Efficient Interconnection Network Components for Programmable Logic Devices

by

Guy Gerard Frederick Lemieux

A thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy
Edward S. Rogers Senior Graduate Department of Electrical and Computer Engineering
University of Toronto

Efficient Interconnection Network Components for Programmable Logic Devices

Doctor of Philosophy 2003

Guy Gerard Frederick Lemieux

Edward S. Rogers Senior Graduate Department
of Electrical and Computer Engineering

University of Toronto

Abstract

The complexity of digital logic systems has increased steadily and rapidly for the last several decades due to a steady trend in technology scaling. As current manufacturing technology reaches the deep-submicron level, an increasing amount of low-level design effort is required to create a working design. This further increases the cost and complexity of these designs.

One way to separate digital systems design from the problems of deep sub-micron design is to use programmable logic device (PLD) technology. This provides a clean interface, allowing systems designers to stop at the RTL level while physical design issues are solved by the PLD designers. Although this is a very convenient abstraction, it suffers from significant inefficiencies in area and delay that make the approach unsuitable for many large systems.

This dissertation improves the area efficiency of PLDs by analysing the design of their largest components, the switch blocks and sparse crossbars found in the interconnection network. The approach taken is to modify their topological organisation, so that the routing network can be more highly utilised, as well as their circuit implementations, so they can be made smaller. Through transistor-level design, the delay of these circuit implementations is also reduced, particularly delay under fanout.

The improvements made to these network components can be applied to a variety of high-level PLD interconnect styles. In this dissertation, they are shown to improve area and delay in a mesh-style PLD architecture.

Acknowledgments

I would like to thank my supervisor, Dr. David Lewis, for his support, encouragement, and wisdom. He reminded me many times to look at things in different ways; hopefully, this has permanently rubbed off on me! His dedication to my research was also remarkable. Thank you.

The long arduous tasks embodied by this dissertation were mitigated in many ways by the support of family and friends. Many thanks are due to many people – there are far too many to list here, but you know who you are! Thank you for the many distractions, lively discussions, and encouragement throughout.

I would also like thank some friends in particular for their technical assistance and feedback on my work: Elias Ahmed, Su Jin Chun, Marcus van Ierssel, Paul Kundarewich, Kostas Pagiamtzis, Ajay Roopchansingh, and Andy Ye.

This dissertation is dedicated to the loving memory of my father and grandfather. They struggled hard to give me this opportunity. Thank you.

Contents

List of Figures

List of Tables

List of Symbols

B	switch or buffer size, $B = W_n/W_{minT}$ of the last stage
c	capacity of a crossbar (number of guaranteed-routable inputs)
BLE	basic logic element, containing a k-input LUT and bypassable flip-flop
CLB	clustered logic block
C block	connection block, connects CLB pins to the routing channels
S block	switch block, connects horizontal and vertical routing channels
F_c	fraction of switching points connected to (clustered) logic block inputs
$F_{c_{in}}$	fraction of switching points connected to LUT/BLE inputs from cluster inputs
$F_{c_{fb}}$	fraction of switching points connected to LUT/BLE inputs from cluster feedback
$F_{c_{out}}$	fraction of switching points connected to LUT/BLE outputs
F_s	switch block flexibility, the number of other wires connected at this switch block
FPGA	field-programmable gate array, aka PLD
k	number of inputs to a LUT
I	number of inputs to a clustered logic block
I_{spare}	number of spare inputs to a clustered logic block, used only for routing (not packing)
L_{wire}	logical wire length, measured by the number of logic blocks (CLB tiles) it spans
L_{min}	minimum transistor length (in μm)
L_n, L_p	NMOS, PMOS transistor lengths (in μm)
L_{tile}	physical length of a logic block tile, measured in μm
LUT	lookup table, used to implement logic functions

