). . 92
5.5 L2 switch pattern results ($B = 64K, N = 16, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$). 92

6.1 Architectural parameters for embedded memory resources. 102
6.2 Minimum number of tracks/channel for various CAD flows. 119
6.3 Comparison of directional and maze-routing algorithm results. 120
6.4 CPU time requirements^2 for router. 121
6.5 Track count using disjoint and non-disjoint switch blocks. 121
6.6 Minimum number of tracks/channel. 122
6.7 CPU time requirements for router. 122

7.1 Circuit statistics. 127
7.2 Architecture statistics. 129
7.3 Net statistics. 140
7.4 Statistics for Scribbler circuit. 162

List of Figures

2.1 Implementing a 256x16 logical memory using two 256x8 basic arrays. 8
2.2 Implementing a 2Kx1 logical memory using two 1Kx1 basic arrays. 8
2.3 Applied Micro Circuits gate array floorplan from [2]. 9
2.4 Mitsubishi gate array floorplan from [3]. 10
2.5 One CLB configured as memory. 11
2.6 XC4000E CLB configured as a 16x1 two-read port, one-write port memory. 11
2.7 One bit slice of a 128x8 memory implemented using Xilinx CLBs. 12
2.8 PFU configured as 16x4 memory in ORCA 2CA and 2TA. 13
2.9 Crosspoint CP20K architecture. 14
2.10 Combining RLTs to construct a larger array. 15
2.11 Floorplan of Altera FLEX 10K architecture. 16
2.12 Embedded array block in an Altera FLEX 10K CPLD. 16
2.13 512x24 memory implemented using six EABs. 17
2.14 Floorplan of Chip Express CX2100 LPGA. 19
2.15 FiRM interconnect structure. 20

3.1 Distribution of number of logical memories required. 25
3.2 Distributions of logical memory widths and depths. 26
3.3 Logical memory connected to data-in, data-out, and address subcircuits. . . 28
3.4 Example clusters. 28
3.5 Forward error correction decoder datapath from [4]. 29
3.6 Cluster statistics. 30
3.7 Logic subcircuit statistics (per cluster). 31
3.8 A single cluster showing data-in and data-out networks. 31
3.9 Point-to-point interconnect patterns for data-in network. 32

3.10 Shared-connection interconnect patterns for data-in network. 33
3.11 Point-to-point pattern with no shuffling. 34
3.12 Resulting distribution of total number of logical memories. 37
3.13 Resulting distribution of total number of bits. 38
3.14 Example of connecting data-in subcircuit to a logical memory. 41

4.1 General architecture for a standalone FCM. 48
4.2 L1 data mapping block. 49
4.3 Two example mappings. 50
4.4 Level 2 data and address mapping block topology ($N = M = Q = 8$). . . . 52
4.5 Implementation of two examples in Figure 4.3. 52
4.6 Two-stage address mapping block. 53
4.7 Bi-directional switches in two-stage L2 data-mapping block. 53
4.8 CAD flows. 55
4.9 Two possible implementations of 896x3. 56
4.10 Netlist for {896x3, 5Kx1}. 58
4.11 Netlist for {896x3, 128x16}. 58
4.12 Exhaustive logical-to-physical mapping algorithm for one logical memory . 61
4.13 First heuristic logical-to-physical mapping algorithm for one logical memory. 62
4.14 Algorithm to eliminate unnecessary data-widths for logical memory i 64
4.15 Efficient way to implement 896x3 that is missed by heuristic algorithm. . . 66
4.16 Solution for {896x3, 5Kx1}. 70
4.17 Partial solution for {128x8, 128x8, 512x12}. 70

5.1 Methodology for standalone memory experiments. 75
5.2 Floorplan of a single array (not to scale). 77
5.3 Assumed layout of memory architecture. 78
5.4 Assumed layout of memory architecture, 2 stages. 78
5.5 Access time predictions: $B = 64K, N = 32, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8\}$. . 84
5.6 Number of failures as a function of number of arrays (N). 86
5.7 Access time as a function of number of blocks (N). 87
5.8 Chip area as a function of number of arrays (N). 87
5.9 Number of failures as a function of L1 mapping block capability. 88

5.10 Access time as a function of L1 mapping block capability. 88
5.11 Chip area as a function of L1 mapping block capability. 88
5.12 Failures as a function of data bus width: $B = 64K$, $N = 16$, 128 data pins. . 90
5.13 Delay/area as a function of data bus width: $B = 64K$, $N = 16$, 128 data pins. 90
5.14 L2 data block/address block switch patterns considered. 92
5.15 A single logical memory that uses 4 data buses and 1 address bus. 94
5.16 FiRM switch pattern for address and L2 data mapping blocks. 95
5.17 Best switch patterns if one switch-set is added. 95
5.18 Best switch pattern if two switch-sets are added. 96
5.19 Bus assignment flexibility results. 96
5.20 Best switch patterns if 3 and 6 switch-sets are added. 97

6.1 Floorplan showing memory and logic blocks. 101
6.2 An alternative floorplan with arrays distributed throughout the FPGA (not considered in this dissertation). 101
6.3 Three different switch blocks. 102
6.4 Routing a net from A to B. 103
6.5 Correspondence between terminal label and physical switch block pin. . . . 104
6.6 Memory/logic interconnect structure. 105
6.7 Memory/logic interconnect block. 106
6.8 CAD flow. 107
6.9 Pseudo-code for simulated annealing algorithm. 109
6.10 Pseudo-code for algorithm to route a single net. 112
6.11 Pseudo-code for algorithm to route all nets. 113
6.12 Nodes visited by the basic maze routing algorithm. 114
6.13 Example in which pruning should be limited to moves that are further in both the X and Y dimensions. 115
6.14 Example in which the cost of a node should reflect both the distance from the source and distance to the destination. 115
6.15 Nodes visited by the directional search algorithm. 116
6.16 Modified pseudo-code for algorithm to route a single net. 117
6.17 Modified pseudo-code for algorithm to route all nets. 118

7.1 Memory/logic interconnect block example from Chapter 6. 125
7.2 Methodology for memory/logic interconnect block experiments. 130
7.3 Bi-directional re-powering switch. 132
7.4 Average track requirement as a function of F_m for FPGA with 16 arrays. . 134
7.5 Bad case for memory/logic interconnect block of Chapter 6. 134
7.6 No. of programmable connections as a function of F_m for FPGA with 16 arrays. 138
7.7 Delay as a function of F_m for FPGA with 16 arrays. 139
7.8 Average track requirement as a function of F_m. 140
7.9 A net connected to three memory blocks: three regions of low flexibility. . . 141
7.10 Effect of removing memory-to-memory connections. 142
7.11 Number of programmable connections as a function of F_m. 143
7.12 Delay as a function of F_m. 143
7.13 Track requirement results for two values of F_c. 145
7.14 Number of programmable connections for two values of F_c. 145
7.15 Floorplan of FPGA with memory arrays at one edge. 146
7.16 Track requirements results for FPGA with memory arrays at one edge. . . . 146
7.17 Area results for FPGA with memory arrays at one edge. 147
7.18 Standard memory/logic interconnect structure for $F_m = 1$. 149
7.19 Circuitous route on standard architecture. 149
7.20 Enhanced memory/logic interconnect structure for $F_m = 1$. 150
7.21 Direct route on enhanced architecture. 150
7.22 Track requirement results for architecture of Figure 7.20. 151
7.23 An example where the router does not take advantage of extra switches. . . 151
7.24 Dedicated memory-memory connection architecture. 153
7.25 Enhanced architecture example. 154
7.26 Original architecture example. 154
7.27 Routing results assuming standard router. 155
7.28 Routing a logic net with and without memory-to-memory switches. 156
7.29 Example of a memory net that uses an extra memory-to-memory switch. . . 158
7.30 Track results for an 8-array FPGA with memory-to-memory switches. . . . 159
7.31 Track results for an 16-array FPGA with memory-to-memory switches. . . . 159

7.32 Area results for an 8-array FPGA with memory-to-memory switches. 160
7.33 Area results for an 16-array FPGA with memory-to-memory switches. . . . 160
7.34 Delay results for an 8-array FPGA with memory-to-memory switches. . . . 161
7.35 Delay results for an 16-array FPGA with memory-to-memory switches. . . . 161
7.36 Track requirement results for Scribbler. 163
7.37 Area results for Scribbler. 163
7.38 Delay results for Scribbler. 163
7.39 Track requirement results for Scribbler assuming memory-to-memory switches. 164
7.40 Area results for Scribbler assuming memory-to-memory switches. 164
7.41 Delay results for Scribbler assuming memory-to-memory switches. 164
7.42 One cluster in Scribbler. 165
7.43 One cluster in Scribbler after logical-to-physical mapping. 166
7.44 Placement of four memory blocks from Figure 7.43. 166

8.1 An alternative floorplan with arrays distributed throughout the FPGA. . . 170

List of Symbols

Stand-Alone Architectural Parameters:

B	Total memory bits
N	Number of arrays
M	Number of external data buses
Q	Number of external address buses
W_{nom}	Nominal width of each array and width of each data bus
W_{eff}	Set of allowable data widths of each array

Embedded Array Architectural Parameters:

B	Total memory bits
N	Number of arrays
W_{eff}	Set of allowable data widths of each array
M	Number of pins in each memory block
G	Number of logic blocks
W	Vertical tracks per logic routing channel
F_m	Number of tracks to which each memory block pin can connect
F_c	Number of tracks to which each logic block pin can connect
F_s	Number of choices offered to each incident track by a switch block
V	Vertical tracks between each memory array
R	Logic blocks per memory block in the horizontal dimension

Circuit and Implementation Parameters:

z	Number of logical memories
d_i	Depth of logical memory i
w_i	Width of logical memory i
n	Total number of arrays to implement all logical memories
n_i	Number of arrays required to implement logical memory i
s_i	Number of mux-groups in implementation of memory i
$n_{i,j}$	Arrays in mux-group j in implementation of memory i
$e_{i,j}$	Effective data width of each array in mux-group j of implementation of logical memory i
g	Number of logic blocks in circuit

Chapter 1

Introduction

1.1 Motivation

Since their introduction in 1985, Field-Programmable Gate Arrays (FPGAs) have rapidly become the implementation medium of choice for many digital circuit designers. The reconfigurable nature of FPGAs provides risk-free large-scale integration that has been applied in areas as diverse as telecommunications, high-speed graphics, and digital signal processing. Unlike Mask-Programmed Gate Arrays (MPGAs), which must be personalized by the MPGA vendor, FPGAs can be programmed by the user in minutes. For small and medium volumes, this reduces the cost, and shortens the time-to-market.

Unfortunately, there is an area and speed penalty associated with user-configurability. Unlike MPGAs, in which circuit elements are connected using metal wires, in an FPGA, programmable switches are used to connect circuit elements. These programmable switches add resistance and capacitance to all connections within a circuit, lowering the achievable clock frequency. The switches also require significant chip area, reducing the amount of logic on each device. In some FPGAs, Static RAM (SRAM) bits are required to control the programming switches, reducing the number of circuit elements even further. For the most part, this has limited FPGAs to the implementation of relatively small logic subcircuits, often the "glue-logic" portions of larger systems.

Recent years, however, have seen dramatic improvements in processing technology. Today, 0.5μm and 0.35μm processes are common, 0.25μm processes are becoming avail-

able [5, 6], and even smaller feature sizes are on the horizon. These smaller feature sizes have lead to impressive improvements in the density of integrated circuits (ICs), which, in turn, have had a profound impact on the possible applications and design of ICs.

The impact of improving process technology is very evident in the evolution of FPGAs. Recently, FPGA vendors have introduced devices capable of implementing relatively large circuits and systems. These large systems are quite different than the smaller logic sub-circuits that have traditionally been the target of FPGAs. One of the key differences is that these systems often contain memory. Therefore, next-generation FPGAs must be able to efficiently implement memory as well as logic. If they can not, the implementation of such a system would require both FPGAs (for the logic portion of the system) and separate memory chips. On-chip memory has several advantages:

- Implementing memory on-chip will likely decrease the number of chips required to fully implement a system, reducing the system cost.
- Implementing memory and logic on separate chips will often limit the achievable clock rate, since external pins (and board-level traces) must be driven with each memory access. If the memory access time is part of the critical path delay of the circuit, on-chip memory will allow a shorter clock period.
- For most FPGA packaging technologies, as the devices get larger, the number of logic elements grows quadratically with the edge-length of the chip. The number of I/O pins, however, grows only linearly. Thus, the availability of I/O pins is increasingly becoming a problem. This problem is aggravated if an FPGA is connected to an off-chip memory, since many I/O pins on the FPGA must be devoted to address and data connections. If the memory is implemented on-chip, these pins can be used for other purposes.

Although several FPGA vendors have included support for memory on their latest devices, there has been no published work that investigates how this memory support can best be provided. In this dissertation, we investigate the requirements of such an FPGA, and evaluate architectures for supporting memory in next-generation FPGAs.

System	Memory Requirements[1]
Graphics Chip [7]	eight 128x22, two 16x27
Neural Networks Chip [8]	16x80, 16x16
Translation Lookaside Buffer [3]	two 256x69 (buffers), 16x18 (register file)
Proof-of-concept Viterbi decoder [9]	three 28x16, one 28x3
Fast Divider [10]	2048x56, 4096x12 (ROM)
Communications Chip #1	two 1620x3, two 168x12, two 366x11
Communications Chip #2	six 88x8, one 64x24
Communications Chip #3	one 192x12

Table 1.1: Example systems.

1.2 Requirements of an FPGA Memory Architecture

Since FPGA memory will be used in many different contexts, it must be flexible. Table 1.1 shows a number of circuits described in recent circuits conferences and journals as well as three circuits obtained from a Canadian telecommunications company; each of these circuits requires a different number of memories and different memory sizes. In this dissertation, we refer to the memory requirements of a single application circuit as a *logical memory configuration*. Each independent memory within a logical memory configuration is referred to as a *logical memory*. If two more more memories share address connections (i.e. receive the same address from the logic at all times) they are considered to be a single logical memory. Many configurations contain more than one logical memory; for example, the Viterbi decoder in Table 1.1 requires four logical memories. The specification of these four memories, with their widths, speeds, and any other special requirements (such as dual-port access) make up the circuit's logical memory configuration. A good FPGA will be able to implement circuits with a wide variety of logical memory configurations.

Another element of flexibility not reflected in Table 1.1 is that the logical memories within a circuit will be used in many different contexts, and will therefore communicate with each other and with the logic in many different ways. Some circuits require a single large bank of memory that is connected directly to logic subcircuits. Other circuits might contain smaller memories connected to a common bus; this bus then might drive (or be driven by) one or more logic subcircuits. A third type of circuit might require many small

[1] In this dissertation, a memory written mxn consists of m words of n bits each.

memories distributed throughout the circuit, each connected to its own logic subcircuit. Again, a good FPGA will be able to implement circuits that interact with memory in many different manners.

Flexibility is rarely free. In general, the more flexible an FPGA architecture, the more programmable switches and programming bits are required. Programmable switches add delay to the paths within a circuit implementation; if these paths are part of the circuit's critical path, the achievable clock speed will suffer. Similarly, the extra switches and programming bits will use area that could otherwise be used for additional circuit elements. The design of a good FPGA architecture involves finding a balance between area, speed, and flexibility.

1.3 Coarse-Grained and Fine-Grained RAM

There are two approaches to creating an FPGA that can implement both memory and logic:

Fine-Grained: It is well known that logic can be efficiently implemented using small (4-input) lookup tables [11]. One approach to implementing RAM is to allow the user to optionally use each of these 4-input lookup tables as a 16-bit RAM. These small RAMs can then be combined to form larger logical memories [12, 13].

Coarse-Grained: A second approach is to use a heterogeneous architecture consisting of two types of resources: small lookup tables to implement logic and larger arrays to implement memory [14, 15].

Commercial examples of each type exist and are described in Chapter 2.

Consider the fine-grained approach. Since each lookup table only contains 16 bits of storage, many are needed to implement large logical memories. In addition, large address decoders and data multiplexors are needed, which require even more lookup tables. Each of these lookup tables has routing and other overhead. Clearly, large memories would quickly fill this sort of FPGA, leaving little room left to implement logic. In addition, connections between the lookup tables must be made using programmable interconnect; this results in longer and less predictable memory access times.

In a coarse-grained architecture, large logical memories can be implemented using only a few memory arrays. In this case, the overhead of each array is amortized over many more bits, leading to much denser and faster memory implementations. In this dissertation, we concentrate on coarse-grained architectures.

There are two significant drawbacks to the coarse-grained approach. First, heterogeneity implies that the chip is partitioned into logic and memory resources when manufactured. Since different circuits have different memory requirements, this "average-case" partitioning may lead to inefficient implementations of logic-intensive or memory-intensive circuits. This inefficiency can be reduced by using unused memory arrays to implement logic; this possibility is beyond the scope of this dissertation.

The second drawback of the coarse-grained approach is that the associated CAD algorithms are likely to be more complex than their counterparts for homogeneous architectures, since they must consider both logic and memory resources simultaneously. In the placement phase, the placements of the logic blocks must influence the placement of the memory blocks, and vice-versa. In the routing phase, the tool must be aware of the differences in flexibility in the routing around the memory and logic blocks. The technology-mapping phase for the heterogeneous architectures is also difficult, since, as will be shown in the next chapter, the memory blocks in these architectures often support multiple memory widths, giving an additional degree of freedom that the technology-mapper must explore. This dissertation addresses all of these issues.

1.4 Research Approach and Dissertation Outline

There are three parts to this research: circuit characterization, stand-alone configurable memory architectures, and embedded memory architectures.

We first performed an extensive analysis of 171 large circuits that contain memory. The statistics gathered include the number of logical memories in each circuit and the widths and depths of the memories. We also examined how the logical memories were connected to the logic portions of each circuit, and identified common interconnect patterns. The results of this analysis were then used to develop a stochastic circuit generator that generates realistic circuits containing both logic and memory. The analysis and resulting circuit generator are

described in Chapter 3.

During the next phase of the research, we investigated configurable memory architectures that are flexible enough to support a wide variety of logical memory configurations, but are fast and small enough to be suitable for inclusion in an FPGA. To isolate the issues associated with the memory architecture, we first focused on *stand-alone* architectures in which the address and data pins are connected directly to I/O pads. Chapter 4 describes such an architecture, as well as algorithms that map a logical memory configuration to the physical device. Chapter 5 investigates the effects of varying various architectural parameters on the device's flexibility, chip area, and access time.

During the final phase of this research, we examined how the configurable memory architecture can be embedded into an FPGA. An architecture containing both logic and memory resources is presented in Chapter 6, along with algorithms that map circuits to the FPGA. One of the key components of such a heterogeneous architecture is the interface between the memory and logic resources. It is vital that the interface be flexible enough to support the many different contexts in which the memories will be used, but small and fast enough that efficient circuit implementations are still possible. In Chapter 7, we focus on the memory/logic interface, and experimentally investigate the effects of various architectural parameters on the routability, area, and speed of the resulting FPGA.

Before any of this research is presented, however, the next chapter puts our work in context by describing commercial FPGAs with memory as well as other related work.

Chapter 2

Previous Work

Virtually every major FPGA company offers FPGAs with some memory capabilities. These existing devices are the focus of this chapter. An overview of the approach taken in standard gate arrays, FPGAs, and a Laser-Programmable Gate Array (LPGA) family is presented in Sections 2.2 to 2.5. Section 2.6 describes an experimental flexible memory implementation. We start, however, by discussing how these devices obtain their required flexibility.

2.1 Implementing Logical Memories

One of the key requirements of any configurable memory architecture is that it must be flexible enough to implement a wide variety of logical memories. The underlying approach for achieving flexibility is the same across all the architectures that will be presented.

Each architecture that will be discussed consists of a collection of basic arrays; the sizes of these arrays range from 16 bits to 2 Kbits. The first way flexibility is achieved is that, in many of the architectures, the aspect ratio of each basic array is configurable. For example, each 2-Kbit array in the Altera 10K architecture (described in Section 2.4.1) can be configured to one of 256x8, 512x4, 1Kx2, or 2Kx1.

The second way flexibility is achieved applies to all architectures with more than one basic array. In these architectures, the arrays can be combined in a programmable manner to implement larger logical memories. To implement wide logical memories, the basic arrays can be combined "horizontally" by sharing address lines between two or more arrays.

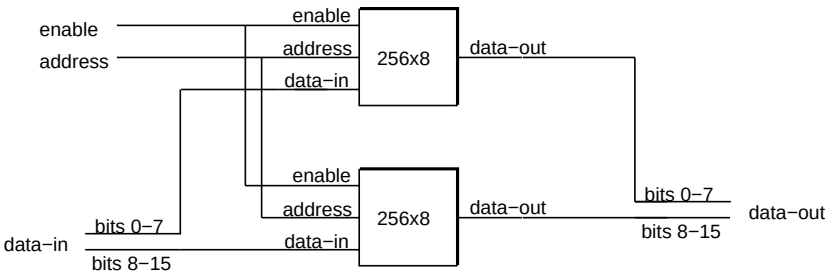


Figure 2.1: Implementing a 256x16 logical memory using two 256x8 basic arrays.

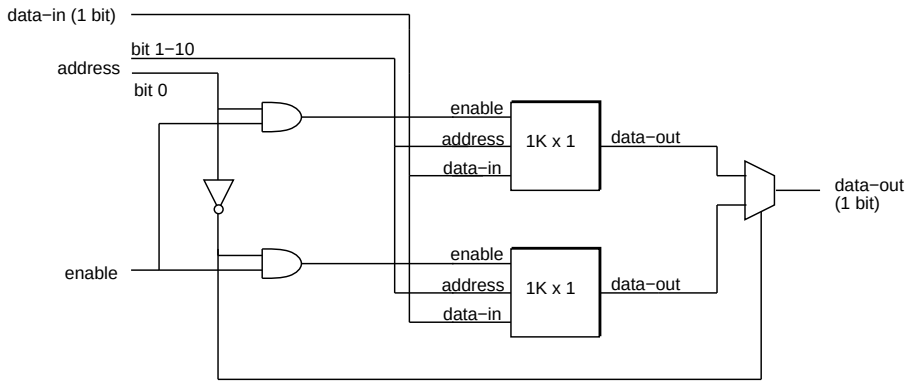


Figure 2.2: Implementing a 2Kx1 logical memory using two 1Kx1 basic arrays.

Figure 2.1 shows such an example in which a 256x16 logical memory is implemented using two 256x8 basic arrays. The enable and address lines are shared between the two arrays, but each array implements a separate 8-bit slice of data. To implement deep logical memories, the physical arrays can be combined "vertically". An example is shown in Figure 2.2, in which a 2Kx1 logical memory is implemented using two 1Kx1 arrays. In this case, the upper array implements all addresses with the LSB equal to 1, while addresses with the LSB equal to 0 are implemented using the lower array. More complex larger memories can be implemented by combining arrays both horizontally and vertically; examples of this will be presented in Sections 2.3.1 and 2.4.1.

The architectures described in the next sections differ widely in the number, size and configurability of their basic blocks, as well as their support for combining arrays.

Company	Reference	Total Bits	Number of Arrays	Basic Organization of Each Array
Applied Micro Circuits	[2]	1280 bits	2	32x20 bits
Hitachi	[16]	4608 bits	1	128x36 bits
Motorola	[17]	4608 bits	2	64x36 bits
Fujitsu	[18]	16384 bits	4	256x16 bits
Mitsubishi	[3]	36864 bits	8	256x18 bits
Fujitsu	[19]	65536 bits	16	256x16 bits
LSI Logic	[20]	up to 576 Kbits		up to 8192 x 72

Table 2.1: Gate arrays with memory.

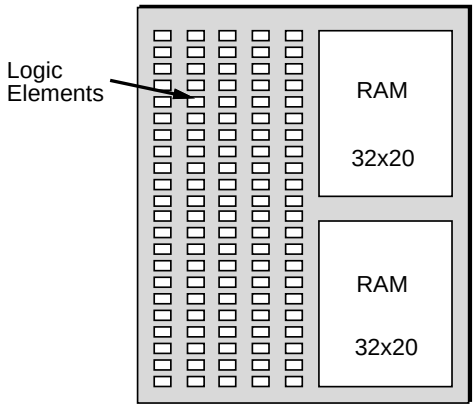


Figure 2.3: Applied Micro Circuits gate array floorplan from [2].

2.2 Memory Resources in Gate Arrays

Embedded memory arrays have been available in mask-programmed gate arrays since the 1980s. Table 2.1 shows seven gate arrays and their memory resources (this table is not meant to be an exhaustive list of all gate arrays with embedded memory). The memory resources consist of a small number of large arrays; in some, the aspect ratio of each array is configurable. The arrays can be combined using logic resources to form deeper and/or wider memories. In addition, the logic cell of one of the gate arrays (from Mitsubishi [3]) was designed such that each logic element can implement one additional bit of memory. In the LSI Logic product, a user-specified memory size and shape is compiled and embedded onto the gate array [20].

Figures 2.3 and 2.4 show the floorplans of two of these devices. In both cases, the arrays were placed on the edge of the chip to allow simple connections between the memory and I/O pads.

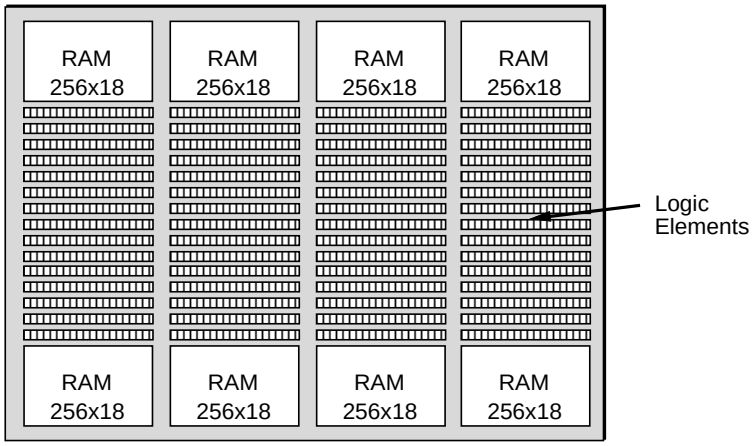


Figure 2.4: Mitsubishi gate array floorplan from [3].

2.3 Fine-Grained Memory Resources in FPGAs

The primary focus of this dissertation is memory resources in Field-Programmable Gate Arrays. As described in the previous chapter, FPGA memory resources can be classified as either coarse-grained or fine-grained. Fine-grained FPGA memory architectures consist of many very small basic arrays distributed over the entire FPGA. Coarse-grained architectures, on the other hand, consist of only a few larger arrays. Both variants can be found in currently available FPGAs. This section describes three commercial FPGAs using a fine-grained memory architecture, while Section 2.4 describes six products employing a coarse-grained approach. Within both sections, the order of the products presented reflects the (approximate) flexibility of the memory resources (most flexible first).

2.3.1 Xilinx 4000 Architecture

The Xilinx XC4000 family of FPGAs was one of the first architectures with fine-grained memory resources [12, 21, 22]. Each FPGA in this family contains between 100 and 1024 configurable logic blocks (CLBs); each CLB contains contains two four-input lookup-tables, a three-input lookup-table, two flip-flops, and glue logic, and can be used to implement either logic or memory. Figure 2.5 shows a block diagram of a single CLB configured as a memory. When used as a memory, each of the two 4-input lookup tables can be used as a 16-bit RAM; the external glue logic allows the user to optionally combine the 16-bit RAMs into a 32x1 memory. The data output pins of each memory can be optionally connected

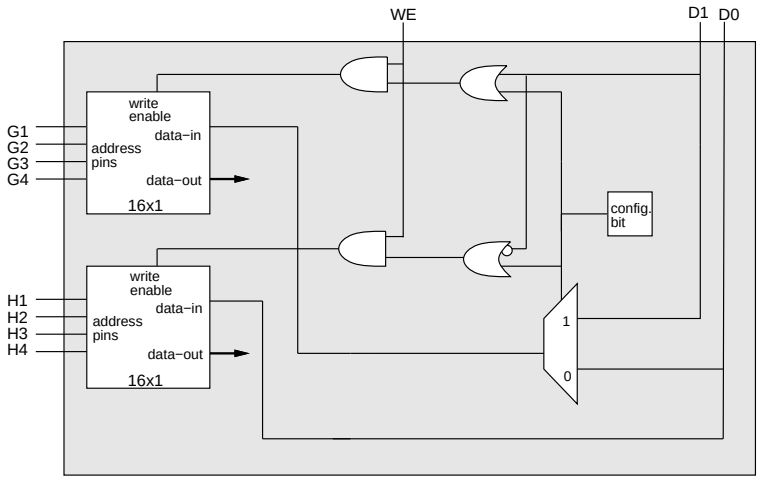


Figure 2.5: One CLB configured as memory.

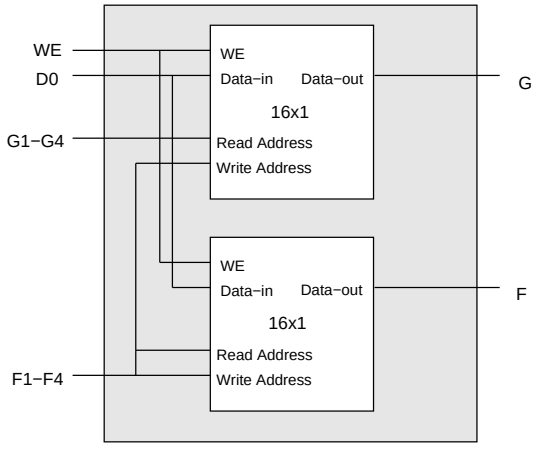


Figure 2.6: XC4000E CLB configured as a 16x1 two-read port, one-write port memory.

to the three-input lookup-table or a flip-flop. In the XC4000E parts, each memory can be synchronous (edge-triggered) or level-sensitive, while in the XC4000 parts, only level-sensitive memories are supported.

The XC4000E parts also have a "dual-port" mode in which the two four-input lookup tables within a CLB are combined to form a 16-bit, two-read port, one-write port memory. The 16 bits are replicated in the two lookup tables, and each replicated copy is used to support one read port, as shown in Figure 2.6. During a write, both replicated copies are updated.

In order to implement memories containing more than 32 bits, CLBs must be combined. Figure 2.7 shows how one bit-slice of a 128x8 bit memory can be implemented on this archi-

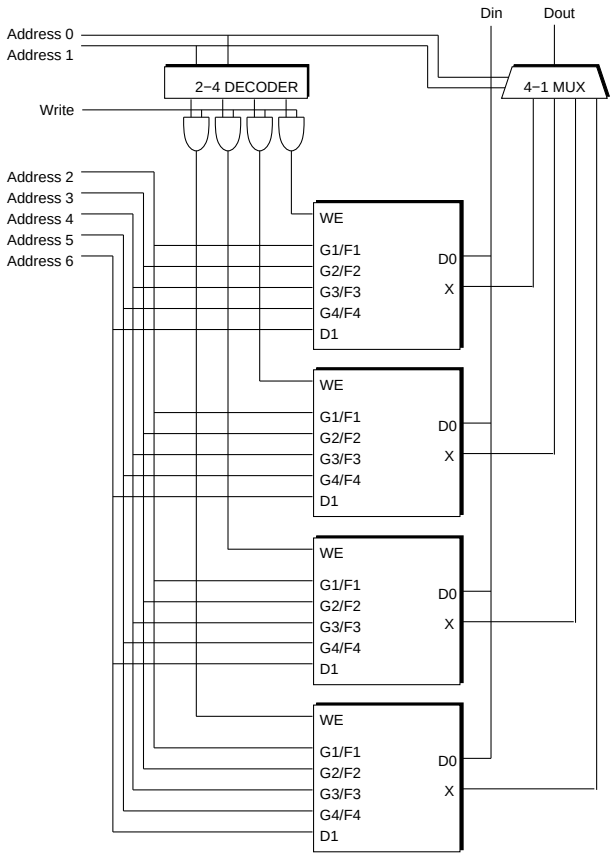


Figure 2.7: One bit slice of a 128x8 memory implemented using Xilinx CLBs.

Logical Memory	CLBs for storage	CLBs for glue logic	Total CLBs
128x8	32	10	42
256x4	32	15	47
1Kx1	32	29	61
4Kx1	128	117	245

Table 2.2: Xilinx CLBs required to implement logical memories (from [1]).

tecture (the CLBs implementing the memory and output multiplexor would be replicated 8 times to implement the complete memory). The output multiplexor and address decoder can also be implemented using CLBs; in this example, 10 CLBs are required to implement the address decoder and eight multiplexors, in addition to the 32 CLBs required for storage. Table 2.2 shows the number of CLBs required to implement larger logical memories (these were obtained using the Xilinx XACT Development System and presented in [1]). Clearly, large logical memories quickly fill the chip, leaving little room left to implement logic.

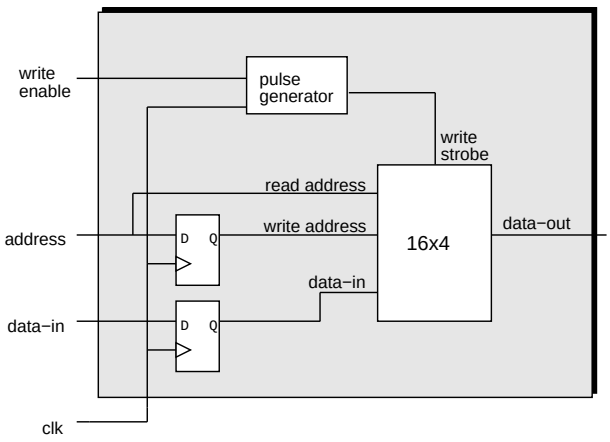


Figure 2.8: PFU configured as 16x4 memory in ORCA 2CA and 2TA.

2.3.2 Lucent Technologies ORCA Architecture

The approach taken in the Lucent Technologies Optimized Reconfigurable Cell Array (ORCA) architecture is similar to that taken in the Xilinx parts. In all ORCA FPGAs, each logic block (called a Programmable Function Unit, PFU) contains four 4-input lookup tables, four latches, and additional glue logic. Each PFU can be configured as either two 16x2 memories or a single 16x4 memory [13, 23, 24].

Further flexibility is provided in the ORCA 2CA and 2TA parts. In these devices, separate read and write addresses to each PFU can be provided by time-multiplexing them onto the single set of address lines. Figure 2.8 shows a simplified view of the PFU architecture when configured as a 16x4 memory. The write address is supplied in one clock phase while the read address is provided in the other. Besides supporting simultaneous reads and writes from each PFU, this architecture also allows the implementation of a two-read, one-write port RAM, similar to the dual-port mode of the Xilinx XC4000E FPGA.

The ORCA PFUs can be combined to implement larger logical memories in the same manner as the Xilinx CLBs. Since the minimum data width of each PFU is two, the minimum data width of any implemented logical memory is also two.

2.3.3 Crosspoint Architecture

The architecture of the Crosspoint CP20K family of FPGAs is shown in Figure 2.9 [25, 26]. The diagram shows two distinct types of building blocks: transistor-pair tiles (TPT) and RAM-logic tiles (RLT). Each TPT contains one n-type transistor and one p-type transistor and is aimed primarily at implementing the logic portions of a circuit. Memory can be implemented using the RLTs; each RLT contains one bit of storage (RLTs also contain additional circuitry to implement wide multiplexors and sequential logic elements).

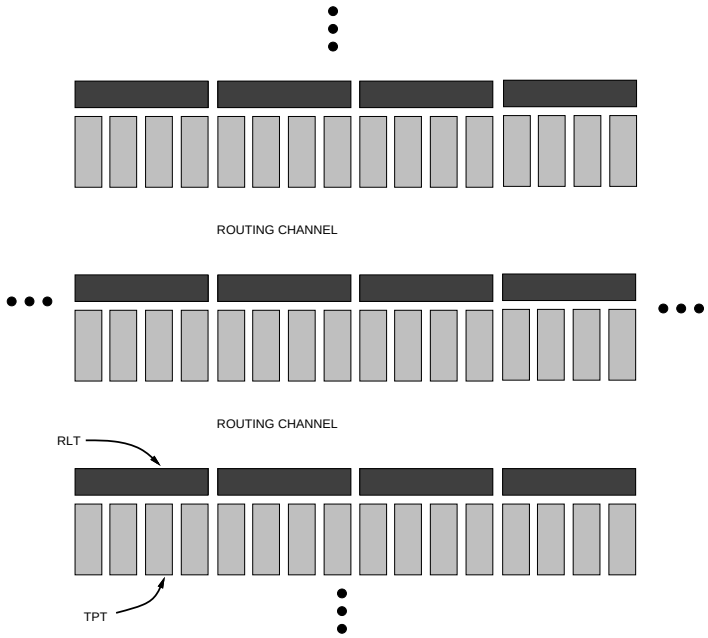


Figure 2.9: Crosspoint CP20K architecture.

Figure 2.10 shows how RLTs can be combined to form logical memories. The RLTs are connected using dedicated wordlines (*read_select* and *write_select*), a dedicated bitline (*data*), and a dedicated column select line (four other pins not normally used when implementing memory are not shown). The dedicated wordlines and column select lines can be connected to TPTs at the edge of the chip; an additional RLT is required to connect the bitline to TPTs (this additional RLT acts as a sense amplifier). The wordlines can be split in the middle of the chip meaning two independent memories can be implemented. The data multiplexors and address decoder are implemented using TPTs. Since the largest FPGA in the CP20K family contains 3684 RLTs, the maximum memory size that can be implemented is slightly less than 3.6 Kbits.

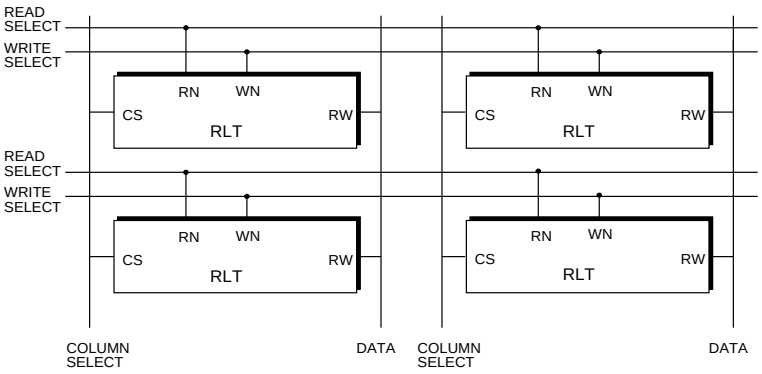


Figure 2.10: Combining RLTs to construct a larger array.

2.4 Coarse-Grained Memory Resources in FPGAs

All of the FPGAs described in the previous section are inefficient when implementing large memories. In this section we describe six commercial heterogeneous architectures which are suitable for implementing circuits with large memory requirements.

2.4.1 Altera FLEX 10K Architecture

The Altera FLEX 10K family of programmable logic devices contain large embedded arrays in which memory can be efficiently implemented, as well as small lookup-tables suitable for implementing logic [14, 27]. Members of this family contain between 3 and 12 *Embedded Array Blocks* (EABs), each of which contains a 2-KBit array. The EABs are positioned in a single column in the middle of the chip, as shown in Figure 2.11. The EABs are connected to the logic array blocks (each of which contains eight 4-input lookup tables) using horizontal and vertical routing channels.

Figure 2.12 shows a single EAB. The memory array within the EAB can be configured as a 256x8, 512x4, 1024x2 or a 2048x1 memory. Registers can be optionally included before the inputs and after the outputs of the array. Although not shown in the diagram, there are also dedicated inputs that can be used to drive the clock inputs of each register and the write enable input of the memory array.

This architecture can implement large memories much more efficiently than the Xilinx architecture described in Section 2.3.1. Each of the first three logical memories in Table 2.2 can be implemented using a single EAB; the fourth logical memory can be implemented

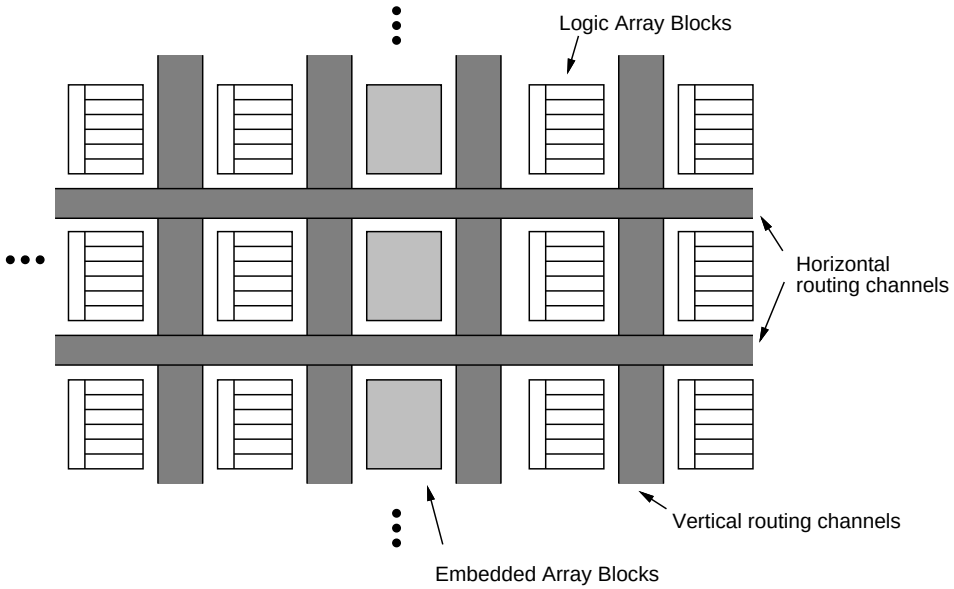


Figure 2.11: Floorplan of Altera FLEX 10K architecture.

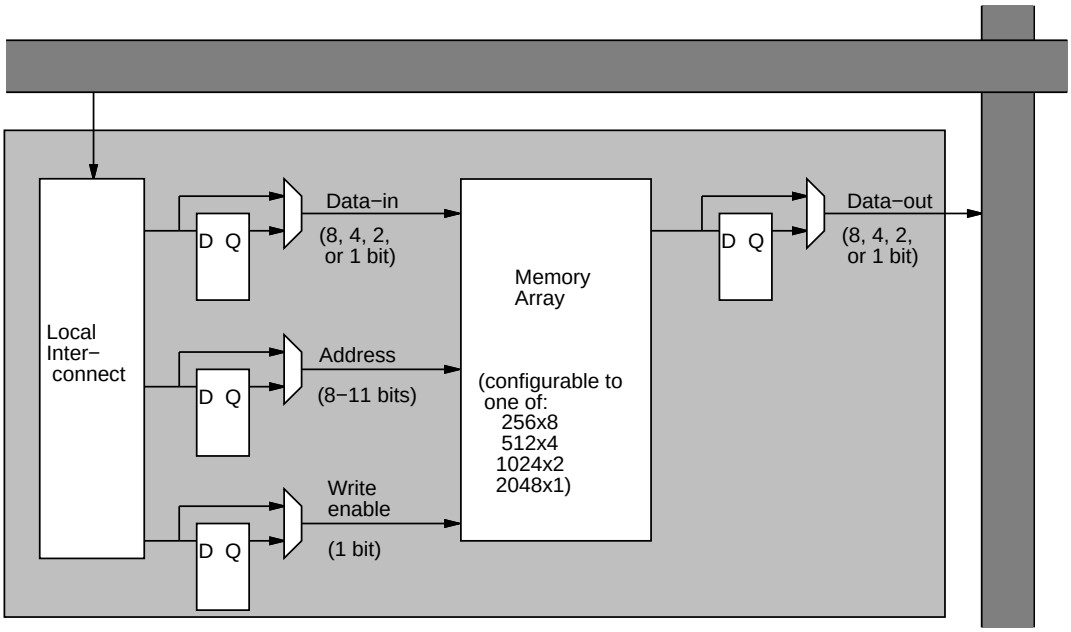


Figure 2.12: Embedded array block in an Altera FLEX 10K CPLD.

using two EABs. Even larger logical memories can be created by combining EABs; Figure 2.13 shows how six EABs, each in their 256x8 mode, can be connected to implement a 512x24 logical memory.

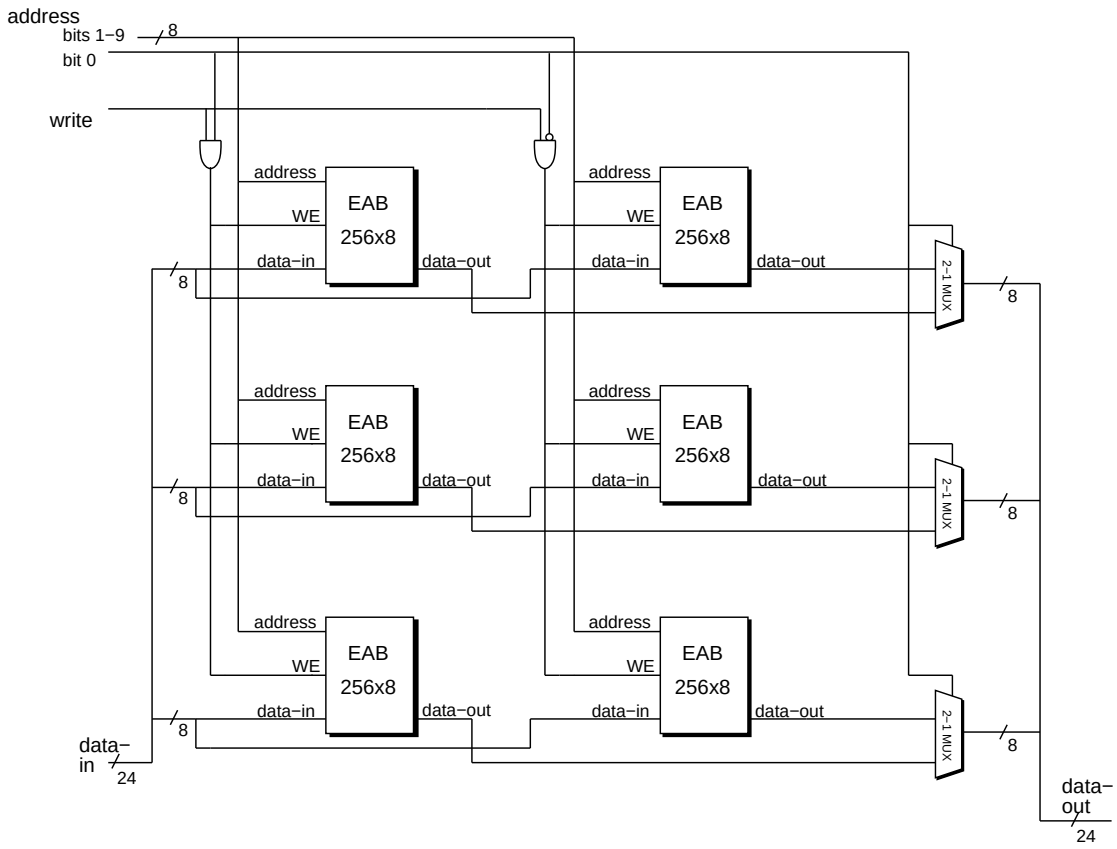


Figure 2.13: 512x24 memory implemented using six EABs.

2.4.2 Actel 3200DX Architecture

A similar approach is used in the Actel 3200DX family [15]. The FPGAs within this family contain between 8 and 16 memory arrays, each of which can be configured as 32x8 or 64x4. Unlike the Altera EABs, each Actel SRAM block contains independent read and write address ports and separate read and write clocks. This is suitable for implementing buffers which must be read and written simultaneously.

2.4.3 Lattice ispLSI 6192 Architecture

Lattice's approach to configurable memory is unique in that they offer three devices which are identical, except for the memory resources [28]. All three parts contain 4608 bits, but differ in the support circuitry for the arrays.

In the 6192FF part, the memory acts as a 256x18 or 512x9 first-in first-out (FIFO) buffer. The FIFO is dual-ported; one port is connected directly to I/O pins, while the other

is connected to the logic resources. I/O signals indicating buffer "full", "empty", "almost full", and "almost empty" are provided. The thresholds of "almost full" and "almost empty" are adjustable by the user.

The 6192SM device contains a standard single-port memory; it can implement one 256x18 or 512x9 single-port logical memory or two 128x18 or 256x9 single-port logical memories. The memory or memories can be either connected directly to I/O pins or to the logic part of the FPGA.

In the 6192DM part, the user can implement one dual-port 256x18 or 512x9 memory (the option of creating two smaller memories is not provided). Since the memory is dual-ported, it can be accessed from the I/O pins and the logic part of the FPGA simultaneously.

2.4.4 Altera FLASHlogic

The Altera FLASHlogic architecture (formerly marketed by Intel) consists of eight configurable function blocks (CFBs) connected using a global interconnect matrix [29]. Each CFB can be configured to act either as a 22V10-like logic block or a 128x10 memory. Unlike the other coarse-grained architectures, the aspect ratio of each memory is not configurable.

2.4.5 Kawasaki Steel FPGA

In 1990, the LSI Division of Kawasaki Steel described a prototype FPGA containing a single 8-Kbit SRAM block [30]. Dedicated FIFO control circuitry was also included, allowing the memory to be used either as a FIFO buffer or a random access memory. The data width could be configured as 4 or 8.

2.4.6 Plus Logic Architecture

One of the earliest FPGA architectures with memory was proposed, but not marketed, by Plus Logic, Inc. in 1989 [31]. Their FPSL5110 FPGA architecture, which was capable of implementing 1000 to 2000 gates, contained a dual-port 1152 bit memory. The single array could be configured as 32x36, 64x18, or 128x9. Like the Kawasaki Steel architecture, it contained only one array, meaning it could be used to implement only a single logical memory.

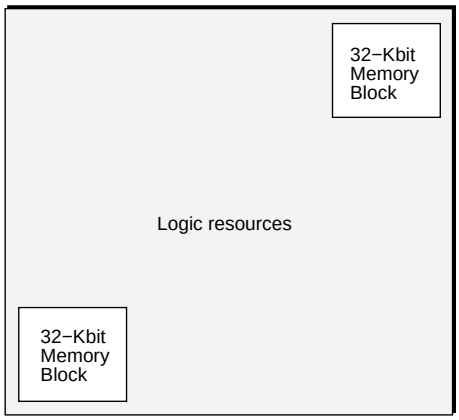


Figure 2.14: Floorplan of Chip Express CX2100 LPGA.

2.5 Chip Express Laser-Programmable Gate Array

The CX2000 Laser-Programmable Gate Array (LPGA) family from Chip Express also contains embedded configurable memory [32]. Unlike an FPGA, an LPGA can not be programmed in the field, but must be programmed by the LPGA vendor. Proprietary laser programming technology, however, results in a turn-around time as short as one day.

The CX2000 family of LPGAs contain between zero and four 32-Kbit blocks. Each block can implement up to four independent single-port logical memories or two dual-port memories. Although the details of the architecture are not available, Chip Express advertises that the word width of these logical memories can be between 1 and 128 bits, and the depth can be between 2 and 16K words. A floorplan of the CX2100 LPGA is shown in Figure 2.14; this device contains two memory blocks, each in one corner of the chip [33].