M	transistor width as a multiple of W_{minT}
m	number of crossbar outputs
N	number of LUTs/BLEs in a clustered logic block
n	number of crossbar inputs
PLD	programmable logic device
p	number of switches in an $n \times m$ crossbar
T	unit of area occupied by one minimum-sized contactable transistor, including spacing
W	number of tracks per routing channel (channel width)
W_n, W_p	NMOS, PMOS transistor widths (in μm)
W_{min}	minimum tracks required to route
W_{minT}	minimum *contactable* transistor width
V_t	transistor threshold voltage

Chapter 1

Introduction

1.1 Motivation

The rapid advancement of semiconductor technology has required the concurrent advancement of the digital system design process. Early integrated circuits such as the Intel 4004 processor were completely hand-designed, including the layout artwork. This was a reasonable effort for a 2,300-transistor device built in 1971 with a 10μm technology process [1]. In contrast, the latest Intel Itanium 2 processor, released in July 2002, contains 220 million transistors in a 0.18μm process [1]. Designing such a large device requires hundreds of engineers who rely upon sophisticated CAD tools to manage the complexity. For the last five to ten years, each new Intel processor has been the pinnacle of design, setting the standard for other devices to follow. However, other new devices are reaching a similar level complexity. For example, the ATI Technologies RADEON 9700 graphics processor [2], also released in July 2002, is reported to contain 107 million transistors in a 0.15μm process [3]. There is no doubt that increasingly larger devices will continue to be designed, so the demand for tools and design methodology advancement will continue.

Coupled with the increase in design complexity is a development cost trend that is spiralling upwards. Future devices will contain more transistors with ever-smaller feature sizes. Designers and CAD tools must not only cope with an increasing transistor count, but also with an increasing number of second-order effects. Solving the problems of IR supply-voltage drops, electromigration, signal noise and integrity, and increasing power

all translate into greater design effort and cost. For example, one way to reduce power without sacrificing performance is to use transistors with lower threshold voltages on the speed-critical portion only [4]. This demonstrates that not only are designs larger and more complex, but they require more detailed design and analysis than ever before. The impact on design team size, tool sophistication and development costs reflect this as well.

Design costs cannot continue to grow unabated without a shift in the design process toward a simpler, more manageable methodology. Programmable logic device (PLD) technology is one solution for reducing design costs. When using PLDs, design can stop at the RTL netlist level because many of the difficult, low-level physical design issues have already been addressed during the physical design of the PLD itself. This saves considerable effort in physical design and verification, not to mention a reduction in tool cost. For example, Nvidia Corporation spent $160 million for CAD tools and $40 million on emulation systems to design the GeForce 4 graphics processor, a custom chip with 63 million transistors [5]. In addition to reducing these types of costs, the PLD abstraction also improves time-to-market, a critical factor for establishing market acceptance and profitability.

Despite their design cost advantage, PLDs impose large area overheads when they are compared with custom silicon alternatives. To illustrate the magnitude of this problem, the area profile for a PLD architecture studied in this dissertation is given in Table 1.1.[1] The top row shows the portion of area used for logic functions, while the remaining rows show the portion of area used for various routing functions. From this table, it can be seen that only 8–16% of the area is used to directly implement logic. This inefficiency makes PLDs very unattractive in per-unit cost. PLDs also suffer from large delay and power overheads. The inability to meet timing requirements, power budgets, or logic capacity constraints make it technically infeasible to use PLDs in the most demanding applications.

[1] To compute the results in Table 1.1, twenty different benchmark circuits are placed and routed using the smallest PLD possible (with 20% additional routing tracks). The architecture is a mesh-based PLD using a cluster with six 4-input LUTs. Half of the routing tracks are unbuffered and the other half are buffered only at *alternate* switching points. This architecture contains very few buffers, yet routing area still dominates. The average area is determined by a geometric average across the benchmark circuits; the range is similarly determined by the minimum or maximum portion across the different circuits. The area data are compiled using the tools and methodology described later in Chapter 3.

Resource	Details	Proportion of Area	
		Range	Average
Logic	flip-flops, lookup tables, lookup table input buffers	8–16%	12%
Routing	lookup table output buffers and switches	8–10%	9%
	lookup table input multiplexers	18–36%	27%
	cluster input multiplexers	11–13%	12%
	cluster input buffers (track buffers)	7–11%	9%
	routing switches	23–39%	31%
	total routing	84–92%	88%

Table 1.1: Area profile of a mesh-based PLD.