2.6 FiRM Architecture

An experimental configurable memory implementation employing the coarse-grained approach is described in [1, 34]. The chip, called FiRM, is stand-alone; that is, the address and data lines are connected directly to external I/O pins.

The memory resources of FiRM consist of four 1-Kbit arrays, each of which can be configured as 128x8, 256x4, 512x2, or 1024x1. There are also four external I/O ports, each of which contains address pins, data pins, and a block enable pin. The configurable arrays and external pins are connected using a novel interconnect structure as shown in Figure 2.15.

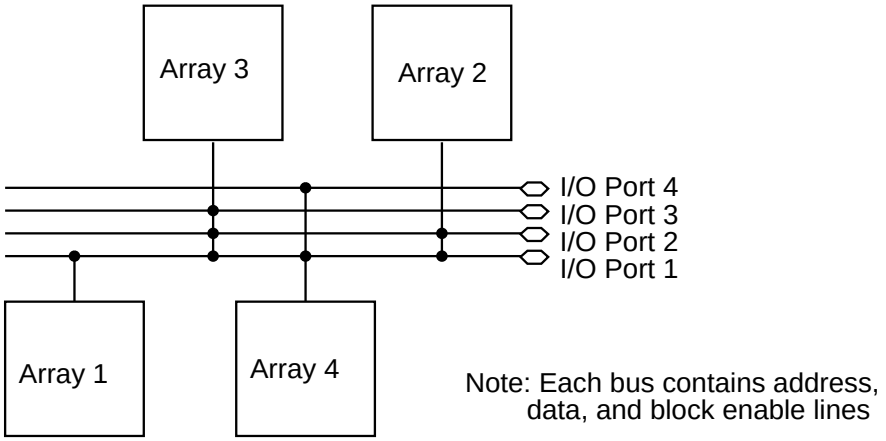


Figure 2.15: FiRM interconnect structure.

This interconnect serves two purposes: it allows arrays to be combined to implement larger logical memories, and it connects the arrays to the external I/O ports. A key feature of this interconnect is that it is not complete; not every array can be connected to every other array and every I/O port. Yet, by taking advantage of the fact that all memory blocks are identical, as are all I/O ports, the interconnect structure can implement as many logical memory configurations as if a complete interconnect structure had been employed. The interconnect requires only 220 pass-transistors and 27 configuration bits. Had a complete interconnect been used, at least 440 pass-transistors and 48 configuration bits would have been required.

Because of the efficient interconnect employed, the complete FiRM chip was only 38% larger than the four memory arrays without any reconfigurability. The access time of FiRM was 89% slower than that of the base array.

An interesting feature of the FiRM architecture is that it can implement a two read-port, one write-port logical memory by replicating data across two memory arrays. This technique was later used in the Xilinx XC4000E FPGAs and the ORCA 2CA and 2TA products as described in Sections 2.3.1 and 2.3.2.

2.7 Summary

Table 2.3 summarizes the capabilities of the commercial FPGAs and LPGA described in this chapter. The devices are classified into two categories: fine-grained devices, which contain many very small basic memory blocks, and coarse-grained devices, which contain only a few large arrays. It is clear that the coarse-grained architectures implement large memories more efficiently, since the per-bit overhead is smaller.

FPGA / LPGA Family	Type	Basic arrays	Size of each basic array	Allowable data widths for each basic array
Xilinx 4000	Fine	100-1024	32 bits	1,2
Lucent ORCA	Fine	100-900	64 bits	2,4
Crosspoint CP20K	Fine	3684	1 bit	1
Altera 10K	Coarse	3-12	2 Kbits	1,2,4,8
Actel 3200DX	Coarse	8-16	256 bits	4,8
Lattice ispLSI	Coarse	1	4608 bits	9,18
Altera FLASHlogic	Coarse	8	1280 bits	10
Kawasaki Steel FPGA	Coarse	1	8 Kbits	4,8
Plus Logic	Coarse	1	1152 bits	9,18,36
Chip Express CX2000	Coarse	0-4	32 Kbits	unknown

Table 2.3: Summary of memory resources in commercial FPGAs.

Although some of the FPGAs described in this chapter have been available for several years, there have been no published studies evaluating and comparing potential configurable memory architectures (outside of the work related to this dissertation). This work is a first step in that direction. In Chapters 4 through 7, important parameters will be identified, and several architectural experiments that give insight into the nature of configurable memory will be presented. Before these experiments are described, however, it is important to understand the applications that will use these devices; this is the focus of the next chapter.

Chapter 3

Circuit Analysis and Generation

In order to design a good FPGA architecture, a thorough understanding of the nature of large digital circuits is essential. Circuit characterization for logic circuits is an active area of research [35, 36]. Many of the circuits that will be implemented on next-generation FPGAs, however, will contain significant amounts of memory. Thus, a characterization of circuits containing both logic *and* memory is required. Section 3.2 presents such an analysis.

As explained in the next section, one of the key reasons for performing this analysis is that it can be used to build a circuit generator that stochastically generates realistic circuits with memory. Such a generator was built, and is described in Section 3.3.

3.1 Motivation

The primary focus of this dissertation is the investigation of FPGAs with on-chip memory. The search for a "good" memory architecture is central to this work. There are two approaches to developing this "good" architecture:

1. Enumerate all possible uses of the FPGA memory resources and create an architecture that can implement them.

2. Experimentally implement many benchmark circuits on many architectures, and measure the implementation efficiency of each device. The architecture that can most efficiently implement the benchmark circuits is deemed the best.

Ideally we could use the first approach. Unfortunately, just as it is impossible to enumerate all possible uses of an FPGA, it is also impossible to enumerate all possible uses of FPGA memory resources. Although some structures, such as queues and stacks, are obvious, there are many uses of memories that are not obvious (such as the "microcoded" state machine implementation used in [9]). Even if it *was* possible to enumerate all such memory structures that we want the FPGA to support, the sizes, shapes, and connectivity of each structure must be considered. A wide shallow stack would place significantly different demands on the FPGA architecture than a narrow deep stack. Finally, even if such an exhaustive list *could* be constructed, it would contain so much information that it would be impossible to deduce a suitable FPGA memory architecture without experimentation.

Thus, in this research, as in many other FPGA architectural studies [11, 37, 38, 39, 40], we have chosen the second approach – we "implement" many benchmark circuits on various architectures, and estimate the efficiency of each implementation. When performing such experiments, it is important that we use enough benchmark circuits to thoroughly exercise our architecture. In architectural studies concerning the logic part of an FPGA, it is common to use 10 to 20 "real" circuits as benchmarks. This works well for these studies; since each of these circuits contains hundreds (or thousands) of logic blocks, we can be confident that the architecture under study is thoroughly exercised using only a few circuits. Most memory circuits, however, only contain a few logical memories. Thus, to thoroughly exercise a configurable memory architecture, we need hundreds (or thousands) of benchmark circuits to achieve the same level of confidence.

Unfortunately, we were unable to gather such a large number of complete circuits. Instead, we generated them stochastically. We have developed a circuit generator that stochastically creates realistic benchmark circuits; it is described in Section 3.3. Stochastic circuit generators have been described elsewhere [35, 36]; however ours is the first generator that generates circuits containing both logic and memory.

For our results to have meaning, it is important that these "circuits" be realistic. We ensure this by basing the generator on the results of a detailed circuit analysis. In Section 3.2, we present a detailed structural analysis of circuits with significant memory requirements. Statistics gathered during the analysis are then used as probability distributions in the circuit generator of Section 3.3.

3.2 Circuit Analysis

This section presents the analysis of circuits containing both logic and memory. It is presented in three parts: Subsection 3.2.1 presents statistics regarding the numbers, shapes, and sizes of logical memories within circuits; Subsection 3.2.2 describes how these memories tend to appear in "tightly-connected" groups or clusters and presents statistics regarding the makeup of these clusters; and Subsection 3.2.3 describes common connection patterns within these clusters.

3.2.1 Logical Memory Configurations

This analysis is based on 171 circuits containing memory. Data regarding these circuits was obtained from several sources: recent conference proceedings, recent journal articles, local designers at the University of Toronto, a major Canadian communications company, and a customer study conducted by Altera. Only incomplete data about these circuits was available; for none were we able to obtain a circuit netlist. The data obtained from Altera consisted of statistics regarding how many memories appear in each of 125 circuits, as well as the width and depth of 115 of the logical memories in these circuits. The data obtained from the Canadian telecommunications company consisted of the number of memories in each of 15 communication chips, as well as the width and depth of all 66 memories in these 15 circuits. The telecommunications company supplied additional information about each memory indicating whether it was used as a RAM or ROM, and whether it was single- or dual-ported. The most complete information was obtained for circuits described in conference and journal articles as well as those from local designers. There were 31 such circuits; for each we obtained information regarding the number of memories in each circuit, the width and depths of the memories, whether the memories were used as RAMs or ROMs, whether the memories were single- or dual-ported, and block diagrams that indicated how the memories were connected to each other and to the logic portion of each circuit. Table 3.1 contains a summary of the information available for the circuits.

Since we wanted to focus our architecture studies on circuits that would use *on-chip* memory, we did not include circuits requiring more than 128 Kbits and circuits with more than 16 logical memories in our study. Most of the circuits were initially designed to be

Source	Number of Circuits	Information Available			
		Memory Width/Depths	RAM/ ROM	single/dual port	block diagrams
Altera	125	yes (for 115 memories only)	no	no	no
Local Designs and Conference/Journal Articles	31	yes (for all 87 memories)	yes	yes	yes
Telecommunications Company	15	yes (for all 66 memories)	yes	yes	no

Table 3.1: Information available for example circuits.

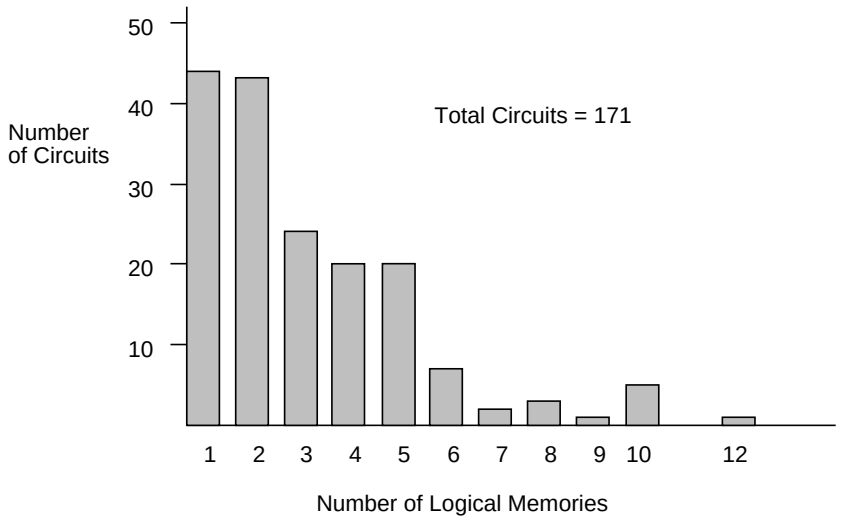


Figure 3.1: Distribution of number of logical memories required.

implemented on a gate array or custom chip.

Figure 3.1 shows the distribution of the number of logical memories in each circuit. If more than one memory in a circuit shares address connections (i.e. receives the same address from the logic at all times), these memories were counted as one logical memory. As the graph shows, circuits with one or two logical memories are common while the demand for higher numbers of logical memories decreases rapidly. Note that the horizontal axis is the number of *logical* memories, not the number of physical arrays required to implement the memories.

Figure 3.2 shows the distribution of widths and depths of the logical memories in the analyzed circuits. Since we did not have memory width/depth information for all circuits from Altera, the results only show the distribution for 268 of the 533 logical memories. As the graphs show, the proportion of memories with depths in each power-of-two-interval

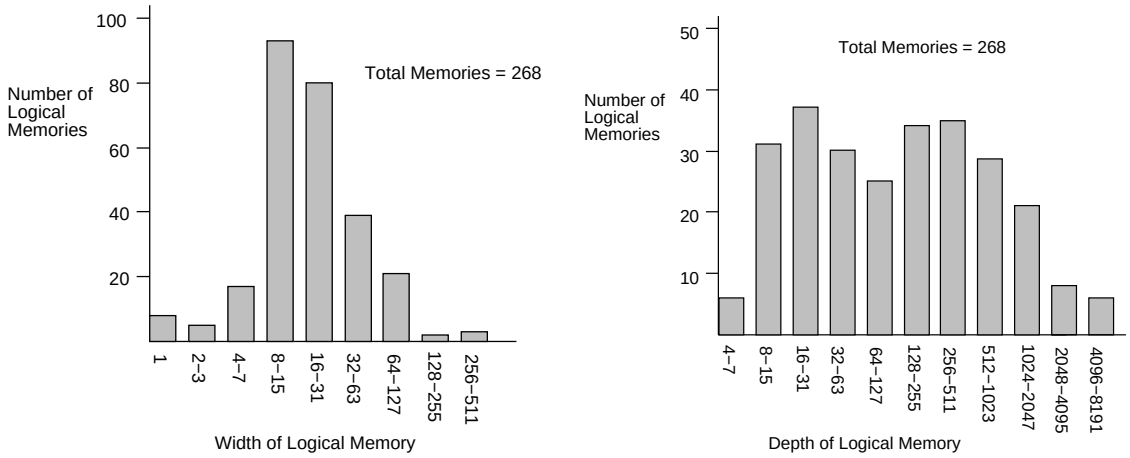


Figure 3.2: Distributions of logical memory widths and depths.

Percentage of Widths that are a Power-of-Two	69%
Percentage of Depths that are a Power-of-Two	74%
Percentage of Memories used as ROMs	16%
Percentage of Memories that are Multi-ported	13%

Table 3.2: Additional logical memory statistics.

between 8 and 2048 is roughly constant, while the memory width distribution peaks at about 8, and falls off below 8 and above 16.

Finally, Table 3.2 shows some additional statistics compiled from our circuits. As shown in the table, 69% of the widths and 74% of the depths are powers-of-two. Also, the majority of logical memories are single-ported RAM blocks.

Note that these measurements are somewhat influenced by the target implementation of the analyzed circuits. Circuits originally intended for discrete-device implementations would tend to use standard memory sizes. However, since most circuits originally had gate-array or custom chip implementations, we believe the observed trends would be similar to those for circuits targeted to an FPGA with memory.

3.2.2 Logical Memory Clustering

In this section, we examine how logical memories are connected to the logic portions of circuits. In our set of circuits, memories often appear in "tightly connected" groups, where all memories within a group perform a similar function. In this dissertation, each of these

groups is called a *cluster*. The following subsection precisely defines a cluster and presents statistics regarding clusters.

Clusters

The logic portion of circuits can be divided into subcircuits. This is a natural consequence of the top-down and bottom-up design approaches employed by most designers. The division between these subcircuits is not always clear; subcircuits often share inputs, and common expressions might be collapsed into or extracted out of each subcircuit. Nonetheless, the discussion in this section will assume that the logic part of a circuit consists of identifiable logic subcircuits.

Memories connect to logic through their address pins, data-in pins, data-out pins, and other control pins (write enable and perhaps a clock). Each of these sets of pins is driven by (or drives) one or more logic subcircuits. Figure 3.3 shows a simple circuit in which separate subcircuits drive the address and data-in ports, while a third subcircuit consumes data from the data-out port. Connections to the write enable and other control lines are not shown. In the remainder of this chapter, we will refer to subcircuits driving the data-in pins as *data-in subcircuits*, subcircuits driving the address pins as *address subcircuits*, and subcircuits driven by the data-out pins as *data-out subcircuits*. Connections to control pins will be ignored, since they make up only a small proportion of all connections, and can often be grouped with the address connections.

A memory may have more than one data-in, data-out, or address subcircuit. In Figure 3.4(a), the memory is driven by two data-in subcircuits and drives two data-out subcircuits. This should not be confused with a dual-port memory; here, either the two data subcircuits are multiplexed onto a single data port, or the data pins within the data port are shared between the two data subcircuits. The next section will consider these memory/logic interconnect patterns in more detail. A single data-in or data-out circuit may be connected to more than one logical memory, as shown in Figure 3.4(b). Examples with several memories and several data-in or data-out subcircuits are also possible.

Figure 3.5 shows a larger example; this is a portion of the datapath of a forward error correction decoder for a digital satellite receiver system [4]. In this circuit, the eight logical memories can be grouped into five sets (denoted by dotted lines) where the memories in each group share common data-in and data-out subcircuits.

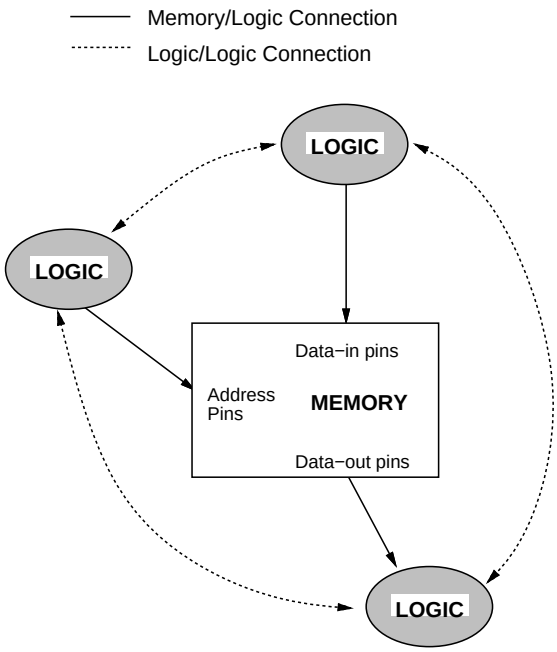
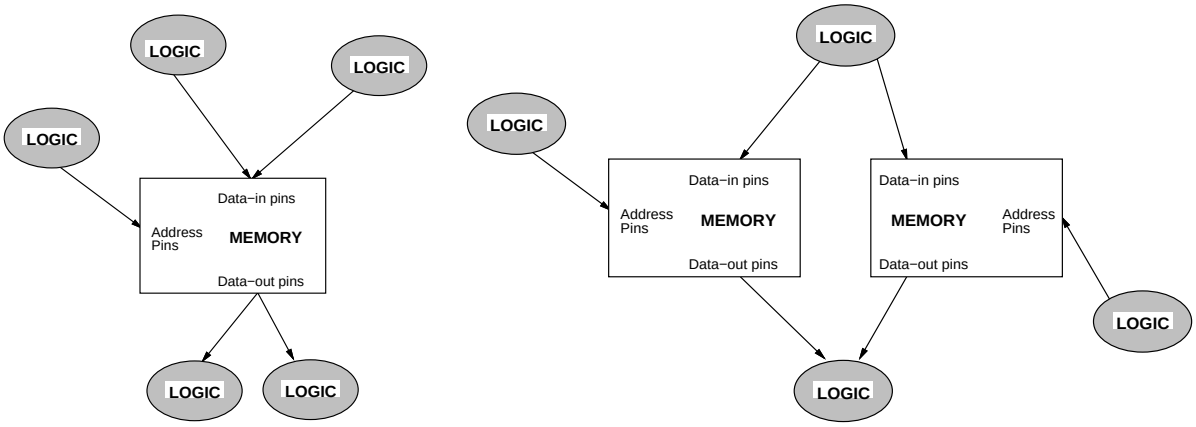


Figure 3.3: Logical memory connected to data-in, data-out, and address subcircuits.



a) one logical memory b) multiple logical memories

Figure 3.4: Example clusters.

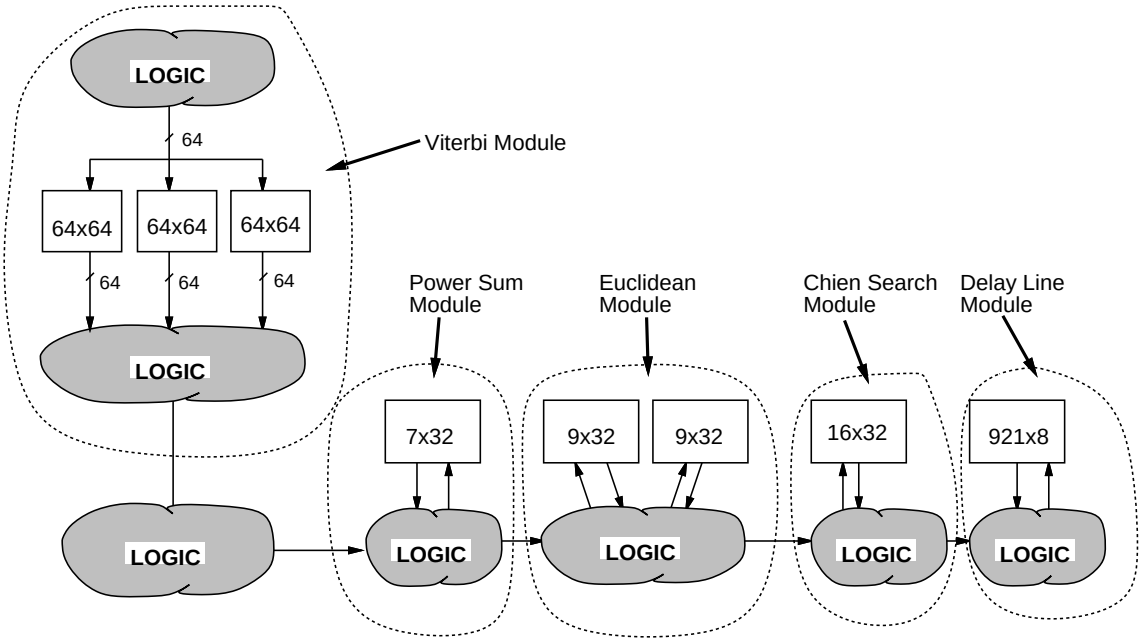


Figure 3.5: Forward error correction decoder datapath from [4].

To speak of these groups more precisely, we define a *cluster* as follows:

cluster: One or more logical memories in which all data-in ports are connected to a common logic subcircuit (or set of logic subcircuits) *or* in which all data-out ports are connected to a common logic subcircuit (or set of logic subcircuits).

The example of Figure 3.5 contains five clusters.

Note that Figure 3.5 does not show any address or control connections. We have chosen to partition memories into clusters based on their data-port connections without regard to the address and control connections. From our circuit survey, it appears that logical memories that do not share a data subcircuit (i.e. are not in the same cluster) rarely share an address or control subcircuit. Further work is needed to analyze address connection patterns and determine what sort of patterns within a cluster are common.

Cluster Statistics

As shown in Table 3.1, we only had block diagrams for 31 of the example circuits. For each of these circuits, we counted the number of clusters and the number of logical memories in

Percentage of Clusters where all Memories Have Same Width	95%
Percentage of Clusters where all Memories Have Same Depth	75%

Table 3.3: Additional cluster statistics.

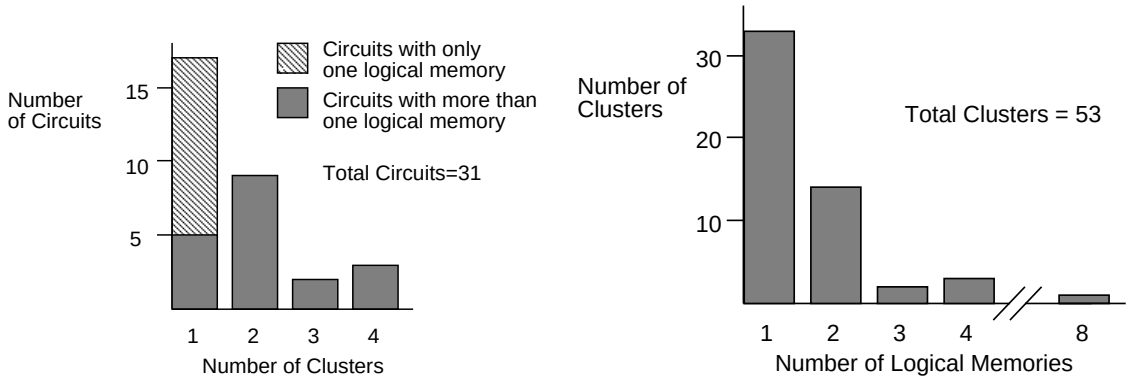
each cluster. Figure 3.6 and Table 3.3 summarize the results. Clearly, circuits with only one logical memory can contain at most one cluster; in Figure 3.6(a), these circuits are denoted with a hatched bar. Of the remaining circuits, half contain two clusters, and just over one third contain only one cluster.

Figure 3.6(b) shows the number of logical memories in each cluster. As the figure shows, 60% of the clusters consist of only one logical memory, while 30% consist of two logical memories. Of those clusters containing more than one logical memory, 95% consist of memories with the same width and 75% consist of memories with the same depth.

Figure 3.7 shows the distribution of the number of data-in and data-out subcircuits connected to the memories in each cluster in the 31 circuits. These measurements are approximate, since in some circuits it is difficult to deduce how a piece of logic can best be represented by a set of subcircuits. As the graphs show, the memories in most clusters are connected to only a single data-in subcircuit and a single data-out subcircuit.

3.2.3 Interconnect Patterns

Figure 3.8 shows a high-level view of a cluster with four memories, three data-in subcircuits, and three data-out subcircuits. The memories and logic are connected through *data-in* and



a) number of clusters distribution b) number logical memories per cluster

Figure 3.6: Cluster statistics.

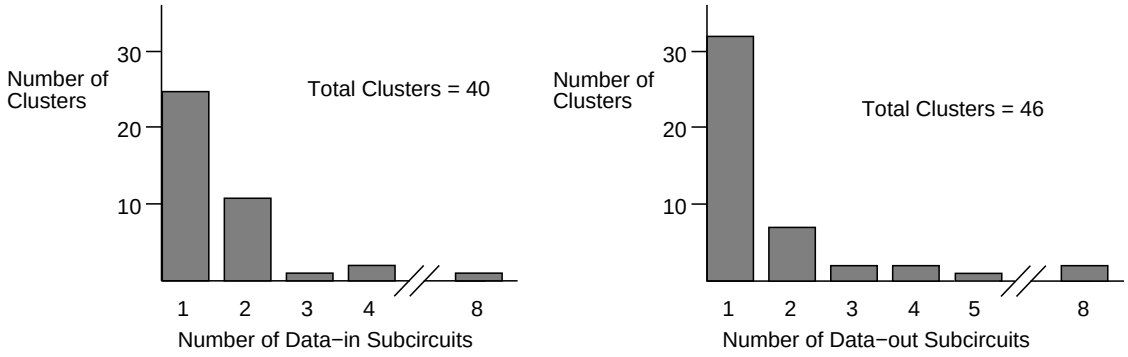


Figure 3.7: Logic subcircuit statistics (per cluster).

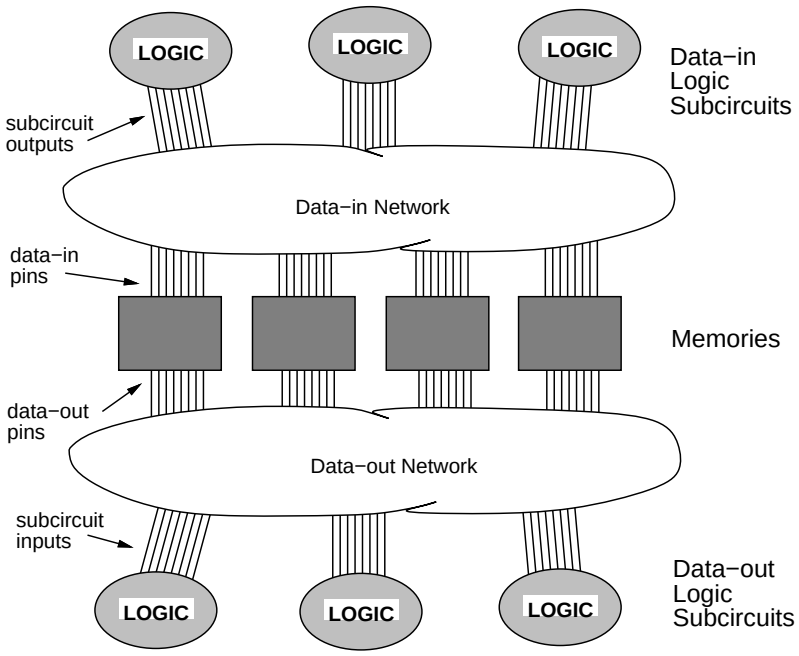


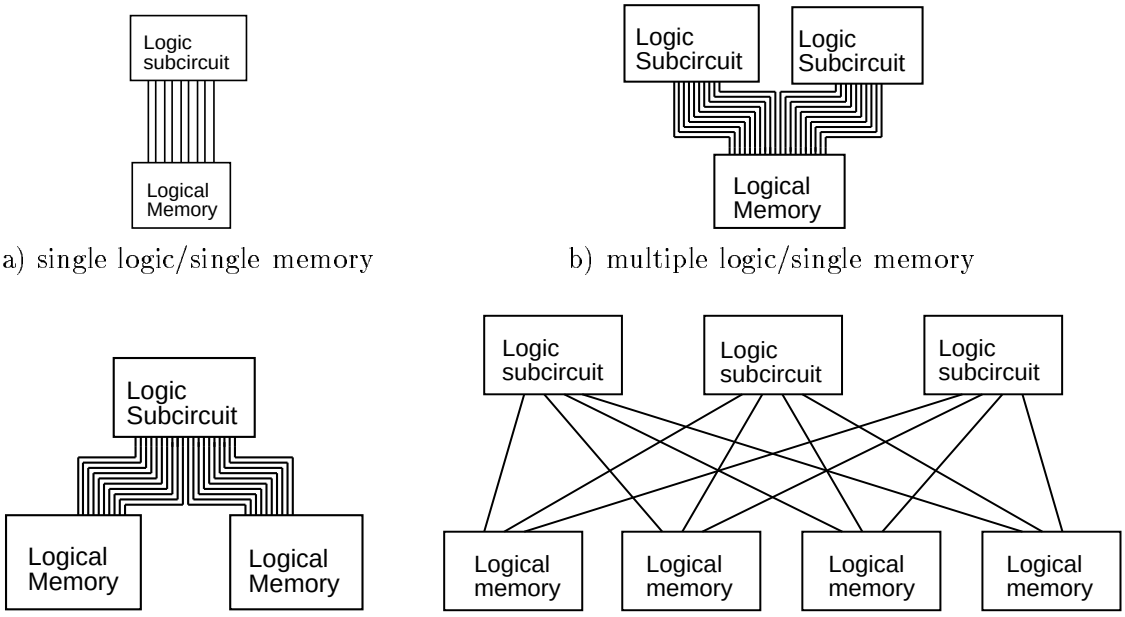
Figure 3.8: A single cluster showing data-in and data-out networks.

data-out networks. In our example circuits, these networks tend to fall into three categories. Each category is discussed in the following subsections.

Category 1: Point-to-Point Patterns

Figure 3.9 shows four interconnect patterns that are common in data-in networks, all of which we classify as *point-to-point* patterns. In all four patterns, each logic subcircuit output pin drives exactly one data-in pin in one memory block.

The simplest pattern, in which a single memory is connected to a single data-in subcircuit, is shown in Figure 3.9(a). Example circuits that employ this pattern can be found



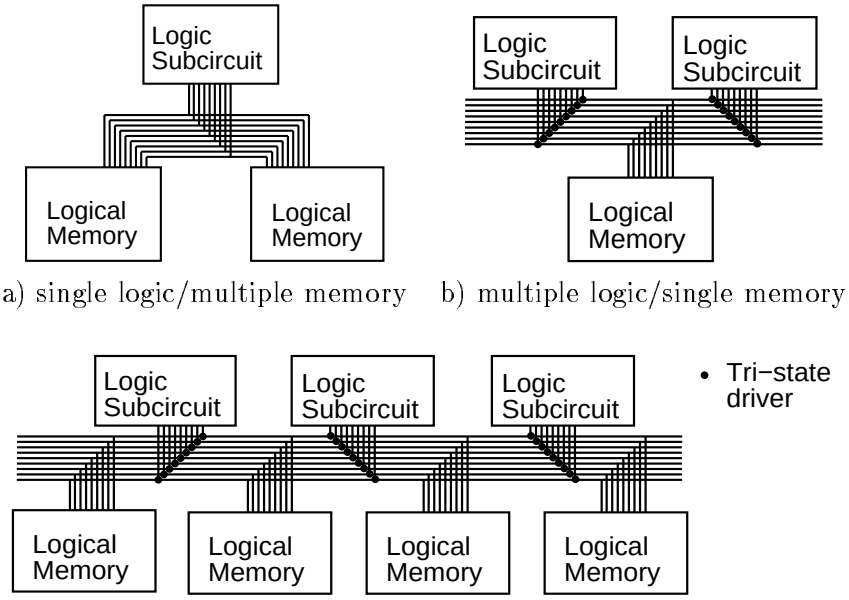
a) single logic/single memory b) multiple logic/single memory

c) single logic/multiple memory d) multiple logic/multiple memory

Figure 3.9: Point-to-point interconnect patterns for data-in network.

in [41, 42, 43, 44]. Figure 3.9(b) shows how the pattern is extended to clusters with more than one data-in subcircuit. Each logic subcircuit drives a subset of the data-in pins. This is common in applications where there are several fields in a single record (eg. a cache line with a valid bit) [45]. The extension of this pattern to more than two logic subcircuits is clear. In Figure 3.9(c), there is one data-in subcircuit but multiple memories. An example of this pattern is in the Viterbi module of Figure 3.5. Other examples are in [7, 41]. Finally, Figure 3.9(d) shows the most complex case; here, there are multiple data-in subcircuits and multiple memories, and the connections between the logic and memory are "shuffled" such that all logic subcircuits drive at least one pin of each memory. An example of this pattern can be found in [46], where input data is appended with a time-stamp as it is put into FIFOs.

The same patterns are common in the data-out network. Examples of the single logic/single memory pattern can be found in [42, 43, 47, 48, 49]. Examples of the multiple logic/single memory pattern can be found in [50, 51]. Examples of the single logic/multiple memory pattern are described in [7, 4, 43, 46, 52]. Finally, examples of the multiple logic/multiple memory pattern can be found in [53].



a) single logic/multiple memory b) multiple logic/single memory

c) multiple logic/multiple memory

Figure 3.10: Shared-connection interconnect patterns for data-in network.

Category 2: Shared-Connection Patterns

The second category of interconnect patterns is the *shared-connection pattern*. Like the point-to-point patterns, the structure of shared-connection patterns depends on the number of memory arrays and logic subcircuits. Figure 3.10 shows three different cases.

In Figure 3.10(a), there is one logic subcircuit and two memory arrays. The data-in port of each memory is driven by the same data (but presumably the memories have different address inputs or enable lines). An example of this can be found in [54] in which the data-in ports of a scratch-pad memory and a FIFO are connected to a bus. As before, the extension of this pattern to larger numbers of memories is clear. In the second pattern (Figure 3.10(b)), any one of the several logic subcircuits can supply data to the memory through multiplexor or tri-state bus. An example of this structure can be found in [49], in which a multiprocessor network interface puts either a pre-formatted request packet or a data-return packet into a queue before sending it on the network. Other examples can be found in [55, 56]. Finally, Figure 3.10(c) shows an example with multiple memory and logic subcircuits; each logic subcircuit is connected to a bus, which then drives the memories. An example is in [57] in which a *constant RAM* and a *data RAM* are connected to a single bus, which is, in turn, connected to several logic subcircuits.

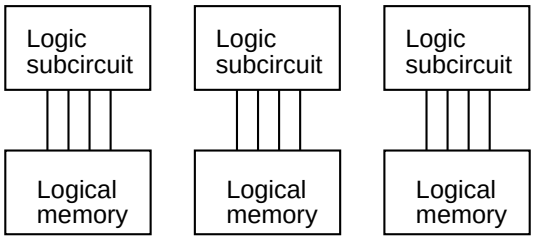


Figure 3.11: Point-to-point pattern with no shuffling.

The patterns in Figure 3.10 can appear in the data-out network as well. Examples of the single logic/multiple memory pattern can be found in [41, 54]. Examples of the multiple logic/single memory pattern are in [54, 55, 58]. Finally, examples of the multiple logic/multiple memory pattern are described in [57, 59].

Category 3: Point-to-Point Pattern with No Shuffling

The final pattern that will be described only occurs when there are the same number of data-in subcircuits as memories (or the same number of memories as data-out subcircuits). In the example of Figure 3.11, the data-in pins of each memory are driven by exactly one subcircuit. Since the three memories are connected to separate logic sources, it is tempting to separate the memories into three clusters. It is possible, however, that these three logical memories share a single data-out subcircuit. In that case, the memories are part of the same cluster even though they do not share the same data-in connections. An example of this is in [53], in which the data-in ports of two memories (a *pixel memory* and a *coefficient memory*) are driven by separate serial-to-parallel converters, while the data-out port of each memory drives separate inputs of a large multiplier. Another example can be found in [52].

Pattern Statistics

We analyzed each of the data-in and data-out networks in each cluster in the 31 circuits for which we had block diagrams, and classified each into one of the three categories described in the last three subsections. In some cases, the pattern did not fall into any of the categories; in those cases, we took the category with the closest match. Table 3.4 shows the results for all data-in and data-out networks except those connecting one memory to one logic subcircuit. As the table shows, all three categories are well represented among the data-

Connection Type	Count
Point-to-Point	5
Shared Connection	9
No Shuffling	7

a) Data-in Connection Types

Connection Type	Count
Point-to-Point	10
Shared Connection	14
No Shuffling	0

b) Data-out Connection Types

Table 3.4: Data-in and data-out interconnection type statistics.

in interconnects in our clusters, while the first two categories were approximately equally common among the data-out interconnects.

3.2.4 Summary of Circuit Analysis

The analysis described in this section was presented in three parts: the number, shapes, and sizes of logical memories, the number and sizes of clusters within circuits, and the interconnect pattern within these clusters. With the exception of Table 3.3, we have not attempted to investigate correlations between the various sets of statistics. The data in Table 3.4 was gathered from only 31 circuits; clearly this is not a large enough sample to estimate meaningful correlations. Gathering more circuits, and measuring correlations between the various statistics is left as future work. Nonetheless, we feel the analysis presented here is sufficient to develop an accurate stochastic circuit generator. The remainder of this chapter describes such a generator.

3.3 Stochastic Circuit Generator

This section describes a stochastic circuit generator developed using the statistics gathered during the circuit analysis. As described earlier, such a generator is essential in the architectural experiments described in the remainder of this dissertation.

Generating a circuit involves the following tasks:

1. Choosing a logical memory configuration and division of logical memories into clusters.
2. Choosing an interconnect pattern for each cluster
3. Choosing the number of data-in/data-out subcircuits associated with each cluster
4. Generating the logic subcircuits
5. Connecting the logic subcircuits and logical memories together

The above tasks are described in the following subsections.

3.3.1 Choosing the Logical Memory Configuration

The logical memory configuration is chosen as follows:

1. The number of clusters is chosen based on the probability distribution shown in the left half of Table 3.5. The numbers in this table were obtained by scaling the statistics in Figure 3.6(a).
2. For each cluster, the probability distribution in the right half of Table 3.5 was used to select the number of logical memories. This probability distribution was obtained by scaling the statistics in Figure 3.6(b). The number of logical memories for each

Number of Clusters	Probability
1	0.548
2	0.290
3	0.065
4	0.097

LMs per Cluster	Probability
1	0.623
2	0.264
3	0.038
4	0.075

Table 3.5: Probability distribution functions used to select the number of clusters and logical memories per cluster.

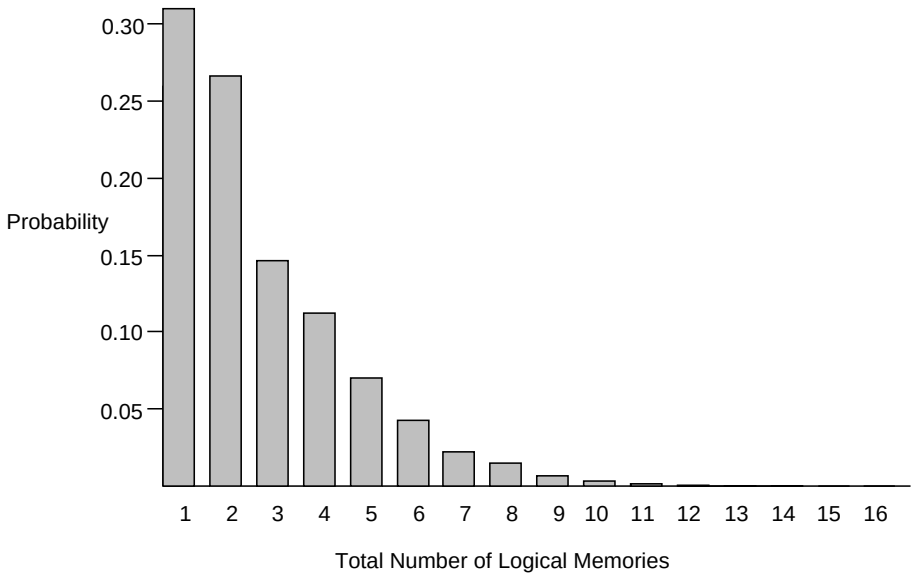


Figure 3.12: Resulting distribution of total number of logical memories.

cluster is chosen independently (and independent of the number of clusters). While these relationships are probably not completely independent, we do not have enough example circuits to accurately estimate the required partial probabilities to make a more accurate approximation. As a partial validation of the assumption, consider Figure 3.12. This figure shows the distribution of total logical memories over all clusters if the process described here is followed. Comparing this to Figure 3.1, the total number of logical memories in our example circuits, we see that the generated distribution is similar to the measured distribution. It is interesting to note that the statistics presented in Figure 3.6 (and therefore the distribution in Figure 3.12) reflect only 31 of our example circuits, while the statistics presented in Figure 3.1 reflect all 171 circuits. The similarity of the two figures (3.12 and 3.1) suggests that we are using enough example circuits to get reasonably accurate probabilities.

3. The width and depth of each logical memory is then determined. The probability distributions in Table 3.6 are used to select a range of depths and widths (these distribution functions were obtained by scaling the statistics in Figure 3.2). According to Table 3.2, 69% of the widths and 74% of the depths in our example circuits were powers-of-two. Thus, with a 69% probability (74% for the depth) we select the power-of-two dimension within the selected range. If we do not select the power-of-two dimension, all other values within the range are equally likely. The same width is

Width	Probability
1	0.030
2-3	0.019
4-7	0.063
8-15	0.347
16-31	0.299
32-63	0.146
64-127	0.078
128-255	0.007
256-511	0.011

Depth	Probability
4-7	0.023
8-15	0.118
16-31	0.141
32-63	0.115
64-127	0.095
128-255	0.130
256-511	0.134
512-1023	0.111
1024-2047	0.080
2048-4095	0.031
4096-8191	0.022

Table 3.6: Probability distribution functions used to select depth and width of each memory.

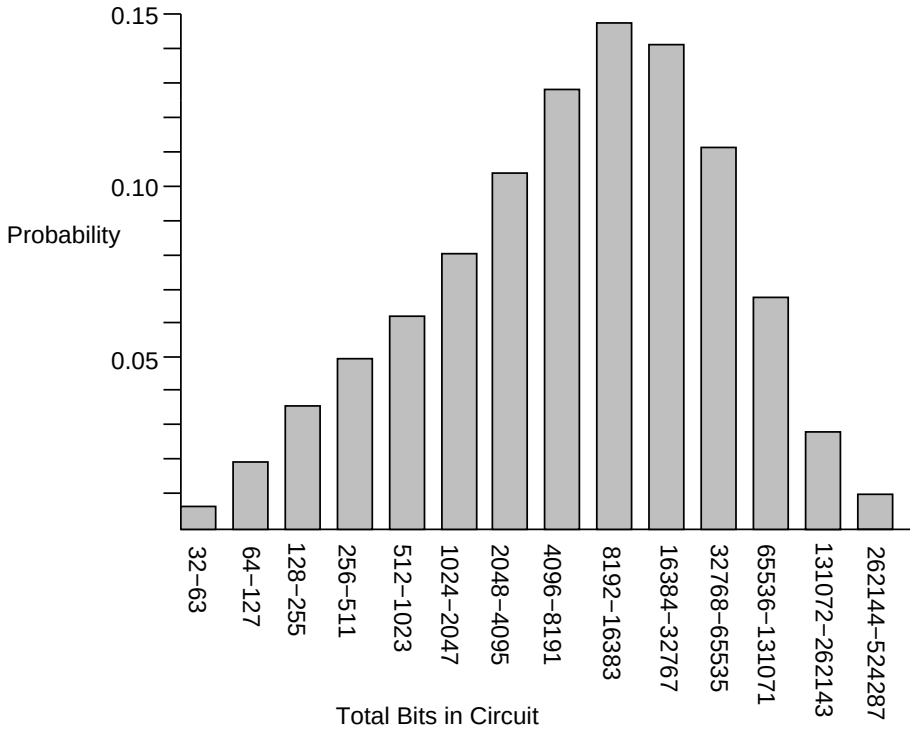


Figure 3.13: Resulting distribution of total number of bits.

used for all memories within each cluster (from Table 3.3, 95% of our example clusters consisted of memories of the same width). The same depth for each memory in a cluster is used with a 75% probability (this number is also from Table 3.3). For the other 25% of the clusters, the depth for each memory is chosen independently.

Combining Figure 3.12 with the distributions in Table 3.6 gives Figure 3.13, which is the distribution of the total number of bits in the generated circuits.

Connection Type	Probability
Point-to-Point	0.34
Shared Connection	0.33
No Shuffling	0.33

a) Data-in Connection Types

Connection Type	Probability
Point-to-Point	0.50
Shared Connection	0.50
No Shuffling	0

b) Data-out Connection Types

Table 3.7: Probability distribution for choosing interconnect type.

4. Finally, we decide if each memory is a ROM or a RAM. According to Table 3.2, 16% of the memories in the example circuits were ROMs. Thus, with a 16% probability, we determine that the memories in a given cluster are ROMs (meaning no data-in subcircuit needs to be generated). Each cluster is considered independently.

The current version of the generator does not generate circuits containing multiported memory. If the model was to be used in a study that needed such circuits, the model could be extended using the appropriate value from Table 3.2.

3.3.2 Choosing Memory/Logic Interconnect Patterns and Number of Data-in and Data-out Subcircuits

For each cluster, the form of the data-in and data-out network is chosen as follows:

1. The interconnection type for the data-out network is chosen according to the probability distribution in Table 3.7. Because of the small numbers of samples in gathering the statistics presented in Table 3.4, it may be misleading to use those numbers to generate probabilities. The conclusion to be drawn from that table is that all three interconnect types are well represented in the data-in case, and the first two are well represented in the data-out case. The probabilities in Table 3.7 reflect this conclusion. For each cluster, the number of data-in and data-out subcircuits is chosen based on the probability distribution in Table 3.8. These probabilities were obtained directly from the statistics in Figure 3.7.

Number	Probability
1	0.63
2	0.28
3	0.02
4	0.05
5	0.00
6	0.00
7	0.00
8	0.02

a) Data-in Subcircuits

Number	Probability
1	0.70
2	0.16
3	0.04
4	0.04
5	0.02
6	0.00
7	0.00
8	0.04

b) Data-out Subcircuits

Table 3.8: Probability distribution for choosing number of data-in and data-out subcircuits.

2. If a shared connection pattern is chosen, a multiplexor or a tri-state bus may be required to connect the logic and memory properly. If this is so, a multiplexor is constructed and included in the circuit. A multiplexor implementation was chosen since tri-state drivers are in short supply on many commercial FPGAs.

3. For clusters not containing ROMs, steps 1 and 2 are repeated for the data-in network.

Note that choosing the number of subcircuits and the associated interconnect pattern are, in general, not independent decisions. A point-to-point pattern with no shuffling can only be chosen if the number of logic subcircuits is the same as the number of logical memories. Similarly, if a regular point-to-point pattern is chosen, there must be enough logic subcircuits to sink (or source) every data pin in every logical memory. The number of subcircuits and the interconnect pattern are chosen independently. If the number of subcircuits and the chosen interconnect pattern conflict, a new number of subcircuits is chosen.

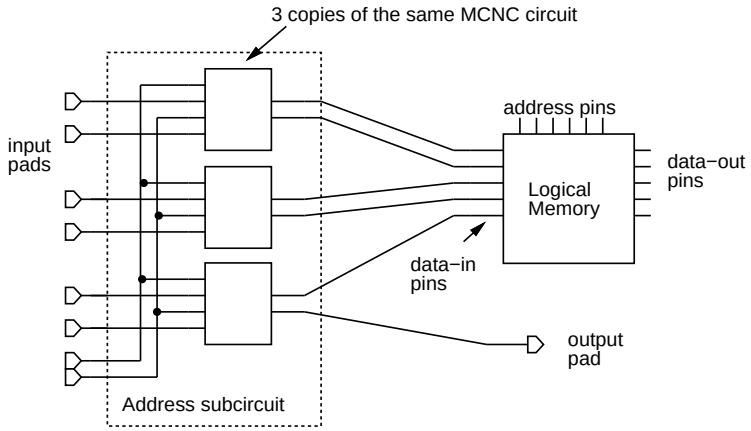


Figure 3.14: Example of connecting data-in subcircuit to a logical memory.

3.3.3 Generating Logic Subcircuits

Once the interconnect patterns and the number of subcircuits associated with each cluster are determined, the subcircuits themselves are generated. These subcircuits are chosen randomly from 38 benchmark circuits from the Microelectronics Center of North Carolina (MCNC) and listed in Table 3.9 [60]. The circuits are replicated enough times to supply the appropriate number of inputs or sink the appropriate number of outputs. The circuits in Table 3.9 are all combinational and contain between 24 and 184 five-input lookup tables. Each circuit was optimized using SIS [61] (logic-independent optimization) and technology mapped using FlowMap [62] into five-input lookup tables.

As an example, consider Figure 3.14. This diagram shows a 64x5 logical memory driven by a single data-in subcircuit (the address and data-out subcircuits are not shown). The MCNC circuit chosen to implement the subcircuit contains only 2 outputs; thus, it was replicated 3 times in order to provide the 5 required data lines. The sixth subcircuit output was not required, so it was connected to an output pad.

In Figure 3.14, a straightforward replication of the MCNC circuit would have resulted in 12 input pins (4 for each copy of the circuit). Assuming a bit-sliced design, it is likely that each copy of the circuit would share some inputs. In the figure, two inputs to each copy are shared among all copies. The extent to which inputs are shared is controlled by the parameter β, which is the proportion of all inputs that should be shared. Subsection 3.3.4 discusses the selection of a value for β.

Circuit	Number 5-LUTs	Number Inputs	Number Outputs
5xp1	40	7	10
9symml*	66	9	1
C432*	79	36	7
C499	66	41	32
C880	151	60	26
C1355	145	33	25
C1908	66	41	32
alu2	165	10	6
apex7	85	49	37
b12	40	15	9
b9	41	41	21
bw	28	5	28
c8	38	28	18
cht	37	47	36
clip	115	9	5
comp*	33	32	3
count	36	35	16
example2	119	85	66
f51m	30	8	8
frg1*	33	28	3
i2*	73	201	1
i5	66	133	66
inc	49	7	9
lal	32	26	19
misex2	32	25	18
my-adder	24	33	17
o64*	44	130	1
sao2	89	10	4
rd73	66	7	3
rd84	147	8	4
t481*	184	16	1
too-lrg*	162	38	3
ttt2	55	24	21
term1	83	34	10
unreg	32	36	16
vg2	43	25	8
x1	123	51	35
x4	144	94	71

Table 3.9: MCNC circuits used as logic subcircuits (* = only used as data sink subcircuit).

Data-out subcircuits are created in the same way, except that input-sharing is not done. If multiple data-in or data-out subcircuits are to be created, a different MCNC circuit is chosen for each, and inputs are not shared between subcircuits.

Also associated with each logical memory is a single address subcircuit. In clusters employing a shared-connection pattern using a multiplexor, a subcircuit is also required to drive the multiplexor select lines. These subcircuit are generated and connected in the same way as the data subcircuits.

Some of the circuits in Table 3.14 are marked with a *; this denotes circuits with a high input pin to output pin ratio. As an extreme example, consider circuit *i2*, which has 102 inputs and only 1 output. If this circuit is used to construct a data-in subcircuit feeding a memory with 8 data bits, the circuit must be replicated 8 times, meaning that the resulting subcircuit will have 816 inputs (less if input sharing is done). Clearly this sort of circuit is not amenable to driving the input pins of a wide memory. Because of this, circuits marked with a * are not considered for data-in or address subcircuits.

In actual memory circuits, the memories are most likely to be connected to datapath-oriented subcircuits. Table 3.9, however, includes a mixture of both datapath and random-logic (non-datapath) subcircuits. This will not likely influence the conclusions of architectural experiments based on these memory circuits, however, since:

1. As explained above, the circuits are replicated enough times to supply the appropriate number of inputs or sink the appropriate number of outputs. Thus, the composite circuits will have many properties of datapath circuits, even though the subcircuits upon which they are based may be random-logic.

2. Currently available placement tools usually have difficulty deducing structure in a datapath circuit [63]. Thus, *after placement*, routing a datapath-oriented circuit is not significantly different than routing a non-datapath oriented (random-logic) circuit.

An alternative to using MCNC circuits as building blocks would to be to use GEN, which generates combinational and sequential logic circuits [35]. This is left as future work.

Parameter	Default Value
β	0.5
γ	0.8

Table 3.10: Default values for β and γ.

3.3.4 Connecting Logic and Memories

For each cluster, the logic subcircuits are connected to the memories using the selected interconnect pattern, and then the clusters are combined into a single circuit, with no connections between the clusters. Unused circuit outputs are then randomly connected to unused inputs to create connections between logic subcircuits, often in different clusters. The parameter γ controls how extensively this is done. For circuits with more inputs than outputs, each output is examined individually, and with a probability of γ, it is connected to a randomly chosen input. For circuits with more outputs than inputs, each input is examined, and connected to a randomly chosen output with a probability of γ.

Estimating values of β and γ from block diagrams is difficult. Rather than choose values for these parameters based on gathered statistics, we have chosen values which result in reasonable I/O pin to logic ratios. The default values are shown in Table 3.10. Clearly, the higher the value of these two parameters, the lower the number of I/O pins.

3.3.5 Additional Constraints

The algorithm as described generates circuits based on the statistics presented earlier in this chapter. Alternatively, the user can further constrain the circuit generation. The constraints and their default values are shown in Table 3.11. After generating a circuit, the circuit is compared to the constraints, and if any constraints are violated, it is thrown away and a new circuit is generated.

Constraint	Default
Minimum, maximum number of clusters	1, 4
Minimum, maximum number of memories per cluster	1, 4
Minimum, maximum number of data-in subcircuits per cluster	0, 4
Minimum, maximum number of data-out subcircuits per cluster	1, 4
Minimum, maximum number of bits per circuit	1, ∞
Minimum, maximum number of logical memories per circuit	1, 16
Minimum, maximum number of 5-LUTs per circuit	1, ∞

Table 3.11: SCIRC constraints.

3.4 Summary

In this chapter, we have presented analysis of circuits containing both logic and memory. The analysis was based on 171 circuits obtained from 3 different sources. These circuits contained a total of 533 logical memories. Statistics regarding the number of memories, the width and depths of each memory, whether the memories were used RAMs or ROMs, whether they were single- or dual-ported were obtained. We also identified three different commonly-occurring patterns in the interconnection between the memory and logic portions of the circuits.

This chapter also described a circuit generator that generates stochastic, yet realistic circuits containing both logic and memory. The statistics gathered during the analysis are used as probability distributions in the selection of memory configurations and memory/logic interconnect patterns. The logic part of the circuits are constructed by connecting many small benchmark logic subcircuits together and to the memory.

Since the generator is based so closely on the circuit analysis, we can be confident that the generated circuits "look" realistic, but since they are generated stochastically, we can generate as many as desired. This will be essential in the architectural experiments that will be described in Chapters 5 and 7.

Chapter 4

Stand-alone Configurable Memories: Architecture and Algorithms

As shown in the last chapter, digital circuits have widely varying memory requirements. In order to implement these circuits on an FPGA or an FPGA-based system, a memory that can be configured to implement different numbers of logical memories, each with different widths and depths, is essential. This chapter describes a family of such architectures, similar to the FiRM Field-Configurable Memory described in [1, 34], along with algorithms to map logical memories to these devices.

The architecture described in this chapter is assumed to be stand-alone; that is, there are no logic blocks, and the address and data pins are connected directly to I/O pads. A key application of such a chip is a reconfigurable system consisting of FPGAs, memory devices, and interconnect [64, 65, 66, 67, 68]. In many of these systems, logical memories are packed into standard off-the-shelf memory chips. Usually, the logical memories will have a different size or aspect ratio than the physical memory devices. When this happens, either memory is wasted, or the logical memories must be time-multiplexed onto the memory chips [69]. By providing memory resources that can better adapt to the requirements of circuits, more efficient circuit implementations are possible.

The architecture described in this chapter can also be embedded into an FPGA to provide on-chip memory. The extension of this architecture to on-chip applications is the focus of Chapters 6 and 7.

The discussion in this chapter has two parts: Section 4.1 describes the family of configurable memory architectures, and Section 4.2 presents algorithms for mapping logical memory configurations to these configurable devices.

An early version the material in Section 4.1 also appears in [70].

4.1 Architectural Framework

The family of configurable memory architectures, illustrated in Figure 4.1, consists of B bits divided evenly among N arrays (each with a single data port) that can be combined to implement logical memory configurations. The effective aspect ratio of each array can be configured using the L1 data mapping block; this is discussed further in Section 4.1.1.

The address and L2 data mapping blocks programmably connect arrays to each other and to the external data and address buses. The external address and data buses are of a fixed width and are connected directly to I/O pads; the user can access memory only through these buses. We assume that a single bus can be used for only one logical memory at a time. Therefore, if a memory does not use the entire bus, the rest is wasted. This assumption simplifies the design of the L2 data mapping and address mapping blocks; the structure of these blocks is the focus of Section 4.1.2.

The parameters used to characterize each member of this architectural family are given in Table 4.1 (a complete list of symbols is given at the beginning of this dissertation). In the next chapter we vary these parameters and measure the effects on area, speed, and flexibility of the device; the range of parameter values considered in the next chapter is also shown in Table 4.1 (this is not meant to indicate *all* reasonable parameter values). Since each logical memory requires at least one array, one address bus, and one data bus, the maximum number of logical memories that can be implemented on this architecture is the minimum of M, Q, and N.

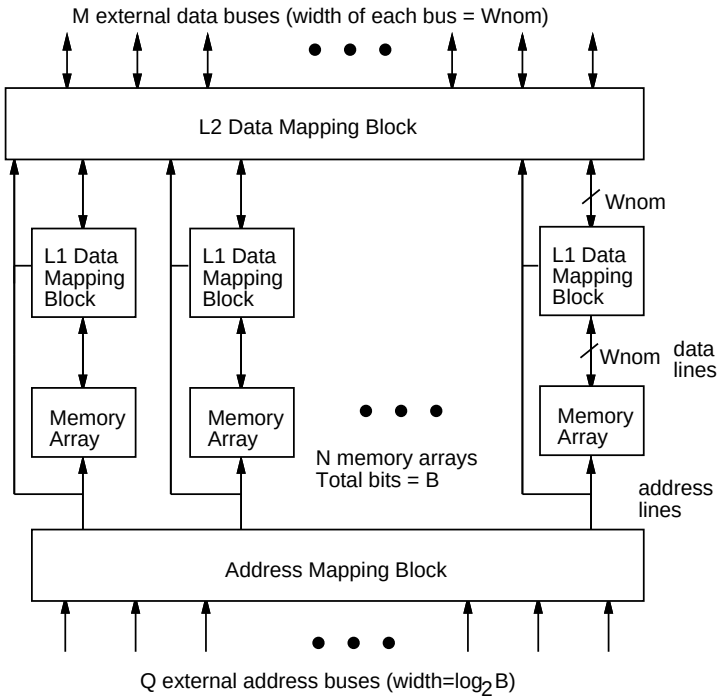


Figure 4.1: General architecture for a standalone FCM.

Parameter	Meaning	Range in next chapter
B	Total bits	4 Kbits - 64 Kbits
N	Number of arrays	4 - 64
M	Number of external data buses	2 - 16
Q	Number of external address buses	4 - 8
W_{nom}	Nominal data width of each array	1 - 16
W_{eff}	Set of allowable effective data widths of each array	several

Table 4.1: Architectural parameters.

4.1.1 L1 Data Mapping Block

The flexibility to implement many different numbers and shapes of memories is achieved by this architecture in two ways: by allowing the user to configure the effective data width of each array (trading width for depth), and by allowing the user to combine arrays to implement larger memories. First consider the effective data width of each array. Each array has a nominal width of W_{nom} and depth of $B/(NW_{\text{nom}})$. This nominal aspect ratio can be altered by the level 1 (L1) data mapping block. Figure 4.2(a) shows an example L1 data mapping block in which $W_{\text{nom}} = 8$. Each dot represents a programmable pass-transistor switch. The set of horizontal tracks to which each vertical track i can be connected

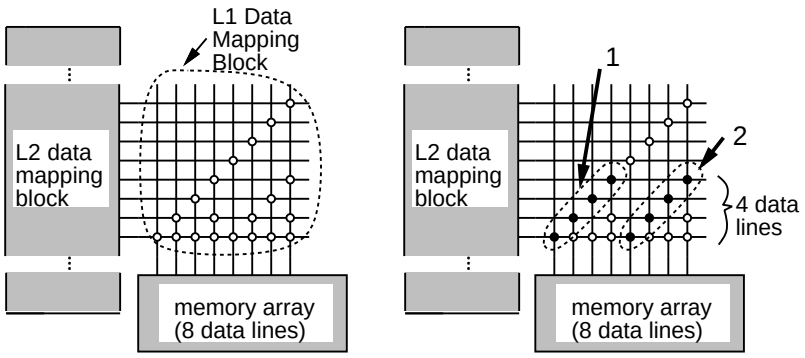


Figure 4.2: L1 data mapping block.

$(0 \leq i < W_{\text{nom}})$ is:

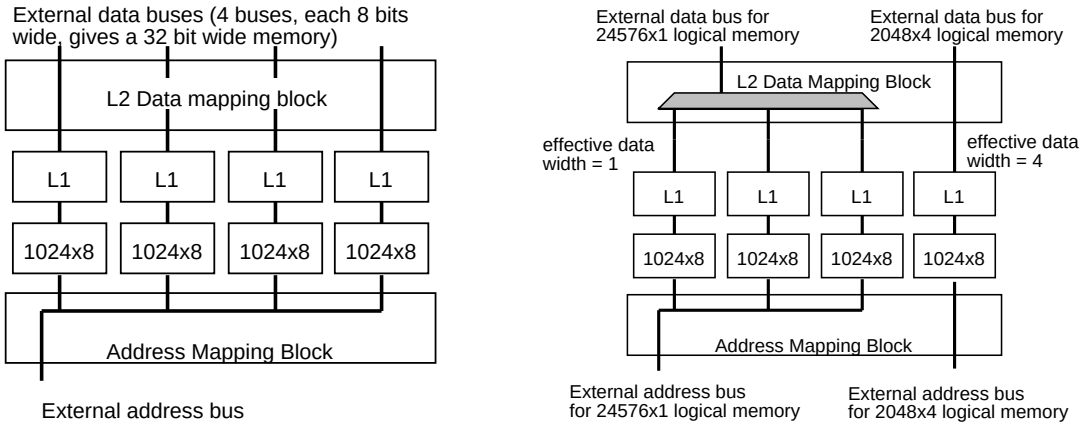
$$T(i) = \{i \bmod j \ : \ j \in W_{\text{eff}}\} \tag{4.1}$$

where W_{eff} is the set of allowable effective data widths of each array.

In the example of Figure 4.2(a), $W_{\text{eff}} = \{1, 2, 4, 8\}$, meaning each array can be configured to be one of $\frac{B}{N}$x1, $\frac{B}{2N}$x2, $\frac{B}{4N}$x4, or $\frac{B}{8N}$x8. Figure 4.2(b) shows two sets of switches, 1 and 2, that are used to implement the $\frac{B}{4N}$x4 configuration. One of the memory address bits is used to determine which set of switches, 1 or 2, is turned on. Each set of switches connects a different portion of the memory array to the bottom four data lines.

Notice that the mapping block need not be capable of implementing all power-of-two widths between 1 and W_{nom}. By removing every second switch along the bottom row of the block in Figure 4.2(a), a faster and smaller mapping block could be obtained. The resulting mapping block would only be able to provide an effective data width of 2, 4, or 8, however, meaning that the resulting architecture would be less flexible. The impact of removing L1 data mapping block switches on area, speed, and flexibility will be examined in the next chapter.

The detailed circuit diagram of the L1 data mapping block used in the FiRM chip can be found in [1].



a) 1024x32 b) 24Kx1 and 2048x4

Figure 4.3: Two example mappings.

4.1.2 L2 Data Mapping Block and Address Mapping Block

Memory flexibility is also obtained by allowing the user to combine arrays to implement larger memories. Figure 4.3(a) shows how four 1024x8 arrays can be combined to implement a 1024x32 logical memory. In this case, a single external address bus is connected to each array, while the data bus from each array is connected to separate external data buses (giving a 32-bit data width). Each L1 data mapping block connects 8 array data lines directly to the 8 L1 outputs.

Figure 4.3(b) shows how this architecture can be used to implement a configuration containing two logical memories: one 24Kx1 and one 2048x4. The three arrays implementing the 24Kx1 memory are each configured as 8192x1 using the L1 data mapping block, and each data line is connected to a single external data line using pass transistors. Two address bits control the pass transistors; the value of these address bits determine which array drives (or is driven by) the external data line. The 2048x4 memory can be implemented using the remaining array, with the L1 block configured in the "by 4" mode.

The extent to which arrays can be combined depends on the flexibility of the L2 data and address mapping blocks. The address mapping block must be flexible enough that each array that is used in a single logical memory can be connected to the same external address bus. The L2 data mapping block must be flexible enough to combine arrays in two ways:

1. Arrays can be combined horizontally to implement wider logical memories. In this case, each array must be connected to its own external data bus. An example of this is shown in Figure 4.3(a).

2. Arrays can be combined vertically to implement a deeper logical memory. In this case, the data buses of all arrays that implement this logical memory must be multiplexed onto a *single* external data bus. An example of this is shown in Figure 4.3(b).

Of course, some logical memories can be implemented by combining arrays both horizontally and vertically.

The topology of the switches in the L2 data mapping block and the address mapping block determines the flexibility of these blocks. If both of these mapping blocks are fully populated, meaning any external bus (both address and data) can be connected to any array, a very flexible architecture would result. Unfortunately, the switches take chip area and add capacitance to the routing lines, resulting in a larger and slower device than is necessary. The switch topologies shown in Figure 4.4 provide a compromise between speed and flexibility. In this figure, each dot represents a set of switches controlled by a single programming bit, one switch for each bit in the bus (W_{nom} in the L2 data mapping block and $log_2 B$ in the address mapping block). This pattern is similar to that of the L1 data mapping block described by Equation 4.1. In the L2 data mapping block, the set of buses to which array i can be connected ($0 \leq i < N$) is:

$$B(i) = \{i \bmod 2^j \ : \ 0 \leq j \leq log_2 M\} \tag{4.2}$$

where i and j are both integers. The pattern in the address mapping block is the same with M replaced by Q. Although in the examples of Figure 4.4, $M = Q = N$, the same formula applies for non-square mapping blocks. In the next chapter, this mapping block topology will be compared to other possible patterns.

Figure 4.5 shows how a mapping block defined by Equation 4.2 with $N = M = Q = 4$ can be used to perform the connections required in the examples of Figure 4.3.

Note that the switch pattern restricts which arrays can be used to implement each logical memory. In the example of Figure 4.5(b), if the first array had been used to implement the 2048x4 logical memory, then the 24Kx1 logical memory could not have been implemented. Algorithms that determine how arrays should be combined, decide which arrays should

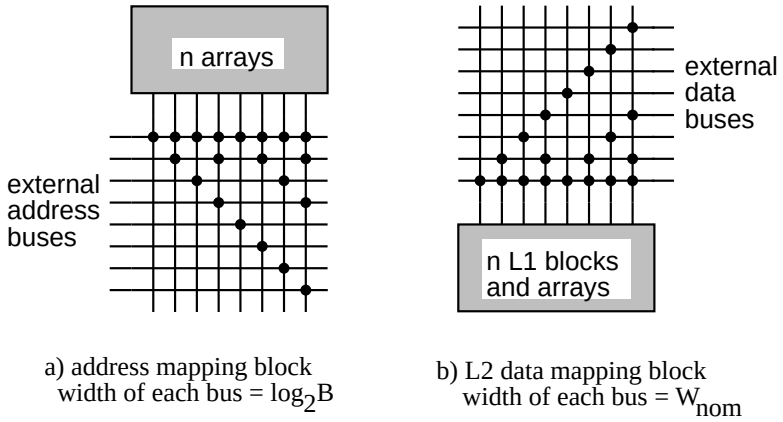
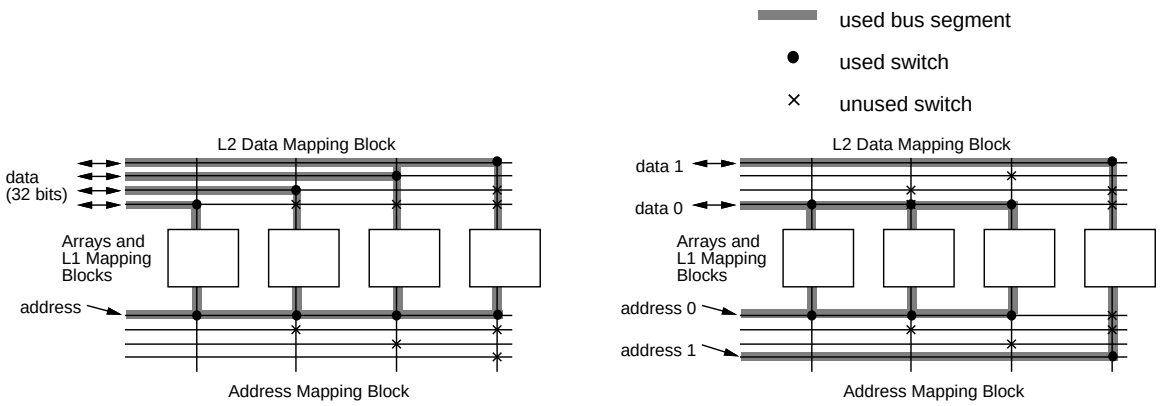


Figure 4.4: Level 2 data and address mapping block topology ($N = M = Q = 8$).



a) 1024x32 b) 24Kx1 and 2048x4

Figure 4.5: Implementation of two examples in Figure 4.3.

implement which logical memories, and decide how they should be connected to the external buses are the focus of Section 4.2.

It is important to note that all the bits of a bus are routed as a unit through the L2 data and address mapping blocks. Thus, a single programming bit can be used to control each set of W_{nom} switches in the L2 data mapping block and $log_2 B$ switches in the address mapping block. This results in a significant savings in area, but also reduces the flexibility of the architecture. In the example of Figure 4.5(b), only one bit of the bottom data bus is used; the others are wasted and can not be used for other logical memories.

In addition to address and data lines, write enable signals are required for each array. The write enable lines can be switched in the L2 mapping block just as the data lines are. In order to correctly update arrays for effective widths less than the nominal width, we assume that the arrays are such that each column in the array can be selectively enabled.

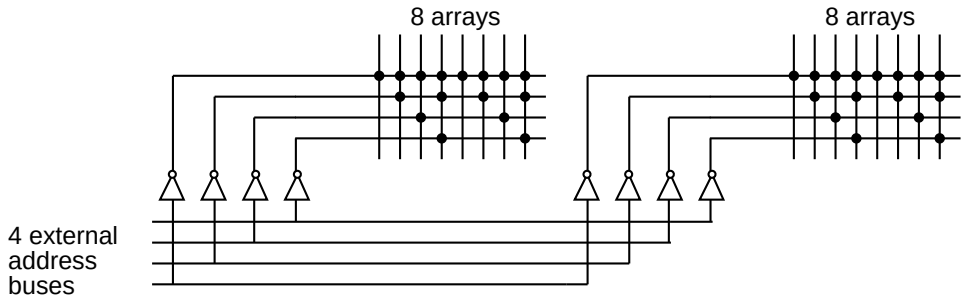


Figure 4.6: Two-stage address mapping block.

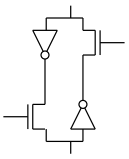


Figure 4.7: Bi-directional switches in two-stage L2 data-mapping block.

The address bits used to control the L1 mapping block can be used to select which array column(s) are updated.

The detailed circuit diagram of the L2 data mapping and address mapping blocks used in the FiRM chip can be found in [1].

4.1.3 Wide Mapping Blocks

For architectures with $N > 8$, the horizontal buses become heavily loaded and start dominating the memory access time. For these wide blocks, the two-stage structure shown in Figure 4.6 is assumed. This diagram shows the address mapping block; the L2 data mapping block is similar, with each inverter replaced by the bidirectional driver in Figure 4.7. The number of sub-buses that each bus should be broken into depends on the width of the original mapping block. Section 5.3 examines how this parameter affects the overall memory access time.

4.2 Algorithms

The family of architectures described in the last section provides an efficient, flexible mechanism for implementing field-configurable memory. Such an architecture is of limited use, however, without effective algorithms to map a user's memory configurations to the device. This section describes such algorithms. The algorithms are general enough to map to architectures with a wide variety of sizes and flexibilities, and provide near-optimal solutions over this wide architectural range.

The implementation of a logical memory on a physical memory architecture has two main steps:

1. assign physical arrays to logical memories and determine the effective data width of each array, and
2. combine these arrays using available routing resources and connecting them to the external buses.

The CAD problem described here is similar to well-known CAD problems (for example, implementing a logic circuit on an FPGA). The main difference is that because memories are somewhat regular, and do not offer as many implementation choices as logic offers, we can write algorithms that perform much better than their corresponding general-purpose counterparts. The algorithms presented in this chapter provide near-optimal solutions, even for architectures with very little flexibility (very few switches).

We have broken our algorithm into two parts as shown in Figure 4.8. The first phase, which we call *logical-to-physical mapping*, maps each logical memory to a set of arrays, and determines the effective data width of each of these arrays. This is analogous to technology mapping, in which a logic circuit is mapped into a set of lookup tables [62, 71].

The second phase in our CAD flow is analogous to placement and routing. In this phase, each mapped array is assigned a physical array such that it can be connected to the required number of external buses. Because there are relatively few items to place and wires to route, the placement and routing can be done simultaneously.

The remainder of this section describes these two algorithms in detail.

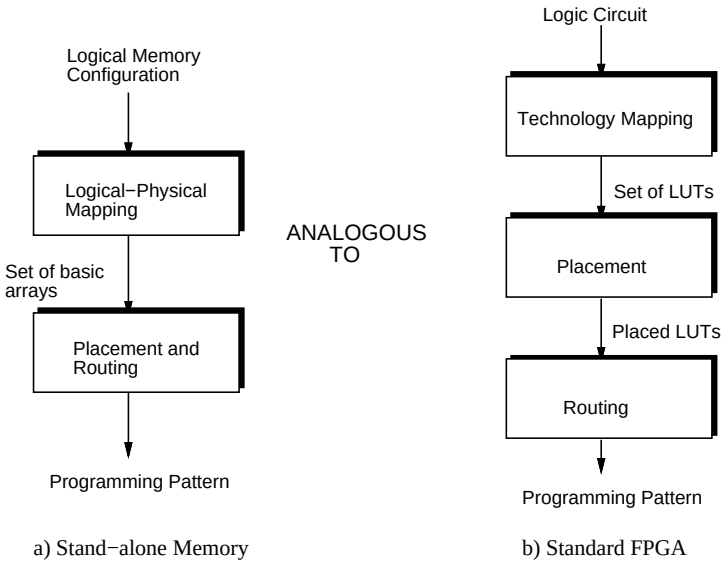


Figure 4.8: CAD flows.

4.2.1 Trivial Checks

Before attempting to map a logical memory configuration to an architecture, we do the following:

1. Ensure that the total number of bits in the logical memory configuration does not exceed the number of bits in the architecture.

2. Ensure that the total number of address and data pins required to implement the logical memory configuration does not exceed the number of address and data pins in the architecture.

3. Ensure that the number of logical memories does not exceed the number of address buses, data buses, and physical arrays in the architecture. Since each bus and array can only be used to implement a single logical memory, configurations that fail this test can not be implemented on the device.

These tests are trivial to perform and help quickly reject configurations which can not possibly fit into the target architecture.

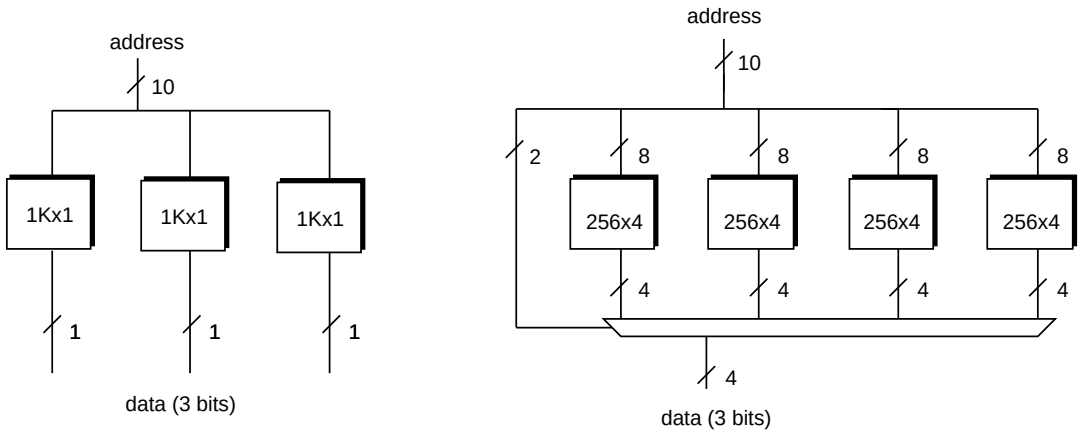


Figure 4.9: Two possible implementations of 896x3.

4.2.2 Logical-to-Physical Mapping

The first step in implementing a logical memory configuration is to map each logical memory onto arrays that are the size of the physical arrays on the target architecture. If a logical memory can fit in one physical array, the mapping is trivial. If more than one physical array is required, there are often several possible solutions; selecting between them is the goal of the logical-to-physical mapping algorithm.

Consider the implementation of the logical memory {863x3} using 1-Kbit physical arrays ($\frac{B}{N} = 1024$) each with allowable data widths of 1,2,4, or 8 ($W_{\text{eff}} = \{1, 2, 4, 8\}$). Figure 4.9 shows two possible implementations. The first requires three arrays configured as 1024x1, while the second requires four arrays configured as 256x4. In the second example, the data buses incident to the arrays are multiplexed onto a single external data bus. The L2 mapping block described in Section 4.1 can implement this. In the first solution, however, the L2 mapping block is not able to connect the data line from each array to a single external data bus, since the L2 mapping block switches each bus as a unit. As explained in Section 4.1.2, this reduces the number of programmable switches, and hence, the mapping block area. Thus, if the first solution were implemented, three external data buses would be used.

Which solution of Figure 4.9 is the best depends on the other logical memories within the configuration. Consider the example in Figure 4.10, in which the logical memory configuration {896x3,5Kx1} is mapped to an architecture with $N = 8$ (number of arrays), $M = 4$

(number of data buses), $\frac{B}{N} = 1024$ (bits per array) and $W_{\text{eff}} = \{1, 2, 4, 8\}$ (allowable data widths for each array). The 5Kx1 logical memory requires at least five arrays. Since the target architecture contains eight arrays in total, only three are left for the 896x3 memory. Thus, only the first solution in Figure 4.9 will lead to a valid implementation.

Now consider the example in Figure 4.11, which shows how the logical memory configuration {896x3,128x16} can be mapped. Here, the 128x16 memory requires at least two external data buses (since the width of each bus is 8 and the width of the memory is 16). Since the target architecture only contains four external data buses, only two are available for the 893x3 memory. Thus, the *second* solution in Figure 4.9 is used. These two examples illustrate the complexities involved in finding a valid logical-to-physical mapping.

In general, logical memory i will be implemented by n_i arrays, divided into s_i subsets or *mux-groups*. The number of arrays in each mux-group is denoted by $n_{i,j}$ $(0 \leq j < s_i)$. The data lines from these $n_{i,j}$ arrays are multiplexed onto a single external data bus; thus, the implementation of logical memory i uses s_i external data buses. The effective width of each array in mux-group j is denoted by $e_{i,j}$ (all arrays within a single mux-group have the same effective width). Table 4.2 shows values for each of these symbols for the two examples of Figures 4.10 and 4.11.

Each mux-group j implements a $e_{i,j}$-bit wide slice of a logical memory. Therefore, the total width of the implemented memory is the sum of the widths of all the mux-groups. If the total number of bits in each array is $\frac{B}{N}$, then the depth of the slice implemented by mux-group j is $\frac{B}{N e_{i,j}} * n_{i,j}$. To implement a logical memory of depth d, *all* slices must have a depth of at least d.

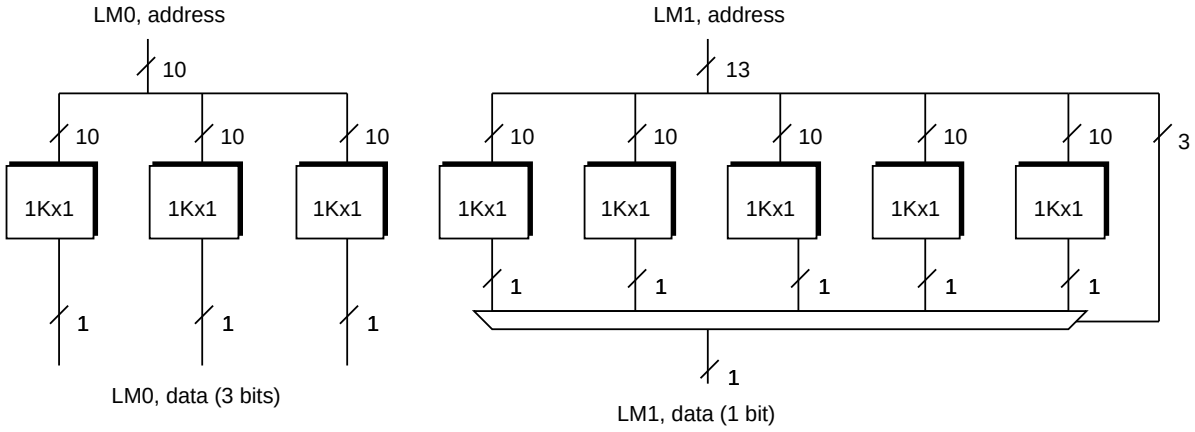


Figure 4.10: Netlist for {896x3, 5Kx1}.

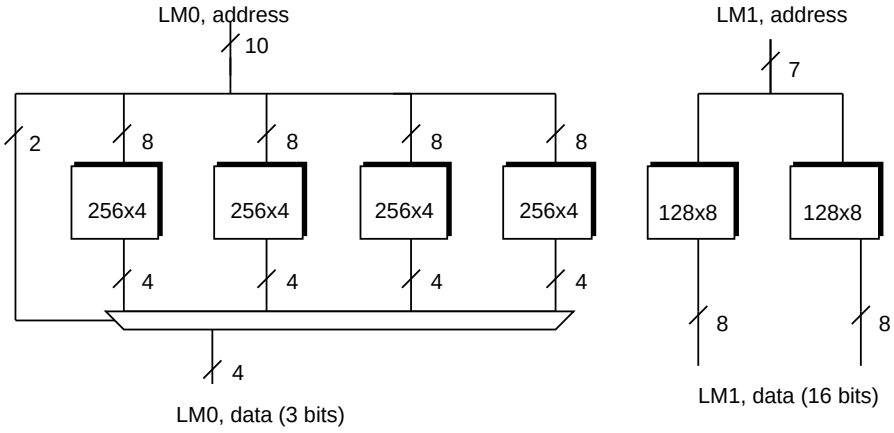


Figure 4.11: Netlist for {896x3, 128x16}.

The goal of the logical-to-physical mapping algorithm is to find values for these symbols. More precisely, the problem definition is:

Given:

1. A logical memory configuration consisting of z logical memories, each with depth d_i and width w_i $(0 \leq i < z)$,
2. A description of an FCM architecture: B, Q, N, M, and W_{eff}.

Find: Values of n_i, s_i, $n_{i,j}$, and $e_{i,j}$ for $0 \leq i < z$ and $0 \leq j < s_i$. These quantities define a logical-to-physical mapping consisting of, for each logical memory i,

1. The number of arrays to implement the logical memory, n_i,
2. A partitioning of the n_i arrays into s_i subsets $(1 \leq s_i \leq n_i)$. The size of each subset j is denoted by $n_{i,j}$,
3. The effective data width of the arrays in each subset j, $e_{i,j} \ \epsilon \ W_{\text{eff}}$.

Such that:

1. The logical-to-physical mapping "implements" the logical memory configuration, i.e. for each logical memory i $(0 \leq i < z)$,

$$\sum_{j=0}^{s_i-1} e_{i,j} \ \geq \ w_i \tag{4.3}$$

and

$$\left(\frac{B}{N e_{i,j}}\right) \ n_{i,j} \geq d_i \qquad \text{for all } j : 0 \leq j < s_i \tag{4.4}$$

2. The mapping uses N or fewer arrays:

$$\sum_{i=0}^{z-1} n_i \ \leq \ N \tag{4.5}$$

3. The mapping uses M or fewer external data buses:

$$\sum_{i=0}^{z-1} s_i \ \leq \ M \tag{4.6}$$

Symbol	Meaning	Figure 4.10	Figure 4.11
z	Number of logical memories	2	2
d_0	Depth of memory 0	896	896
w_0	Width of memory 0	3	3
d_1	Depth of memory 1	5K	128
w_1	Width of memory 1	1	16
n_0	Arrays for logical memory 0	3	4
s_0	Mux-groups for logical memory 0	3	1
$n_{0,0}$	Arrays for mux-group 0, logical memory 0	1	4
$n_{0,1}$	Arrays for mux-group 1, logical memory 0	1	-
$n_{0,2}$	Arrays for mux-group 2, logical memory 0	1	-
$e_{0,0}$	Effective data width, mux-group 0, log.mem 0	1	4
$e_{0,1}$	Effective data width, mux-group 1, log.mem 0	1	-
$e_{0,2}$	Effective data width, mux-group 2, log.mem 0	1	-
n_1	Arrays for logical memory 1	5	2
s_1	Mux-groups for logical memory 1	1	2
$n_{1,0}$	Arrays for mux-group 0, logical memory 1	5	1
$n_{1,1}$	Arrays for mux-group 1, logical memory 1	-	1
$n_{1,2}$	Arrays for mux-group 2, logical memory 1	-	-
$e_{1,0}$	Effective data width, mux-group 0, log.mem 1	1	8
$e_{1,1}$	Effective data width, mux-group 1, log.mem 1	-	8
$e_{1,2}$	Effective data width, mux-group 2, log.mem 1	-	-

Table 4.2: Symbol values for examples of Figure 4.10 and 4.11

The above definition does not restrict the number of address buses in the logical-to-physical mapping to be less than Q, even though this is clearly required for a valid implementation. Since the address ports of all arrays that implement a logical memory are connected together, the output logical-to-physical mapping will always require the same number of address buses as there are logical memories. Assuming the trivial checks in Section 4.2.1 pass, there will be at most Q logical memories, meaning the logical-to-physical mapping will require Q or fewer address buses.

The logical-to-physical mapping solution is not concerned with the mapping from logical-to-physical mapping elements to specific physical arrays on the device (i.e. placement), nor the assignment of logical-to-physical mapping inputs and outputs to specific physical buses on the device (i.e. routing). Section 4.2.3 will describe an algorithm that performs placement and routing.

Exhaustive Algorithm

An exhaustive algorithm for solving the logical-to-physical mapping problem for a single logical memory i is outlined in Figure 4.12. The outer loop runs through all possible values of s_i. For each value of s_i, all possible assignments of $e_{i,0}$ through e_{i,s_i-1} are found. Since each $e_{i,j}$ can take on one of $|W_{\text{eff}}|$ values, a naive algorithm would run through all $|W_{\text{eff}}|^{s_i}$ assignments. Many of these assignments are redundant, since the *order* of the j assignments is irrelevant. For example, if $s_i = 2$, the assignments $e_{i,0} = 2$ and $e_{i,1} = 4$ are equivalent to $e_{i,0} = 4$ and $e_{i,1} = 2$. A more accurate count of the number of possible combinations is obtained from the following formula (from [72]):

$$\text{number of combinations, } C = \left(\begin{array}{c} |W_{\text{eff}}| + s_i - 1 \\ s_i \end{array} \right) = \frac{(\ |W_{\text{eff}}| + s_i - 1\ )!}{s_i!\ (\ |W_{\text{eff}}| - 1\ )!} \tag{4.7}$$

For each of the C assignments for $e_{i,0}$ through e_{i,s_i-1}, the value of $n_{i,j}$ for each mux-group can be calculated from Equation 4.4, and n_i can be found by summing all $n_{i,j}$ for $0 \leq j < s_i$. It is then simple to check if the inequalities in Equations 4.3, 4.5, and 4.6 hold. If they do, a valid implementation has been found. The run-time of this algorithm is $O(M^2C)$.

For configurations with more than one logical memory, the logical memories can not be considered independently. Therefore, Equations 4.3, 4.5, and 4.6 must be calculated for all values of s_i, and all possible assignments for $e_{i,0}$ through e_{i,s_i-1} across all logical memories. If there are z logical memories, the run time of the algorithm is:

$$O((M^2C)^z) = O(M^{2z}C^z)$$

If $M = 4, |W_{\text{eff}}| = 4$, and $z = 4$, then $C = 35$ and $M^{2z}C^z = 9.83\text{x}10^{10}$. Clearly, this algorithm is not suitable for even moderate architecture sizes.

```
for s_i = 1 to M {
    for all possible assignments of e_{i,j} for 0 ≤ j < s_i {
        for j = 0 to s_i {
            calculate n_{i,j} from Equation 4.4
        }
        n_i = sum of n_{i,j} for 0 ≤ j < s_i
        if Equations 4.3, 4.5, and 4.6 hold then solution found, exit loop
    }
}
```

Figure 4.12: Exhaustive logical-to-physical mapping algorithm for one logical memory

Heuristic Algorithm

The run-time of the algorithm can be improved by making the restriction that each array that implements a single logical memory has the same effective data width (i.e. $e_{i,j} = e_i$ for all $j : 0 \leq j < s_i$) and the number of mux-groups implementing a single logical memory, s_i, is $\lceil w_i/e_i \rceil$. Figure 4.13 shows the new algorithm for a single logical memory. With this restriction, the run-time for a single logical memory is $O(|W_{\text{eff}}|)$. If there are z logical memories, the run-time is simply

$$O\left(|W_{\text{eff}}|^z \right)$$

If $|W_{\text{eff}}| = 4$ and $z = 4$, then $|W_{\text{eff}}|^z = 256$. Clearly, this is better than the exhaustive version, but is still too slow to be practical for architectures with $|W_{\text{eff}}|$ or z larger than 8.

If there are z logical memories, there are $|W_{\text{eff}}|^z$ combinations of e_0 through e_{z-1} that must be checked. The algorithm can be made more efficient by noting that, for most logical memories, many effective widths (values of e_i) do not make sense, and can be eliminated without considering the other logical memories in the configuration. For example, had the 128x16 memory in the second example of Section 4.2.2 been implemented with anything other than 128x8 arrays, the logical memory would require more than 2 arrays and more than 2 data buses (see Table 4.3). Thus, if the implementation using 128x8 arrays does not lead to a valid solution, none of the other implementations will.

A 893x3 memory is an example of a memory for which it is not obvious which array aspect ratio is the best choice. From Table 4.4, it can be seen that the 128x8 and 512x2 aspect ratios can be ruled out immediately. Since the 128x8 implementation requires the same number of data buses as the 256x4 implementation, but requires more arrays, the 128x8 implementation will never lead to a valid mapping if the 256x4 implementation doesn't. Similarly, the 512x2 implementation requires the same number of arrays as the

```
for each e_i ∈ W_eff {
    s_i = ⌈w_i/e_i⌉
    calculate n_{i,0} from Equation 4.4      /* n_{i,j} the same for all j */
    n_i = n_{i,0} * s_i
    if Equations 4.3, 4.5, and 4.6 hold then solution found, exit loop
}
```

Figure 4.13: First heuristic logical-to-physical mapping algorithm for one logical memory.

Array Organization	Arrays Required, n_i	Data buses required, s_i
1024x1	16	16
512x2	8	8
256x4	4	4
128x8	2	2

Table 4.3: Array and data bus requirements for 128x16.

Array Organization	Arrays Required, n_i	Data buses required, s_i
1024x1	3	3
512x2	4	2
256x4	4	1
128x8	7	1

Table 4.4: Array and data bus requirements for 896x3.

256x4 implementation, but needs an extra data bus. The remaining organizations, 1024x1 and 256x4, however, must both be considered as potential organizations for the arrays that implement this logical memory. The two examples in Section 4.2.2 show cases where each of the two organizations is the only one that can be chosen in order to implement the entire configuration.

In order to investigate how many array organization combinations can be eliminated in this way, we generated 1,000,000 logical memory configurations using the circuit generator described in Chapter 3. Of these, 422,156 passed the trivial checks described in Section 4.2.1. These 442,156 configurations contained a total of 609,597 logical memories. For each logical memory, we eliminated organizations according to the following rules (an algorithm that implements these rules is shown in Figure 4.14):

1. If several array organizations lead to implementations that require the same number of arrays but different numbers of data buses, eliminate all organizations except those that require the fewest number of data buses.

2. If several array organizations lead to implementations that require the same number of data buses but different numbers of arrays, eliminate all except those that require the fewest number of arrays.

3. If several array organizations lead to implementations that require the same number of arrays *and* data buses, eliminate all but one (it doesn't matter which).

```
output = φ
for j = 1 to m {
    arrays[j] = ∞
}
for all e ∈ W_eff {
    s_i = ⌈w_i/e_i⌉
    calculate n_{i,0} from Equation 4.4      /* n_{i,j} the same for all j */
    n_i = n_{i,0} * s_i
    if arrays[s_i] > n_i then {
       arrays[s_i] = n_i
       beste[s_i] = e
    }
}
ℓ = ∞
for j = 1 to m {
    if ( arrays[j] < ℓ ) {
       ℓ = arrays[j]
       output = output ∪ beste[j]
    }
}
```

Figure 4.14: Algorithm to eliminate unnecessary data-widths for logical memory i

The architecture described earlier was assumed: $N = 8$ (number of arrays), $M = 4$ (number of data buses), $\frac{B}{N} = 1024$ (bits per array) and $W_{\text{eff}} = \{1, 2, 4, 8\}$ (allowable data widths for each array). Table 4.5 shows a break-down of how many organizations remained after the elimination process was applied for each of the 609,597 logical memories. For most logical memories, only one sensible organization was found.

Number of Remaining Organizations $\lvert output \rvert$	Count
4	54
3	2232
2	16191
1	591120

Table 4.5: Breakdown of how many organizations remain after elimination process.

The elimination process described above is performed for each logical memory independently. Once a list of potential organizations for each memory is constructed, all combinations are considered (one element from each list), and those that violate the total array or total data bus constraints are discarded. Since most lists will contain only one element, this is very fast. Any combinations remaining represent valid solutions to the logical-to-physical mapping problem.

Occasionally, more than one valid solution is found. Without information regarding the location of the switches in the L2 data mapping block and the address mapping block, it is impossible to ascertain which of the valid solutions (if any) will result in a successful placement/routing (described in the next section). To avoid making an arbitrary decision at this point, *all valid solutions* are passed to the placer/router. The placer/router is then free to use which ever logical-to-physical mapping results in a successful place and route. This is in contrast to most technology mapping algorithms for logic circuits, in which only one valid mapping is passed to the placer/router.

As mentioned at the beginning of this section, this heuristic algorithm assumes that all arrays used to implement the same logical memory have the same aspect ratio. This assumption can sometimes cause the heuristic algorithm to miss a solution that the exhaustive algorithm would find. For example, the 896x3 memory considered earlier can more efficiently be implemented as shown in Figure 4.15 (the symbol values are in Table 4.6). When combined with another logical memory that requires 5 arrays and 2 data buses, this mixed-organization implementation of 896x3 is the only implementation that would work.

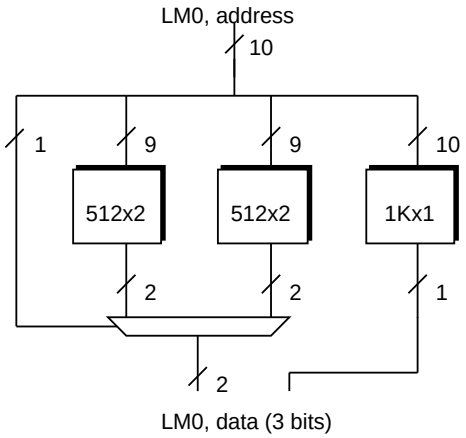


Figure 4.15: Efficient way to implement 896x3 that is missed by heuristic algorithm.

Symbol	Meaning	Figure 4.15
z	Number of logical memories	1
d_0	Depth of memory 0	896
w_0	Width of memory 0	3
n_0	Arrays for logical memory 0	3
s_0	Mux-groups for logical memory 0	2
$n_{0,0}$	Arrays for mux-set 0, logical memory 0	2
$n_{0,1}$	Arrays for mux-set 1, logical memory 0	1
$e_{0,0}$	Effective data width, mux-set 0, log.mem 0	2
$e_{0,1}$	Effective data width, mux-set 1, log.mem 0	1

Table 4.6: Symbol values for examples of Figure 4.15

.

4.2.3 Place and Route

Once the logical memories have been mapped to physical arrays, the next step is to assign these arrays physical locations in the configurable memory architecture, and to connect them to external pins:

Given:

1. A set of valid logical-physical mappings from the algorithm described in the previous subsection
2. A description of an FCM architecture: B, Q, N, M, and W_{eff}.

Find:

1. A one-to-one mapping from each array in the input logical-to-physical mapping to a physical array in the architecture,
2. A one-to-one mapping from each address bus in the input logical-to-physical mapping to a physical address bus in the architecture such that all required address bus to array connections can be made using the address block switches,
3. A one-to-one mapping from each data bus in the input logical-to-physical mapping to a physical data bus in the architecture such that all required data bus to array connections can be made using the L2 data block switches.

Determining the first mapping in the above list is analogous to the placement problem in a logic-based FPGA CAD flow and generating the remaining two mappings is analogous to the routing problem.

Because of the scarcity of routing switches in the architectures described in Section 4.1, the place and route tasks must be solved simultaneously. This is feasible because the mapped circuits from the logical-to-physical mapper contain only a few (not more than N) arrays, and the number of connections that must be routed is bounded by $Q + M$. This is in contrast to standard FPGA placement and routing problems in which the complexity of the architecture and circuits often force the two problems to be solved separately (although algorithms that combine both have been proposed [73, 74, 75]).

Exhaustive Algorithm

The following algorithm exhaustively tests all possible placements and routings for a given circuit on a given architecture:

```
for all input logical-to-physical mappings {
   for all array placements {
      for all address bus assignments {
         if sufficient switches for address bus and array assignment {
            for all data bus assignments {
               if sufficient switches for data bus and array assignment {
                  solution found, exit algorithm
               }
            }
         }
      }
   }
}
```

The outer loop runs through all logical-to-physical mappings from the algorithm described in the previous section (recall that if more than one valid mapping is found, they are all sent to the placer/router). The next loop runs through all possible array assignments. In the worse case, there are N arrays and $N!$ such assignments. The inner loops run through all possible data bus and address bus assignments. For each possible assignment, the algorithm determines if there are sufficient switches to connect the external buses to the arrays. If there are, a valid mapping has been found. If there are Q address buses and M data buses to assign, the overall worst case complexity of this algorithm is $O(N! * M! * Q!)$ (ignoring the possibility of multiple logical-to-physical mappings). Clearly, this algorithm is not feasible for large architectures (N, Q, or M greater than 8).

Heuristic Algorithm

In this section, we describe a heuristic algorithm to perform the same task. Like the heuristic algorithm for the logical-to-physical translation, we process each memory sequentially, beginning with the one that requires the fewest number of arrays. The algorithm, and its application to the example of Figure 4.10, is described below:

1. Sort the logical memories by number of arrays required for implementation.

2. Start with the logical memory that requires the *least* number of arrays. Let n_i be the number of arrays required for this memory. In our example, we start with the 896x3 memory, for which $n_i = 3$.

3. Find the *least flexible* available address bus (the one with the fewest possible connections) that can connect to at least n_i arrays. In our example, this would be address bus 7, since all less flexible address buses can connect to only 1 or 2 arrays (see Figure 4.16).

4. Divide the number of arrays, n_i, by the number of data buses required for this logical memory, s_i. This gives the number of arrays to which each data bus must connect, n_i/s_i. In our example, the 896x3 memory requires 3 arrays and 3 data buses; therefore, $n_i/s_i = 1$.

5. Find the s_i least flexible available data buses which each connect to n_i/s_i different arrays. Only consider arrays that can connect to the current address bus. In our example, data buses 1, 3, and 5 would be chosen (buses 2 and 4 can not be connected to arrays which can also connect to address bus 7).

6. If s_i such data buses can not be found, repeat step 5 for the next least flexible address bus. If there are no more address buses, repeat steps 1 through 5 for the next logical-to-physical mapping. If there are no more logical-to-physical mappings, the place and route fails.

7. Mark the chosen address and data buses, as well as the chosen arrays, and repeat steps 3 through 6 for the all remaining logical memories. Figure 4.16 shows the entire solution for our example.