There is a considerable demand for PLDs which use less area and have lower delay. One general method used to make PLDs more efficient is to search for improvements to the numerous algorithmic steps which map a logic circuit into a PLD. Improvements to the logic synthesis step, for example, can reduce the amount and depth of logic needed. Also, improvements to the partitioning, placement, and clustering steps can reduce the amount of interconnect needed and reduce delay by encouraging the use of shorter or more local connections. Similarly, improvements to the routing step can better map critical delay paths to faster connections. Defining a PLD architecture is the challenge of fixing the logic and routing resources so that these algorithms produce the most efficient results possible. Since both the algorithm and architecture can be simultaneously defined, there is a significant amount of interaction which can influence the final result. The scope of this design problem has motivated a considerable amount of research to improve the efficiency of PLDs.

The primary goal of this research is to improve the area efficiency of PLDs by examining the building blocks used in the routing network. The approach taken is to examine the largest portions of the routing network and make micro-architectural or circuit-level optimisations which can reduce area or delay while remaining independent of the CAD algorithms being used and the higher-level architectural organisation.

Two building blocks commonly used in PLD architectures are switch blocks and sparse

crossbars. These blocks also form the largest parts of the PLD. From Table 1.1 it can be seen that the largest component is the routing switches (31%) which make up the switch blocks. The next two largest routing components are the lookup table input multiplexers (27%) and cluster input multiplexers (12%), both of which can be implemented as crossbars or sparse crossbars. These three components are the focus of this work.

The design of switch blocks and sparse crossbars is important to the creation of area-efficient interconnect for PLDs. Chapters 4 through 7 explore ways to make these components use less area. In particular, Chapter 4 examines the topological organisation of a sparse crossbar to make it as routable as possible. Chapter 5 uses these sparse crossbars within a cluster of lookup tables to reduce the lookup table input multiplexer area in half. Chapter 6 suggests two circuit-level optimisations to reduce switch area: a) by using a new switch design which is smaller than those used previously and has lower delay under fanout, and b) by selectively replacing some buffers with pass transistors, which are smaller, so that delay is not increased. To further reduce switch area, Chapter 7 explores new topological organisations of switch blocks.

It should be noted that the topological changes considered in Chapters 4 and 7 do not directly reduce area themselves. Instead, the improvements to topology lead to a network which can reach higher levels of utilisation by the CAD algorithms, hence require fewer routing tracks. This, in turn, reduces the routing area needed.

A secondary goal of this dissertation is to develop techniques for designing PLD routing networks of arbitrary size. Since each circuit presents a unique set of requirements to a PLD, it can be advantageous to develop a different routing network for each one. For example, some circuits may have many high-fanout signals which benefit from long wires with buffers, while other circuits may be dominated by very regular connections to close neighbours which benefit from short wires with pass transistors.

One way to characterise interconnect demand is with Rent's rule [6], an empirical observation which predicts that the amount of wiring needed as a circuit increases in size is related to $P = k \cdot B^R$. In this formula, P is the number of wiring pins, k is a constant, B is the number of blocks, and R is a parameter known as the Rent exponent. The value of R is known to vary depending on the interconnect demand of the circuit, but it is typically in

the range of 0.5 to 0.75 [7]. A PLD routing network that is designed to suit most circuits must grow at a rate close to $R = 0.75$, yet this is considerably overdesigned for the circuits which only require $R = 0.5$. This suggests there is a need to customise the routing network for different classes of circuits.

The idea of customisation has been considered before, but recently the focus has shifted towards the routing architecture. For example, early work by Betz and Rose introduced the notion of creating a "family" of different architectures, with each member designed to better suit a different type of circuit [8]. Betz and Rose demonstrate the usefulness of the approach by using members with different logic blocks. In comparison, more recent work [9, 10] focuses on customising the routing network for a few pre-specified circuits rather than a general class of circuits.

PLD vendors currently produce PLDs with varying amounts of logic, I/O pins, and different speed grades, yet they do not offer PLDs with different amounts of interconnect at a fixed logic capacity. This is surprising since the interconnect consumes nearly 90% of the chip area! Some reasons for not offering a variety of interconnect sizes are inventory control, the impact of marketing and sales of seemingly inferior or unroutable devices, and the large amount of engineering effort that is required to develop a single device. This last reason can be partially addressed by further automating the PLD interconnect design stage with some of the techniques developed in this dissertation.

Another strong motivation for designing PLD networks of arbitrary size is provided by the trend of implementing system-on-a-chip (SoC) and platform-based designs. These designs would benefit from embedding programmable logic cores on-chip by combining multiple implementations (*i.e.*, different feature sets) into one piece of silicon. A number of embedded PLD core products are being developed by companies such as Actel [11], eASIC [12], Leopard Logic [13] and a Xilinx/IBM partnership [14].

There are numerous other reasons to prefer using a embedded PLD core over regular custom logic in SoC or platform applications. First, the use of embedded PLD cores retains all of the traditional benefits of PLDs, such as making changes to the circuit late in the manufacturing cycle to correct design errors or to comply with emerging standards. It also raises the new possibility of mapping around or tolerating some types of manufacturing

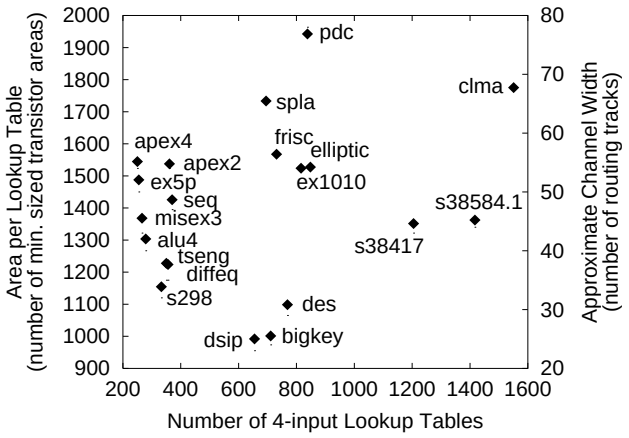


Figure 1.1: Range of area requirements for different circuits.

defects. Lastly, it increases the number of markets for the device, since it can be more easily adapted to work in different customer environments or to include entirely new features.

One way to improve the efficiency of embedded PLD cores is to customise the core for the types of circuits it will be implementing. To see this, Figure 1.1 plots the area per logic cell required by each of the twenty different benchmark circuits used in this dissertation. The left *y*-axis gives the area per logic cell[2] for each circuit. This is plotted against the number of logic cells or lookup tables of the circuit on the *x*-axis. The right *y*-axis shows an approximation to the channel width for each circuit (the approximation error is less than 3 tracks). For these small circuits, the routing demands are not strongly correlated to the circuit size. If an embedded PLD core is only required to implement circuits similar to `dsip` or `bigkey`, for example, very little routing is required and considerable area can be saved. Another interpretation of the results in this figure is that a well-designed PLD of fixed logic capacity may contain twice as many transistors and four times as many routing tracks than what is required by a circuit. This further supports the notion that PLDs should be offered with a variety of wiring resources.

Currently, the design of a custom PLD core requires the specialized tools and the expertise of a PLD architect. This makes it impractical to apply this methodology for each embedded application. However, a future goal in extending this dissertation is to encapsulate this knowledge into a set of tools that can be used by an embedded system engineer to

[2]Area per logic cell includes a 4-input lookup table, a register, and all associated routing switches.

design a fully custom PLD core. For example, the work in Chapter 4 describes a general method for designing routable sparse crossbars of any size with any number of switches. This represents a significant advancement since all prior work places some restriction on the number of inputs, outputs, or switches. Although a significant amount of future work is required to construct a complete PLD design tool, the results of this dissertation can be used as a starting point.