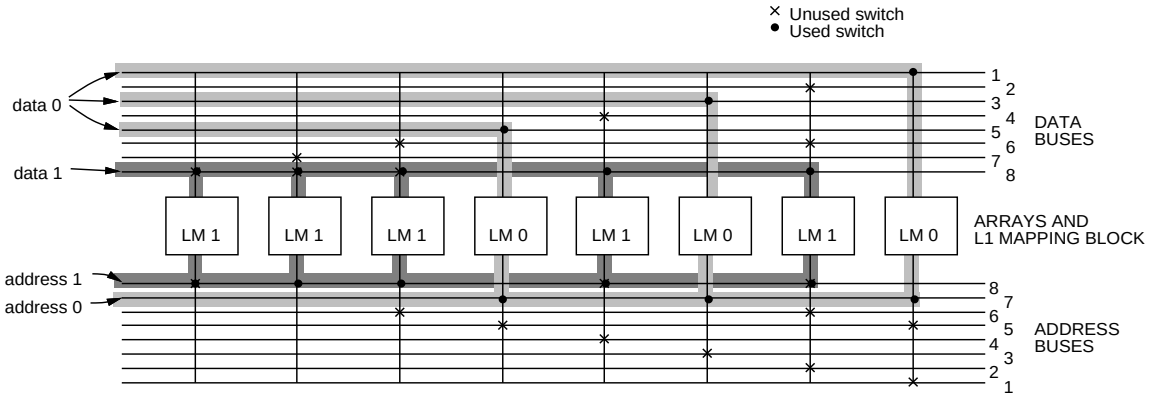


Figure 4.16: Solution for {896x3, 5Kx1}.

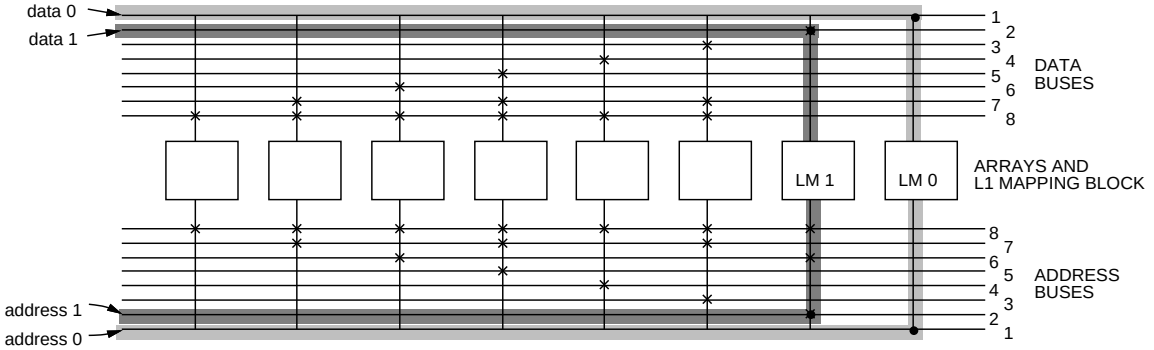


Figure 4.17: Partial solution for {128x8, 128x8, 512x12}.

As a second example, consider mapping the configuration {128x8, 128x8, 512x12} on the same architecture. The algorithm begins with the two smaller logical memories; the partial solution for these two memories is shown in Figure 4.17. The third memory requires 3 data buses; each of these buses must connect to 2 unused arrays. As can be seen in Figure 4.17, three such buses are not available. Had the second 128x8 memory been implemented using address bus 3 and data bus 3, however, the third memory could be implemented. This is an example where the exhaustive algorithm would find a solution that the heuristic algorithm misses. As will be shown in Section 4.2.4, such configurations are rare.

4.2.4 Mapping Results

In this section, we examine the effectiveness of the algorithms described in the previous subsections. In particular, we compare the heuristic place and route algorithm with the exhaustive version. We did not implement the exhaustive logical-to-physical algorithm, so heuristic vs. exhaustive comparisons for that problem are not available.

There are four reasons why the above algorithms might fail to map a given logical memory configuration to a physical architecture:

1. One of the trivial checks in Section 4.2.1 might fail.

2. The logical-to-physical mapper may be unable to find a mapping that requires N or fewer arrays. This can occur even if the total number of bits in the configuration is less than that of the architecture, since each physical array can only be used to implement a single logical memory at a time. If a logical memory does not use an entire array, those bits are wasted and can not be used for another logical memory. As will be shown in Section 5.4.1, increasing the basic array granularity (making more, smaller, basic arrays) reduces the number of mapping attempts that fail because of this reason.

3. The logical-to-physical mapper may be unable to find a mapping that requires M or fewer data buses. In architectures with $N = M$, this will never cause a failure if there are sufficient arrays. In architectures with $N > M$, however, there may be sufficient arrays but insufficient data buses.

4. The switches in the mapping blocks may not be sufficient for the placer/router to make the required connections between the basic arrays and external buses.

The algorithms described in the previous subsections were used to attempt to map 100,000 logical memory configurations onto an architecture with $B = 4096$, $N = Q = M = 4$, and $W_{\text{eff}} = \{1, 2, 4, 8\}$ (this is the FiRM architecture described in [34]). Configurations that failed any of the trivial checks in Section 4.2.1 were immediately discarded, as were configurations that used less than 75% of the total bits on the target architecture.

Table 4.7 shows mapping results obtained using the heuristic algorithms described in this chapter. As can be seen, of those configurations that passed the initial trivial checks,

Mapping Outcome	Heuristic P&R
Successful	88.49%
Insufficient physical arrays [1]	11.51%
Insufficient data buses	0%
Insufficient switches	0%

Table 4.7: Mapping results for FiRM architecture.

Mapping Outcome	Heuristic P&R	Exhaustive P&R
Successful	88.27%	88.62%
Insufficient physical arrays	8.58%	8.58%
Insufficient data or address buses	2.34%	2.34%
Insufficient switches	0.81%	0.46%

Table 4.8: Mapping results: $B = 8192$, $N = 8$, $M = Q = 4$, $W_{\text{eff}} = \{1, 2, 4, 8\}$.

88.5% could be successfully implemented by the heuristic algorithms. All the failures were because the logical-to-physical mapper could not find a mapping using four or fewer arrays. There were no failures because of insufficient data buses; any configurations requiring more than four data buses would also require more than four arrays. As stated in [34], the switch pattern is sufficient to perform all required mappings; the final number in Table 4.7 agrees with this result.

Because there were no failures in the place and route stage in this example, the exhaustive place and route algorithm would perform no better. As an example of an architecture where the exhaustive algorithm might find solutions that the heuristic place and router might not, consider an architecture with $B = 8192$, $N = 8$, $M = Q = 4$, $W_{\text{eff}} = \{1, 2, 4, 8\}$. Table 4.8 shows the mapping results obtained for 10,000 logical memory configurations using the heuristic and exhaustive place and route algorithms (the heuristic logical-to-physical mapper was used in both cases). As can be seen, the failure rate of the heuristic algorithm was only slightly worse than that of the exhaustive algorithm. The run-times of the two algorithms were very different however; the exhaustive algorithm took approximately 14 hours on an unloaded Sparc-4, while the heuristic algorithm took about 2 minutes.

[1] Includes mappings for which there were *both* insufficient arrays and data buses

4.3 Summary

This chapter has described a family of standalone memory architectures consisting of an interconnected set of arrays. There are two levels of configurability: the effective data width of each array and the interconnection between these arrays. Together, they result in an architecture that is flexible enough to implement a wide variety of logical memory configurations.

We have also described heuristic mapping algorithms that are able to take advantage of the architecture's flexibility. The first algorithm is analogous to the technology mapping phase in FPGA logic implementation; a logical memory configuration is mapped to a set of arrays, each of which can be implemented by a single physical array in the architecture. The second algorithm places the arrays in the configurable memory architecture and performs the required routing between the arrays and external pins. Although this is a heuristic algorithm, it was shown to perform virtually the same as an exhaustive algorithm for the same problem.

In the next chapter, we continue with the configurable memory architecture, and examine the effect that each of the architectural parameters in Table 4.1 has on the flexibility, speed, and area of the architecture.

Chapter 5

Evaluation of Stand-Alone Architectures

The standalone configurable memory architecture and mapping algorithms described in the previous chapter form the framework for a very flexible memory that can be used in a wide variety of applications. In order to create a good configurable memory, however, it is crucial to understand how various architectural parameters affect the flexibility and implementation cost of the device. In this chapter, we explore these tradeoffs. Specifically, we examine the effects of changing the array granularity, data bus granularity, L1 data mapping block structure, and the L2 mapping block switch patterns.

There are four competing goals when creating an FCM architecture:

1. The architecture should use as little chip area as possible.
2. The memories implemented on the architecture should have as low an access time as possible.
3. The architecture should be flexible enough to implement as many different logical memory configurations as possible.
4. For each logical memory configuration that the architecture can implement, the external pin assignments should be as flexible as possible.

After describing our experimental methodology in Sections 5.1 to 5.3, Section 5.4 compares memory architectures based on the first three criteria listed above. An early version of some material in Section 5.4 also appears in [70]. The fourth criteria, external pin flexibility, is the focus of Section 5.5.

5.1 Methodology

Although both SRAM technology and FPGA architectures have been studied extensively, this is the first (published academic) study that combines the two in order to study the architecture of field-configurable memory. As such, we can not rely on well-accepted experimental methods.

Figure 5.1 shows our experimental approach. For each architecture under study, we use the circuit generator of Chapter 3 to generate 100,000 benchmark circuits. The generator is constrained so that all circuits use between 75% and 100% of the available bits in the target device in order to stress the architecture. As explained in Chapter 3, the common technique of using 10 to 20 "real" benchmark circuits will not suffice for configurable memory architecture studies. Since circuits typically contain only a few logical memories, hundreds would be required to thoroughly exercise the architectures. Only by using the circuit generator of Chapter 3, can we create enough circuits for our experiments.

Each of the logical memory configurations in the 100,000 benchmark circuits is mapped to each architecture using the heuristic algorithms described in Chapter 4. Each mapping attempt either succeeds or fails; we count the number of failures and use the count as a flexibility metric. The architecture with the fewest failures is deemed the most flexible.

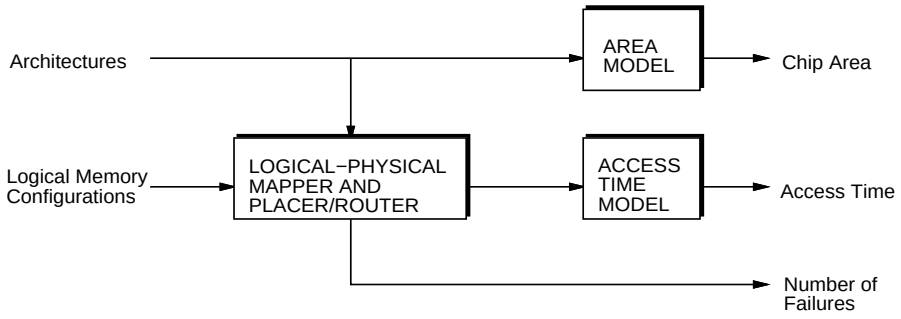


Figure 5.1: Methodology for standalone memory experiments.

In addition to flexibility results, we also obtain area and access time estimates for each architecture. Sections 5.2 and 5.3 describe detailed area and access time models developed specifically for this purpose.

5.2 Area Model

Chip area is an important consideration in the design of any integrated circuit. The larger a chip, the more expensive it will be, both because fewer larger chips can fit on a single wafer, and because the larger a chip, the higher the probability of a defect rendering the chip useless. Area comparisons will thus be an important part of the results in Section 5.4. This section describes a model that estimates the chip area required by the standalone configurable memory architecture as a function of the architectural parameters in Table 4.1.

In order to obtain a degree of process independence, a unit of area called a *memory bit equivalent* (mbe) is used. One mbe is the area required to implement one 6-transistor SRAM memory cell. This is the same approach as that employed in [76] and [77].

There are two components in the model: the area of each array and the area of the routing between the arrays (data and address mapping blocks). Each will be described separately.

5.2.1 Array Area Model

Figure 5.2 shows the assumed floorplan of a single array. The following equations are used to estimate the area of the storage array, the x-decoder, the x-driver, the y-decoder, the sense amplifiers, and the control block. As mentioned above, all area measurements are in mbe's.

$$
\begin{aligned}
A_{storage} &= x_{width} * y_{width} = B/N \\
A_{xdec} &= x_{width} * (\text{XDECFIXED} + \text{XDECPERBIT} * log_2(x_{width})) \\
A_{xdrive} &= \text{XDECDRIVER} * x_{width} \\
A_{ydec} &= y_{width} * (\text{YDECFIXED} + \text{YDECPERBIT} * log_2(y_{width}/W\text{nom})) \\
A_{sense} &= \text{SENSEAREA} * W\text{nom} \\
A_{control} &= \text{CONTROLAREA}
\end{aligned}
$$

The quantities in capital letters are technology dependent constants; Subsection 5.2.4 shows values and units for these constants estimated from three Bell-Northern Research memories and the FiRM layout [34]. The values of x_{width} and y_{width} indicate the dimensions (in bits) of the storage array.

The total area of one array is simply the sum of these quantities:

$$A_{array} = A_{storage} + A_{xdec} + A_{xdrive} + A_{ydec} + A_{sense} + A_{control}$$

In Section 5.2.2, the dimensions of each array will be required (the units for each dimension is mbe$^{0.5}$):

$$\begin{aligned} y_{array} &= y_{width} + \text{XDECDRIVER} + \text{XDECFIXED} + \text{XDECPERBIT} * log_2(x_{width}) \\ x_{array} &= x_{width} + \text{YDECFIXED} + \text{YDECPERBIT} * log_2\left(\frac{y_{width}}{w}\right) + \frac{A_{sense}}{y_{width}} \end{aligned}$$

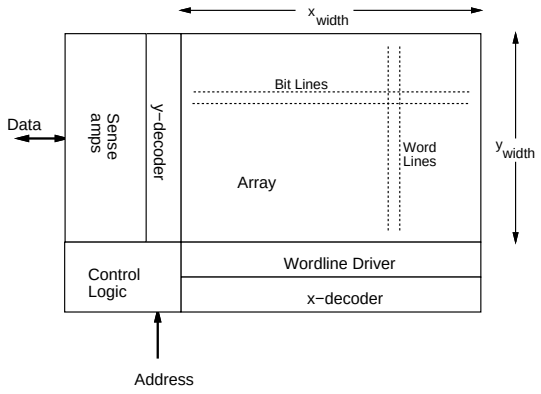


Figure 5.2: Floorplan of a single array (not to scale).

5.2.2 Routing Area Model

As described in Section 4.1.3, the L2 data and address mapping blocks can consist of either one or two stages. For architectures with one stage mapping blocks, the floorplan of Figure 5.3 is assumed. The address and data buses run down the middle of the layout, with arrays on either side. The floorplan for architectures with a two-stage mapping block is shown in Figure 5.4. Here, the upper level buses run horizontally and connect to the lower level buses at their intersections.

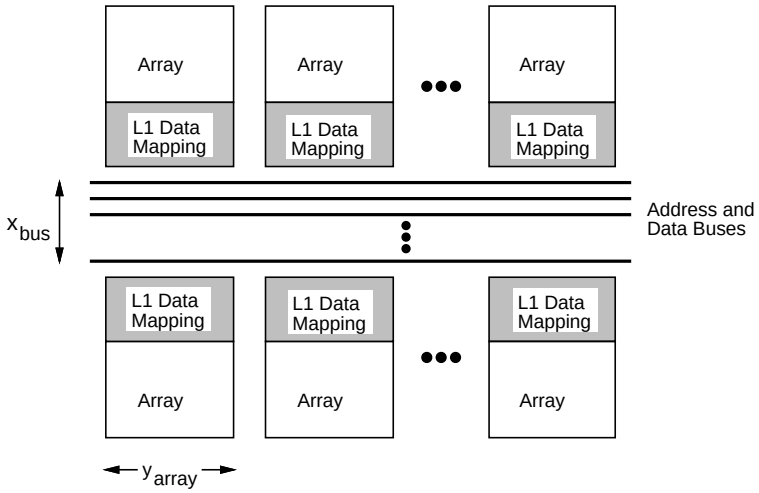


Figure 5.3: Assumed layout of memory architecture.

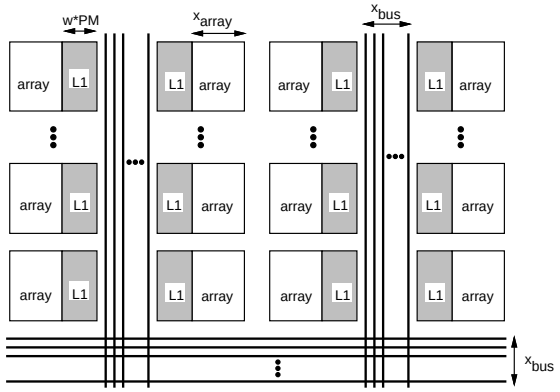


Figure 5.4: Assumed layout of memory architecture, 2 stages.

L1 Mapping Block

The area of the L1 mapping block shown in Figure 4.2 can be approximated by:

$$A_{L1} = y_{array} * w * \text{PM} + p_{bits,L1} * \text{PROGBIT}$$

where PM is the minimum metal pitch (minimum distance between two adjacent metal lines), PROGBIT is the area of one programming bit, and $p_{bits,L1}$ is the number of programming bits in the L1 mapping block.

Single level L2 data and address mapping blocks

In Figure 5.3, there are $M * W_{\text{nom}}$ data lines and $Q * \lceil log_2(B/W_{min}) \rceil$ address lines (where W_{min} is the smallest value in W_{eff}). Thus, the width of the routing channel is:

$$x_{bus} = Q * \left\lceil log_2 \left(\frac{B}{W_{min}} \right) \right\rceil * \text{PM} + M * W_{\text{nom}} * \text{PM}$$

The length of each of these lines is $N * y_{array}/2$; thus,

$$A_{bus,1} = x_{bus} * \frac{N * y_{array}}{2}$$

The total routing area for architectures with a single-level mapping block can be written as:

$$A_{routing} = N * A_{L1} + A_{bus,1} + p_{bits} * \text{PROGBIT}$$

where p_{bits} is the total number of programming bits required for the particular connection topology used, and PROGBIT is the size of a single programming bit.

Two-level L2 data and address mapping blocks

In these architectures, we must account for the area required by the upper-level buses. As shown in Figure 5.4, the length of these buses is $M_{sub} * (x_{bus} + 2 * w * \text{PM} + 2 * x_{array})$ where M_{sub} is the number of sub-buses and the total width is x_{bus}. Therefore,

$$A_{bus,2} = x_{bus} * M_{sub} * (x_{bus} + 2 * W_{\text{nom}} * \text{PM} + 2 * x_{array})$$

Parameter	Value
XDECDRIVER	6.4 $mbe^{0.5}$
XDECFIXED	0.46 $mbe^{0.5}$
XDECPERBIT	0.50 $mbe^{0.5}$/bit
YDECFIXED	3.0 $mbe^{0.5}$
YDECPERBIT	0.60 $mbe^{0.5}$/bit
SENSEAREA	30 mbe
CONTROLAREA	260 mbe
PROGBIT	7.8 mbe
PM	0.23 $mbe^{0.5}$

Table 5.1: Technology parameters for the area model.

The total routing area for these architectures is then:

$$A_{routing} = N * A_{L1} + A_{bus} + A_{bus,2} + p_{bits} * \text{PROGBIT}$$

5.2.3 Total Area

The total area is simply:

$$A = A_{routing} + N * A_{array}$$

5.2.4 Technology and layout parameters

Table 5.1 shows the technology parameters estimated from three Bell-Northern Research memories and Tony Ngai's FiRM layout[34].

5.2.5 Area Measurements

Table 5.2 presents the model predictions for three architectures. The first architecture is the FiRM architecture, containing four 1-Kbit arrays connected using four address and four data buses. The second architecture contains eight 1-Kbit arrays, connected using

Component	$B = 4K, N = 4,$ $M = Q = 4$ $W_{\text{eff}} = \{1, 2, 4, 8\}$	$B = 8K, N = 8,$ $M = Q = 4$ $W_{\text{eff}} = \{1, 2, 4, 8\}$	$B = 64K, N = 16,$ $M = Q = 8$ $W_{\text{eff}} = \{1, 2, 4, 8, 16\}$
memory bits	4096	8192	65536
x-driver	819	1638	6554
x-decoder	379	758	3543
y-decoder	538	1075	4301
sense amplifiers	960	1920	7680
control	1040	2080	4160
L1 data mapping blocks	928	1857	10339
L2 data mapping block	671	1343	30463
address mapping block	975	2104	30463
total	10407	20966	163039

Table 5.2: Area model predictions (results in mbe's).

four address and four data buses. The third architecture contains sixteen 4-Kbit arrays connected with eight address and eight data buses.

The routing structures make up 25%, 25% and 44% of the total area of the three architectures respectively. The area due to the L2 data mapping block and address mapping block is considerably higher in the third architecture than the first or second; this is because the larger architecture uses two-stage blocks as described in Section 4.1.3 to improve its access time. In the next section, we describe an access time model, and show the effect that a two-stage mapping block has on the speed of the device.

The first set of results in Table 5.2 can be compared to the FiRM layout. One mbe in a $1.2\mu m$ CMOS technology is approximately 580 μm^2. From [1], the core area of FiRM is 7.08 mm^2; this translates into 12204 mbe's. The difference between this number and the first total in Table 5.2 is because FiRM employs a different, less dense bus routing structure than the one assumed here. Unfortunately, we have no other physical devices to which we can compare the model.

5.3 Access Time Model

Another important characterization of a device is the speed at which it can operate. The advantages of a configurable memory will not ensure its viability if its access time is far greater than that of a comparable non-configurable off-the-shelf SRAM part. Thus, we need an accurate way to compare the access times of our proposed architectures.

This section describes such a model. The model consists of two components:

1. access time of an array
2. routing delay through mapping blocks

Each of these will be discussed separately.

5.3.1 Array Delay

The array access time model was based on the CACTI model [78, 79], which we originally developed to estimate the access time of on-chip microprocessor caches. This subsection highlights the relevant parts of the model; more details can be found in [78, 79]. Note that this portion of the model does not include the L1 data mapping block; this block will be considered as part of the routing delay in Subsection 5.3.2.

The array access time is composed of four components: the delay of the address decoder, the wordline rise time, the bitline and column multiplexor delay, and the sense amplifier delay. The original CACTI model contains other cache-specific components that were not used in this study.

The delay of each component was estimated by decomposing each component into several equivalent RC circuits, and then using an equation due to Horowitz [80] to estimate the delay of each stage. Horowitz's equation accounts for non-zero rise/fall times of component inputs, which we found to have a significant effect on the overall delay. The equation in [80] assumes equal threshold voltages among all gates; we modified the equation slightly to account for gates with different threshold voltages. The bitline circuitry is the most complex of the components; rather than using Horowitz's equation for the bitline delay, we used a novel technique described in [78] in order to account for non-zero input rise/fall times.

In [78], results from CACTI are compared to those obtained from Hspice. As shown in the paper, the CACTI estimates are within 6% of the Hspice measurements.

Component	Model Prediction (ns)
address mapping network	0.456
array decoder	0.850
array wordlines	0.450
array bitlines and sense amplifiers	0.369
data mapping network	0.506
total	2.63

Table 5.3: Model predictions ($B = 8$Kbits, $N = 8, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$).

5.3.2 Routing Delay

The L1 and L2 data and address mapping blocks were represented by an RC-tree, and the delays estimated using the first-order Elmore delay [81]. As described in Section 4.1.3, mapping blocks in architectures with a high N (more than 8) are assumed to consist of two stages; the delay of each stage is estimated independently, and the results combined to give the overall routing delay. The resistance used for each pass transistor was the "full-on" resistance from [78].

5.3.3 Delay Results

Table 5.3 shows the access time model predictions for an architecture consisting of eight 1-Kbit arrays connected using four address and four data buses. CACTI assumes a 0.8μm CMOS technology; we divided all access times by 1.6 to more closely reflect a 0.5μm technology (this approach was also used in [77]). As the table shows, the overhead in the two mapping networks represents about 37% of the overall memory access time. Note that the timing values will vary slightly depending on which memory configuration is implemented; the results in Table 5.3 are for a 8Kx1 logical memory.

The architecture in Table 5.3 is small, and thus, can most efficiently be implemented using a single-stage L2 mapping block and address block. As described in Section 4.1.3, however, a hierarchical interconnect scheme can be used for wider mapping blocks in order to reduce the memory's access time. Timing results for an architecture for which this is appropriate are shown in Figure 5.5. This architecture consists of 32 arrays and 8 data and address buses. The horizontal axis in this graph is the number of sub-buses in each mapping block. As can be seen, the minimum overall access time occurs for a mapping

block with 4 sub-buses, each connecting $\frac{32}{4}$ arrays. The speed improvement obtained by using the hierarchical routing structure is approximately 20% for this architecture.

Since FiRM uses a different technology (1.2μm vs. 0.5μm) and uses a slightly different decoder, sense amplifier, and wordline driver size than those assumed in the CACTI model, direct comparisons between the model and the measured FiRM delay are not meaningful.

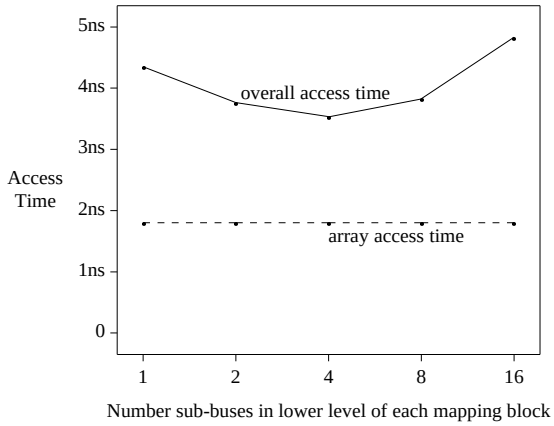


Figure 5.5: Access time predictions: $B = 64K, N = 32, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8\}$.

5.4 Architectural Study

As stated at the beginning of this chapter, in order to create a flexible but efficient configurable memory architecture, it is vital to understand how the architectural parameters in Table 4.1 affect the flexibility, access time, and chip area of the resulting device. In this section, we examine four parameters: the number of basic arrays (N), the capability of the L1 mapping block (W_{eff}), the number of data buses (M), and the L2 mapping block switch patterns. Our goal here is not to uncover the single best architecture, but rather to understand how each of these design decisions affects the efficiency and flexibility of the device.

5.4.1 Number of Basic Arrays

One of the key architectural parameters is the number of arrays, N. This subsection examines the impact of changing N while keeping the total number of bits constant.

The motivation for increasing N is flexibility. Since each array can be connected to at most one address bus at a time, it can only be used to implement one logical memory. If a logical memory does not use the entire array, the remaining bits are wasted. This is especially a problem when implementing configurations with many small logical memories. As an example, an architecture with four 1-Kbit arrays can implement at most four logical memories, no matter how small they are. An architecture with eight 512-bit arrays, however, can implement configurations with up to eight logical memories if there are sufficient data buses.

Figure 5.6 shows flexibility results as a function of N for two architectures: the first architecture contains 8 Kbits of memory and four address and data buses, while the second contains 64 Kbits of memory and eight address and data buses. In each case, N is varied from its minimum value (it does not make sense to have $N < Q$) to 64. The vertical axis in each graph shows the proportion of the 100,000 stochastically generated logical memory configurations that could not be mapped. In each graph, the configurations that could not be mapped are broken into three categories:

1. The lower dashed line indicates the mapping attempts that failed because the logical-to-physical mapper could not find a mapping using N or fewer arrays.

2. The distance between the two dashed lines indicates the configurations for which the logical-to-physical mapper could not find a mapping requiring M or fewer external data buses.

3. The distance between the upper dashed line and the solid line indicates the configurations that failed in the placer/router stage; that is, the switches in the L2 mapping block and address mapping blocks were not sufficient to perform the required mapping.

Configurations that fail for more than one of the above reasons are classified into the first appropriate failure category.

As the two graphs show, for low values of N, the predominant cause of failures is that the logical-to-physical mapper can not find a mapping requiring N or fewer arrays. As N increases, the blame shifts to an insufficient number of external data buses or an insufficient number of switches in the mapping blocks. The total number of failures, however, drops as N increases. For the first architecture, the number of failures is reduced by 28% over the

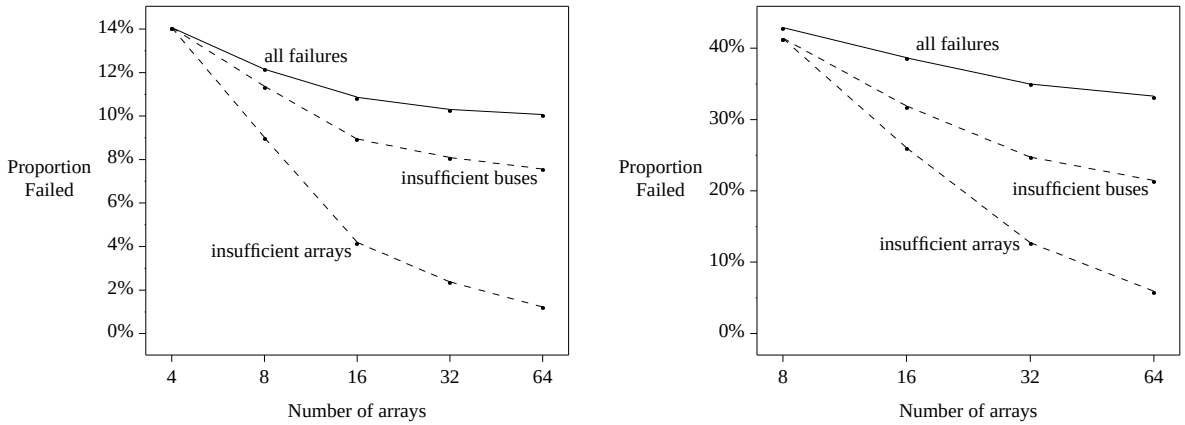
range of the graph, while for the second architecture, the number of failures is reduced by 22%.

Figure 5.7 shows how the memory access time is affected by N for the same two architectures. The access time is broken into two components: the array access time and the delay due to the mapping blocks. For each value of N we tested both one and two level mapping blocks; for the two level blocks we varied the number of sub-buses in the lower level. The mapping block that resulted in the minimum access time was chosen.

As N increases, each array gets smaller; these smaller arrays have lower access times as shown by the dotted lines in Figure 5.7. This is overshadowed, however, by an increase in the delay due to the mapping blocks; as N increases, these mapping blocks get larger, causing the overall access time to increase. For the smaller architecture, the access time increases by about 38% over this range, while for the larger architecture, it increases by about 24%.

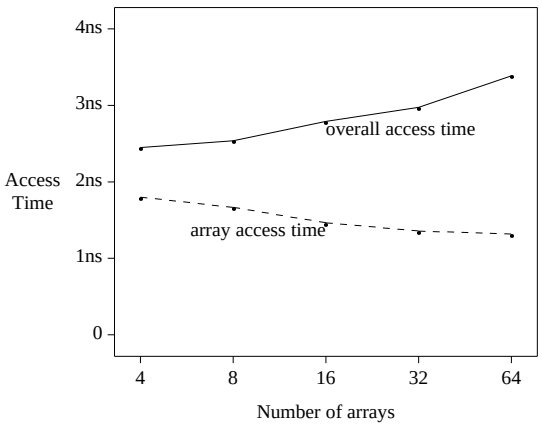
Finally, Figure 5.8 shows the chip area required by each architecture as a function of N, again broken into two components: the area of the arrays (along with their support circuitry) and the area due to the mapping blocks. As N increases, the arrays get smaller, meaning the overhead due to the array support circuitry increases. The mapping blocks also get larger as N increases.

Combining the access time, area, and flexibility results, we can conclude that a good FCM architecture has a moderate number of basic arrays (in our architectures, 2 or 4 times

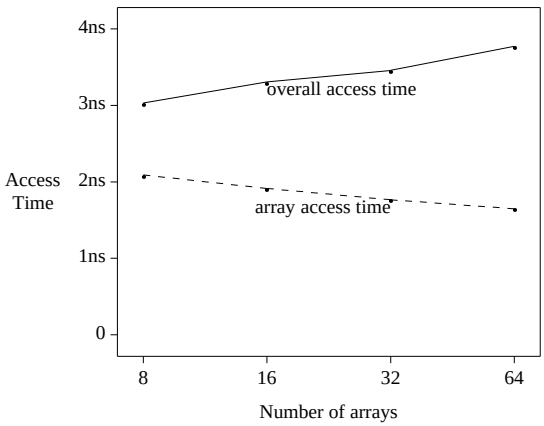


a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$ b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Figure 5.6: Number of failures as a function of number of arrays (N).

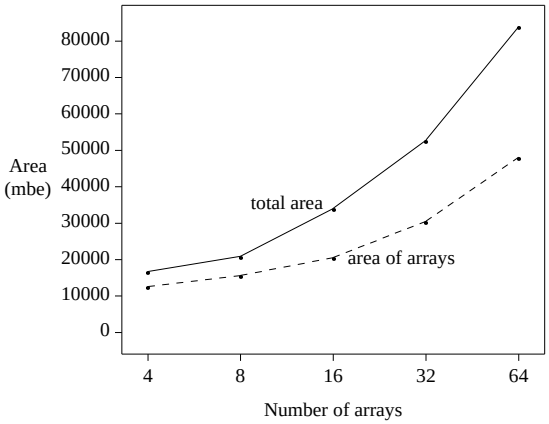


a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$

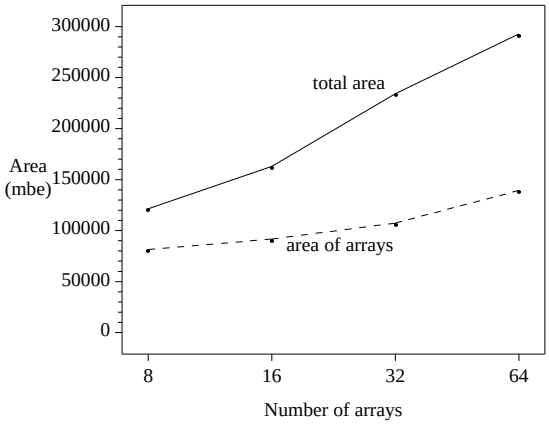


b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Figure 5.7: Access time as a function of number of blocks (N).



a) $B = 8K, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$



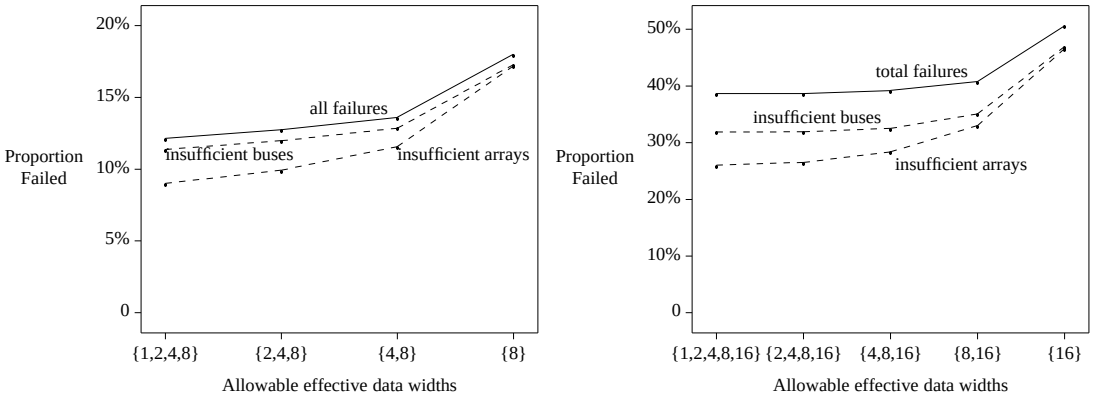
b) $B = 64K, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$

Figure 5.8: Chip area as a function of number of arrays (N).

the number of address buses is a good choice). If more arrays are used, the area increases considerably, but without as much of an increase in flexibility.

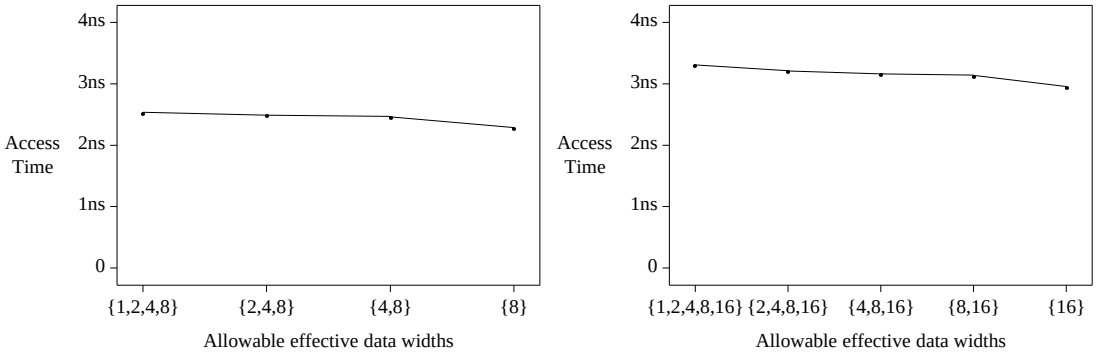
5.4.2 L1 Mapping Block Capability

The effective data width of each array can be set by configuring the L1 data mapping blocks. In the previous set of results, it was assumed that the set of effective output widths, W_{eff}, consisted of all powers-of-two between 1 and the nominal array width W_{nom}. Section 4.1.1 described how a faster, but less flexible architecture could be obtained by removing some of the capability of the L1 data mapping block. In this section, we investigate the effects of changing the minimum effective data width (smallest value of W_{eff}).



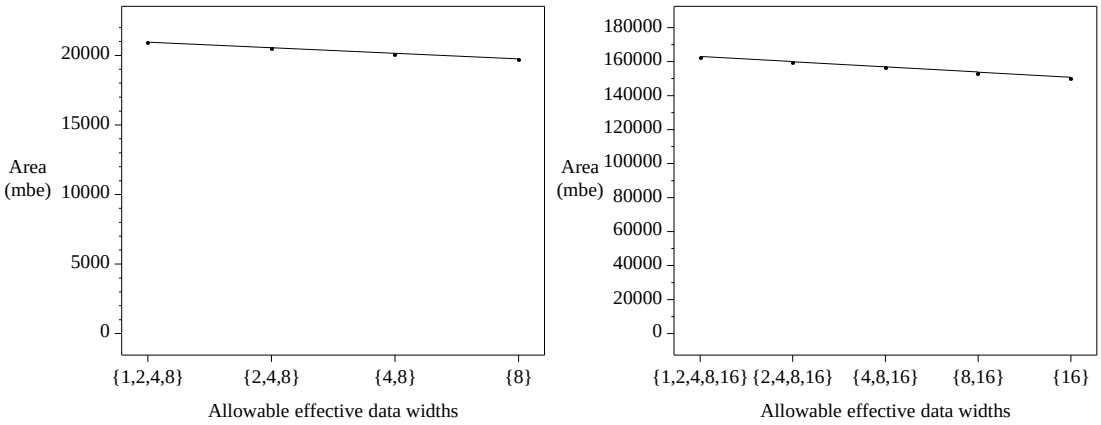
a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

Figure 5.9: Number of failures as a function of L1 mapping block capability.



a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

Figure 5.10: Access time as a function of L1 mapping block capability.



a) $B = 8K, N = 8, M = Q = 4$ b) $B = 64K, N = 16, M = Q = 8$

Figure 5.11: Chip area as a function of L1 mapping block capability.

Intuitively, the higher the minimum data width, the less flexible the L1 mapping block, and hence, the less flexible the architecture. As shown in Figure 5.9, this is indeed the case; decreasing the L1 block flexibility causes the logical-to-physical mapper to fail more often. The difference is more noticeable in the smaller architecture. Recall that we are only including configurations that use between 75% and 100% of the available bits; configurations aimed at the larger architecture are more likely to have wide data widths, meaning an L1 block that can be configured in the "x1" or "x2" configuration is less important.

Figures 5.10 and 5.11 show that the speed and area benefit of the simpler L1 mapping blocks is small. For both architectures, the access time of a memory employing the least flexible L1 mapping block is only 10% less than that for a memory employing the most flexible mapping block. The area requirements of a memory with the least flexible mapping block are only 6% and 7% less than if the most flexible block was used for the two architectures considered. Thus, we conclude that, especially for small architectures, a flexible L1 mapping block is most appropriate.

5.4.3 Data Bus Granularity

In Section 5.4.1, the number of failures due to insufficient arrays was reduced by increasing the array granularity (creating more, but smaller, arrays). Unfortunately, most of the gain was lost because the architecture did not contain enough external data buses. In this section, we vary the number of external data buses in an attempt to reduce this effect.

In order to perform fair comparisons, we fix the total number of data pins. In the results in this section, there are 128 data pins; we examine architectures in which these 128 pins are broken into 2, 4, 8, and 16 buses (of 64, 32, 16, and 8 bits respectively). In order to concentrate on the data bus effects, we fix the number of arrays at 16 (each containing 4 Kbits) and the number of address buses at 8.

Figure 5.12 shows the failure rates for the three architectures. As the number of data buses is increased, the failures due to insufficient buses is reduced, as expected. This is offset, however, by an increase in the number of failures due to insufficient arrays. The increase in failures due to insufficient arrays is because as the data width decreases, each array becomes less capable, and thus, more of them are needed to implement wider logical memories. For example, in the 16-bus architecture, each bus is only 8 bits wide. This means

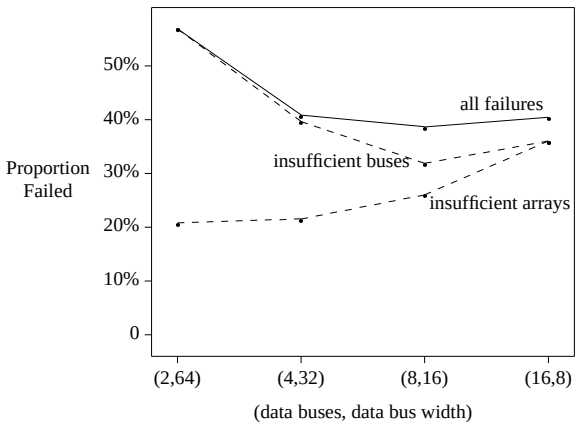


Figure 5.12: Failures as a function of data bus width: $B = 64K$, $N = 16$, 128 data pins.

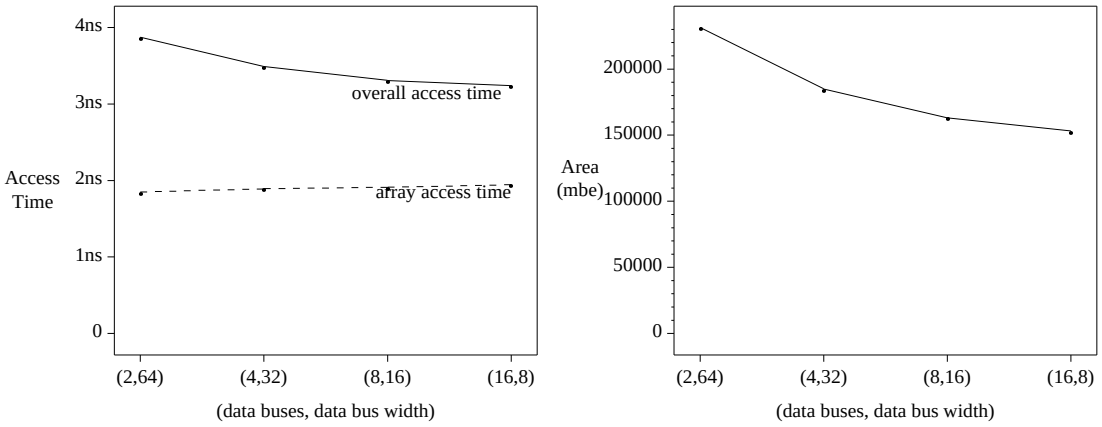


Figure 5.13: Delay/area as a function of data bus width: $B = 64K$, $N = 16$, 128 data pins.

that the maximum data width (highest value in W_{eff}), and hence the maximum number of data bits that can be extracted from each array, is 8. Thus, a 32-bit wide logical memory will require at least four arrays, regardless of the memory's depth. This is in contrast to the 4-bus architecture, in which 32 bits can be extracted from each array. The same wide logical memory on this architecture would require only one array (as long as the depth is less than 128). Overall, the number of failures drops by 29% over the range of the graph.

Figure 5.13 shows the area and delay results for the same set of architectures. There is an area advantage for architectures with more buses; a narrower bus means fewer sense amplifiers are required in each array. A narrower bus also means a less complex L1 mapping block. These two factors offset the fact that the architectures with more external data buses have more complex L2 mapping blocks. Over the range of the graph, the access time drops 16% and the area drops 34%.

Overall, the results indicate that an architecture with many narrow data buses is better in terms of speed, area, and (with the exception of the last data point) flexibility.

5.4.4 L2 Switch Patterns

Until now, all results have assumed that the L2 mapping block and address mapping block employ the switch pattern defined by Equation 4.2. In this section, we compare this pattern to three other patterns; the four patterns are shown in Figure 5.14. For each pattern, an equation giving the set of buses to which each array can connect is also given. Intuitively, the more switches within the mapping blocks, the more flexible the overall architecture will be, but at the expense of slower logical memory implementations and more chip area.

Table 5.4 shows the flexibility, delay, and area results for an architecture with eight 1-Kbit arrays and four address and data buses. The column labeled "P&R Failures" shows the proportion of all configurations that failed *during* the place-and-route algorithm (clearly, the switch pattern has no effect on the logical-to-physical mapper, so configurations that failed during that phase are not included in the table). Pattern 1 does not provide sufficient flexibility; 85% of the configurations could not be mapped. Pattern 2, which is the pattern assumed in all other results in this section, can be used to implement all but 7.8% of the configurations. The area and delay penalties when moving from pattern 2 to 3 are small (0.3% and 0.7% respectively). With pattern 3, however, every configuration could be mapped. Thus, for this architecture, pattern 3 seems to be the best choice.

Table 5.5 shows the results for a larger architecture (16 arrays and 8 address and data buses). For this architecture, the delay penalty from moving from pattern 2 to 3 is 4%, and the area penalty is 17%. The dramatic increase in area is because more sub-buses are required; each of the sub-buses needs a second-level driver. If the number of sub-buses was held constant, the access times for patterns 3 and 4 would be considerably larger than they are. From the results in the two tables of this section, we conclude that the FiRM-style mapping block (pattern 2) is flexible enough to perform most required mappings, but at a lower area and speed cost than patterns 3 and 4 (especially for large architectures).

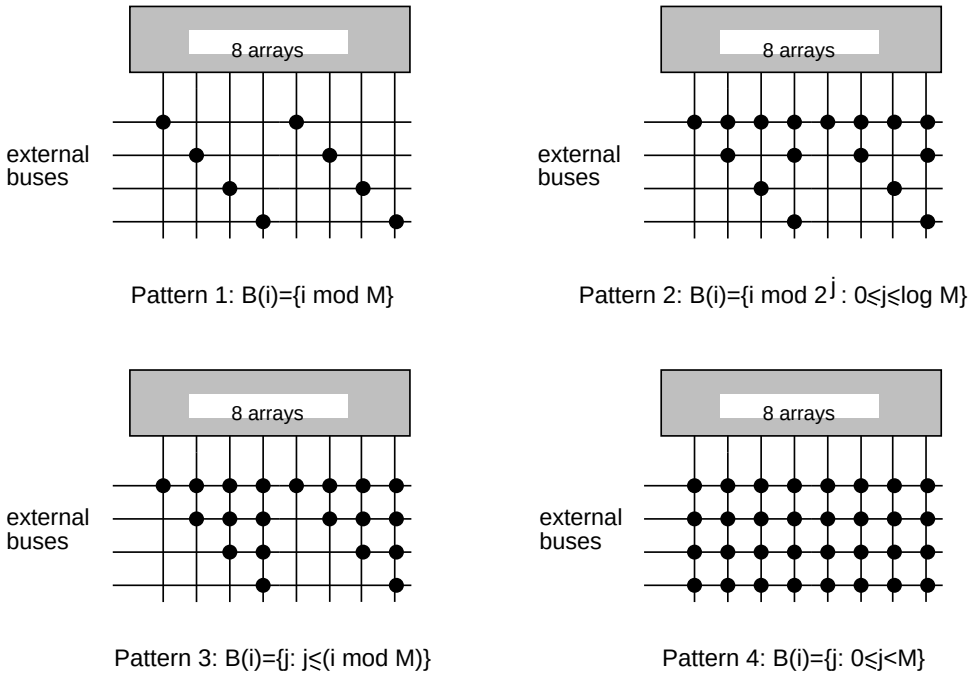


Figure 5.14: L2 data block/address block switch patterns considered.

Pattern	P&R Failures	Total Delay	Total Area
1	84.6%	3.53 ns	20841
2	7.79%	4.06 ns	20966
3	0%	4.09 ns	21029
4	0%	4.20 ns	21216

Table 5.4: L2 switch pattern results ($B = 8K, N = 8, M = Q = 4, W_{\text{eff}} = \{1, 2, 4, 8\}$).

Pattern	P&R Failures	Total Delay	Total Area
1	68.7%	4.13 ns	137153
2	6.84%	5.29 ns	163039
3	0.06%	5.50 ns	189050
4	0%	5.81 ns	189924

Table 5.5: L2 switch pattern results ($B = 64K, N = 16, M = Q = 8, W_{\text{eff}} = \{1, 2, 4, 8, 16\}$).

5.5 External Pin Assignment Flexibility

The results in the previous section assumed that the place/route algorithms are free to assign any I/O signal to any external pin. In practice, there are many applications in which this is unacceptable. For example,

- In large systems where the configurable memory is one of many chips mounted on a PC-board, once the board has been constructed, the pin assignments of all chips are fixed. If future system changes involve reconfiguring the FCM, the new pin assignments must be the same as the old assignments.
- In reconfigurable systems which contain FPGAs and FCMs, although each chip is configurable, the interconnections between these chips are often fixed [66, 67]. If there is little flexibility in the FCM pin assignments, severe restrictions will be placed upon the system level partitioner, placer, and router.

In order to evaluate the viability of FCMs in these applications, a measure of pin assignment flexibility is required. This section defines a metric for this flexibility, and evaluates this flexibility for the FCM architecture described in the previous chapter.

In any memory, there is complete functional flexibility in the pin assignments within a single address or data bus; i.e. all address pins are equivalent, as are all data pins. In this section, we are concerned with a higher level of flexibility: flexibility in the assignment of buses. As an example, consider Figure 5.15. In this example, a single logical memory that requires four data buses is implemented. There is no flexibility at all in the address bus assignment; only address bus 4 has enough pins to connect to all four arrays. There is complete flexibility in the data bus assignments, however, since each bus need only connect to one array. If two data buses are swapped, the arrays implementing the corresponding data fields can also be swapped, guaranteeing a successful implementation.

Consider a logical configuration requiring s data buses and z address buses mapped onto an architecture with M data buses and Q address buses ($M \geq s$ and $Q \geq z$). If the address mapping block contains switches at every horizontal/vertical bus interconnection,

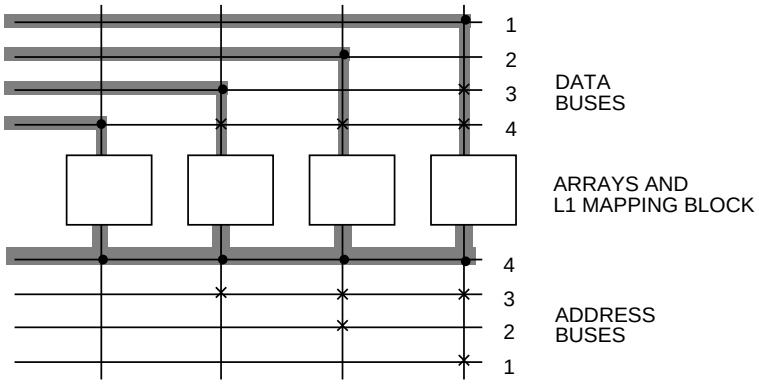


Figure 5.15: A single logical memory that uses 4 data buses and 1 address bus.

the number of possible address bus assignments is

$$A_{a,max} = \begin{pmatrix} Q \\ s \end{pmatrix} s! = \frac{Q!}{(Q-s)!}$$

Similarly, if the L2 data mapping block contains switches at every intersection, the number of possible data bus assignments is

$$A_{d,max} = \begin{pmatrix} M \\ z \end{pmatrix} z! = \frac{M!}{(M-z)!}$$

Combining these two results gives the number of possible bus assignments given complete mapping block flexibility:

$$A_{max} = A_{a,max} * A_{d,max} = \frac{Q!M!}{(Q-s)!(M-z)!}$$

For architectures with mapping blocks that do not contain switches at every intersection (such as the FiRM architecture), not all A_{max} bus assignments will lead to a valid implementation. Averaging the ratio of the number of valid bus assignments to A_{max} over many logical memory configurations gives the *bus assignment flexibility* of a particular architecture. The maximum value of an architecture's bus assignment flexibility is 1.

In order to evaluate the bus assignment flexibility of our architectures, we used an exhaustive algorithm that iteratively steps through all possible address bus assignments and determines whether a valid mapping is possible. Because of the heavy computational re-

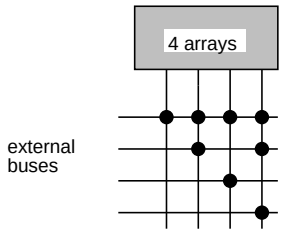


Figure 5.16: FiRM switch pattern for address and L2 data mapping blocks.

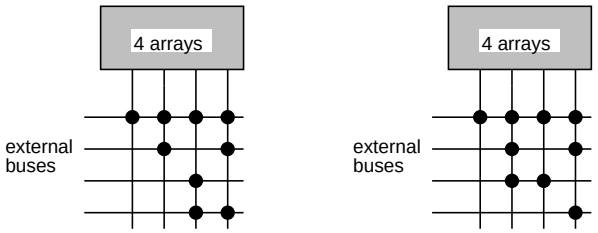


Figure 5.17: Best switch patterns if one switch-set is added.

quirements of this algorithm, we were limited to studying an architecture with 4 arrays, 4 data buses, and 4 address buses (the FiRM architecture). Nonetheless, this simple architecture provides insight into the behaviour that we expect out of larger configurable memories.

Using the above procedure, we measured the bus assignment flexibility of the FiRM architecture with the FiRM switch pattern to be 0.0829. In other words, on average, only 8.3% of all possible bus assignments result in valid implementations. For many applications, this is unacceptable.

In order to increase FiRM's bus assignment flexibility, we can add switches to the L2 data mapping block and address mapping block. The mapping block employed in FiRM contains 8 intersections with switches and 8 intersections without switches (see Figure 5.16). Consider adding switches (a switch-set, since each wire in the bus needs its own switch) to a single intersection. Since there are 8 intersections without switches, there are 8 locations where we can add this switch-set. We experimentally tried each, and the mapping blocks that gave the best results are in Figure 5.17 (both blocks shown are equivalent). If either of these mapping blocks are used in the FiRM architecture, its address bus flexibility approximately doubles to 0.157. The area and speed overhead in adding one extra set of switches is small, so clearly, for many applications, either of the mapping blocks in Figure 5.17 results in a better FCM architecture.

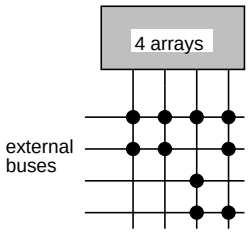


Figure 5.18: Best switch pattern if two switch-sets are added.

We can go further and add switches to two intersections. There are 28 ways these two switch-sets can be added; the best combination is shown in Figure 5.18. This mapping block gives a bus assignment flexibility of 0.217.

This process was repeated for 3,4,5,6,7, and 8 switch sets. The graph in Figure 5.19 gives the bus assignment flexibility for each number of switch sets. For each point, all possible switch patterns were considered, and the best chosen. As the graph shows, the curve is roughly linear, meaning there is no point beyond which adding switches gives little benefit. Thus, we conclude that in order to get a bus assignment flexibility of close to 1, the mapping block should be fully populated.

Note that there are two points on the graph that stand out: 3 and 6 switch sets. Figure 5.20 shows the mapping block patterns corresponding to these points. Figure 5.20(a) is the first pattern which has two horizontal buses that can connect to all 4 arrays; Figure 5.20(b) is the first pattern containing three horizontal buses that can connect to all 4 arrays. Logical memory configurations in which all arrays must be connected to each other

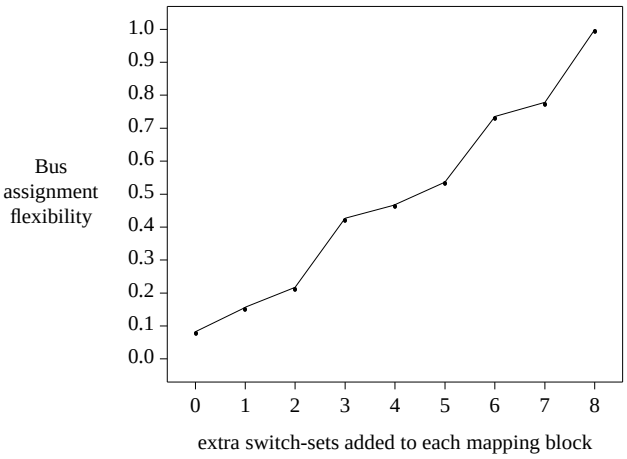
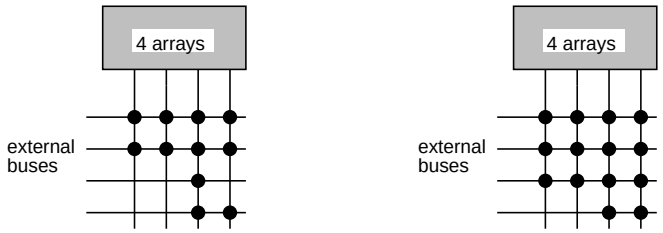


Figure 5.19: Bus assignment flexibility results.



a) Three switch-sets added b) Six switch-sets added

Figure 5.20: Best switch patterns if 3 and 6 switch-sets are added.

are common; thus, the more buses on which this can be done, the higher the bus assignment flexibility.

5.6 Conclusions

In this chapter, we studied the standalone memory architecture described in Chapter 4, examining how the architectural parameters described in that chapter affect the chip area, access time, and flexibility of the device. Specifically, we have shown that:

- As the array granularity is increased, the flexibility of the architecture increases (more configurations can be implemented), but that this is accompanied by a corresponding increase in access time and a significant increase in area. We considered two architectures: one with four address and data buses and 8 Kbits of memory, and one with eight address and data buses and 64 Kbits of memory. For the first architecture, the number of mapping failures decreased by 28% as the number of arrays was varied from 4 to 64, while for the second architecture, the number of mapping failures decreased by 22% as the number of arrays was varied from 8 to 64. Over the same two ranges, the access times of the two architectures increased by 38% and 24% respectively, and the area increased by 400% and 142% respectively. Overall, we found that, in our architectures, setting the number of arrays to be 2 or 4 times the number of address buses was a good choice.

- The more flexible the L1 mapping block, the more flexible the overall architecture will be. We considered the same two architectures as in the array granularity experiments, with the number of arrays fixed at 8 and 16 respectively. We varied the capability of the L1 mapping block from fully flexible (any power-of-two width between 1 and

the data bus width can be chosen) to not flexible at all (the effective data width is fixed). The fully flexible L1 mapping block resulted in 33% and 24% fewer mapping failures than the non-flexible L1 mapping block for the two architectures. There was found to be very little speed and area penalty as the flexibility of the L1 mapping block is increased. For both architectures, the access time of a memory employing the non-flexible L1 mapping block is only 10% less than that for a memory employing the fully-flexible mapping block. The area requirements of a memory with the non-flexible mapping block are only 6% and 7% less than if the fully-flexible block was used for the two architectures considered. Overall, we concluded that a fully-flexible L1 mapping block is best, especially for small architectures.

- An architecture with more, narrower data buses is a win in terms of speed, area, *and* flexibility. We considered an architecture with 64 Kbits of memory and 16 arrays, and varied the number of data buses was varied from 2 through 16, while the total number of data pins was held constant at 128. Over this range, the access time dropped by 16%, the area by 34% and the number of mapping failures by 29%.

- The L2 data mapping block and address mapping block switch patterns are critical to the overall flexibility of the architecture. Although a FiRM-like pattern provides near-perfect flexibility when the algorithms are free to assign buses, it is not nearly flexible enough when pins must be pre-assigned. In these applications, a mapping block in which every bus can connect to every array is the most appropriate.

The architectures in this chapter have been stand-alone; that is, the address and data pins are assumed to connect directly to I/O pads. As stated in the introduction to Chapter 4, one of the key applications of a configurable memory architecture is that it can be embedded onto an FPGA to create on-chip memory. This is the focus of the remainder of this dissertation.

Chapter 6

Embedded Arrays: Architecture and Algorithms

The architecture studied in Chapters 4 and 5 is a stand-alone memory intended for use in reconfigurable systems containing both FPGAs and memory chips. As a result of the ongoing improvements in integrated circuit technology and FPGA architectures, many applications that once required these multi-chip systems are becoming suitable for implementation on a single FPGA. Since memory is an important part of these applications, on-chip memory support in an FPGA is critical. As discussed in Chapter 1, on-chip memory will help relax I/O constraints, will improve the speed of many circuit implementations, and will reduce the number of chips required to implement large circuits, leading to lower system costs.

In this chapter, we present the architecture of an FPGA with on-chip memory. The memory resources consist of large blocks of RAM, similar to those discussed in Chapter 4. The logic resources consist of five input lookup tables connected using horizontal and vertical channels. The memory and logic are interconnected through memory/logic interconnect blocks. Section 6.1 describes the architecture in more detail.

This chapter also describes custom place and route tools (called SPLACE and SROUTE) that map circuits to the FPGA. Placement and routing in an FPGA with both memory and logic blocks is more difficult than placement and routing for FPGAs without memory arrays since both types of resource must be considered simultaneously. In the placement phase, the placements of the memory blocks must influence the placement of the logic blocks, and

vice versa. In the routing phase, the tool must be aware of differences in flexibility between the connections to the memory blocks and connections to the logic blocks. Section 6.2 will describe our solutions to these problems.

6.1 Basic Architecture

In the following subsections, we will describe the memory resources, logic resources, and the memory/logic interconnect. Much of this discussion also appears in [82].

6.1.1 Memory Resources

As described in Chapter 1, we concentrate on heterogeneous architectures with large blocks of RAM. The memory resources consist of a set of identical arrays, similar to those employed in the stand-alone architecture of Chapter 4. As in the stand-alone architecture, the number of bits in each array is fixed, but the aspect ratio of each array can be configured by the user. Unlike the stand-alone architecture, here we assume that each array has a separate data-in and data-out port (as well as an address port).

As shown in Figure 6.1, the memory arrays are assumed to be positioned in a single row across the middle of the chip. This is similar to the Altera FLEX 10K architecture. As will be shown, connections between two or more memory arrays are common; these connections are shorter if the memory arrays are positioned close together on the chip, resulting in a potentially faster and more routable device. Another approach would be to "spread" the memory arrays around the chip as evenly as possible; Figure 6.2 shows one possible floorplan for this sort of architecture. This would tend to shorten the memory-to-logic connections but lengthen the memory-to-memory connections. This sort of architecture is not considered in this dissertation; a full exploration of these tradeoffs is left as future work.

Table 6.1 shows the parameters that characterize the embedded memory resources. The parameters B, N, and W_{eff} are the same as those in the stand-alone architecture. In Chapter 7, we vary the number of arrays, N, from 2 to 16, while fixing the array size (B/N) at 2048 bits and the set of allowable data widths, W_{eff}, at $\{1, 2, 4, 8\}$. The parameters W, F_c, and F_s were defined in [37, 83], and F_m, M, V, and R will be described in Section 6.1.3.

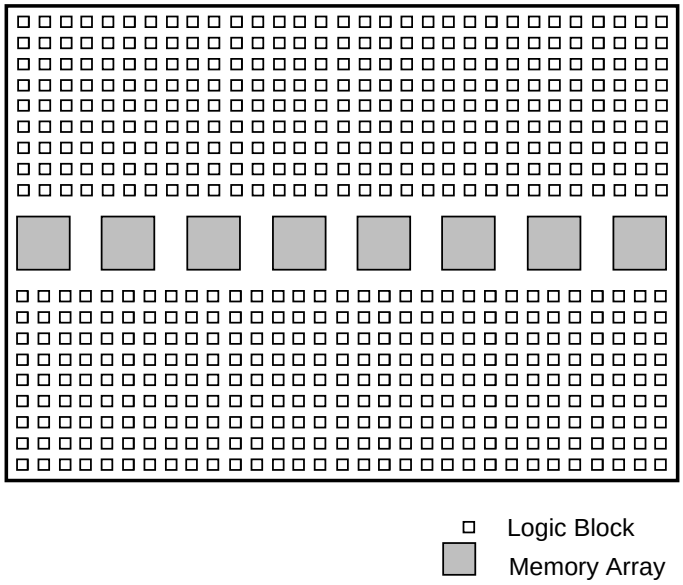


Figure 6.1: Floorplan showing memory and logic blocks.

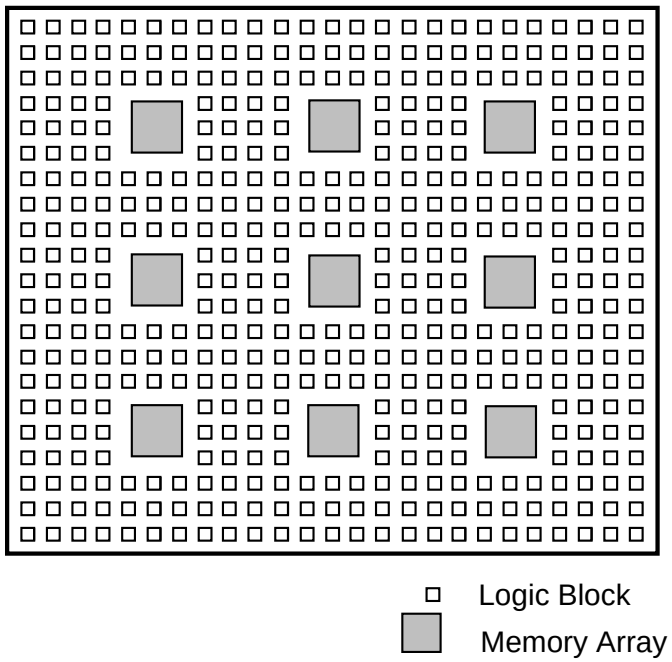


Figure 6.2: An alternative floorplan with arrays distributed throughout the FPGA (not considered in this dissertation).

Parameter	Meaning
B	Total bits
N	Number of arrays
W_{eff}	Set of allowable effective data widths of each array
M	Number of pins in each memory block
G	Number of logic blocks
W	Parallel tracks per logic routing channel
F_m	Number of tracks to which each memory block pin can connect
F_c	Number of tracks to which each logic block pin can connect
F_s	Number of choices offered to each incident track by a switch block
V	Vertical tracks between each memory array
R	Logic blocks per memory block in the horizontal dimension

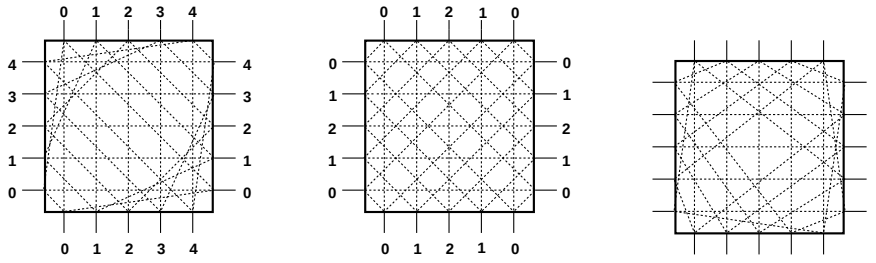
Table 6.1: Architectural parameters for embedded memory resources.

6.1.2 Logic Resources

The logic resources of the FPGA are assumed to consist of five-input lookup tables interconnected using horizontal and vertical channels, similar to the Xilinx 4000 FPGA, the Lucent Technologies ORCA FPGA, and the architecture studied in [37].

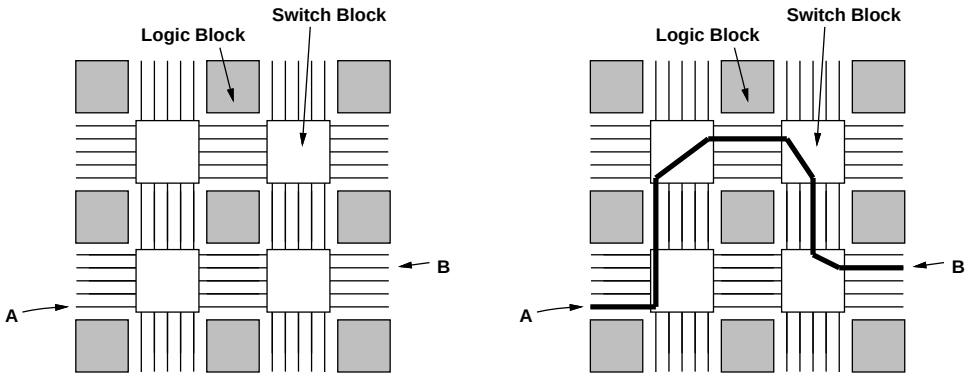
Of particular interest is the switch block that is found at the intersection of each horizontal and vertical channel. Figures 6.3(a) and 6.3(b) show two switch blocks used in previous work; in both, each incoming wire is offered three possible connections (denoted by $F_s = 3$ in [37]). Each dotted line represents one of these connections.

Neither of these switch blocks works well with the limited memory/logic interconnect that will be described in the next subsection. As will be shown, an inflexible memory/logic interconnect structure will often make it necessary to route a net to a specific track within a channel. Figure 6.4 illustrates the problem. Consider the implementation of a portion of



a) from [12, 84] b) from [85, 86] c) new switch block

Figure 6.3: Three different switch blocks.



a) Impossible using switch block of fig. 6.3(a) or 6.3(b)

b) Possible using switch block of fig. 6.3(c)

Figure 6.4: Routing a net from A to B.

a net that must connect track A to track B. If either of the switch blocks in Figures 6.3(a) or 6.3(b) is used, the connection is impossible. Each track in Figure 6.3(a) and (b) is labeled; both of these switch blocks are such that a track labeled i can only be connected to another track labeled i. Since track A would have a different label than track B, the connection is impossible, no matter how many switch blocks are traversed along the route. In [87, 88], such a routing structure is termed *disjoint*.

To alleviate this problem, we have used the switch block shown in Figure 6.3(c). This is similar to the switch block in Figure 6.3(b), except the diagonal connections have been "rotated" by one track. A precise description of the switch block can be written by representing it by a graph $M(T, S)$ where each node in T represents a terminal (incident track) of the switch block and each edge in S represents a programmable switch that connects two terminals. T is partitioned into four subsets, each with W terminals (each subset represents the tracks incident to one side of the switch block). Each terminal in T is labeled $t_{m,n}$ where m is the subset number ($0 \leq m \leq 3$) and n is the terminal number within the subset ($0 \leq n \leq W-1$). Figure 6.5 shows the correspondence between terminal label and physical switch block pin. The set of edges, S, for the non-disjoint switch block is then:

$$S = \bigcup_{i=0}^{W-1} \{ \quad (t_{0,i}, t_{2,i}) \ , \ (t_{1,i}, t_{3,i}) \ , \ (t_{0,i}, t_{1,(W-i) \bmod W}), \quad (t_{1,i}, t_{2,(i+1) \bmod W}) \ , \ (t_{2,i}, t_{3,(2W-2-i) \bmod W}) \ , \ (t_{3,i}, t_{0,(i+1) \bmod W}) \} \tag{6.1}$$

In Section 6.2.3, this switch block will be compared to the others in Figure 6.3.

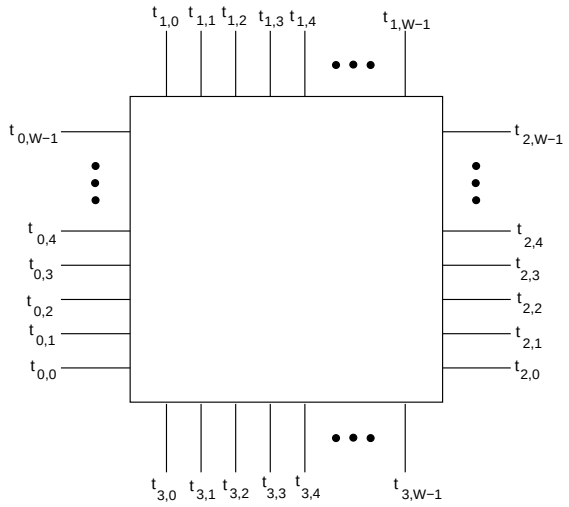


Figure 6.5: Correspondence between terminal label and physical switch block pin.

6.1.3 Memory/Logic Interconnect

Figure 6.6 shows the assumed interconnect structure between the logic and memory portions of the FPGA. Each memory pin is connected to one or more vertical routing tracks using a memory/logic interconnect block.

An example memory/logic interconnect block is shown in Figure 6.7. The vertical tracks connect to the logic routing resources above and below the memory row as shown in Figure 6.6. The memory pins can be programmably connected to these vertical tracks. Each dot in the figure represents a programmable connection. The interconnect block is characterized by the switch pattern, the number of memory pins, M, and the number of vertical tracks, V that run between each memory array. To calculate V, let N be the number of memory blocks and R the ratio of the number of logic blocks in the horizontal dimension to the number of memory blocks. There are then $(RN + 1)$ vertical channels connecting the top and bottom halves of the chip, each containing W parallel tracks. Then,

$$V = \left\lfloor \frac{(RN + 1)W}{N + 1} \right\rfloor$$

We define the flexibility of the memory/logic interconnect, F_m, as the number of vertical tracks to which each memory pin can connect. The example of Figure 6.7 shows $F_m = 4$. The maximum value of F_m is V (a switch at every pin/track intersection) and the minimum value is 1. As will be shown in Chapter 7, the value of F_m has a significant effect on the

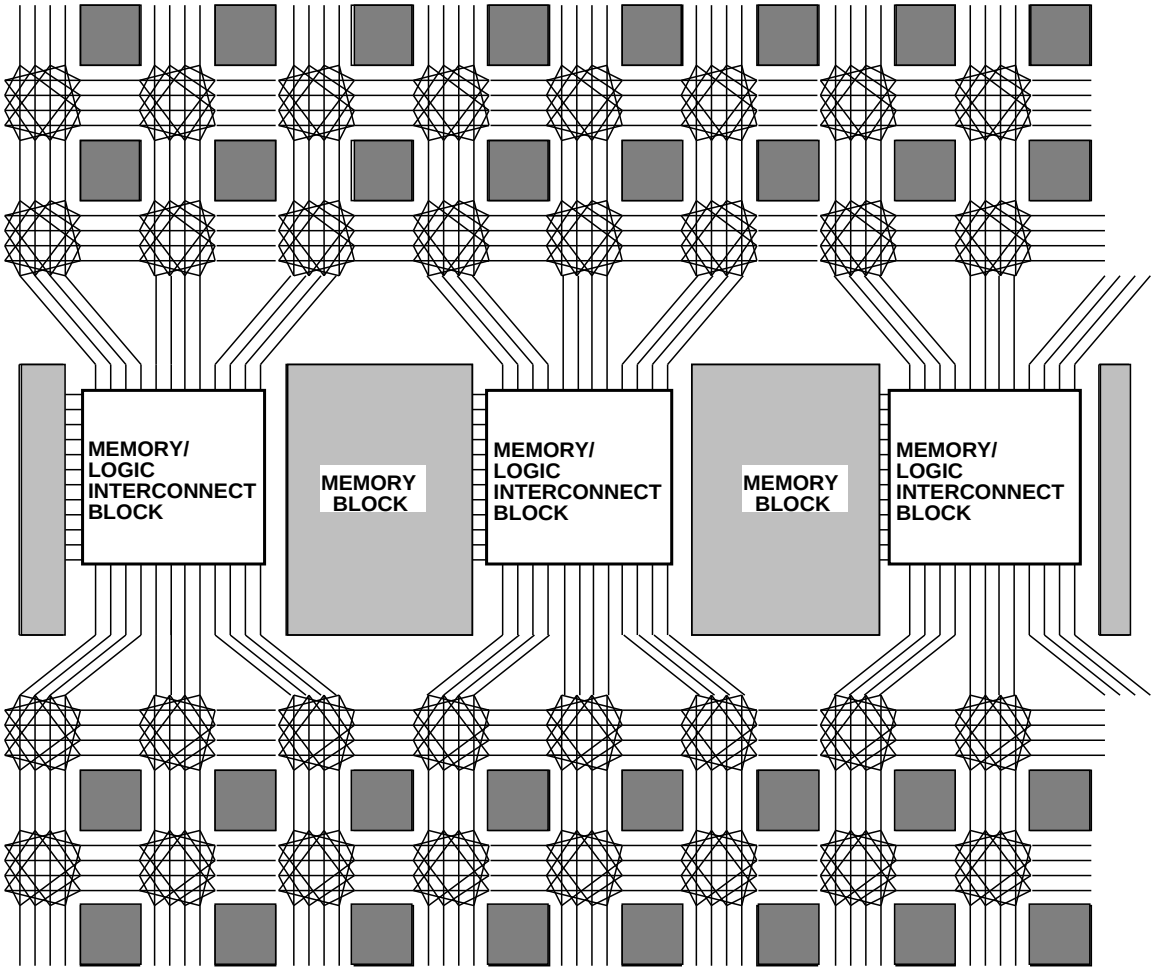


Figure 6.6: Memory/logic interconnect structure.

routability of the architecture.

The switch pattern shown in Figure 6.7 is just one of many possible switch patterns for $F_m = 4$. The choice of switch pattern in each memory/logic interconnect block is important since a poor pattern can cause unnecessary competition for one or more of the vertical tracks. The pattern we use is as follows: the set of F_m tracks to which pin j can connect are:

$$T_j = \left\{ \left(\left\lfloor \frac{jV}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor i \right) \bmod V \quad : \quad 0 \leq i < F_m \right\} \tag{6.2}$$

(the tracks are numbered starting at the left-most track and the memory pins are numbered from the top).

The motivation for choosing this formula is that it tends to "spread out" tracks to which a pin is connected as much as possible. Although we have not thoroughly explored all possible switch patterns, the patterns obtained from the above formula were found to

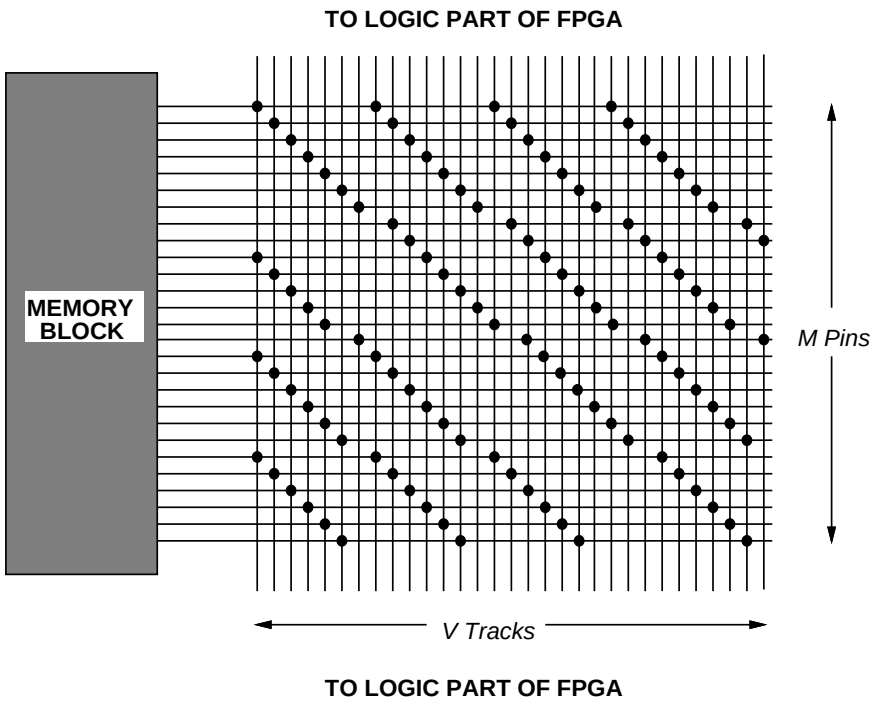


Figure 6.7: Memory/logic interconnect block.

work well (although in Section 7.1.2 we will show an anomaly where this formula produces a pattern that performs poorly). Finding an *optimum* pattern is left as future work.

6.2 Implementation Tools

As mentioned in Section 6.1, programs that map circuits to heterogeneous FPGA architectures are likely to be more complex than those for homogeneous FPGAs. Without effective CAD software, however, a heterogeneous architecture is of little practical use. In this section, we describe how the mapping algorithms of Chapter 4 can be combined with existing logic optimization and technology mapping tools and new place and route programs to produce a CAD flow that is simple, yet results in efficient circuit implementations.

Our main goals in creating this CAD flow were:

1. To create a tool suite that was flexible enough to map to a wide variety of architectures. In the next chapter, different memory/logic interconnect structures are considered; a single tool suite that can map to all of these architectures is essential.

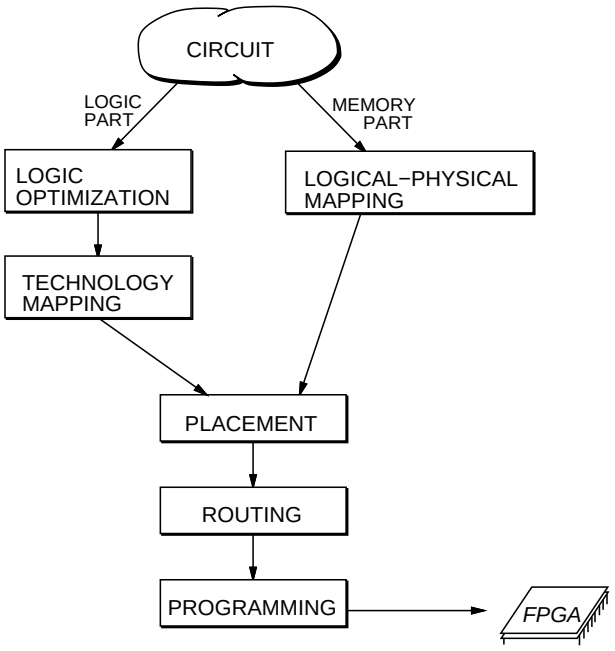


Figure 6.8: CAD flow.

2. To minimize the development time of the CAD tools. Since the primary focus of this dissertation is to consider configurable memory architectures, we did not wish to spend an inordinate amount of time optimizing the CAD tools.

Two secondary goals were:

1. To provide results that were not significantly worse than results reported elsewhere.

2. To provide tools with run times that were short enough to allow many repetitive experiments to be run using a limited amount of CPU time.

The CAD flow is shown in Figure 6.8. Until the placement phase, the logic and memory portions of the circuit are processed independently, since optimization and technology-mapping is very different than the corresponding routines for memory. The optimization and technology-mapping stages for logic have been well studied; we use SIS [61] for logic optimization and FlowMap [62] for logic technology-mapping. The logical-to-physical mapping problem for memory was presented in Chapter 4; we use the heuristic algorithm from Section 4.2.2.

Since we wish to consider both memory and logic during the placement and routing phases, existing tools that support only logic circuits will not suffice. Instead, we have

created custom tools; these tools are described in the following two subsections. In developing these tools, we have followed the traditional approach of separating the placement and routing tasks and using separate algorithms to solve each. Although combined placement/routing programs exist [73, 74, 75], and would likely provide superior circuit implementations, such a tool would be more complex to develop. Since the development time of our tools was of the upmost importance, we decided to employ separate algorithms.

The final phase of the CAD flow, programming, is highly dependent on the technology used in the FPGA. We do not consider this in this dissertation.

6.2.1 Placement Program

During the placement phase, circuit elements (logic blocks, memory blocks, and I/O blocks) are assigned physical locations on the target FPGA. Although there have been several published placement tools for FPGAs [75, 73, 74, 89, 90], none of them support an FPGA architecture with both logic and memory. The placement algorithm described in Chapter 4 works well (almost optimally) for the standalone architecture described in that chapter, but is not likely suitable for the architecture in this chapter, since the interconnect between the memory arrays is much more flexible than that of the standalone architecture. In the architecture of this chapter, any array can be connected to any other array and to any logic block through the chip's routing resources. The costs of each of these connections, however, is difficult to estimate, since the tracks used to connect memory arrays are also needed to connect logic blocks to each other. In addition, the placement of logic blocks should influence the placement of the memory blocks, and vice-versa. These complications motivate us to develop a new placement tool.

Our placement program, called SPLACE, is based on the popular simulated annealing optimization algorithm [91]. The algorithm has been used in many standard cell placement programs [92, 93, 94, 95, 96, 97], and later in placement programs specific to FPGAs [74, 90, 98]. The appeal of the simulated annealing algorithm is that it can be used to place both logic and memory blocks simultaneously (as well as I/O blocks) and provides a natural mechanism for the placement of logic blocks to influence the placement of memory blocks, and vice versa.

The simulated annealing algorithm as applied to placement starts with a random place-

```
P = Random Placement
T = Initial Temperature
While (exitCondition(T) == False) {
    Repeat for a fixed number of iterations {
        P_new = generateMove(P)
        ΔC = cost(P_new) - cost(P)
        if ( random(0,1) < exp(-ΔC/T) ) {
            P = P_new
        }
    }
    T = updateTemperature(T)
}
```

Figure 6.9: Pseudo-code for simulated annealing algorithm.

ment, and iteratively swaps random elements until some stopping criteria is achieved. The key to the annealing algorithm is that each swap may or may not be accepted. This section describes the algorithm, concentrating on our selection of annealing schedule, cost function, swap acceptance probability, and the stopping criteria.

Figure 6.9 shows psuedo-code for the algorithm. Within each iteration, a move is chosen (legal moves will be discussed below), and the change in cost ΔC is calculated. The cost of a placement is the total sum of the half-perimeters (sum of the X and Y dimensions) of the bounding box for each net. The motivation for choosing this cost function is that we wish to minimize the routing area in the final circuit implementation; hence, we seek to minimize the total wire-length over all nets. Since an easily computable cost function is essential for simulated annealing, we approximate the wire-length of a net by the half-perimeter of its bounding box. Nets that connect to memory are treated in the same manner as those that connect to logic. If the change in cost is negative, meaning the move improves the placement, the move is performed. If the change is positive, the move is accepted with a probability of $\exp(\frac{-\Delta C}{T})$. This possible acceptance of seemingly bad moves allows the algorithm to escape from local minima in order to find a better placement.

As shown, the acceptance rate depends on the system "temperature", T. T is initialized to a large enough value that most bad moves are accepted. During each call to *updateTemperature*, T is updated as follows:

$$T_{new} = \text{MAX}\left(T \exp\left(\frac{-\lambda T}{\sigma}\right) \ , \ 0.75T\right)$$

where σ is the standard deviation of the costs obtained during the current temperature step (iteration of the outer loop) and λ is a constant. This formula was used in [94]. We used $\lambda = 0.7$ (this value was also used in [94]).

For each value of T, the inner loop executes

$$\text{MIN}(5(\text{elements in circuit})^{4/3}\ ,\ 20000)$$

times, where *elements in circuit* is the total number of memory arrays, logic blocks, and I/O pads in the netlist. This schedule, which is a scaled version of the formula used in [97], was found to produce good results in a reasonable amount of time.

A legal move is the swapping of two blocks or the movement of a block to an unoccupied location. The blocks are selected at random; they can be either memory, logic, or I/O blocks, but both selected blocks must be of the same type (if a movement of a block to an unoccupied location is chosen, the destination site must be of the appropriate type). The selection is subject to the constraint that the current physical distances between the two blocks is less than R_x in the x-direction and R_y in the y-direction. Initially R_x and R_y are set to the size of the device, but after each temperature update, they are updated as follows:

```
if the acceptance rate for the last temperature is less than a constant α {
    decrease R_x and R_y by 20%
} else {
    increase R_x and R_y by 20%
}
```

In no case do we let R_x or R_y become smaller than the length of 3 logic blocks, nor do we let them become larger than the size of the device. This dynamic sizing of range windows ensures that the acceptance rate stays as close to some desired rate α as possible, leading to a reduction in CPU time [95, 97]. In [95, 97], $\alpha = 0.44$ was used; we use the same value here.

Finally, the algorithm is terminated when either T or the standard deviation of the costs obtained during the previous temperature step drops below a preset value. A final greedy stage is performed, in which moves are accepted only if they improve the cost function (equivalent to $T = 0$).

The placement algorithm was implemented. Rather than presenting a quantitative comparison with existing placement tools, Section 6.2.3 will combine the placer and router, and compare the composite results with existing placer/router combinations.

6.2.2 Routing Program

Once the placement program has assigned physical locations for all circuit elements, the next step is to use the prefabricated routing resources to implement all required connections between these circuit elements. Once again, many FPGA routing tools have been developed, but none are general enough to map to our heterogeneous architecture. This section describes a custom FPGA router that is general, efficient, and provides good results.

In many FPGA CAD systems, the routing task is solved in two stages: global routing [99, 100, 101, 98], in which nets are assigned to channels, and detailed routing [102, 84], in which individual routing segments within the assigned channels are chosen. Because our router must be flexible enough to work with very low-flexibility connections (in the next chapter, architectures with $F_m = 1$ will be considered), the split global/detailed scheme will not suffice. Consider a global router searching for a route for a net starting in a memory block in the architecture of Figure 6.6. Each memory/logic interconnect block can connect to six logic switch blocks (three above the memories and three below). Without knowledge of the internals of the memory/logic interconnect block (the detailed routing architecture), it is impossible to determine which of these switch blocks can be connected to the starting pin. Single-stage routers that combine the global and detailed routing phases have also been proposed [88, 103, 104, 105, 106]. Since single-stage routers take into account the detailed routing architecture at all phases of the routing, it is more suitable for architectures containing very low flexibility connections.

Our router, called SROUTE, employs a multi-pass maze router, with enhancements that significantly speed up the routing and lower the algorithm's memory requirements. The next subsection describes the basic algorithm, while the subsequent subsection describes the enhancements.

Basic Maze Routing Algorithm

The maze router is based on the maze routing algorithms by Lee and Moore [107, 108], both of which use the shortest-path algorithm of Dijkstra [109]. Each net is processed sequentially. For each net, the shortest path connecting the source to all sinks that does not conflict with previously routed nets is found; the algorithm that routes one net is shown in Figure 6.10. The algorithm operates on a graph in which each node represents a track or pin in the physical architecture, and each edge represents a programmable connection between nodes (this is the same graph structure used in [106]). A breadth-first algorithm is used to traverse the nodes; thus, at each stage, the closest unvisited node to the source is expanded. This ensures that each node need only be visited once, and that the shortest path for this net is found. In the next subsection, we will describe how this strategy can be modified to speed up the search.

```
heap = null
T = null /* T will contain route of net */
mark all nodes as 'unused'
put source of net in heap with label=0
while there are still pins to connect for this net{
    if heap is empty {
        routing for this net failed, exit algorithm
    }
    from heap, remove node with lowest label, l
    if this node is a goal pin (a sink of the net) {
        backtrace through graph, adding nodes to T, and marking resources as 'used'
        heap = null
        place all nodes in T into heap with label=0
    } else {
        if we have not yet visited this node {
            for each neighbour {
                if this neighbour is not marked 'used' {
                    place neighbour in heap with label = l + 1
                }
            }
        }
    }
}
```

Figure 6.10: Pseudo-code for algorithm to route a single net.

The success of this algorithm will depend on the ordering of the nets. Rather than attempting to determine a priori which nets should be routed first, we perform the routing using several passes, re-ordering the nets between each pass. The nets that fail in one pass

```
N = list of nets to route
sort N by number of memory connections
N' = N
passes = 0
done = 0
repeat until (passes = 10 or done = 1) {
    for each net i in N {
        route net according to algorithm of Figure 6.10
        if routing for this net fails {
            move element i to head of N'
        }
    }
    if all nets were routed successfully {
        done = 1
    }
    N = N'
    add 1 to passes
}
```

Figure 6.11: Pseudo-code for algorithm to route all nets.

are moved to the head of the list, and routed first in the next pass, as shown in Figure 6.11. This "move-to-front" heuristic was also used in [105].

The initial order of nets (for the first pass) was found to have little influence on the quality of the final solution, but it was found to have an effect on the number of passes required to successfully route circuits. In general, the nets with the most connections to memory were harder to route (see Chapter 7); thus, before the first pass, we sort nets by the number of memory pins on each net, and route those with more memory pins first. During subsequent passes, the nets are re-ordered as described above. When this strategy was employed, we found 10 iterations sufficient to find a successful routing in most circuits (if one exists).

Directional Algorithm

For large architectures, the search space for the maze router can be very large. Consider Figure 6.12. Each solid track in the diagram represents a node that must be visited during the search when routing from the source to the destination. Because of the breadth-first nature of the algorithm, the set of visited nodes expands in a "diamond" pattern around the source. As the nodes get farther apart, the number of nodes that must be visited (and hence the run-time and memory requirements of the algorithm) increases quadratically.

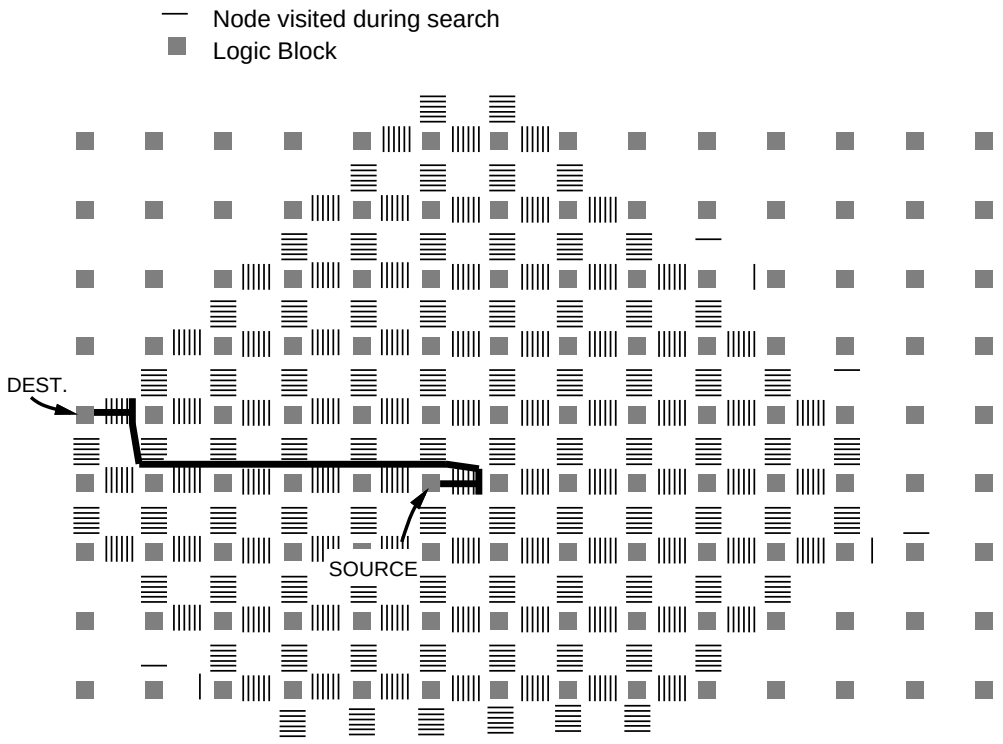


Figure 6.12: Nodes visited by the basic maze routing algorithm.

There have been many published techniques to reduce the run-time of the maze routing algorithm. These fall into two categories: algorithms that attempt to reduce the state stored for each node, and algorithms that attempt to reduce the number of nodes that must be visited during a search. We employ the latter approach. The remainder of this section describes our enhancements, which are similar to the depth-first algorithm described in [110].

The enhancements are motivated by the observation that for most connections, the chosen route has few "loops" or "detours" to avoid congestion. To take advantage of this, we precede the execution of the algorithm in Figure 6.10 with a first pass that attempts to route the net quickly without searching for loops or detours. Only if the first pass fails, is the full maze router of Figure 6.10 used.

This initial pass is similar to the algorithm of Figure 6.10 with two modifications. First, during the expansion of a node, neighbours that are geometrically farther away from the destination than the current node (in both the X and Y direction) are not placed into the heap. This has the effect of disallowing "backwards" turns. The reason for limiting

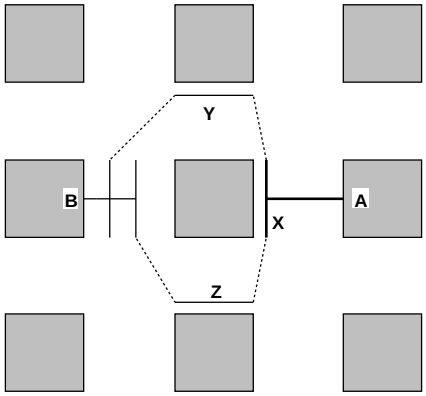


Figure 6.13: Example in which pruning should be limited to moves that are further in both the X and Y dimensions.

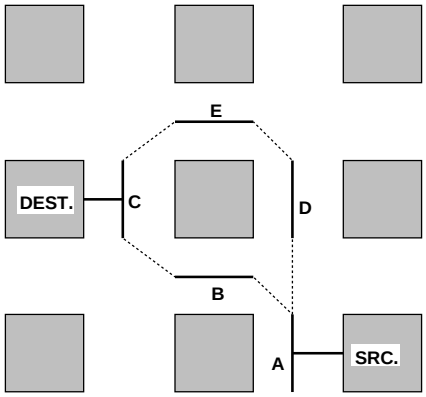


Figure 6.14: Example in which the cost of a node should reflect both the distance from the source and distance to the destination.

the pruning to moves that are further in *both* dimensions (as opposed to using Manhattan distance to determine which nodes should be pruned) can be seen in Figure 6.13 in which block A is to be connected to block B. Assume the router is currently expanding around track X. In terms of Manhattan distance, both tracks Y and Z are further away from the destination than X, but clearly Y or Z must be chosen to complete the route. These tracks are closer to the destination than X in terms of their X-dimensions; therefore, our algorithm would place Y and Z in the heap, leading to a valid solution.

Second, we modify the order in which nodes in the heap are visited. Rather than using a breadth-first search, we visit nodes according to their geometric location. At each step, we extract the node in the heap that is closest to the destination (Manhattan distance) and expand around it.

Figure 6.14 shows a potential problem with this approach. Here, tracks B, D, and E

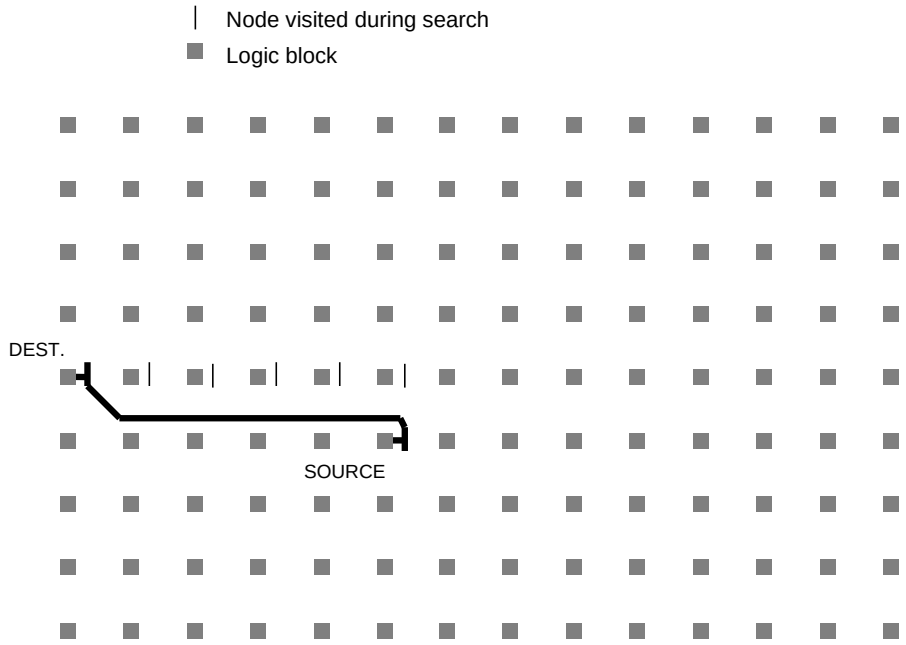


Figure 6.15: Nodes visited by the directional search algorithm.

are equally far from the destination (recall that we use the Manhattan distance). Using the algorithm described, path $\{A,D,E,C\}$ might be chosen, when clearly path $\{A,B,C\}$ is a better choice. To avoid this problem, we select a node from the heap based not only on the distance of the node to the destination, but also the path length between this node and the source.

This can be implemented by changing the "then" clause of the inner-most "if" statement in the algorithm of Figure 6.10 to:

$$\text{place neighbour in heap with label} = \text{distance to destination} + \alpha(l+1)$$

By experimentation, we found that $\alpha = 0.75$ works well.

Figure 6.15, shows the nodes visited during the routing of the net of Figure 6.12 using the enhanced algorithm. In this example, only five nodes that were not part of the final path were visited.