1.2 Contributions

The contributions made in this dissertation are summarised in the following paragraphs.

Chapter 4 presents a new way of statistically measuring the routability of a sparse crossbar. The routability measurement is based on standard Monte Carlo and network flow techniques, but the application is novel.[3]

This chapter establishes that Hall's theorem [16], a well-known result found in set and graph theory, provides the insight needed to understand the relationship between the routability of a sparse crossbar and the precise pattern of switches used. It suggests that a good sparse crossbar design algorithm needs to connect every subset of inputs to as many outputs as possible. Expander graphs have this property (their *expansion* property), but there are few known construction techniques [17, 18]. In addition, these constructions are impractical because they are too restrictive in the number of inputs, outputs, or switches.

This chapter also presents a greedy heuristic technique for designing sparse crossbar switch placements to maximise routability. The heuristic is the first known general method for generating sparse crossbar switch placements with good routability for an arbitrary number of inputs, outputs, and switches.

The analysis of a few of sparse crossbar design examples leads to the observation that it is better to intentionally under-utilise the output capacity. Having additional outputs seems wasteful, but the design examples show that this technique saves 10–40%

[3] A similar technique has been used in other work to design switch blocks [15], but not sparse crossbars.

of the switches (or transistors) while keeping the routability level constant (for example, 95% or better).

Chapter 5 shows that a 10–18% area savings can be realised in PLDs by using a sparse crossbar inside logic clusters (rather than a full crossbar). The use of actual routing experiments demonstrates the viability of the approach.

The experiments also show that area savings increases with lookup table size, causing 6-input lookup table architectures to use less area than ones with the traditional 4-input lookup table. Since 6-input lookup tables result in lower critical-path delay, a significant 22% area·delay reduction is observed. This presents a compelling case to re-examine the benefits of 6-input lookup tables using larger and more realistic benchmark circuits, such as those used internally by PLD vendors.

Chapter 6 proposes new routing switch circuits that virtually eliminate delay increases caused by fanout at the switch. These new switches are also slightly smaller in area. They lead to an area reduction of 2% and a delay reduction of 7%.

This chapter shows that it is inappropriate to scale routing switch sizes larger as wire load increases beyond a certain point. This counters a practice used by Betz *et al* in [19], where switches are doubled in size whenever wire length doubles.

This chapter finds that the careful selection and substitution of some buffers with pass transistors is a viable way to reduce area by 8–13% while keeping delay approximately constant. The buffer-substitution strategies employed permit a signal to alternate between buffers and pass transistors. Although this improves the delay of some long connections, it does not improve critical-path delay due to fanout effects.

Chapter 7 presents a new way to design switch blocks by maximising diversity, or the number of paths that reach different routing tracks. This is the first switch block design technique to directly consider the routing fabric in a global, not just local, perspective at the initial interconnect design stages. The switch blocks produced are demonstrated to have measurably superior diversity and achieve similar routability to the best known switch blocks.

1.3 Summary

The continuing trend of designing larger chips requires simpler design methodologies and abstractions. PLD technology presents one simple methodology which allows designers to stop at the RTL level, but it suffers from significant area and delay overheads. The largest portion of area comes from the interconnect, consisting primarily of switch blocks and sparse (or full) crossbars. This dissertation studies ways to efficiently design and implement these interconnect components. Most of this design process can be automated and used in tools to generate custom PLDs for SoC or embedded applications. Automatically designing custom PLDs is a promising new research area which may help us reach the ultimate objective of providing a simple digital design methodology.

1.4 Outline

The remainder of this dissertation is organised as follows. Chapter 2 describes previous work which has been done in the areas of PLD architecture and routing network design. Chapter 3 explains the methodology, tools, and metrics used for many of the experiments. Chapter 4 presents a technique for designing routable sparse crossbars with an arbitrary number of inputs, outputs, and switches. Chapter 5 uses these sparse crossbars as a replacement for full crossbars to reduce area within clusters of lookup tables. Chapter 6 presents three new circuit designs which are based on fan-in rather than fan-out. It also explores the idea of replacing buffered interconnect with a combination of buffers and pass transistors to reduce both area and delay. Chapter 7 defines new types of switch blocks which maximise diversity, defined as the number of different tracks that can be reached near a destination. Chapter 8 provides a final summary and concludes the work. Rather than listing them at the end, comments on the potential for future work have been embedded at the end of each of the technical chapters, 4 through 7.