The above algorithm assumes that we know the location of the destination pin. For multi-sink nets, however, there are several possible destination pins. To route multi-sink nets, we choose the *closest* pin to the source, and use that as the "target". When the target is found, the remaining pins are examined to determine which is closest to *any single node*

of T (the partial route for this net). This closest pin is the new target, the single closest node of T is placed in the heap, and the process repeated. The modified algorithm is shown in Figures 6.16 and 6.17.

```
heap = null
T = null /* T will contain route of net */
mark all nodes as 'unused'
put source of net in heap with label=0
target = closest sink to source
while there are still pins to connect for this net{
    if heap is empty {
        routing for this net failed, exit algorithm
    }
    from heap, remove node with lowest label, l
    if this node is a goal pin (a sink of the net) {
        backtrace through graph, adding nodes to T, and marking resources as 'used'
        heap = null
        target = closest unconnected sink to any node in T
        place node in T that is closest to target into heap
    } else {
        if we have not yet visited this node {
            for each neighbour {
                if this neighbour is not further in both the X and Y dimensions {
                    if this neighbour is not marked 'used' {
                        place neighbour in heap with label =
                                distance between neighbour and target + α(l+1)
                    }
                }
            }
        }
    }
}
```

Figure 6.16: Modified pseudo-code for algorithm to route a single net.

As described above, if the directional algorithm fails, the search is repeated using a breadth-first search. This ensures that if a route exists, it will be found. However, often the route found by the directional algorithm will be sub-optimal, and this sub-optimal route may lead to failures when subsequent nets are routed. In the next section, we place and route several benchmark circuits, and quantify the effects of using the directional algorithm.

Pin Equivalence

Both the standard and directional maze routing algorithms make use of the fact that all input pins to a lookup table are equivalent. Thus, when routing towards a lookup table

```
N = list of nets to route
sort N by number of memory connections
N' = N
passes = 0
done = 0
repeat until (passes = 10 or done = 1) {
    for each net i in N {
        route net according to algorithm of Figure 6.16
        if routing for this net fails {
            route net according to algorithm of Figure 6.10
            if routing still fails {
                move element i to head of N'
            }
        }
    }
    if all nets were routed successfully {
        done = 1
    }
    N = N'
    add 1 to passes
}
```

Figure 6.17: Modified pseudo-code for algorithm to route all nets.

input, the first available input found is used. Similarly, all address pins of a memory block are equivalent. Data pins are also equivalent with the restriction that once a connection is made to a data-in pin, the corresponding data-out pin is no longer equivalent to the other data-out pins (and vice versa). For example, if the fifth data-in pin of a given memory block is used to implement a connection to bit 0 of a logical memory, the fifth data-out pin of that block must also correspond to bit 0. Our router takes advantage of this when possible.

6.2.3 Validation of Placement and Routing Tools

This section presents a quantitative comparison of the placer/router combination described in this chapter with existing placer/router combinations. Since all published results are for logic-only circuits, we first present comparisons for an FPGA with no memory arrays. In Subsection 6.2.3, we examine the algorithms' performance on two circuits containing memory.

Circuit	Altor/ LR/ Sega [111, 100, 84]	Altor/ GPB [88]	FPR [75]	Altor/ GC [103]	Altor/ IKMB [105]	Altor/ TRACER [106]	SPLACE/ SROUTE	Altor/ SROUTE
9symml	9	9	9	9	8	6	7	7
alu2	10	11	10	9	9	9	8	9
alu4	13	14	13	12	11	11	9	12
apex7	13	11	9	10	10	8	6	9
example2	17	13	13	12	11	10	7	11
k2fix	16	17	17	16	15	14	11	15
term1	9	10	8	9	8	7	5	8
too_large	11	12	11	11	10	9	8	11
vda	14	13	13	11	12	11	10	12
TOTAL	112	110	103	99	94	85	71	94

Table 6.2: Minimum number of tracks/channel for various CAD flows.

Logic Circuits

Table 6.2 compares place and route results from our CAD flow to those obtained from five other sources using 9 small logic benchmark circuits (54 to 358 logic blocks). Each entry in the table is the smallest number of tracks within each channel for which the circuit can be completely routed. In each case, we assumed an FPGA size and aspect ratio consistent with those assumed in the previous studies.

As Table 6.2 shows, the results obtained by SPLACE and SROUTE compare favorably with the previous published results. Overall, we require 16.5% fewer tracks than the next best set of results, and 36.6% fewer tracks than the Altor/LocusRoute/Sega combination [111, 100, 84]. Note that each of the referenced papers (except [75]) presents results for a routing algorithm using an existing placement. The Altor program, used to create the placements for each of the other studies, is a standard cell placement tool, and not optimized for FPGAs. To determine how much of an effect the new placement has on the final routing results, we re-routed the circuits using SROUTE with the same placement that was used in the other studies. As the final column in table 6.2 shows, SROUTE routed the circuits using 15% fewer tracks than the LocusRoute/Sega combination, and required more tracks that only one of the previous routing algorithms. Thus, both SPLACE and SROUTE perform well, and are suitable for the architectural experiments we present in the next chapter.

Circuit	SPLACE/SROUTE (directional)	SPLACE/SROUTE (maze-routing only)
9symml	7	7
alu2	8	8
alu4	9	9
apex7	6	6
example2	7	7
k2fix	11	11
term1	5	5
too_large	8	8
vda	10	9
TOTAL	71	70

Table 6.3: Comparison of directional and maze-routing algorithm results.

Table 6.3 shows the results with and without the directional enhancements described in Section 6.2.2. As the table shows, the directional algorithm performs nearly as well as the full maze routing algorithm. In only one circuit was an extra track required. Table 6.4 shows the difference in CPU time required by the two routing algorithms for the three largest circuits of Table 6.2. The CPU time required depends on the "difficulty" of the routing problem; this, in turn, depends on the number of tracks in each channel. For each benchmark in Table 6.4, two sets of results are presented: one for an architecture with 14 tracks per channel (an "easy" problem), and one for an architecture with the minimum number of tracks in each channel (a more difficult problem). In each case, the directional algorithm requires less CPU time than the maze router algorithm, but the difference is not as dramatic as might be expected. This is because these are small circuits; as shown in the next subsection, the difference is larger for large circuits.

The results in Table 6.2 assume that the switch block in Figure 6.3(a) is employed (this is the same across all CAD flows presented). As explained in Section 6.1.2, this switch block will not work well in an architecture with a low memory-logic flexibility. Table 6.5 shows the track count results for the other two switch blocks in Figure 6.3. As explained in Section 6.1.2, the non-disjoint switch block of Figure 6.3(c) is superior when making connections between low-flexibility memory pins; these results show that this switch block works well, even if no memory blocks are present.

Circuit and Tracks/Channel	Maze only	Directional
alu4:		
14 tracks/channel	4.22 sec	1.75 sec
minimum tracks/channel	10.6 sec	2.02 sec
k2fix:		
14 tracks/channel	8.88 sec	2.58 sec
minimum tracks/channel	7.35 sec	7.08 sec
vda:		
14 tracks/channel	4.00 sec	1.27 sec
minimum tracks/channel	5.87 sec	3.62 sec

Table 6.4: CPU time requirements[2] for router.

	Disjoint		Non-disjoint
Circuit	Fig.6.3(a)	Fig.6.3(b)	Fig.6.3(c)
9symml	7	7	6
alu2	8	9	8
alu4	9	10	9
apex7	6	6	6
example2	7	8	6
k2fix	11	12	10
term1	5	6	5
too_large	8	9	8
vda	10	11	9
TOTAL	71	78	67

Table 6.5: Track count using disjoint and non-disjoint switch blocks.

Circuits with Memory

In this section, we quantify the performance of the placer and router for circuits and architectures with both memory and logic. Since there are no comparable published studies, the results are presented for our tools only.

Table 6.6 shows the minimum number of tracks required in each channel for two circuits. The first circuit, obtained from Hewlett-Packard Labs, finds the "edit distance" between two strings (representing handwritten characters) [112]. The circuit, called Scribbler, contains 486 five-input lookup tables, 84 I/O pads, 8 memory blocks, and 573 nets. It was placed and routed on an architecture with 504 logic blocks and 8 memory arrays. The second circuit was

[2] All timing results in this chapter were obtained on a 110Mhz Sparc-5 with 32MB of memory.

generated by the stochastic circuit generator described in Chapter 3 and contains 1848 five-input lookup tables, I/O pads, 15 memory blocks, and 2156 nets. It was placed and routed on an architecture with 1872 logic blocks and 16 memory arrays. For both architectures, $F_m = 4$ was assumed (other values of F_m will be investigated in the next chapter).

As Table 6.6 shows, the directional algorithm finds results with only slightly more tracks in each channel than does the full maze routing algorithm. The difference in CPU time requirements is shown in Table 6.7; again, results are presented for both "easy" and "more difficult" routing problems. Clearly the difference in CPU time is more significant than it was for the smaller circuits, and justifies the use of the directional algorithm in architectural experiments.

Circuit	Directional	Maze only
Scribbler	12	11
Stochastic	17	15

Table 6.6: Minimum number of tracks/channel.

Circuit and Tracks/Channel	Maze only	Directional
Scribbler:		
17 tracks/channel	39.3 sec	3.10 sec
minimum tracks/channel	258.6 sec	33.1 sec
Stochastic:		
20 tracks/channel	420 sec	16.8 sec
minimum tracks/channel	1296 sec	16.9 sec

Table 6.7: CPU time requirements for router.

6.3 Summary

This chapter has described a heterogeneous FPGA architecture that can efficiently implement circuits containing both memory and logic. The memory resources consist of a set of identical arrays, similar to those discussed in Chapter 4, while the logic resources consist of a grid of five-input lookup tables connected using horizontal and vertical channels.

The memory and logic resources are connected through memory/logic interconnect blocks; within each block, each memory pin can be programmably connected to one (or more) of F_m tracks.

We have also presented a CAD tool suite that maps circuits to our heterogenous FPGA. The optimization and technology-mapping phases are performed separately for the logic and memory portions of the circuits while the custom placer and router processes the memory and logic portions simultaneously. We have shown that the placer and router perform well compared to several existing published CAD tools. Overall, we require 16.5% fewer tracks than the next best set of results, and 36.6% fewer tracks than the Altor/LocusRoute/Sega combination [111, 100, 84]. When using a fixed placement, our router finds solutions using fewer tracks than all but one of the algorithms that we compared our results to, and 15% fewer tracks than the LocusRoute/Sega combination.

The next chapter focuses on the memory/logic interconnect. In particular, we first determine how flexible the interconnect must be, and then examine enhancements to the interconnect presented here that improves the routability and speed of the FPGA.

Chapter 7

The Memory/Logic Interface

The previous chapter described an FPGA architecture containing both logic and memory resources. The logic blocks of this device are five-input lookup tables connected using horizontal and vertical channels, similar to those in the Xilinx XC4000 and Lucent Technologies ORCA FPGAs [12, 13]. The memory resources consist of a set of configurable arrays similar to those described in Chapter 4. The focus of this chapter is the interface between these memory and logic resources.

The design of a good memory/logic interface is critical. If the interface is not flexible enough, many circuits will be unroutable, while if it is too flexible, it will be slower and consume more chip area than is necessary. This chapter concentrates on two aspects of the memory/logic interface. First, Section 7.1 investigates how flexible the interface must be, and how flexibility affects the area and speed of the device. Second, Section 7.2 shows that the routability and speed of the FPGA can be improved by adding architectural support for nets that connect more than one memory array. The results in both of these sections were obtained experimentally using benchmark circuits from the generator described in Chapter 3. In Section 7.3, we compare the results to those obtained using a single "real" benchmark circuit.

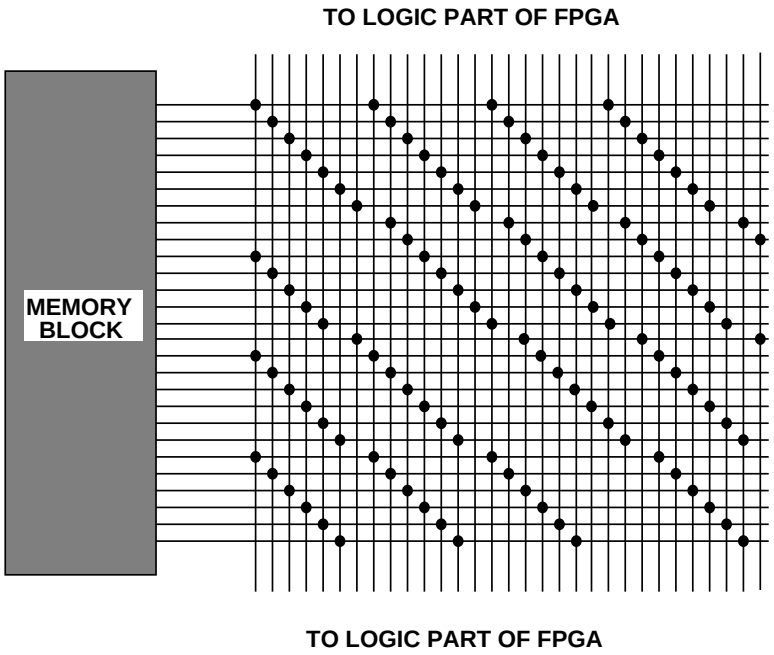


Figure 7.1: Memory/logic interconnect block example from Chapter 6.

7.1 Memory/Logic Interface Flexibility

As described in Section 6.1.3, the flexibility of the interconnect structure is quantified by a parameter F_m which indicates the number of tracks to which each memory pin can be connected. The example of that section is repeated in Figure 7.1. In this figure, each dot represents a programmable switch that can optionally connect a single horizontal pin to a single vertical track. Since each pin in Figure 7.1 has four of these switches, $F_m = 4$. The question we address in this section is: *what value (or range of values) of F_m results in the most area-efficient and speed-efficient FPGA?*

Note that we are concerned about the area and speed of the entire FPGA, not just the memory/logic interconnect region. Clearly, the lower F_m, the smaller the memory/logic interconnect block will be because fewer programming bits and pass transistors are needed. Decreasing F_m, however, places additional demands on the rest of the FPGA since it makes it more difficult to route nets between the logic and memory blocks. One way the designer of an FPGA can compensate for this reduction in routability is to add extra tracks to each routing channel across the entire FPGA. The area cost of these additional tracks must be considered when determining a good value of F_m.

Similarly, the lower F_m, the fewer switches there are on any path into and out of the memory blocks, and hence, the shorter the memory access times. However, the lower-flexibility memory/logic interconnect blocks may result in circuitous routes that connect logic and memory. The extra delay due to these circuitous routes must also be considered when determining a good value for F_m.

An early version of part of this section appeared in [82].

7.1.1 Methodology

We employ an experimental approach in which we "implement" many benchmark circuits on FPGAs with different values of F_m and measure the area and delay of the resulting implementations. The following subsections describe the benchmark circuits, the architectures explored in the experiments, the implementation tools employed, and the area and timing models used to evaluate the architectures.

Benchmark Circuits

As described in Chapter 3, the usual approach of using a handful of benchmark circuits is not suitable for the exploration of configurable memory architectures. Since most memory circuits contain only a few logical memories, we need hundreds of benchmark circuits to thoroughly exercise the memory/logic interconnect architecture. Rather than gather that many circuits, we use the same approach used in in Chapter 5; that is, we use stochastically generated circuits from the circuit generator described in Chapter 3. The use of the 171 circuits evaluated during the analysis of Chapter 3 is not an option since we could not obtain the netlists for these circuits. Unlike Chapter 5, here we use the entire generated circuits, not just the logical memory configurations.

It is important that the circuits used as benchmarks be realistic representatives of circuits that would actually be implemented on the device in question. Since the circuits generated by the generator described in Chapter 3 vary widely in terms of size, memory usage, and I/O pin requirements, it is necessary to constrain the generator to ensure that it generates circuits suitable for our study.

Architecture to which circuits are targeted	Number	Logic Blocks		Memory Blocks		I/O Blocks		Nets	
		Avg	St.Dev	Avg	St.Dev	Avg	St.Dev	Avg	St.Dev
2 arrays	100	535	279	2	0	123	75	651	307
4 arrays	100	657	373	3.63	0.81	134	85.9	788	415
8 arrays	100	913	467	7.72	0.57	146	85.5	1069	497
16 arrays	100	849	500	15.2	1.33	154	102	1030	500

Table 7.1: Circuit statistics.

The following restrictions are placed on the generated circuits:

1. The number of logic blocks in each circuit must be between 200 and 2500. The upper bound ensures that the circuits can be placed and routed using a reasonable amount of CPU time.

2. Each circuit must use no more than 400 I/O pins. This ensures that the generated circuits do not require an inordinately large FPGA.

3. The total number of memory bits in each circuit's logical memory configuration must be between 75% and 100% of the total bits in the target architecture. The lower limit ensures that we are "stressing" the architecture in each experiment.

4. The logical-to-physical mapper must be able to find a mapping using N or fewer arrays, where N is the number of arrays in the target architecture. As described in Section 4.2.4, even if an architecture contains enough bits, there is no guarantee that the memory resources will be able to implement a circuit's logical memory configuration using N or fewer arrays.

Circuits that do not meet these constraints are discarded, and replacements generated.

In the results of Section 7.1.3, we examine four sizes of FPGAs: FPGAs with two, four, eight, and sixteen 2-Kbit memory arrays. Since each size of FPGA will have a different total number of memory bits, we must create a separate set of benchmark circuits for each size of FPGA. For each of the four sizes, we created circuits until there were 100 that met all of the above criteria. Table 7.1 shows the number of logic blocks, memory blocks, I/O blocks, and nets in the accepted circuits.

Architectural Considerations

As mentioned above, we employ an experimental approach in which each benchmark circuit is "implemented" on an FPGA. The architecture on which the circuits are implemented is the focus of this section.

In selecting a particular FPGA architecture from the family of architectures described in the previous chapter, there are four independent decisions: the number of memory blocks (N), the number and positions of the logic blocks, the value of F_m, and the number of parallel tracks in each routing channel (W). In the next section, we present results for FPGAs with 2, 4, 8, and 16 memory arrays, and for each, we vary F_m from 1 to its maximum value. Thus, the values of F_m and N are constrained by the experiment.

A common approach used to select the number of logic blocks in the target architecture is to choose the smallest square FPGA onto which the benchmark circuit fits [11, 98, 113]. This means that a different architecture is used for each benchmark circuit. The motivation for allowing the FPGA size to float with the circuit size is that this results in "nearly-full" FPGAs, which is the case FPGA designers wish to optimize.

We use a similar approach. Our choice is complicated by the fact that we must ensure a reasonable ratio of logic blocks to memory blocks in the horizontal dimension. Given a desired ratio, the memory block designer can choose an appropriate aspect ratio for the memory block such that the desired memory block to logic block ratio is achieved. For example, if there are to be 8 memory arrays, and 32 logic blocks in the horizontal dimension, the memory block designer can lay out the memory block such that it is four times as wide as one logic block. On the other hand, if there are to be 16 memory arrays and 32 logic blocks in the horizontal dimension, the memory block designer would be hard-pressed to lay out the memory block such that it is only twice as wide as one logic block. By using area models for logic and memory blocks, we have determined that it would be difficult to lay out a memory block with a width less than that of three logic blocks.

Thus, to choose an appropriate number and position of logic blocks, we do the following:

1. If g is the number of logic blocks in the circuit, we choose an architecture with G logic blocks where G is the smallest value such that $G \geq g$ and $\sqrt{G}$ is an integer. Half of these blocks are positioned above the memory row and half below, creating an

Class of architectures	$R = \frac{\text{num. logic blocks in horiz. dimension}}{\text{num. memory blocks}}$		aspect ratio	
	Average	St.Dev	Avg	St.Dev
2 arrays	12.1	3.35	1	0
4 arrays	6.60	1.99	1	0
8 arrays	3.93	0.86	1.08	0.25
16 arrays	3.01	0.01	3.16	1.72

Table 7.2: Architecture statistics.

approximately square chip.

2. We assume that an architecture with G logic blocks contains $8\sqrt{G}$ I/O blocks on the periphery of the chip (two I/O blocks per logic block on each edge). If the circuit requires more I/O pins, G is adjusted to give a square FPGA with enough pins.

3. The above will create an architecture with $R = \sqrt{G}/N$ logic blocks per memory block in the horizontal dimension, where N is the number of memory blocks. If R is less than three, we create the smallest *rectangular* FPGA with enough logic blocks and I/O blocks such that $R = 3$.

Table 7.2 shows the FPGA aspect ratio and logic blocks per memory block averaged over all 100 circuits for each class of FPGA architecture.

The final architectural parameter to be discussed is the number of tracks per routing channel, W. In our experiments, we let this parameter float, as was done in [11, 98, 113]. For each implementation, we determine the *minimum number of tracks per channel* for which all nets in the circuit can be routed. This value of W is then used in the area model described below.

Implementation Tools

Figure 7.2 shows the CAD flow used to implement each benchmark circuit. The logical-to-physical mapper was described in Section 4.2.2, the automatic placement tool in Section 6.2.1, and the automatic routing tool in Section 6.2.2. The routing is performed several times for each circuit; between attempts, we vary the number of tracks in each channel. From this, we determine the minimum channel width required to route all nets in the circuit. In each case, we assume a uniform channel width across the FPGA.

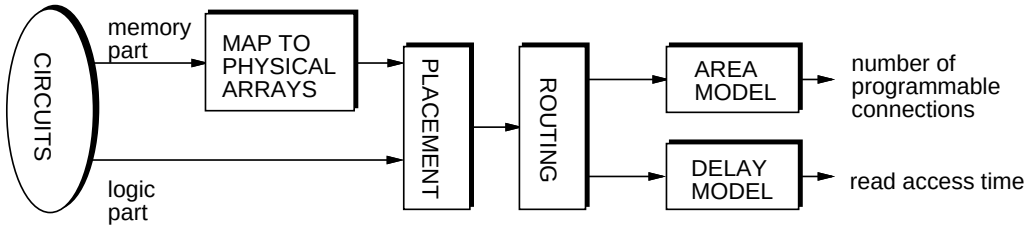


Figure 7.2: Methodology for memory/logic interconnect block experiments.

We do not need to perform logic optimization or technology mapping for the logic since the logic portions of the circuits from the circuit generator have already been optimized and are expressed in terms of five-input lookup tables.

Area Model

The area required by an FPGA architecture is the sum of the area required by the logic blocks, memory blocks, and routing resources. Since the value of F_m does not affect the number of memory blocks or logic blocks required to implement a circuit, we focus on the routing area.

The routing resources are made up of three components: programmable switches, programming bits, and metal routing segments. Since estimates of the area required by the metal routing segments are difficult to obtain without performing a detailed layout, we concentrate on the area due to the programming bits and switches. Unlike the standalone architecture of Chapter 4, the area due to switches and programming bits makes up a significant portion of the FPGA area.

The programming bits and switches appear in three places: connections between logic blocks and neighbouring routing tracks, connections between memory blocks and neighbouring routing tracks (within the memory/logic interconnect blocks), and connections between routing tracks in switch blocks. First consider the connections between logic blocks and routing tracks. The number of such connections is:

$$\text{programmable connections between logic and routing} = PF_cG$$

where G is the number of logic blocks, P is the number of pins incident to each logic block, and F_c is the number of tracks to which each pin can be connected. In our architecture,

$P = 6$ and $F_c = W$; thus,

$$\text{programmable connections between logic and routing} = 6WG$$

Within each switch block, there are $2 * F_s * W$ programmable bidirectional connections; in our architecture $F_s = 3$ and there are approximately G switch blocks, thus,

$$\text{programmable connections in switch blocks} = 6WG$$

Within each memory/logic interconnect block, there are $M * F_m$ programmable connections. In the architecture assumed in this Chapter, there are 8 data-in pins, 8 data-out pins, and 11 address pins incident to each array. Since there are N arrays,

$$\text{programmable connections in memory/logic interconnect} = 27NF_m$$

Combining the above gives:

$$\text{number of programmable connections} = 12WG + 27NF_m \tag{7.1}$$

The number of programmable connections estimated by Equation 7.1 is used as an area metric. Of course, not all programmable connections are the same size; the W connections to the input of a logic block do not each require their own programming bit since the W bits can be encoded. Nonetheless, the number of programmable connections gives a good indication of the relative sizes of routing architectures.

Since the number of logic blocks is typically much larger than the number of memory arrays, Equation 7.1 suggests that the area depends much more on the number of tracks in each channel than on F_m.

Delay Model

A detailed delay model is used to compare the delays of circuit implementations for FPGAs with different values of F_m. Since the circuits from the circuit generator do not include flip-flops, it is difficult to identify the critical path of each circuit. Thus, rather than comparing critical paths, we compare the *maximum read access time* of the memories in each circuit. The read access time is the time to read an item from memory, including the delay due

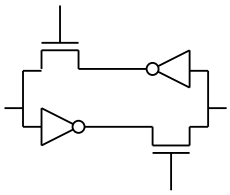


Figure 7.3: Bi-directional re-powering switch.

to the routing between the address pins and the driving logic block and the delay due to the routing between the data-out pins and the logic sink. The delays of the logic blocks on either end are not included. A memory typically has several address and data connections; the longest connection for each is chosen. For circuits with more than one memory, the memory with the longest read access time is chosen.

Like the delay model used in the stand-alone architecture studies, the model used here contains two parts: the delay through the memory array and the delays of the input and output networks. The first component is estimated using the access time model described in Section 5.3.1, which takes into account the delay due to the decoder, wordlines, bitlines, and sense amplifiers. The second component, the network delay, is found by constructing RC trees for each routed net, and finding the Elmore delay of each tree [81].

The delay of a routed net depends significantly on the switch architecture of the FPGA. If a single pass transistor is used for each switch, the delay increases quadratically as the net length increases. To speed up high fan-out and long nets, commercial FPGAs either contain re-powering buffers distributed through the FPGA, or the associated CAD tools use logic blocks to re-power long or high fan-out nets. The identification of the optimum buffer placement strategy is an active area of research. In our model, we make the pessimistic assumption that *every* switch in each switch block and memory/logic interconnect block is replaced by the bidirectional re-powering switch of Figure 7.3. Although this architecture is not likely to be used in practice because of the large area requirements of each switch block, we believe that the delay estimates obtained by assuming such an architecture are similar to those that would be obtained had a more intelligent buffer placement policy been employed.

7.1.2 Effect of F_m on FPGA with 16 Memory Blocks

The first set of results is for an FPGA with sixteen 2-Kbit memory arrays. We will first discuss how F_m affects FPGA routability, and then compare FPGAs with different values of F_m in terms of area and speed.

Track Requirement Results

As explained in Section 7.1.1, the results are based on the implementation of 100 stochastically generated circuits. For each circuit, the minimum channel width required to route all nets in the circuit is determined, assuming that all channels across the FPGA have the same width. This channel width (number of tracks) is then averaged over all 100 benchmark circuits, giving the average number of tracks required to route the circuits. This is repeated for several values of F_m, giving the solid line in Figure 7.4. The dotted lines show the track requirement plus and minus one standard deviation. As expected, as the memory/logic interconnect flexibility (F_m) increases, the average track requirement drops. It is interesting, however, that beyond $F_m = 4$, very little further drop is seen. The final point on the horizontal axis is the case in which each pin can be connected to *all* of the tracks in the neighbouring vertical channel. Even with this high degree of flexibility, almost no improvement beyond that obtained for $F_m = 4$ is seen.

The graph shows an anomaly at $F_m = 3$. To understand why $F_m = 3$ is a bad choice in this particular architecture, it is necessary to examine the memory/logic interconnect pattern employed. Consider Figure 7.5, which shows a particularly bad case for the pattern described in Section 6.1.3. In this example, the number of memory pins, M, is 27, the number of vertical tracks incident to each memory/logic interconnect block, V, is 57, and the number of these tracks to which each pin can connect, F_m, is 3. As can be seen, the 81 memory connections ($M * F_m$) connect to only 27 of the vertical tracks, leaving 30 vertical tracks unconnected. Forcing the 27 memory pins to compete for only 27 vertical tracks when 57 are available is clearly bad for routability. In the next section, we derive a general condition for which the memory/logic interconnect block described by Equation 6.2 leaves unconnected vertical tracks, and in the subsequent section we show how this general result relates to our experiments.

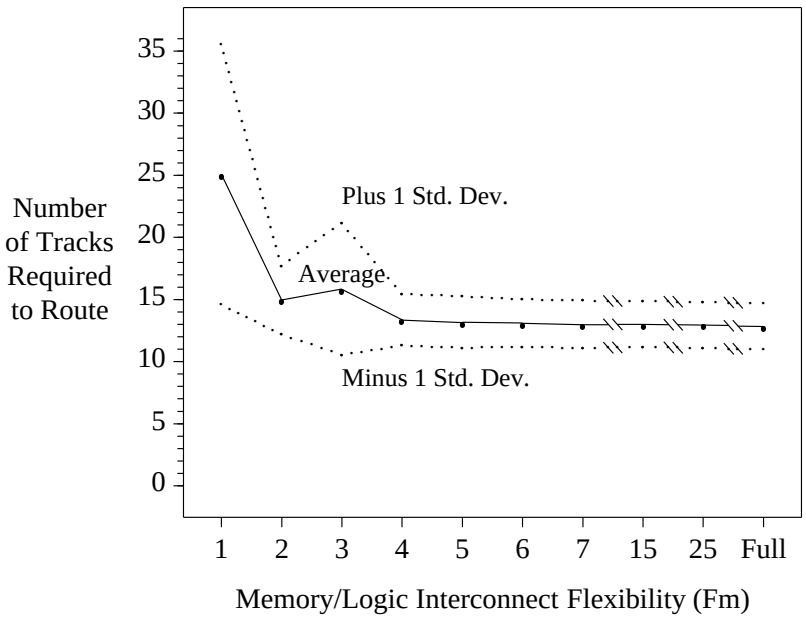


Figure 7.4: Average track requirement as a function of F_m for FPGA with 16 arrays.

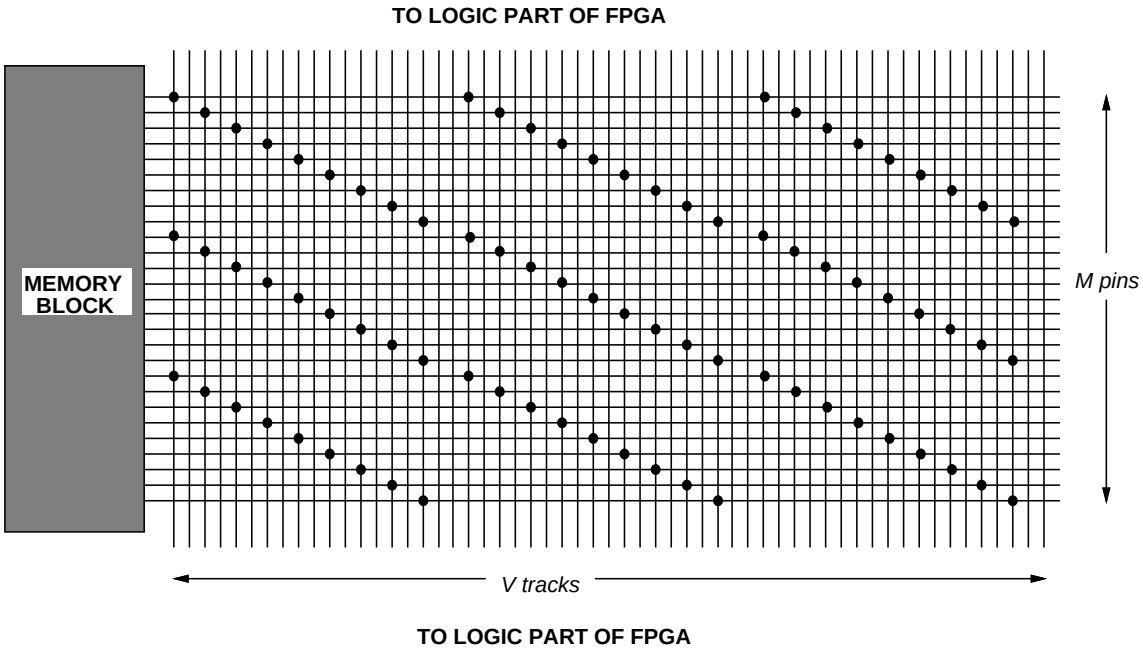


Figure 7.5: Bad case for memory/logic interconnect block of Chapter 6.

General condition for poor memory/logic interconnect blocks

In this subsection, we show that the memory/logic interconnect block described by Equation 6.2 results in $V - M$ vertical tracks being left unconnected if V and M are both divisible by F_m and $V > M$.

We use the following notation: $T_i(p)$ indicates the track number of the i'th connection to pin p ($0 \leq i < F_m$, $0 \leq p < M$, and $0 \leq T_i(p) < V$). From Equation 6.2,

$$T_i(p) = \left\{ \left\lfloor \frac{pV}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor i \right\} \bmod V \tag{7.2}$$

We will use T_i to denote the set of all elements of $T_i(p)$ for all $0 \leq p < M$.

Assume that V and M are both divisible by F_m and $V > M$. We will first show that each of the M elements of T_0 are distinct. From Equation 7.2, the first operand of the modulus operation is never larger than V, since $p < M$. Thus:

$$T_0(p) = \left\lfloor \frac{pV}{M} \right\rfloor$$

Since $V > M$, it is clear that no two elements of T_0 are the same.

We now show that $T_{i+1}(p) = T_i(p + M/F_m)$ (this result will be needed later). From Equation 7.2,

$$T_{i+1}(p) = \left\{ \left\lfloor \frac{pV}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor (i+1) \right\} \bmod V$$

From our initial assumptions, we know V/F_m is an integer. Taking advantage of this, and simplifying, gives:

$$\begin{aligned} T_{i+1}(p) &= \left\{ \left\lfloor \frac{(p + \frac{M}{F_m})V}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor i \right\} \bmod V \\ &= T_i\left(p + \frac{M}{F_m}\right) \end{aligned} \tag{7.3}$$

We further show that $T_i(M + d) = T_i(d)$ for all integers d. From Equation 7.2,

$$\begin{aligned} T_i(M+d) &= \left\{ \left\lfloor \frac{(M+d)V}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor i \right\} \bmod V \\ &= \left\{ V + \left\lfloor \frac{Vd}{M} \right\rfloor + \left\lfloor \frac{V}{F_m} \right\rfloor i \right\} \bmod V \\ &= T_i(d) \end{aligned} \tag{7.4}$$

Having established the above, we now show that if x is an element of T_i, it is also an element of T_{i+1} under our assumptions. In other words,

$$T_i(p) = T_{i+1}(k) \text{ for some } k : 0 \leq k < M \quad \forall p : 0 \leq p < M \tag{7.5}$$

If $M/F_m \leq p < M$, then from Equation 7.4,

$$T_i(p) = T_{i+1}\left(p - \frac{M}{F_m}\right)$$

while if $0 \leq p < M/F_m$,

$$\begin{aligned} T_i(p) &= T_{i+1}\left(p - \frac{M}{F_m}\right) \\ &= T_{i+1}\left(p + M - \frac{M}{F_m}\right) \\ &= T_{i+1}\left\{p + \frac{M(F_m - 1)}{F_m}\right\} \end{aligned}$$

where we have used Equation 7.5 to give a pin number in the required range. In both cases, the argument of T_{i+1} is between 0 and M; therefore, Equation 7.5 is true.

Since T_i and T_{i+1} each have M elements, and since every element in T_0 is unique, from Equation 7.5 we know that each set T_i for $0 \leq i < F_m$ contains the same M unique elements. Since each element is between 0 and $V - 1$, and since $V > M$, there are $V - M$ integers that are *not* within T_i for any $0 \leq i < F_m$. Thus, the vertical tracks that are represented by these missing elements are not connected to any memory pins.

Application of General Result to our Experiments

In all of our experiments, each array contains 11 address pins, 8 data-in pins, and 8 data-out pins, giving $M = 27$.

The value of V depends on the number of tracks in each logic routing channel:

$$V = \left\lfloor \frac{(RN + 1)W}{N + 1} \right\rfloor$$

where N is the number of memory blocks, W is the width of a single channel between two logic blocks, and R is the number of logic blocks per memory block in the horizontal

dimension. From Table 7.2, in the experiments using an FPGA with 16 memory arrays, $R = 3$ for most circuits. This (along with $N = 16$) gives:

$$V = \left\lfloor \frac{49W}{17} \right\rfloor$$

From the above equation, V is divisible by 3 when $18 \leq W \leq 25$. Although the average W for this set of experiments is below 18, there are many circuits that require a W in the above range. For these circuits, both M and V are divisible by 3. From the result in the previous subsection, we know that the resulting memory/logic interconnect block will contain unconnected vertical tracks when $F_m = 3$. Thus, we would expect a larger average track requirement for this value of F_m; Figure 7.4 shows that this is the case.

Area Results

As explained in Section 7.1.1, simply counting the number of tracks needed may not be an accurate reflection of the area required for routing. Instead, we use the number of programmable connections in the routing resources of the FPGA as an area metric. Figure 7.6 shows the number of programmable connections vs. F_m, averaged over the 100 circuits, as well as the results plus and minus one standard deviation. As expected, this follows the track requirement curve from Figure 7.4 very closely. Since the logic routing resources contain many more switches than the memory/logic interconnect does, the increase in the memory/logic interconnect area as F_m increases is swamped by the decrease in the area of the logic routing resources due to the decrease in the number of tracks per channel. Thus, in this architecture, the best area efficiency is obtained for $F_m \geq 4$. For very large values of F_m, the area increases slightly due to the larger memory/logic interconnect blocks and the inability of the extra switches to reduce the track requirement. The final point on the graph shows the number of programmable connections when F_m equals its maximum value; in this case, only slightly more area is required than when $F_m = 7$.

Delay Results

Figure 7.7 shows the average read access time of the memories in the 100 benchmark circuits. As explained above, the read access time is the time to perform a read including the delay of the address-in and data-out networks. The extremes of $F_m = 1$ and F_m equal to its

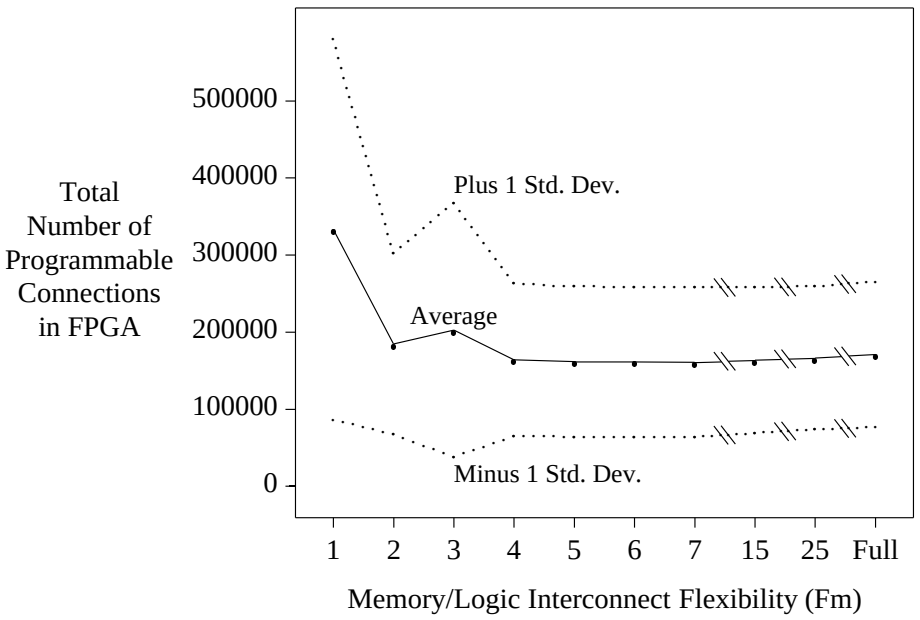


Figure 7.6: No. of programmable connections as a function of F_m for FPGA with 16 arrays.

maximum value are both bad choices. If $F_m = 1$, circuitous routes are required to connect nets to the low flexibility memory pins. These circuitous routes pass through more switch blocks than would otherwise be necessary, leading to longer net delays. When F_m is its maximum value, the large number of switches in the memory/logic interconnect block adds a significant amount of extra capacitance to the routing wires, again leading to longer routing delays. Between $F_m = 2$ and $F_m = 7$, the delay is roughly constant.

Combining the area and delay results, we see that the area and speed efficiency of the FPGA is roughly the same over a wide range for F_m. Architectures with $4 \leq F_m \leq 7$ all have approximately the same area and speed efficiency.

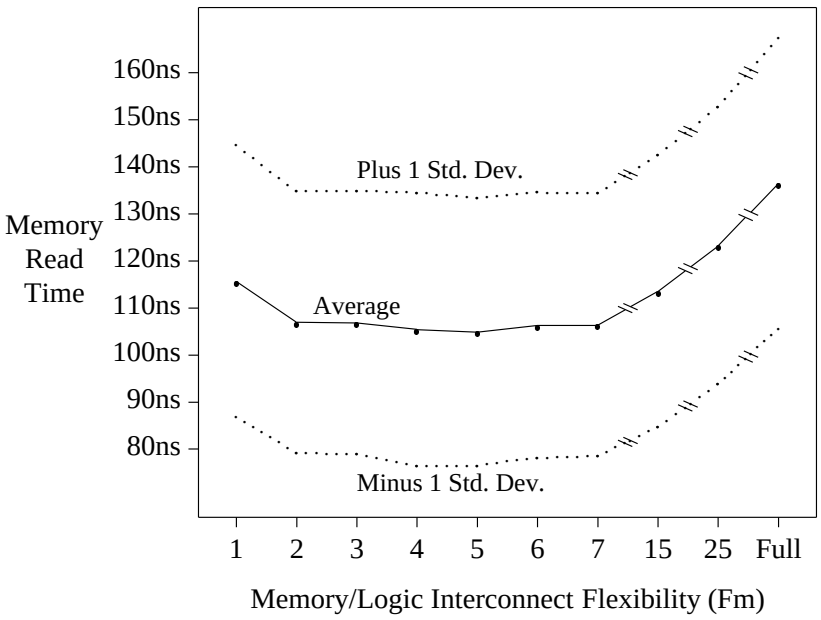


Figure 7.7: Delay as a function of F_m for FPGA with 16 arrays.

7.1.3 Effect of F_m on FPGAs with Fewer Memory Blocks

The results in the previous section were for an FPGA with sixteen arrays. Here we compare the results to those obtained for FPGAs with two, four, and eight 2-Kbit arrays.

Track Requirement Results

Figure 7.8 shows the number of tracks required in each channel averaged over all circuits for all four FPGA sizes. Recall from Section 7.1.1 that the benchmark circuits used to evaluate each architecture depend on the total number of memory bits in the architecture. Thus, each line in Figure 7.8 represents a different set of 100 benchmark circuits.

The most interesting trend in Figure 7.8 is that the results for the smaller FPGA architectures are less sensitive to F_m than those for the larger architectures. In fact, for the 2 and 4 array FPGAs, an F_m of 1 or 2 provides sufficient flexibility. To understand why the smaller architectures can tolerate such low values of F_m, it is necessary to examine the circuits that are used to evaluate each architecture.

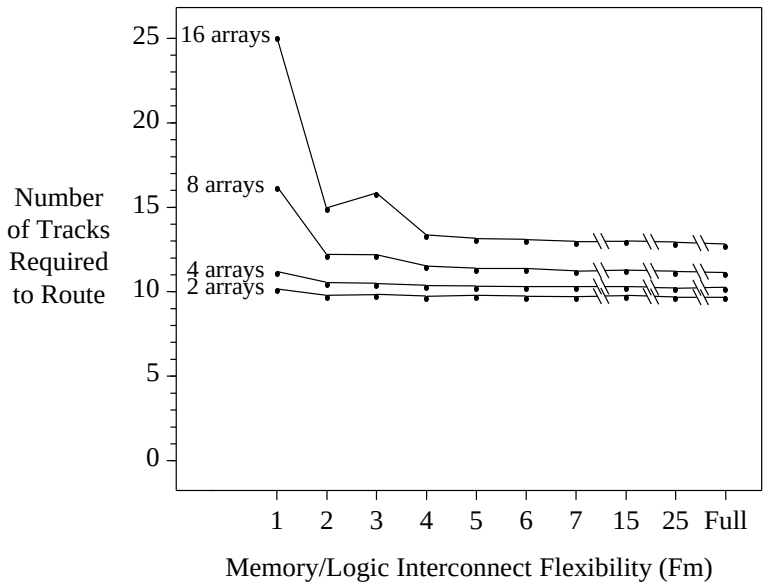


Figure 7.8: Average track requirement as a function of F_m.

Table 7.3 breaks down all nets in the benchmark circuits into three categories:

1. nets which connect only logic blocks
2. nets that connect exactly one memory block to one or more logic blocks
3. nets that connect *more* than one memory block to one or more logic blocks

Consider the third category: nets that connect to more than one memory block. In the generated circuits, these nets serve two purposes:

1. When arrays are combined to implement a larger logical memory, the address pins (and possibly data-in pins) of the arrays are connected together.
2. In clusters with more than one logical memory, often the data-in pins are driven by a common bus (the shared-connection patterns described in Section 3.2.3).

Architecture to which circuits are targeted	Category 1 nets	Category 2 nets	Category 3 nets	Memory Pins per category 3 net
2 arrays	94.2%	4.32%	1.45%	2.00
4 arrays	93.0%	5.24%	1.78%	2.77
8 arrays	92.2%	5.93%	1.91%	4.24
16 arrays	89.4%	8.17%	2.43%	7.19

Table 7.3: Net statistics.

These memory-to-memory nets are particularly hard to route with small values of F_m. Figure 7.9 illustrates a net that connects to three memory pins. Assuming a low value of F_m, there are three regions of low flexibility; the logic routing resources must be used to connect to specific tracks incident to each of these three low-flexibility regions. Given the relatively few options available within each switch block, circuitous routes are often required to make connections between these low flexibility regions, causing routability problems.

Intuitively, these nets will appear more often in circuits aimed at larger architectures, since there are likely more memory blocks to connect. Table 7.3 shows that this intuition is correct. Thus, the FPGAs used to implement the larger circuits need a higher value of F_m.

To investigate this further, we removed all memory-to-memory connections from the circuits, and repeated the experiment. Figure 7.10 shows the results for the 16-array case. The solid line shows the original results from Figure 7.4. The dashed line shows the results obtained from the circuits with the memory-to-memory connections removed. The dotted line shows the results obtained from the circuits with *all* memory connections removed (clearly, this is independent of F_m). As the graph shows, removing just the memory-to-memory connections gives a routability only slightly worse than that obtained by removing all memory connections (only slightly more tracks are required). This motivates us to study these memory-to-memory connections more closely; in Section 7.2 we will present architectural enhancements aimed at efficiently implementing memory-to-memory connections.

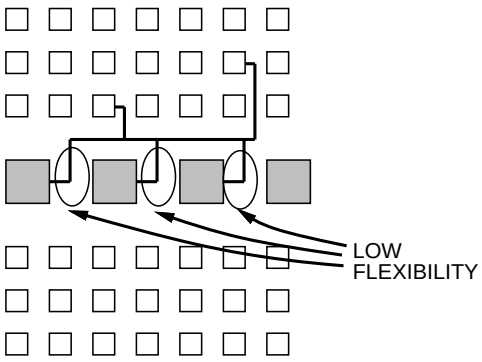


Figure 7.9: A net connected to three memory blocks: three regions of low flexibility.

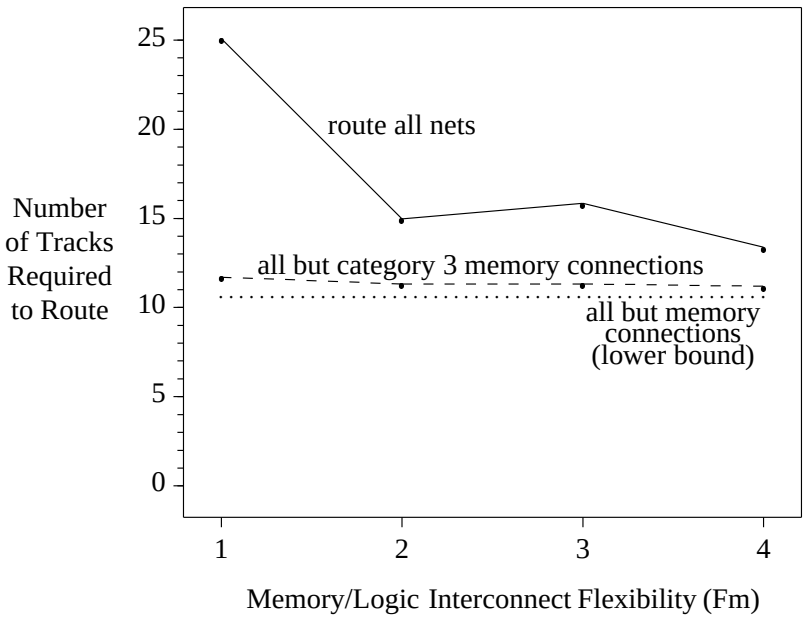


Figure 7.10: Effect of removing memory-to-memory connections.

Area Results

Figure 7.11 shows the number of programmable connections required in each architecture as a function of F_m. As before, the curves closely follow the track requirement results of Figure 7.8, since the number of programmable connections is dominated by the connections within the logic routing resources.

Delay Results

As shown in Figure 7.12, the delay curves for the smaller FPGAs follow the same trends as the 16-array FPGA examined earlier. Since the nets connecting the memories have a larger fanout in the larger FPGAs, we might expect the delay of the larger FPGAs to be more sensitive to F_m. Because of the buffering assumptions described in Section 7.1.1, however, it is the *depth* of the tree implementing a net that determines its delay. Branches that are not part of the longest source-to-sink path of a tree do not contribute significantly to its delay.

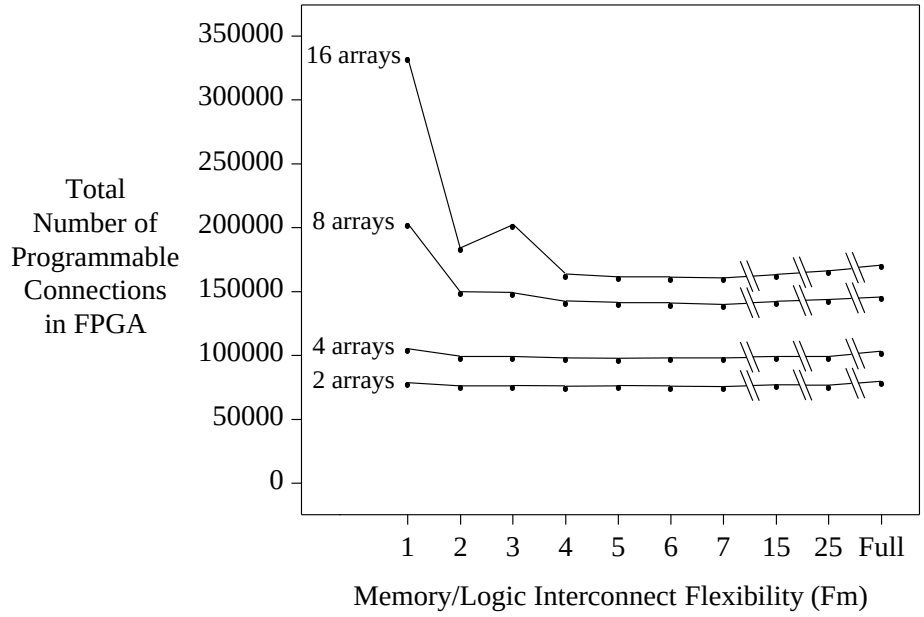


Figure 7.11: Number of programmable connections as a function of F_m.

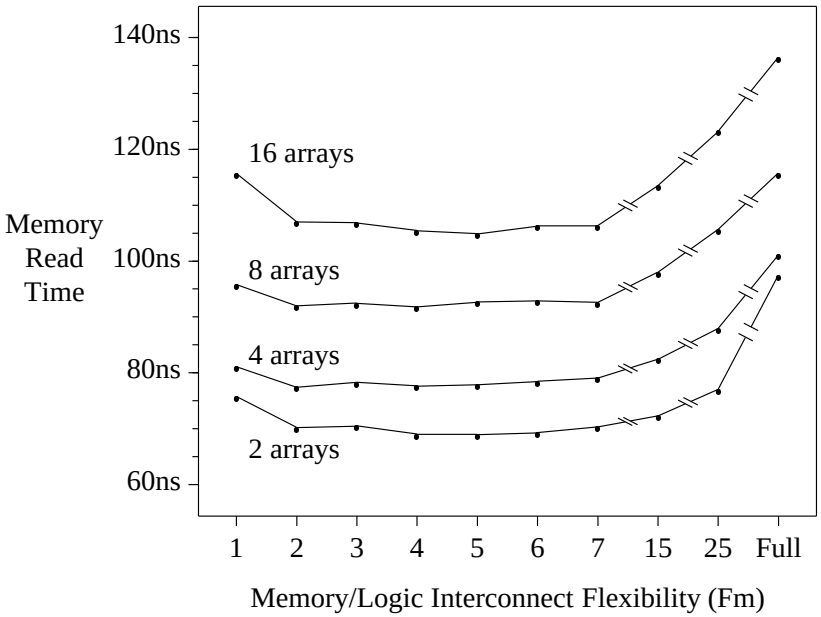


Figure 7.12: Delay as a function of F_m.

7.1.4 Effect of Connection Block Flexibility

In the next two sections, we vary the base architecture slightly and repeat the experiments. In this section, we reduce the flexibility of the logic routing resources, and in the next section, we assume the arrays are positioned at the edge of the FPGA rather than in the middle.

To create a less flexible logic architecture, we reduce the number of tracks to which each logic block pin can connect (F_c in [37]). The results presented earlier assume $F_c = W$; that is, each logic block pin can connect to any of the tracks in its adjacent channel. Figure 7.13 compares these previous track requirement results to those obtained if $F_c = 0.5W$ for an FPGA with 8 memory arrays. As the graph shows, reducing F_c means more tracks per channel are required; this result is not new and was reported in [37]. It is interesting, however, that the dependence on F_m is the same for both values of F_c (the "shapes" of the two curves are similar). In the preceding section, we showed that changing F_m primarily affects the memory-to-memory connections; the difficulty in routing these connections does not depend on F_c. Thus, the fact that the dependence on F_m is the same for both curves in Figure 7.13 gives additional evidence that the reasoning in the previous section was correct.

Figure 7.14 shows the number of programmable connections as a function of F_m. Again, the dependence on F_m closely matches that in the track requirement graph of Figure 7.13. Note that the architecture with $F_c = 0.5W$ requires fewer programmable connections; this is because there are fewer programmable connections attached to each logic block pin.

7.1.5 Memory at Edge of FPGA

The results so far have assumed that the memory arrays are positioned in the middle of the chip, as shown in Figure 6.1. The motivation for placing the arrays in the middle of the FPGA was to minimize the average wire-length of the logic-to-memory connections. It is conceivable that in some circuits, however, these memory arrays might "get in the way", causing extra contention in the middle of the device. In this section, we position the memory arrays at the edge of the chip, as shown in Figure 7.15, and repeat the experiments described above.

Figure 7.16 compares the track requirements of an 8-array FPGA with the memory

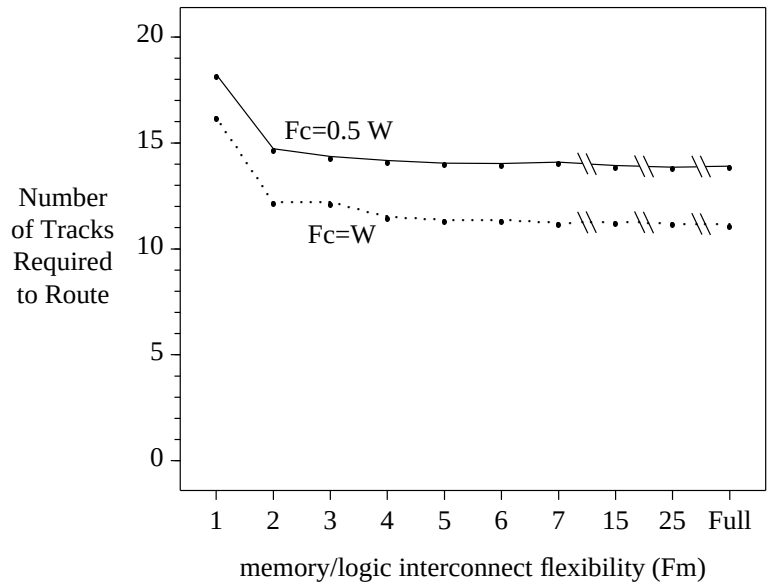


Figure 7.13: Track requirement results for two values of F_c.

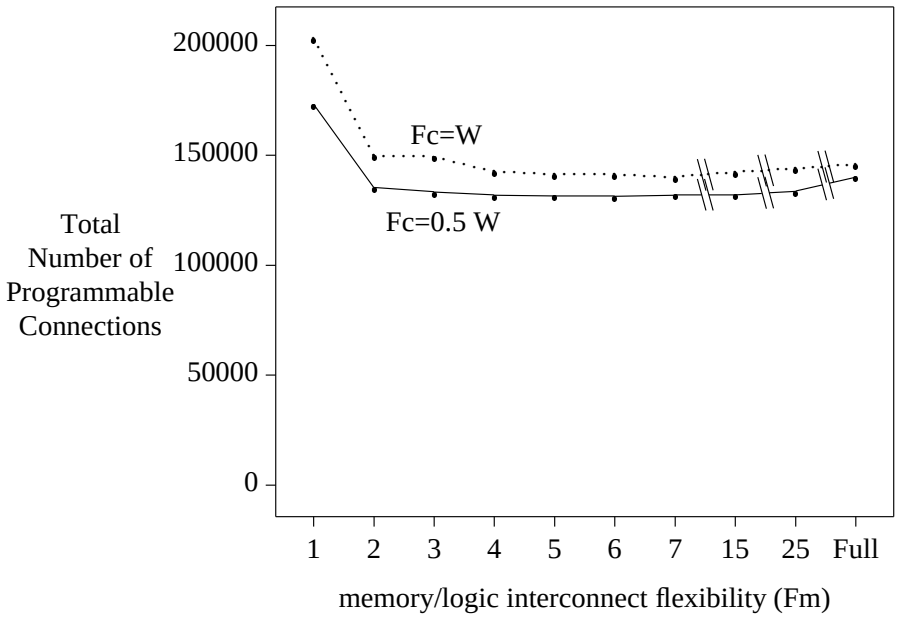


Figure 7.14: Number of programmable connections for two values of F_c.

arrays at the edge (solid line) to the same FPGA with the memory arrays in the middle (dotted line). As the graph shows, the FPGA with the arrays at the edge is slightly less routable; approximately one extra track per channel is required (four extra tracks if $F_m = 1$). The effect of F_m on the track requirement, however, is roughly the same. The area results in Figure 7.17 follow the track requirement results closely, as before.

A potential advantage of placing the arrays near the edge of the chip is that direct

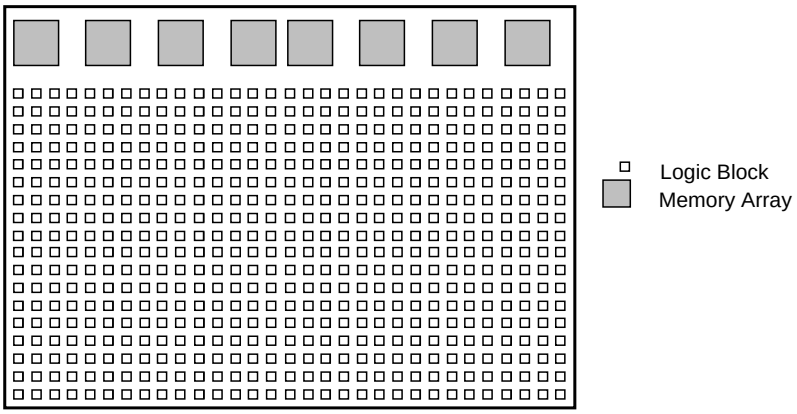


Figure 7.15: Floorplan of FPGA with memory arrays at one edge.

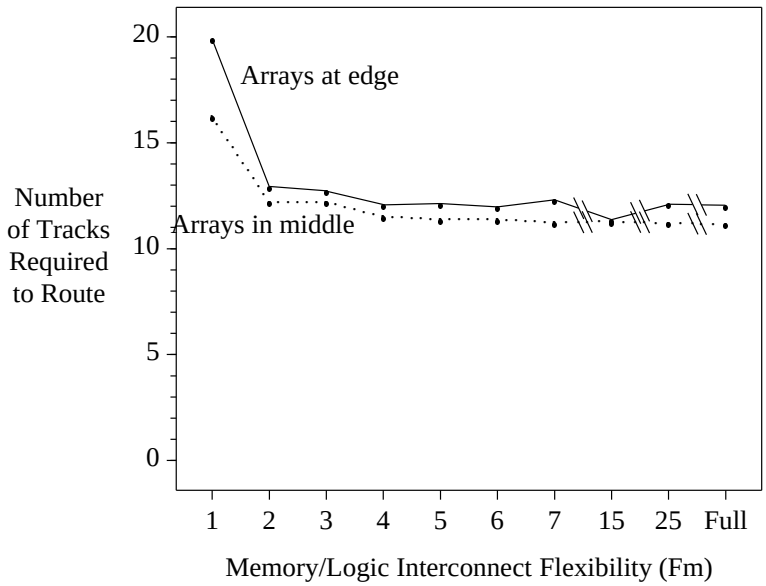


Figure 7.16: Track requirements results for FPGA with memory arrays at one edge.

connections between memory arrays and I/O pins would be easier. Providing direct access to the memory arrays can result in a more testable circuit since the contents of the memories can be more easily read and written by an external test circuit. Also, arrays at the edge of the device may result in faster I/O to memory transfers, which may be advantageous in some applications. We have not attempted to quantify these advantages, but we believe that the difference in track requirement shown in Figure 7.16 should motivate an FPGA designer to investigate alternative ways of providing connections between the arrays and I/O pins.

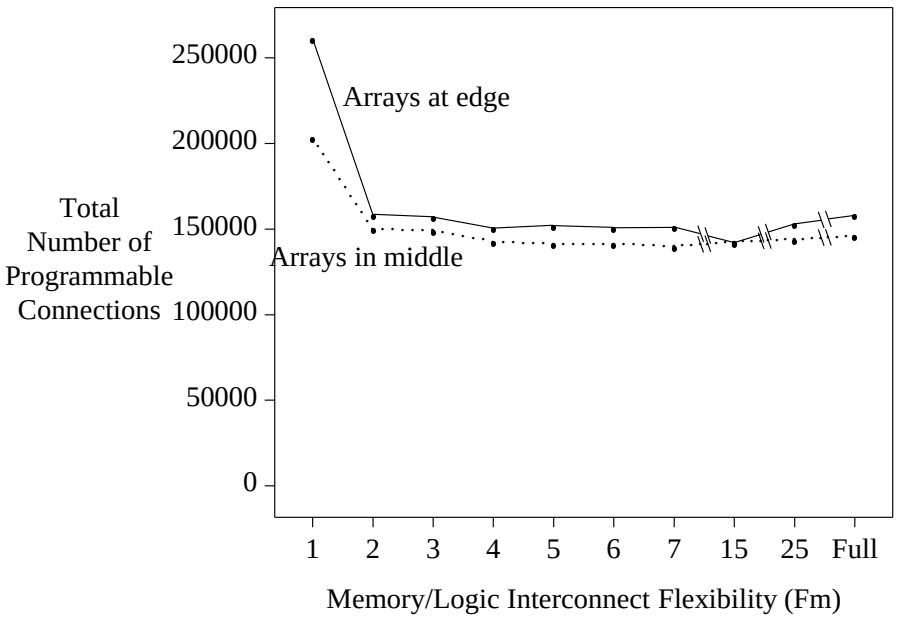


Figure 7.17: Area results for FPGA with memory arrays at one edge.

7.1.6 Summary of Flexibility Results

In this section, we have shown that in the architecture described in the previous chapter, the most area-efficient FPGA occurs when $F_m \geq 4$. This conclusion holds for FPGAs with 2, 4, 8, and 16 memory arrays, as well as devices with $F_c = 0.5W$ and devices with the memory arrays at one edge of the chip. The most speed-efficient FPGA occurs for $2 \leq F_m \leq 7$.

Combining the area and speed results, the most efficient architecture occurs for $4 \leq F_m \leq 7$. As F_m approaches its maximum value, the speed-efficiency is reduced considerably, and the area-efficiency somewhat.

The amount of flexibility required in the memory/logic interconnect is determined to a large extent by connections between memory arrays. In the next section, we consider two architectural enhancements intended to reduce the track requirements of the FPGA by supporting these memory-to-memory connections.

7.2 Enhancements to Support Memory-to-Memory Connections

In this section, we consider two architectural enhancements intended to efficiently implement memory-to-memory connections, and hence improve the area- and speed-efficiency of the resulting FPGA.

7.2.1 Pins on Both Sides of Arrays

The first enhancement is aimed at common connections between neighbouring memory blocks. In the standard architecture, shown in Figure 7.18, each memory pin is connected to F_m tracks in the channel to its right. If a net is to be connected to two neighbouring memory blocks, the connection must be performed using the logic routing network, as shown in Figure 7.19 (the circuitous route shown may be a result of contention for tracks or may be a result of the limited connectivity within a switch block). Figure 7.20 shows an enhanced interconnect structure; each input pin of a memory is connected to tracks in both the right *and* left neighbouring channels. The specific track(s) connected within both channels are the same. Figure 7.21 shows how this enhanced architecture can implement the net of Figure 7.19. Since the net is to drive the same pin in each array, a single vertical track between the two arrays can be used.

Since the outputs of our memory arrays can not be placed in a high-impedance state, it does not make sense to share output pins this way. As shown in Figure 7.20, only input pins are connected to both neighbouring channels.

Figure 7.22 compares the track requirement results for this new scheme to those obtained earlier for the 8-array architecture. To present a fair comparison, the horizontal axis of this graph is the number of switches in each memory/logic interconnect block. In both architectures, each array has 19 input and 8 output pins. In the standard architecture, this translates into $27F_m$ switches per interconnect block, while in the enhanced architecture, there are $2F_m$ switches for each input and F_m switches for each output, leading to $46F_m$ switches per interconnect block. Each point is labeled with F_m.

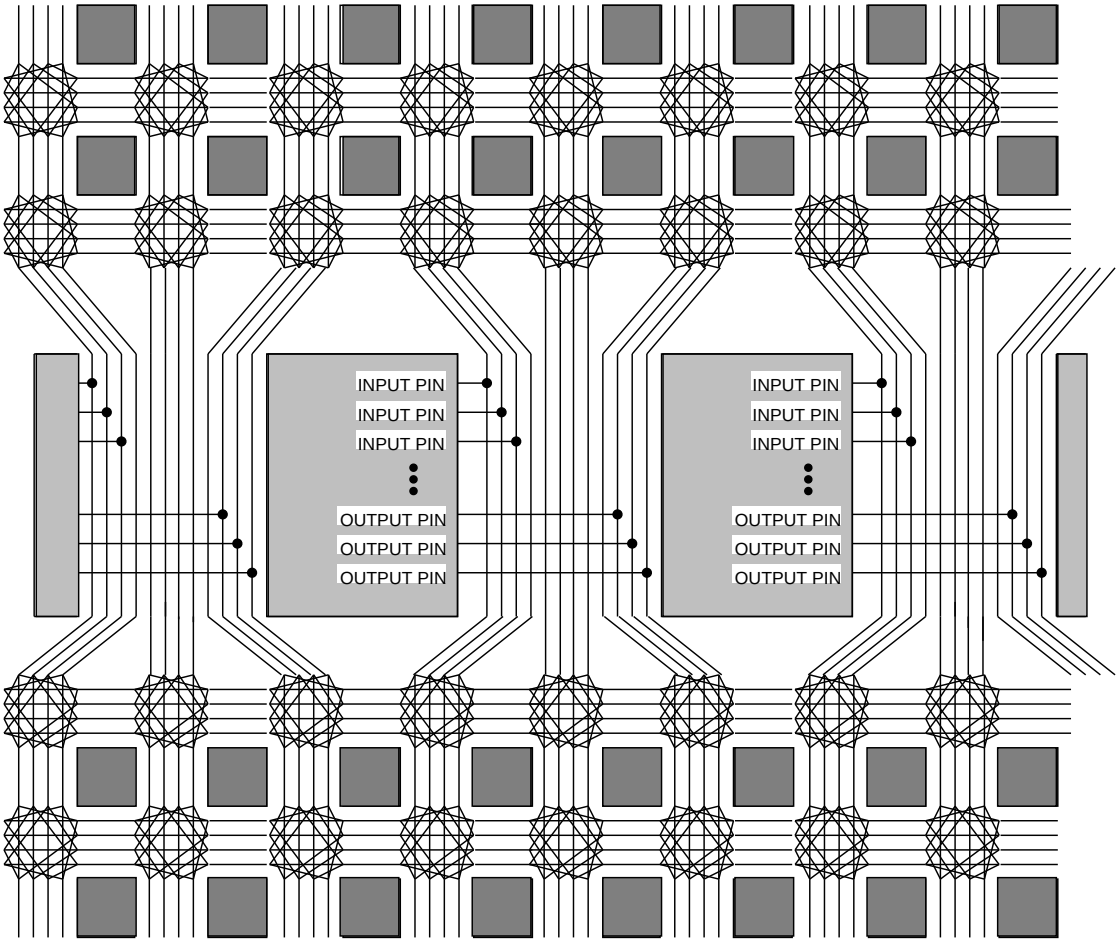


Figure 7.18: Standard memory/logic interconnect structure for $F_m = 1$.

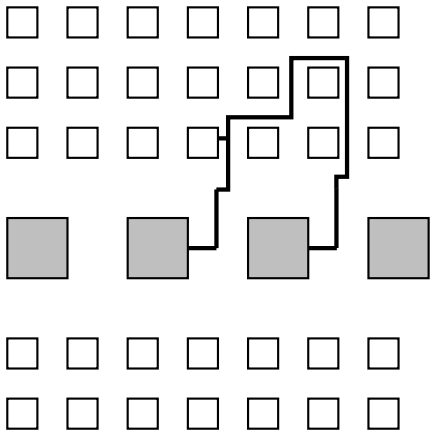


Figure 7.19: Circuitous route on standard architecture.

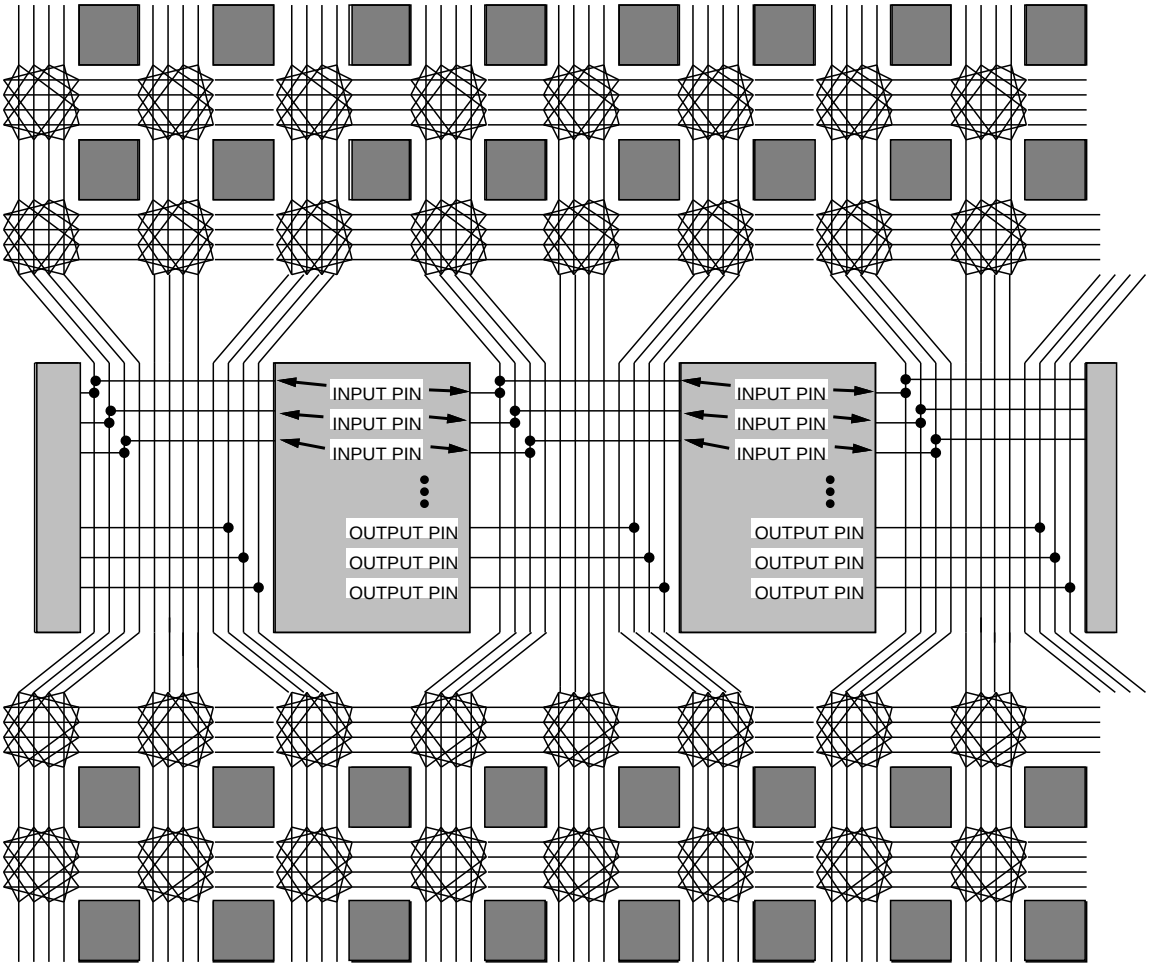


Figure 7.20: Enhanced memory/logic interconnect structure for $F_m = 1$.

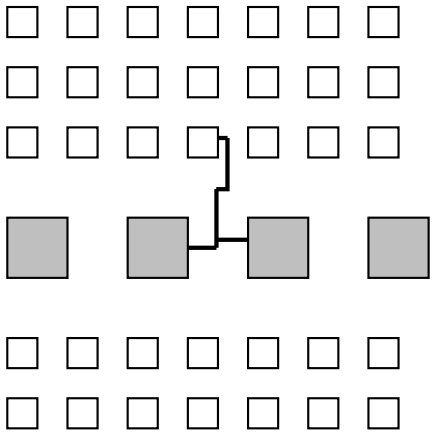


Figure 7.21: Direct route on enhanced architecture.

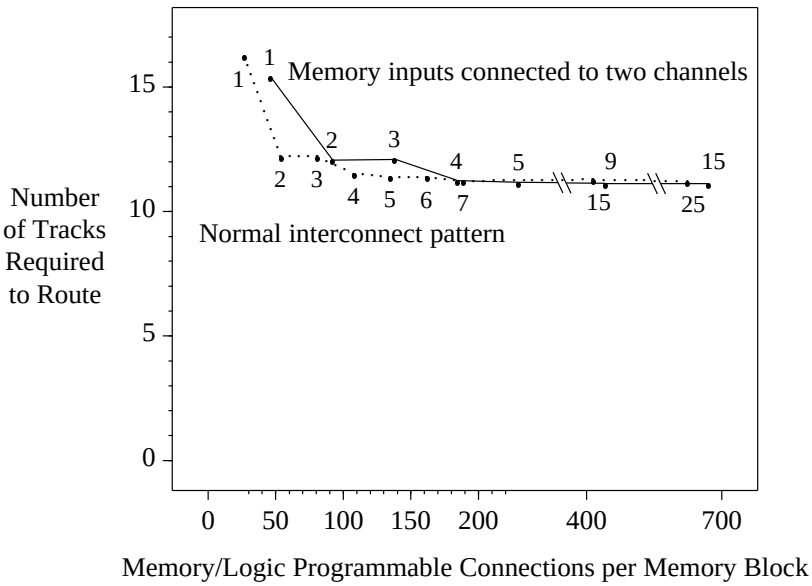
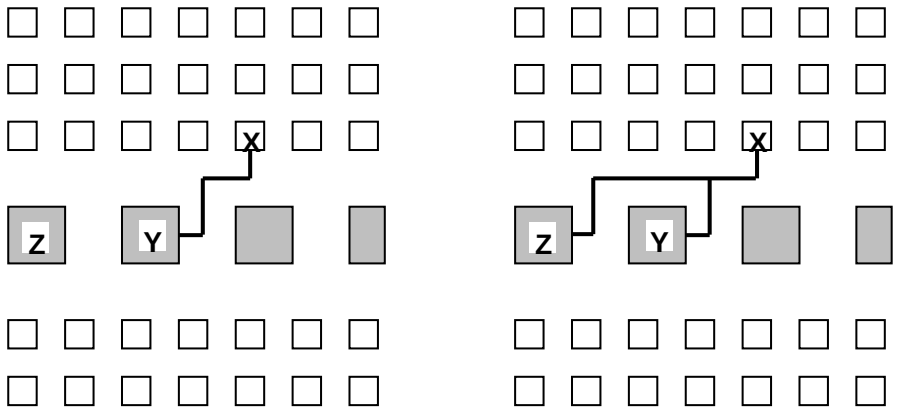


Figure 7.22: Track requirement results for architecture of Figure 7.20.

As the results of Figure 7.22 show, the new architecture is not effective at improving routability. Consider Figure 7.23(a), in which logic block X is to be connected to memory blocks Y and Z. In the absence of contention, the router will route the first segment of the net as shown in Figure 7.23(a). Since track-to-track connections via a memory pin are not allowed (in most commercial FPGAs, pins are unidirectional, making such connections impossible), the remaining connection must be performed using the logic routing resources as in Figure 7.23(b). A better route would be to connect each array to a single vertical track between the two arrays. Although we have added extra switches to the architecture, in this case, the router is unable to take advantage of them.



a) First connection routed b) Second connection routed

Figure 7.23: An example where the router does not take advantage of extra switches.

Even if the router was modified to properly take advantage of the extra switches, the improvement may still not be dramatic. This architecture allows easy connections between neighbouring memory blocks, but does not allow the efficient connection of more than two memory blocks. As shown in Table 7.3, the average net connecting more than one memory block actually connects four such blocks (seven in the 16-array architecture circuits). In the next section, we will present an alternative architecture that efficiently supports these higher memory-fanout nets.

7.2.2 Dedicated Memory-to-Memory Switches

In this section, we present a more aggressive approach to supporting memory-to-memory connections. Some of this material also appears in [114].

Architecture

Figure 7.24 illustrates the second enhanced architecture. Each vertical wire incident to a memory/logic interconnect block can be programmably connected to the corresponding track in the two neighbouring memory/logic interconnect blocks. This connection is made through a pass transistor denoted by a rectangle in Figure 7.24. We refer to each of these pass transistors as a *memory-to-memory switch*. The connections shown as dots are non-programmable permanent connections.

Figure 7.25 shows an example of a net connecting three arrays implemented on the enhanced architecture. Two memory-to-memory switches are used to connect the three memory arrays. The same net implemented on an FPGA with no memory-to-memory switches is shown in Figure 7.26; in this architecture, the net must be implemented using the logic routing resources that could otherwise be used to route signals between logic blocks.

The area cost of this enhancement is small; if there are N arrays and V vertical tracks per memory/logic interconnect block, then NV extra switches are required.

This enhancement is related to the broader channel segmentation issue in FPGAs [22, 38]. There are, however, several major differences from this previous work:

1. In a segmented routing architecture, all channels across the chip usually contain an identical distribution of segment lengths. In the architecture of Figure 7.24, there are

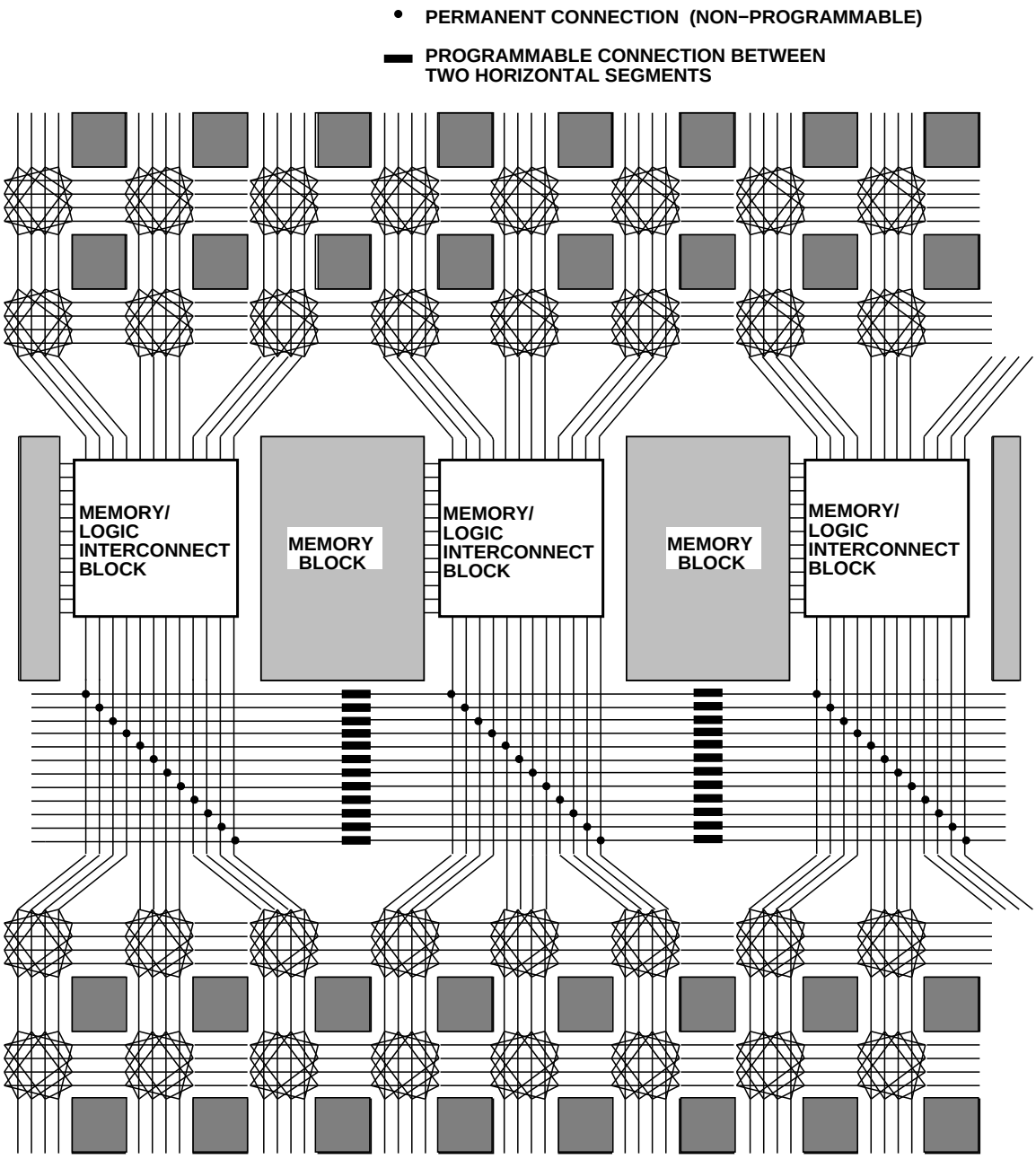


Figure 7.24: Dedicated memory-memory connection architecture.

exactly V additional horizontal tracks, regardless of how many logic routing channels exist on the chip. Thus, the routing architecture is heterogeneous to better match the heterogeneous logic/memory block architecture.

2. Each memory-to-memory connection consists of a programmable switch connecting two tracks. This is topologically different than a standard routing track (of any length), in which two programmable switch blocks are connected using a fixed track.

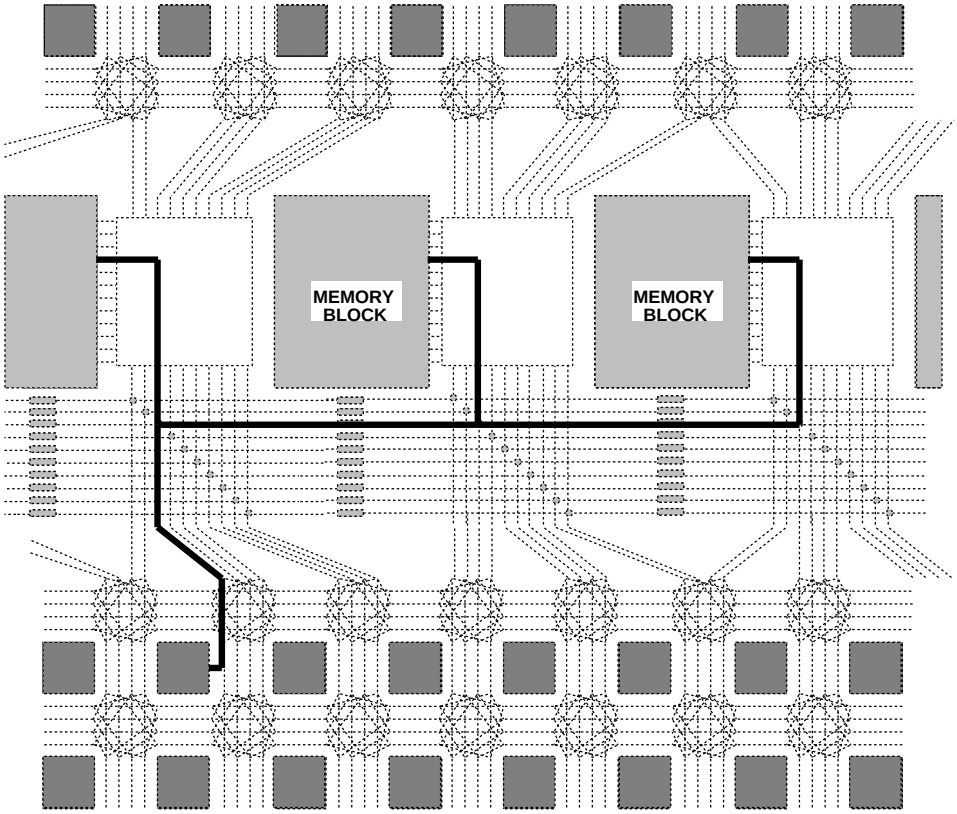


Figure 7.25: Enhanced architecture example.

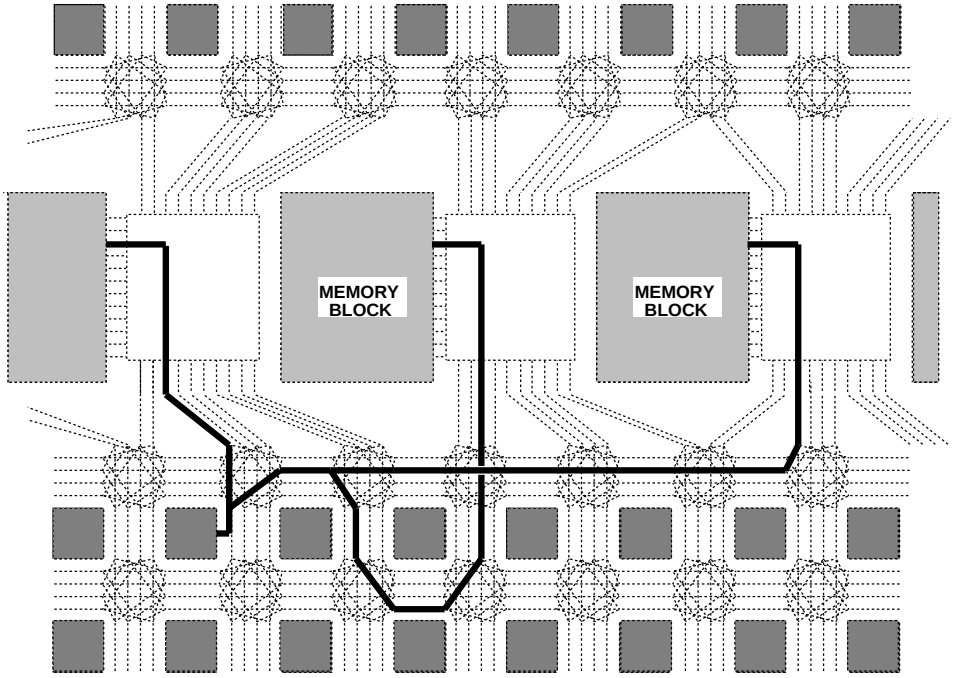


Figure 7.26: Original architecture example.

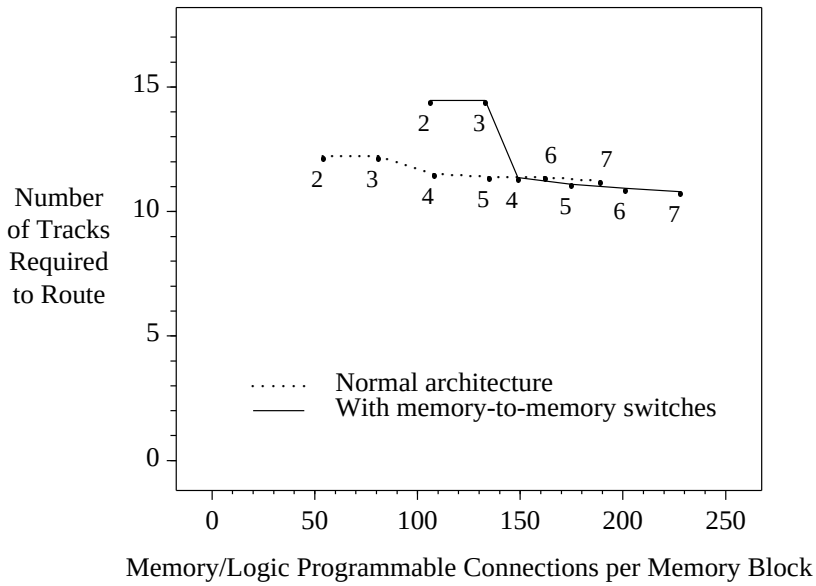


Figure 7.27: Routing results assuming standard router.

CAD Issues

Figure 7.27 shows track requirement results for eight-array FPGA with and without the memory-to-memory switches. The horizontal axis is the number of programmable switches in the memory/logic interconnect region (not the entire FPGA) per memory block. Both architectures contain $27F_m$ switches in each memory/logic interconnect block. In the enhanced architecture there are also V memory-to-memory switches per memory block; V is approximated by:

$$V = \left\lfloor \frac{(RN+1)W}{N+1} \right\rfloor$$

where R is the average R from Table 7.2, $N = 8$, and W is from the measured data. Each point in the graph is labeled with F_m.

As the graph shows, the average track count is increased by approximately 2 tracks for $F_m = 2$ and 3, and relatively unchanged for higher values. At $F_m = 1$, in the enhanced architecture, more that half of the circuits could not be routed using less than 45 tracks; we do not present results for this case.

The primary reason for these disappointing results is that non-memory nets will often use the memory-to-memory switches as a low-cost route to travel from one side of the chip to the other. Consider Figure 7.28, which shows the connection between two distant logic blocks. If the net is implemented using only the logic routing resources, at least six switch

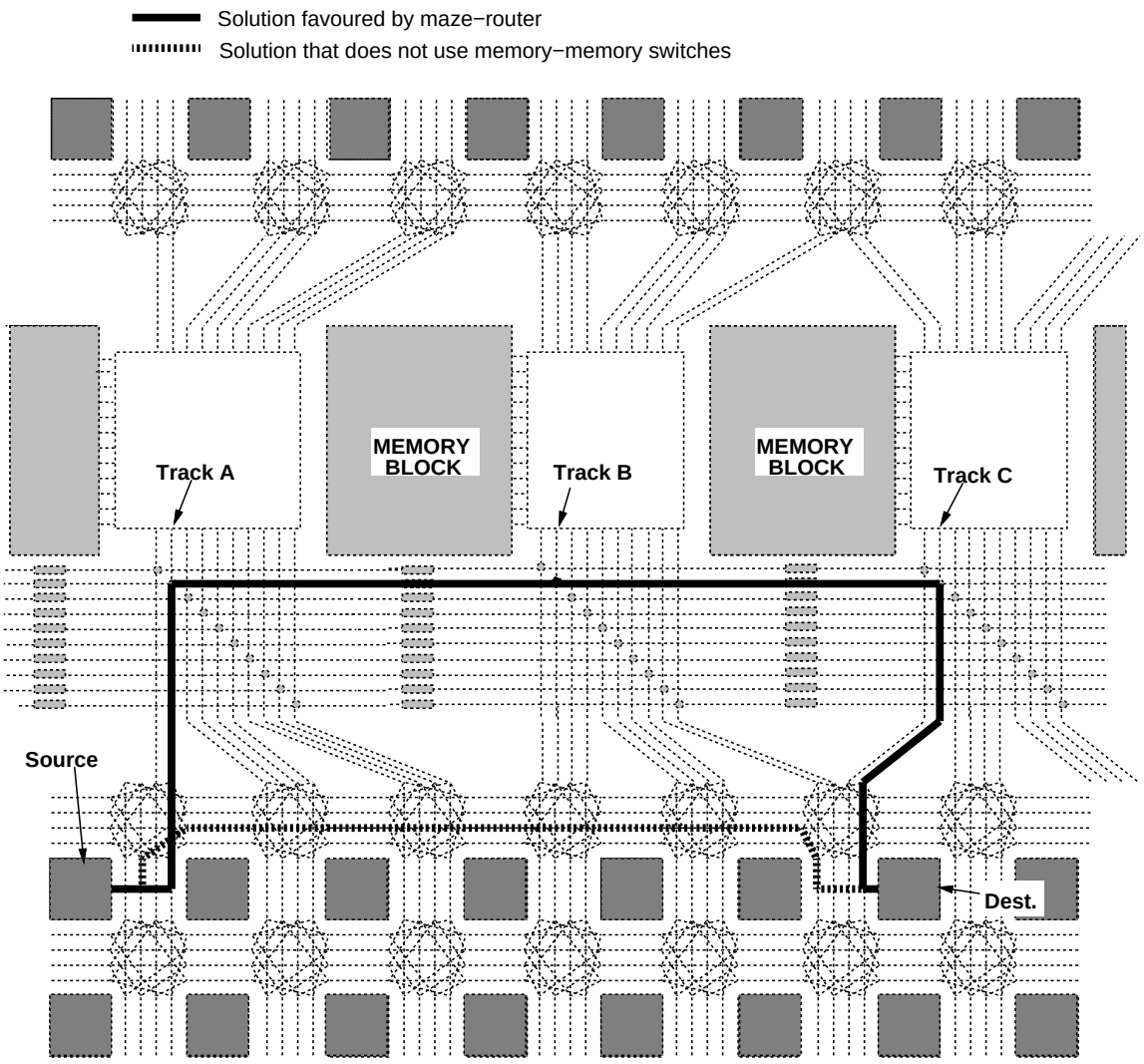


Figure 7.28: Routing a logic net with and without memory-to-memory switches.

blocks would lie on the path between the two logic blocks. Using the memory-to-memory switches, only two switch blocks and two pass transistors (one under each memory block) must be traversed. Since the latter alternative is cheaper, it will be favoured by the router.

Although this provides an efficient implementation of this net, the vertical tracks labelled A and C in the diagram are unavailable for future nets (recall that the router processes nets one at a time). If future nets require connections to the memory, the loss of vertical tracks A and C may severely hamper the routing of these nets, especially in low-F_m architectures. Also, since the connections between the vertical tracks incident to each memory/logic interconnect block and the horizontal tracks connecting the memory-to-memory switches is

permanent, the track labelled B will also be unavailable for future nets.

There are several possible solutions. The most obvious is to route all the memory nets first, and routing nets that do not connect to the memory arrays only after all the memory connections have been made in each pass. The memory-to-memory switches would then only be used by non-memory nets if they are not needed to implement memory connections. There are several problems with this approach:

1. It reduces the flexibility of the "move-to-front" heuristic employed in the router (see Section 6.2.2). This heuristic attempts to route "difficult" nets first; if additional constraints are placed on the net ordering, the heuristic will be less effective.

2. As explained in Section 6.2.2, the address pins of each memory block are equivalent; that is, the connections to any two (or more) address pins of an array can be swapped, and the circuit will function identically. Similarly, the data pins are permutable with the restriction that once an assignment is made to a data-in pin, the corresponding data-out pin assignment is fixed (and vice-versa). Routing all memory nets reduces the potential benefits obtained by this extra flexibility, since if the memory nets are routed first, all memory pin assignments are made when there is little contention in the chip (near the start of each phase).

3. Even the memory nets can interfere with each other as shown in Figure 7.29. In this example, the memories are connected to a logic block using an extra memory-to-memory switch to the right of the right-most memory array to be connected. The track labelled C is then unavailable for future memory nets. Had the logic routing resources been used instead of the right-most memory-to-memory switch, track C would remain available.

A second solution is to replace the permanent connections below each memory/logic interconnect block with programmable switches. This would free track B in the example of Figure 7.28, since the vertical track can be "disconnected" from its corresponding horizontal track. Tracks A and C, however, are still unavailable. The main problem with this approach is the extra area required to implement the switches and programming bits.

A third solution is to use the memory-to-memory switches *only* to implement memory-to-memory connections. Although this means that these tracks are wasted if a circuit

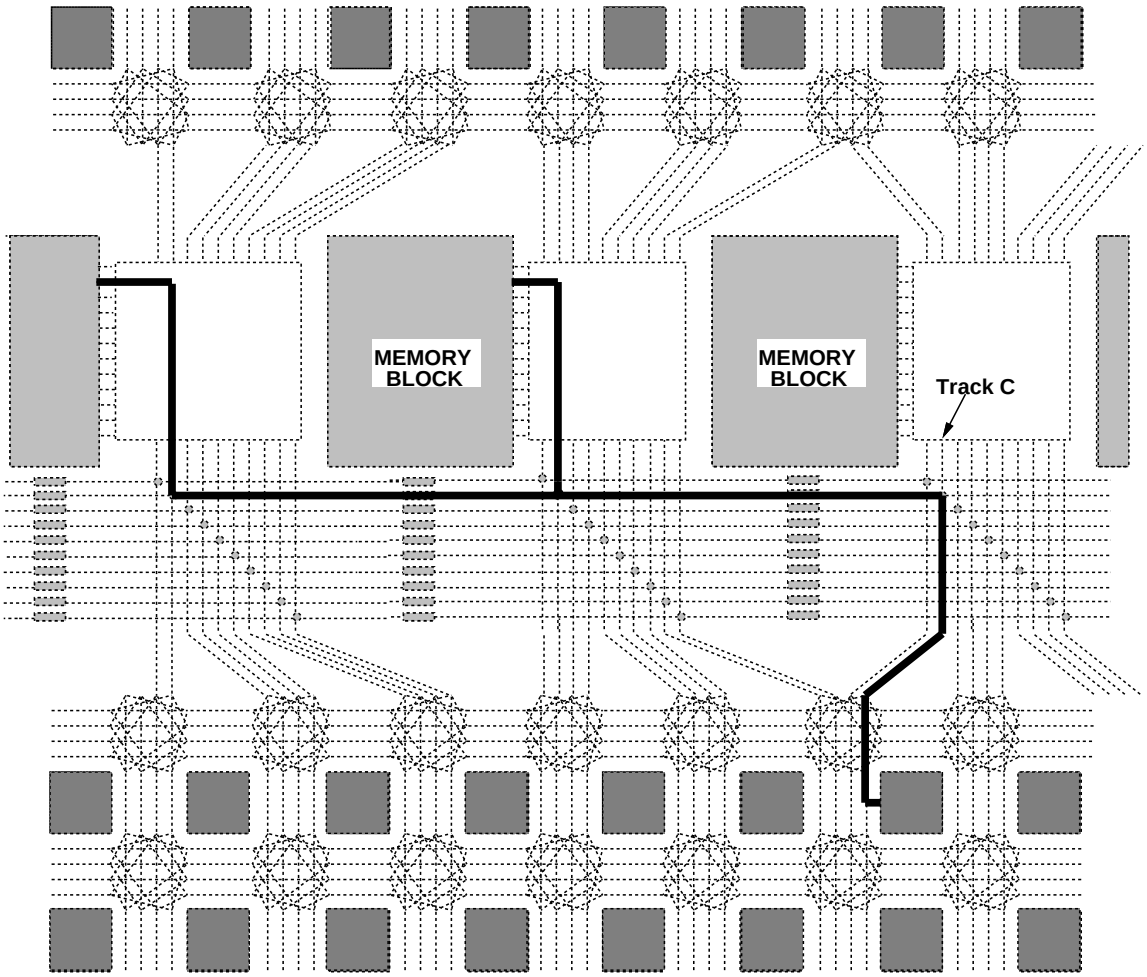


Figure 7.29: Example of a memory net that uses an extra memory-to-memory switch.

contains no (or few) memory-to-memory connections, it alleviates all of the problems described above. To investigate the effectiveness of this approach, we modified the router such that during the normal directional expansion, the memory-to-memory switches are ignored. When routing a net that connects to memory, the algorithm proceeds until the first memory pin is found. The router then determines if the corresponding pins in either of the two neighbouring memory blocks should be on the same net. If so, the connection is made using the proper memory-to-memory switch, the neighbour of the new array is checked, and the process repeated. Once no neighbouring memory blocks can be connected in this way, the normal directional expansion is once again employed for the rest of the net. In this way, the memory-to-memory switches are only used to implement connections between memory arrays. All other connections are made using the normal logic routing resources.

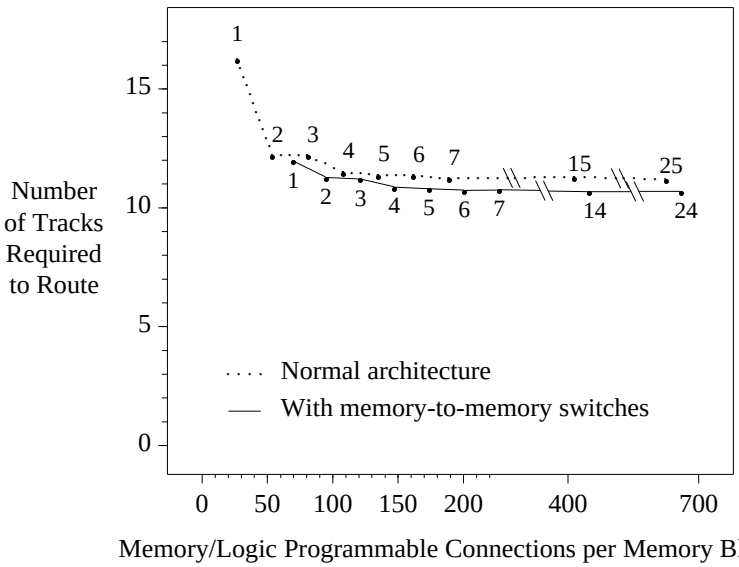


Figure 7.30: Track results for an 8-array FPGA with memory-to-memory switches.

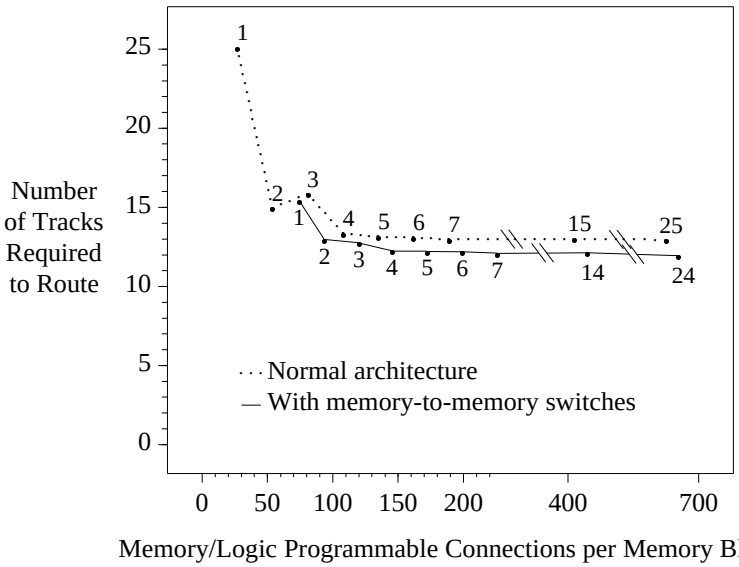


Figure 7.31: Track results for an 16-array FPGA with memory-to-memory switches.

Figures 7.30 and 7.31 show the track requirement results obtained using this algorithm for the 8 and 16 array FPGAs respectively. Again, the horizontal axis is the number of switches in the memory/logic interconnect region per memory array (including the memory-to-memory switches in the enhanced architecture). The most significant comparison is between the lowest point of the two curves; the enhanced architecture requires, on average, half of a track less in each channel than the original architecture for the 8-array FPGA, and almost a full track less in the 16-array FPGA.

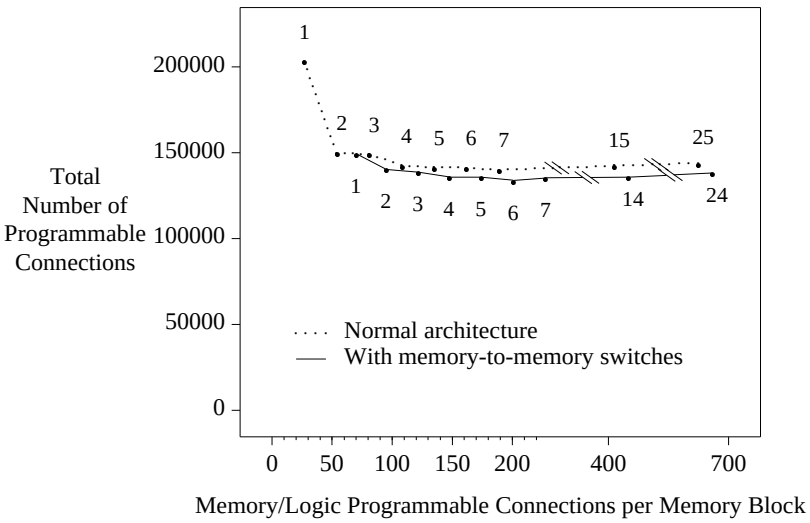


Figure 7.32: Area results for an 8-array FPGA with memory-to-memory switches.

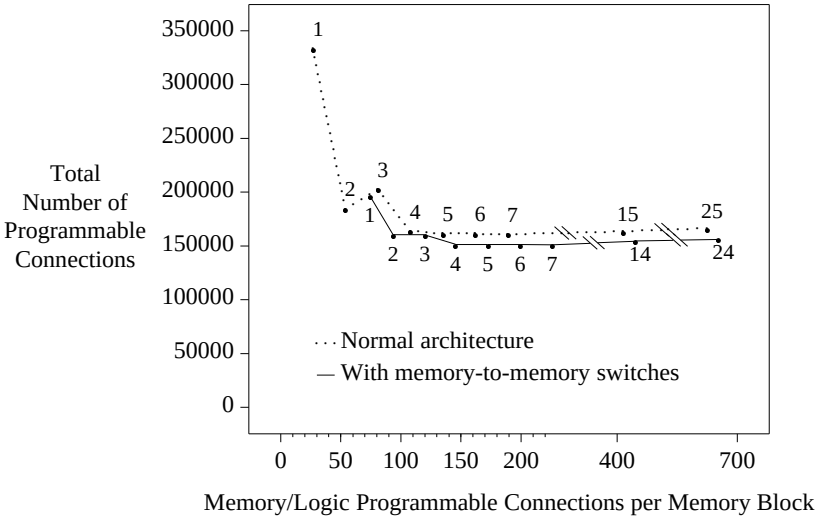


Figure 7.33: Area results for an 16-array FPGA with memory-to-memory switches.

Figures 7.32 and 7.33 show the area results for the 8 and 16 array FPGAs. As before, the area results closely match the track requirement measurements.

Figures 7.34 and 7.35 show delay estimates. As the graphs indicate, the enhancements reduce the memory read time by as much as 25%. Nets that connect to multiple memory arrays usually dominate the memory read time; the memory-to-memory switches provide more direct and therefore faster routes for these paths.

These results show that even with this relatively unaggressive use of the memory-to-memory switches, area is improved somewhat, and speed is improved significantly. The development of algorithms that use these tracks more aggressively is left as future work; it is

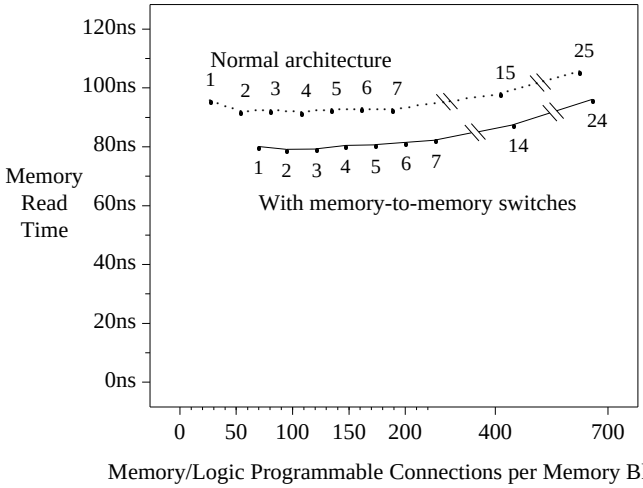


Figure 7.34: Delay results for an 8-array FPGA with memory-to-memory switches.

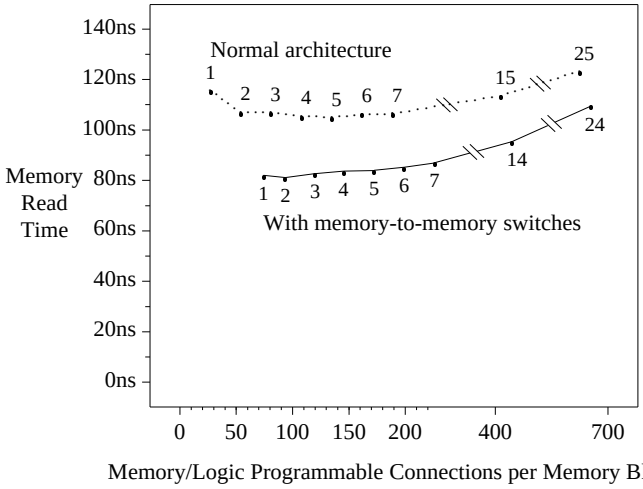


Figure 7.35: Delay results for an 16-array FPGA with memory-to-memory switches.

likely that such algorithms would give improvements beyond those presented in Figures 7.30 through 7.35.

7.3 Scribbler Circuit

The results in the previous two sections were obtained using benchmark circuits from the stochastic circuit generator of Chapter 3. In this section, we repeat the key experiments using a real circuit, called Scribbler, obtained from Hewlett-Packard Labs [112]. Besides providing additional evidence to support our conclusions, this section serves as a first step in the validation of the circuit model of Chapter 3.

Table 7.4 gives the important circuit and implementation properties of the Scribbler circuit. The circuit contains six logical memories, which require eight 2-Kbit arrays. Thus, the results will be compared to the 8-array results from the previous two sections.

Figure 7.36 compares the track requirement results from Section 7.1 (dashed line) to those obtained using the Scribbler circuit (solid line). The dotted lines indicate one standard deviation above and below the dashed line. Comparing Tables 7.4 and 7.3, it is clear that Scribbler contains more than the average number of memory-to-memory connections in the stochastically generated circuits. Thus, we would expect Scribbler to be slightly harder to route than the average stochastically generated circuit. Figure 7.36 shows that this is the case, but that the track requirement of Scribbler is still within one standard deviation of the average.

Area and delay comparisons are presented in Figures 7.37 and 7.38. As can be seen in Figure 7.37, Scribbler requires fewer programmable connections than the average stochastic circuit, even though in Figure 7.36, we showed that Scribbler requires more than the average tracks per channel. This is because Scribbler contains fewer logic blocks than the average

Number of Logical Memories	6
Number of Memory Blocks after Mapping	8
Number of Logic Blocks	486
Number of I/O Blocks	84
Number of Nets	573
Proportion of Category 1 Nets	78.5%
Proportion of Category 2 Nets	15.7%
Proportion of Category 3 Nets	5.76%
Memory Pins per Category 3 Net	2.0
r of FPGA used for experiments	3
Aspect Ratio of FPGA used for experiments	1.14

Table 7.4: Statistics for Scribbler circuit.

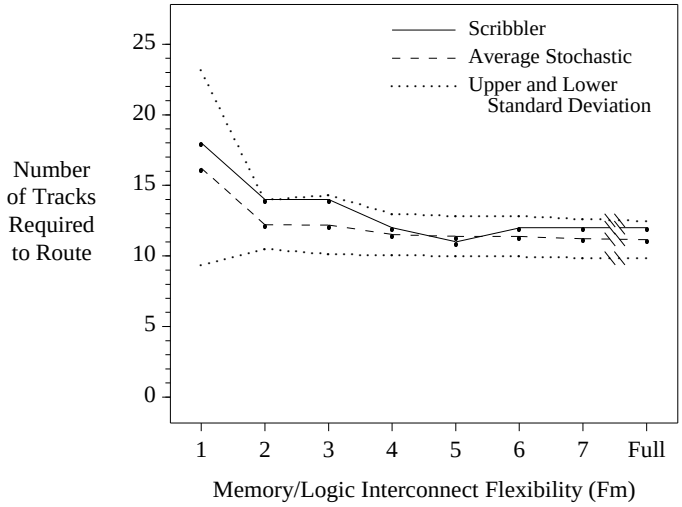


Figure 7.36: Track requirement results for Scribbler.

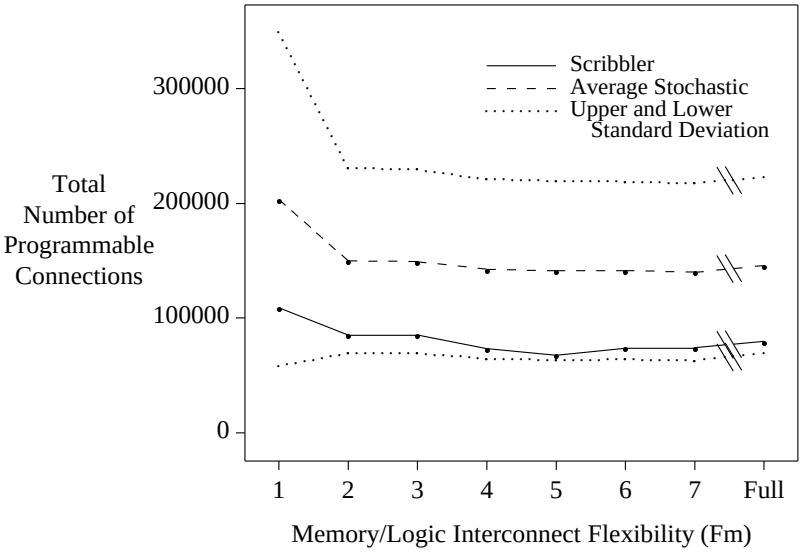


Figure 7.37: Area results for Scribbler.

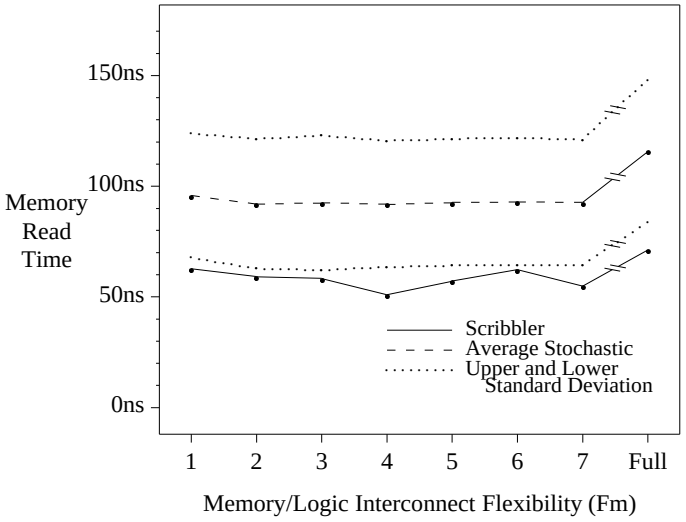


Figure 7.38: Delay results for Scribbler.

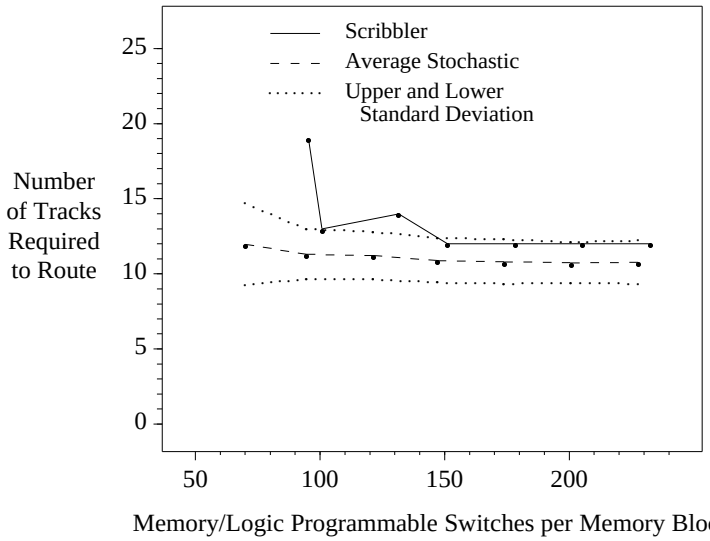


Figure 7.39: Track requirement results for Scribbler assuming memory-to-memory switches.

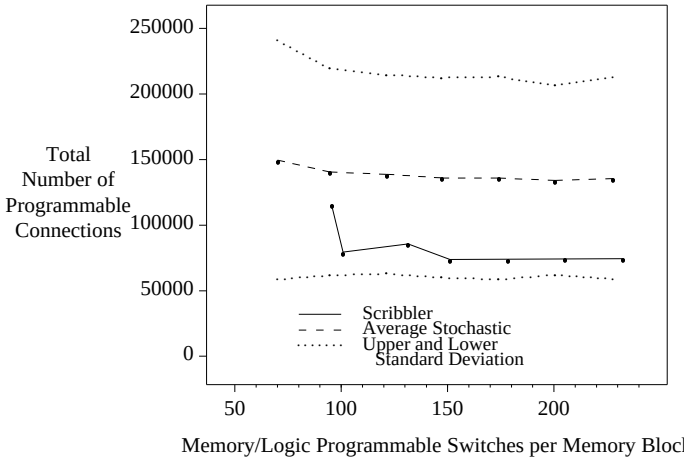


Figure 7.40: Area results for Scribbler assuming memory-to-memory switches.

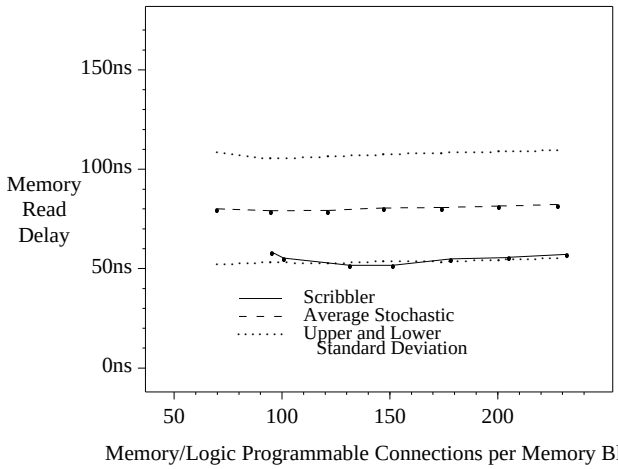


Figure 7.41: Delay results for Scribbler assuming memory-to-memory switches.

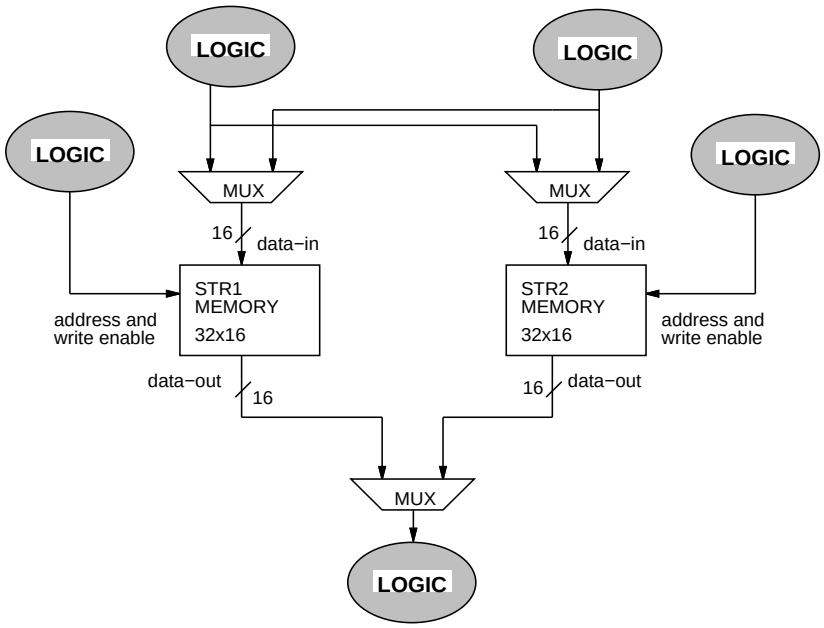


Figure 7.42: One cluster in Scribbler.

stochastic circuit.

Figure 7.38 shows the estimated memory read time of Scribbler. The results are significantly lower than the average from the stochastic circuits. Each memory-to-memory connection in Scribbler goes to only two memory blocks; thus, the typical memory-to-logic distance in Scribbler is less than the average of that in the stochastically generated circuits.

Figures 7.39 to 7.41 show the same comparisons if the FPGA contains memory-to-memory switches. As in Section 7.2, the horizontal axis in each graph is the number of routing switches per memory array.

By comparing Figures 7.36 and 7.39 it is clear that the memory-to-memory switches do not help Scribbler as much as they do the stochastic circuits, even though, from Table 7.4, this circuit has more than the average number of memory-to-memory connections. To understand why this is so, consider Figure 7.42 which shows a high-level view of part of the circuit (two of the six logical memories). Since each logical memory has 16 data bits, the logical-to-physical mapper requires two memory blocks to construct each as shown in Figure 7.43. Since the memory blocks labelled STR1UPPER and STR1LOWER share address connections, we would expect that the placer would place those two memory blocks next to each other. The logic subcircuit connected to the data pins, however, is such

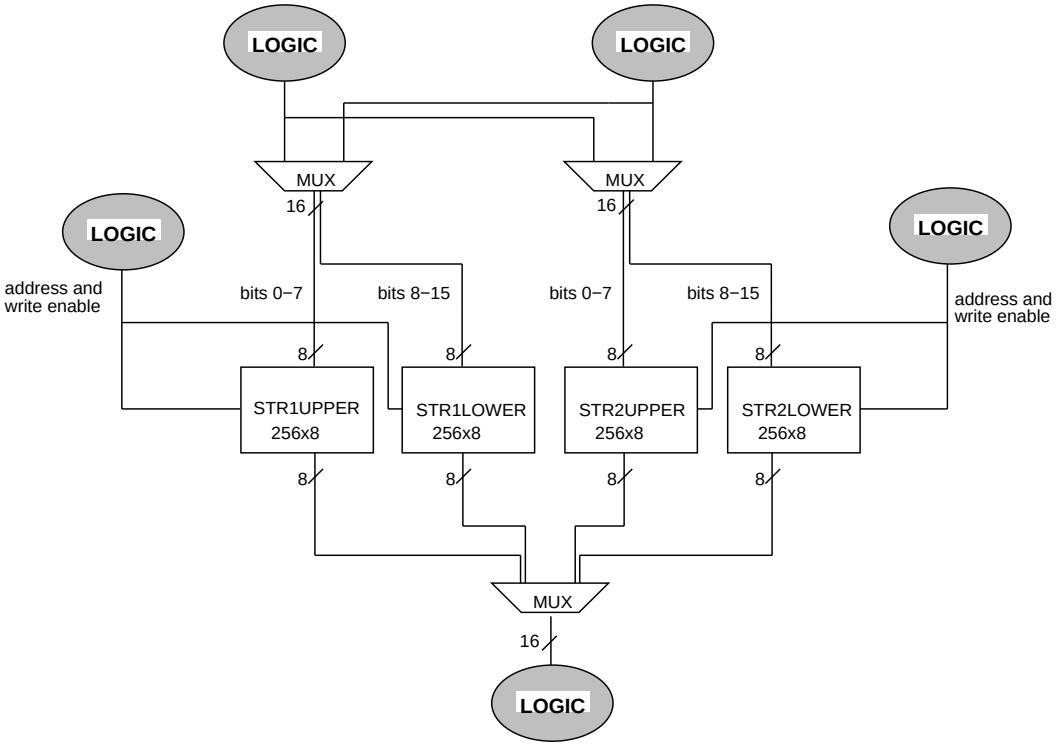


Figure 7.43: One cluster in Scribbler after logical-to-physical mapping.

that each data pin i of the STR1 memory is "tightly connected" to the same data pin i of the STR2 memory. In order to make the data connections shorter, the placer tries to place STR1UPPER next to STR2UPPER and STR1LOWER next to STR2LOWER. Since there are fewer address pins than data pins, the placer favours the latter, resulting in the placement in Figure 7.44. With this placement, the memory-to-memory connections span more than one memory array, and thus, can not be implemented using the memory-to-memory switches. Of the other six logical memories, two do share data-in connections, and these are implemented using the memory-to-memory switches, but it is the routing around the STR1 and STR2 memories that, for the most part, determines the track requirements.

This example shows the danger of using only a few circuits to evaluate configurable memory architectures. The interconnect pattern in Scribbler is such that the memory-to-memory switches are of no use. Yet, by perturbing the circuit slightly, we could have reached

Figure 7.44: Placement of four memory blocks from Figure 7.43.

a different conclusion. If the STR1 and STR2 memories had been deeper, the attraction of the address lines would have been stronger, and may have caused the placer to place STR1UPPER next to STR1LOWER and STR2UPPER next to STR2LOWER. With this placement, the memory-to-memory switches might have helped routability and lowered the track requirements. Since such small changes can so significantly affect the conclusions, it is impossible to find only a few circuits that adequately represent all circuits that will ever be implemented on the device. In order to obtain meaningful results, many circuits are required; thus, a circuit generator such as the one we have used in this research is vital.

7.4 Conclusions

In this chapter, we examined the memory/logic interface in the heterogeneous FPGA architecture described in Chapter 6. We first investigated the effects of F_m on the area and delay of the resulting FPGA. FPGAs with 2, 4, 8, and 16 memory arrays were considered. In each experiment, 100 stochastically generated circuits from the circuit generator of Chapter 3 were used as benchmark circuits. The most area and speed-efficient architecture occurs for $4 \leq F_m \leq 7$. The area conclusions do not change if flexibility is removed from the logic connection block, or if the arrays are moved to the edge of the chip.

We also showed that by adding switches between the vertical tracks incident to each memory/logic interconnect block, the speed performance of the circuits implemented on these devices is improved by as much as 25%. The area efficiency is also slightly improved.

Finally, we repeated the experiments using a real circuit and arrived at the same conclusions. The memory-to-memory switches, however, did not improve the routability of the FPGA because of the peculiarities of the circuit's interconnect structure. This underlines the need for a stochastic circuit generator that can generate many realistic circuits rather than relying on results from only a few benchmark circuits.

Chapter 8

Conclusions and Future Work

8.1 Dissertation Summary

Constant improvement in process technology is changing the way FPGAs are used and designed. In the past, the relatively low capacities of these devices has limited their use to small logic subcircuits, but as FPGAs grow, designers are able to use them to implement much larger systems. These larger systems look significantly different than small logic subcircuits. One of the key differences is that these systems often contain memory. Memory, therefore, will be a crucial component of future FPGAs. This dissertation is the first step in examining the architecture of these next-generation devices.

In Chapter 3, an extensive circuit analysis was presented that examines the use of memory in circuits, and a circuit generator was described. This generator stochastically creates realistic circuits and was used in the architecture studies in the rest of the dissertation. Such a generator is essential to obtain realistic experimental results, since it is the only feasible way to gather enough benchmark circuits to adequately exercise a configurable memory architecture.

In Chapter 4, the FiRM stand-alone configurable memory architecture was generalized and extended to larger memory sizes, and novel algorithms were created that map logical memory configurations to these configurable memory architectures. These algorithms were then used in the architectural experiments described in Chapter 5. The purpose of these experiments was to examine the effects of various architectural parameters on the flexibility,

delay and area of the configurable memory architecture.

Finally, Chapter 6 discussed issues involved in embedding memory resources onto an FPGA. An important part of this work was the generation of tools to map circuits to these devices. In Chapter 7, custom placement and routing programs were used to investigate the flexibility requirements of the interconnect between the memory and logic resources. It was shown that a surprisingly small amount of flexibility results in the most area- and speed-efficient FPGAs, especially in devices with only a few memory arrays. We also examined specific architectural support for memory-to-memory connections, and found that the access times of the memories can be improved by up to 25% by adding switches that allow direct connections between neighbouring memory arrays.

An important part of this work is the use of stochastically generated circuits in the architectural studies. In the last section of Chapter 7, we examined one real example circuit, and showed that architectural conclusions were very dependent on the peculiarities of the circuit, and suggested that if the circuit had been modified even slightly, the conclusions could change. This highlights the need to base architectural conclusions on many circuits; these circuits can only be feasibly obtained using a stochastic circuit generator.

8.2 Future Work

This work is the first published research in this area and has laid the foundation for much more research into configurable memories. Some of the avenues of future work that will likely prove fruitful are discussed in this section. Since architecture studies, algorithm studies, and circuit analysis all make up important parts of FPGA research, we will discuss elements of each.

8.2.1 Architecture Studies

There are many open architectural questions. Chapter 4 briefly described an alternate floorplan in which memory arrays are distributed around the FPGA; the example is repeated in Figure 8.1. This scheme has a number of potential advantages, the primary one being that the contention resulting from memory connections would be spread over the entire chip, rather than concentrated in a single stripe down the middle of the device. Also, if

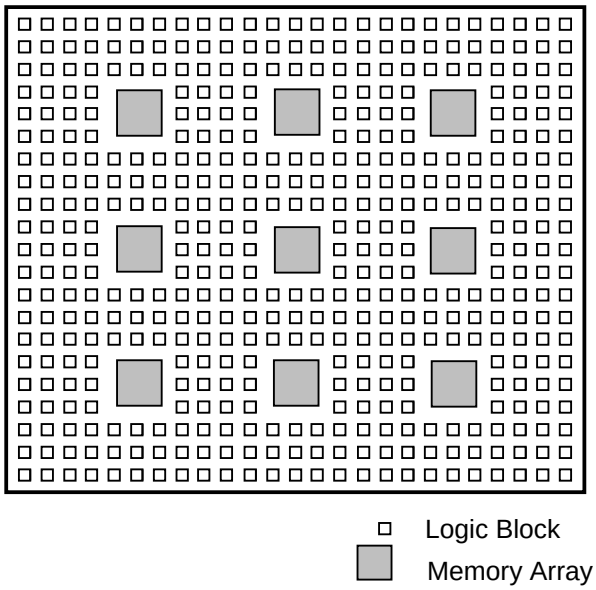


Figure 8.1: An alternative floorplan with arrays distributed throughout the FPGA.

circuits with many small logical memories are to be implemented, the average memory-to-logic distances would likely be less, resulting in lower memory access times and higher potential clock frequencies.

The challenge in creating an architecture with this alternate floorplan is that memory-to-memory nets still must be efficiently supported. Likely, additional memory-to-memory tracks would be required, but it is not obvious how prevalent these tracks should be, and how they should be connected to the rest of the FPGA.

A hybrid scheme in which memory blocks are grouped into sets, and these sets distributed throughout the FPGA, might be a suitable compromise. In this architecture, arrays which are to be tightly connected can reside in the same set, while arrays that make up unrelated logical memories (memories from different clusters) can be placed in different sets. Further experimental architectural studies are needed to evaluate these potential floorplans.

8.2.2 Algorithm Studies

FPGA architectures can not be created in isolation. An integral part of any architectural decision is the set of algorithms that will be used to map circuits to the device. Much work remains to be done in finding suitable algorithms for the heterogeneous devices studied

in this dissertation. One of the most compelling avenues of research is the mapping of random logic into unused memory blocks. Logic can be implemented very efficiently in large memories, since the per-bit overhead is much smaller than that for lookup tables. Algorithms that map logic to FPGAs with both memory arrays and logic blocks would likely result in unprecedented improvements in FPGA logic density and speed.

In [115, 116], an algorithm that maps circuits to an FPGA consisting of two different-sized lookup tables was described. This algorithm does not take into account lookup tables with more than one output, nor memories whose width can be traded for depth. The mapping of logic to large memory arrays was studied in [117]; the envisaged application in that work allowed for sequential function evaluation, as might be common in a circuit emulation system. When mapping logic to embedded memory arrays in an FPGA, we likely want to be able to evaluate all functions simultaneously. Thus, a new mapping algorithm is required.

8.2.3 Circuit Analysis Studies

It is difficult to propose suitable FPGA architectures or algorithms without a firm grasp of the nature of the circuits or systems the device is to implement. The circuit analysis (and accompanying generator) described in Chapter 3 is a step in this direction.

One important extension to this would be to study the interaction between clusters. Currently, the generator constructs clusters independently and glues them together. This was sufficient for this study, as it was the interaction between the memory and logic that was of concern, but future architectural experiments may need a more accurate model of how these clusters interact.

The current model does not distinguish between different types of logic subcircuits. Intuitively, the data pins of memories in many circuits are connected to datapath logic, while the address lines might be driven by a counter with control logic. The theoretical modeling of logic circuits is the focus of [35]; combining the model presented in this dissertation with that from [35] would likely result in a useful and accurate picture of the large digital systems that will be implemented on next-generation FPGAs.

8.2.4 Long-Term Research

Memory is only one way in which large systems differ from the small logic circuits that have been targeted to FPGAs in the past. These large systems often contain datapath portions; dedicated datapath FPGA resources might become important. Options include providing datapath-oriented routing structures in part of the chip [118], or including an embedded arithmetic/logic unit (ALU) that is flexible enough to implement a wide variety of datapath circuits. In the very long term, even larger embedded blocks such as digital signal processing components or even simple CPUs are likely. A thorough understanding of the issues involved with these heterogeneous devices will be crucial as FPGAs grow to encompass even larger systems.

References

[1] T. Ngai, "An SRAM-programmable field-reconfigurable memory," Master's thesis, University of Toronto, 1994.

[2] R. T. Masumoto, "Configurable on-chip RAM incorporated into high speed logic array," in *Proceedings of the IEEE 1985 Custom Integrated Circuits Conference*, pp. 240–243, May 1985.

[3] H. Satoh, T. Nishimura, M. Tatsuki, A. Ohba, S. Hine, and Y. Kuramitsu, "A 209K-transistor ECL gate array with RAM," *IEEE Journal of Solid-State Circuits*, vol. 24, pp. 1275–1279, Oct. 1989.

[4] D. A. Luthi, A. Mogre, N. Ben-Efraim, and A. Gupta, "A single-chip concatenated FEC decoder," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pp. 13.2.1–13.2.4, May 1995.

[5] R. Wilson and B. Fuller, "IBM pushes its ASICs to 0.18um." Electronic Engineering Times, May 6 1996.

[6] M. Gold and R. Wilson, "TI enters 0.25um race." Electronic Engineering Times, May 27 1996.

[7] I. Agi, P. J. Hurst, and K. W. Current, "A 450 MOPS image backprojector and histogrammer," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 6.2.1–6.2.4, May 1992.

[8] R. Mason and K. Runtz, "VLSI neural network system based on reduced interchip connectivity," in *Proceedings of the Canadian Conference on Very Large Scale Integration*, pp. 7.69–7.74, Nov. 1993.

[9] G. Feygin, P. Chow, P. G. Gulak, J. Chappel, G. Goodes, O. Hall, A. Sayes, S. Singh, M. B. Smith, and S. Wilton, "A VLSI Implementation of a Cascade Viterbi Decoder with Traceback," in *Proceedings of the 1993 IEEE International Symposium on Circuits and Systems*, pp. 1945–1948, May 1993.

[10] K. Kaneko, T. Okamoto, M. Nakajima, Y. Nakakura, S. Gokita, J. Nishikawa, Y. Tanikawa, and H. Kadota, "A VLSI RISC with 20-MFLOPS peak, 64-bit floating point unit," *IEEE Journal of Solid-State Circuits*, vol. 24, pp. 1331–1339, October 1989.

[11] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of programmable gate arrays: The effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1217–1225, October 1990.

[12] Xilinx, Inc., *The Programmable Logic Data Book*, 1994.

[13] AT&T Microelectronics, *Data Sheet: Optimized Reconfigurable Cell Array (ORCA) Series Field-Programmable Gate Arrays*, March 1994.

[14] Altera Corporation, *Datasheet: FLEX 10K Embedded Programmable Logic Family*, July 1995.

[15] Actel Corporation, *Datasheet: 3200DX Field-Programmable Gate Arrays*, 1995.

[16] Y. Nishio, F. Murabayashi, I. Masuda, H. Maejima, S. Owaki, K. Yamazaki, and S. Kadono, "0.45ns 7K Hi-BiCMOS gate array with configurable 3-port 4.6K SRAM," in *Proceedings of the IEEE 1987 Custom Integrated Circuits Conference*, pp. 203–204, 1987.

[17] P. S. Bennett, R. P. Dixon, and F. Ormerod, "High performance BICMOS gate arrays with embedded configurable static memory," in *Proceedings of the IEEE 1987 Custom Integrated Circuits Conference*, pp. 195–198, 1987.

[18] Y. Sugo, M. Tanaka, Y. Mafune, T. Takeshima, S. Aihara, and K. Tanaka, "An ECL 2.8 ns 16K RAM with 1.2K logic gate array," in *Proceedings of the 1986 IEEE International Solid State Circuits Conference*, pp. 256–257, Feb. 1986.

[19] M. Kimoto, H. Shimizu, Y. Ito, K. Kohno, M. Ikeda, T. Deguchi, N. Fukuda, K. Ueda, S. Harada, and K. Kubota, "A 1.4ns/64kb RAM with 85ps/3680 logic gate array," in *Proceedings of the IEEE 1989 Custom Integrated Circuits Conference*, pp. 15.8.1–15.8.4, 1989.

[20] LSI Logic, *LCB500K Preliminary Design Manual*, November 1994.

[21] 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," in *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, pp. 31.2.1–31.2.7, 1990.

[22] Xilinx, Inc., *XC4000E: Field Programmable Gate Array Family, Preliminary Product Specifications*, September 1995.

[23] B. K. Britton, D. D. Hill, W. Oswald, N.-S. Woo, and S. Singh, "Optimized reconfigurable cell array architecture for high-performance field-programmable gate arrays," in *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, pp. 7.2.1–7.2.5, March 1993.

[24] T. Ngai, S. Singh, B. Britton, W. Leung, H. Nguyen, G. Powell, R. Albu, W. Andrews, J. He, and C. Spivak, "A new generation of ORCA FPGA with enhanced features and performance," in *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, pp. 247–250, May 1996.

[25] Crosspoint Solutions, Inc., *CP20K Field Programmable Gate Arrays*, November 1992.

[26] D. Marple and L. Cooke, "An MPGA compatible FPGA architecture," in *Proceedings of the ACM/SIGDA International Workshop on Field-Programmable Gate Arrays*, pp. 39–42, Feb. 1992.

[27] S. Reddy, R. Cliff, D. Jefferson, C. Lane, C. Sung, B. Wang, J. Huang, W. Chang, T. Cope, C. McClintock, W. Leong, B. Ahanin, and J. Turner, "A high density embedded array programmable logic architecture," in *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, pp. 251–254, May 1996.

[28] Lattice Semiconductor Corporation, *Datasheet: ispLSI and pLSI 6192 High Density Programmable Logic with Dedicated Memory and Register/Counter Modules*, July 1996.

[29] D. E. Smith, "Intel's FLEXlogic FPGA architecture," in *Compcon Spring '93*, pp. 378–384, February 1993.

[30] K. Kawana, H. Keida, M. Sakamoto, K. Shibata, and I. Moriyama, "An efficient logic block interconnect architecture for user-reprogrammable gate array," in *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, pp. 31.3.1–31.3.4, May 1990.

[31] Plus Logic, *FPSL5110 Product Brief*, October 1989.

[32] Chip Express, *Application Note: CX2000 Reconfigurable Single & Dual Port SRAM/ROM Macro*, May 1996.

[33] Chip Express, *Product Description: 0.6u High Performance Fast Turn ASIC*, May 1996.

[34] T. Ngai, J. Rose, and S. J. E. Wilton, "An SRAM-Programmable field-configurable memory," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pp. 499–502, May 1995.

[35] M. Hutton, J. Grossman, J. Rose, and D. Corneil, "Characterization and parameterized random generation of digital circuits," in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 94–99, June 1996.

[36] J. Darnauer and W. W. Dai, "A method for generating random circuits and its application to routability measurement," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 66–72, Feb. 1996.

[37] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 277–282, March 1991.

[38] S. Brown, G. Lemieux, and M. Khellah, "Segmented routing for speed-performance and routability in field-programmable gate arrays," *Journal of VLSI Design*, vol. 4, no. 4, pp. 275–291, 1996.

[39] D. Hill and N.-S. Woo, "The benefits of flexibility in lookup table-based FPGA's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 349–353, February 1993.

[40] J. L. Kouloheris and A. E. Gamal, "PLA-based FPGA area versus cell granularity," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 4.3.1–4.3.4, 1992.

[41] M. Toyokura, M. Saishi, S. Kurohmaru, K. Yamauchi, H. Imanishi, T. Ougi, A. Watabe, Y. Matsumoto, T. Morishige, H. Kodama, E. Miyagoshi, K. Okamoto, M. Gion, T. Minemaru, A. Ohtani, T. Araki, K. Aono, H. Takeno, T. Akiyama, and B. Wilson, "A video DSP with a macroblock-level-pipeline and and a SIMD type vector-pipeline architecture for MPEG2 CODEC," in *Proceedings of the 1994 IEEE International Solid-State Circuits Conference*, pp. 74–75, Feb. 1994.

[42] S. Ti and C. Stearns, "A 200 MFlop CMOS transformation processor," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 6.1.1–6.1.4, May 1992.

[43] C. Benz, M. Gowan, and K. Springer, "An error-correcting encoder and decoder for a 1 Gbit/s fiber optic link," in *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference*, pp. 7.1.1–7.1.4, May 1991.

[44] P. Tong, "A 40-Mhz encoder-decoder chip generated by a reed-solomon code compiler," in *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, pp. 13.5.1–13.5.4, May 1990.

[45] J. L. Hennessy and D. A. Patterson, *Computer Achitecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[46] R. DeMara and D. Moldovan, "The SNAP-1 parallel AI prototype," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 2–11, May 1991.

[47] A. Curiger, H. Bonnenberg, R. Zimmerman, N. Felber, H. Kaeslin, and W. Fichtner, "Vinci: VLSI implementation of the new secret-key block cipher idea," in *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, pp. 15.5.1–15.5.4, May 1993.

[48] K. Fisher, P. Bednarz, J. Kouloheris, B. Fowler, J. Cioffi, and A. ElGamal, "A 54 Mhz BiCMOS digital equalizer for magnetic disk drives," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 19.3.1–19.3.4, May 1992.

[49] S. J. E. Wilton and Z. G. Vranesic, "Architectural Support for Block Transfers in a Shared Memory Multiprocessor," in *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pp. 51–54, Dec. 1993.

[50] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1895–1904, December 1992.

[51] L. K. Tan and H. Samueli, "A 200-Mhz quadrature digital synthesizer/mixer in 0.8um CMOS," in *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference*, pp. 4.4.1–4.4.4, May 1994.

[52] M. Matsui, H. Hara, K. Seta, Y. Uetani, L.-S. Kim, T. Nagamatsu, T. Shimazawa, S. Mita, G. Otomo, T. Oto, Y. Watanabe, F. Sano, A. Chiba, K. Matsuda, and T. Sakurai, "200 MHz video compression macrocells using low-swing differential logic," in *Proceedings of the 1994 IEEE International Solid-State Circuits Conference*, pp. 76–77, Feb. 1994.

[53] S. Molloy, B. Schoner, A. Madisetti, and R. Jain, "An 80k-transistor configurable 25MPixels/s video-compression processor unit," in *Proceedings of the 1994 IEEE International Solid-State Circuits Conference*, pp. 78–79, Feb. 1994.

[54] T. Karema, T. Husu, T. Saramaki, and H. Tenhunen, "A filter processor for interpolation and decimation," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 19.2.1–19.2.4, May 1992.

[55] M. Kuczynski, W. Lao, A. Dong, B. Wong, H. Nicholas, B. Itri, and H. Samueli, "A 1Mb/s digital subscriber line transceiver signal processor," in *Proceedings of the 1993 IEEE International Solid-State Circuits Conference*, pp. 26–27, Feb. 1993.

[56] N. Sollenberger, "An experimental TDMA modulation/demodulation CMOS VLSI chip-set," in *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference*, pp. 7.5.1–7.5.4, May 1991.

[57] E. Vanzieleghem, L. Dartios, J. Wenin, A. Vanwelsenaers, and D. Rabaey, "A single-chip GSM vocoder," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 10.3.1–10.3.4, May 1992.

[58] Y. Kondo, Y. Koshiba, Y. Arima, M. Murasaki, T. Yamada, H. Amishiro, H. Shinohara, and H. Mori, "A 1.2GLOPS neural network chip exhibiting fast convergence," in *Proceedings of the 1994 IEEE International Solid-State Circuits Conference*, pp. 218–219, Feb. 1994.

[59] K. Ueda, T. Sugimura, M. Okamoto, S. Marui, T. Ishikawa, and M. Sakakihara, "A 16b lower-power-consumption digital signal processor," in *Proceedings of the 1993 IEEE International Solid-State Circuits Conference*, pp. 28–29, Feb. 1993.

[60] S. Yang, "Logic synthesis and optimization benchmarks," tech. rep., Microelectronics Center of North Carolina, 1991.

[61] E. Sentovich, "SIS: A system for sequential circuit analysis," Tech. Rep. UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, May 1992.

[62] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 48–53, November 1992.

[63] R. X. Nijssen and J. A. Jess, "Two-dimensional datapath extraction," in *Proceedings of the 5th ACM/SIGDA Physical Design Workshop*, pp. 111–117, April 1996.

[64] J. M. Arnold and D. A. Buell, "Splash 2," in *Proceedings of the 4th Annual ACM Symposium on Parallel and Distributed Algorithms and Architectures*, pp. 316–324, 1992.

[65] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: A Performance Assessment," in *Research on Integrated Systems: Proceedings of the 1993 Symposium*, MIT Press, 1993.

[66] D. E. van den Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "Anyboard: An FPGA-based, reconfigurable system," *IEEE Design and Test of Computers*, pp. 21–30, September 1992.

[67] Altera Corporation, *Reconfigurable Interconnect Peripheral Processor (RIPP10) Users Manual*, May 1994.

[68] D. Galloway, D. Karchmer, P. Chow, D. Lewis, and J. Rose, "The Transmogrifier: the University of Toronto field-programmable system," in *Proceedings of the Canadian Workshop on Field-Programmable Devices*, June 1994.

[69] D. Karchmer and J. Rose, "Definition and solution of the memory packing problem for field-programmable systems," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 20–26, 1994.

[70] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Architecture of centralized field-configurable memory," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 97–103, 1995.

[71] R. J. Francis, *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.

[72] E. Kreyszig, *Advanced Engineering Mathematics*. John Wiley and Sons, Inc., 1983.

[73] N. Togawa, M. Sato, and T. Ohtsuki, "Maple: A simultaneous technology mapping, placement, and global routing algorithm for field-programmable gate arrays," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 156–163, 1994.

[74] S. K. Nag and R. A. Rutenbar, "Performance-driven simulataneous place and route for row-based FPGAs," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 301–307, 1994.

[75] M. J. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins, "Performance-oriented placement and routing for field-programmable gate arrays," in *Proceedings of the European Design Automation Conference*, Sept. 1995.

[76] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 98–106, Feb. 1991.

[77] N. P. Jouppi and S. J. E. Wilton, "Tradeoffs in Two-Level On-Chip Caching," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[78] S. J. E. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, May 1996.

[79] S. J. E. Wilton and N. P. Jouppi, "An enhanced access and cycle time model for on-chip caches," Tech. Rep. 93/5, Digital Equipment Corporation Western Research Lab, 1993.

[80] M. A. Horowitz, "Timing models for MOS circuits," Tech. Rep. Technical Report SEL83-003, Integrated Circuits Laboratory, Stanford University, 1983.

[81] W. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, vol. 19, pp. 55–63, Jan. 1948.

[82] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory/logic interconnect flexibility in FPGAs with large embedded memory arrays," in *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, pp. 144–147, May 1996.

[83] S. D. Brown, *Routing Algorithms and Architectures for Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.

[84] G. G. Lemieux and S. D. Brown, "A detailed router for allocating wire segments in field-programmable gate arrays," in *Proceedings of the ACM Physical Design Workshop*, April 1993.

[85] Y.-W. Chang, D. Wong, and C. Wong, "Universal switch modules for FPGA design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 80–101, January 1996.

[86] Y.-W. Chang, D. Wong, and C. Wong, "Universal switch module design for symmetric-array-based FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 80–86, Feb. 1996.

[87] Y.-L. Wu and D. Chang, "On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 362–366, 1994.

[88] Y.-L. Wu and M. Marek-Sadowska, "An efficient router for 2-D field programmable gate arrays," in *Proceedings of the European Design Automation Conference*, pp. 412–416, 1994.

[89] S. Raman, C. Liu, and L. Jones, "Timing-based placement for an FPGA design environment," in *More FPGAs* (W. Moore and W. Luk, eds.), pp. 110–119, Abingdon EE&CS Books, 1993.

[90] C. Ebeling, L. McMurchie, S. A. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 3, pp. 473–482, December 1995.

[91] S. Kirkpatrick, J. C.D Gellat, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[92] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE Journal of Solid-State Circuits*, vol. SC-20, pp. 510–522, April 1985.

[93] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A new standard cell placement and global routing package," pp. 432–439, 1986.

[94] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An efficient general cooling schedule for simulated annealing," in *Proceedings of the IEEE Conference on Computer-Aided Design*, pp. 381–384, 1986.

[95] J. Lam and J.-M. Delosme, "Performance of a new annealing schedule," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 306–311, 1988.

[96] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 387–396, March 1988.

[97] W. Swartz and C. Sechen, "New algorithms for the placement and routing of macro cells," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 336–339, 1990.

[98] V. Betz and J. Rose, "Directional bias and non-uniformity in FPGA global routing architectures," in *to appear in International Conference on Computer-Aided Design*, November 1996.

[99] R. Nair, "A simple yet effective technique for global wiring," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 165–172, March 1987.

[100] J. Rose, "Parallel global routing for standard cells," *IEEE Transactions on Computer-Aided Design*, vol. 9, pp. 1085–1095, Oct. 1990.

[101] Y.-W. Chang, S. Thakur, K. Zhu, and D. Wong, "A new global routing algorithm for FPGAs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 356–361, 1994.

[102] S. Brown, J. Rose, and Z. G. Vranesic, "A detailed router for field-programmable gate arrays," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 620–628, May 1992.

[103] Y.-L. Wu and M. Marek-Sadowska, "Orthogonal greedy coupling - a new optimization approach to 2-D FPGA routing," in *Proceedings of the ACM/IEEE Design Automation Conference*, 1995.

[104] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 111–117, 1995.

[105] M. J. Alexander and G. Robins, "New performance-driven FPGA routing algorithms," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 562–567, June 1995.

[106] Y.-S. Lee and A. C.-H. Wu, "A performance and routability driven router for FPGAs considering path delays," in *Proceedings of the ACM/IEEE Design Automation Conference*, June 1995.

[107] C. Lee, "An algorithm for path connection and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[108] E. Moore, "Shortest path through a maze," in *Proceedings of the International Symposium on Switching Circuits*, pp. 285–292, Harvard University Press, 1959.

[109] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[110] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley and Sons, 1990.

[111] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "ALTOR: an automatic standard cell layout program," in *Proceedings of the 1985 Canadian Conference on Very Large Scale Integration*, pp. 169–173, November 1985.

[112] I. Kostarnov, J. Osmany, and C. Solomon, "Riley DPMatch - an exercise in algorithm mapping to hardware," Tech. Rep. ACD 95-09-02, Hewlett Packard Laboratories, Nov. 1995.

[113] K. K. Chung, *Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections.* PhD thesis, University of Toronto, 1994.

[114] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory-to-memory connection structures in FPGAs with embedded memory arrays," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1997.

[115] J. He and J. Rose, "Technology mapping for heterogeneous FPGAs," in *Proceedings of the ACM International Workshop on Field Programmable Gate Arrays*, Feb 1994.

[116] J. He, "Technology mapping and architecture of heterogeneous field-programmable gate arrays," Master's thesis, University of Toronto, 1994.

[117] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," in *Proceedings of the International Workshop on Logic Synthesis*, May 1995.

[118] D. Cherepacha and D. Lewis, "A datapath oriented architecture for FPGAs," in *Proceedings of the ACM/SIGDA International Workshop on Field-Programmable Gate Arrays*, Feb. 1994.