Chapter 2

Background

The investigation of efficient routing networks for programmable logic spreads across many areas. This chapter surveys the most pertinent work related to PLD interconnect design, with particular emphasis being placed on mesh networks. Then, work in the area of switching networks for telephony and graph theory is presented to provide a comparative context.

2.1 PLD Routing Architectures

The routing network is responsible for connecting the primary inputs, the logic cells, and the primary outputs together. Historically, PLD routing architectures have been broadly categorized into three types of networks: row, hierarchical, and array-based or mesh [20]. A mesh-based PLD which uses lookup tables to implement logic is often referred to as an FPGA, but the more general term of PLD will be used throughout this dissertation to emphasize that a variety of interconnect styles and logic elements can be used. Another type, the complete network, is the basis for PALs, one of the simplest PLDs. These four networks are illustrated in Figure 2.1 and are further described below.

Commentary

In the past, commercial PLDs have often been distinguished according to this taxonomy. However, modern architectures contain many optimisations and do not strictly adhere to

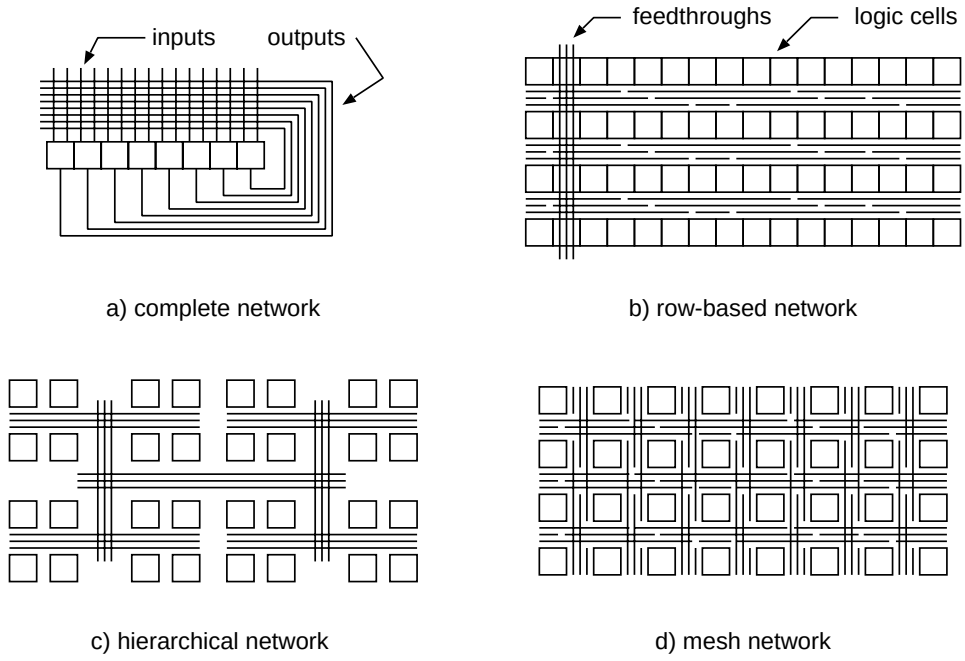


Figure 2.1: Common PLD routing network types.

the ideals of each definition. For example, the Altera FLEX6000 architecture contains direct connections between the LABs, their lowest level of hierarchy [21, 22]. As well, architecture details are sometimes obscured to protect intellectual property. For example, the Actel ProASIC device is said to be “segmented hierarchical,” but the explanation of what this means is not clear [23, 11].

2.1.1 Complete Networks

A complete network is formed when all of the primary inputs and logic cell outputs are provided as inputs to all of the logic cells. This form is inefficient, since the network grows quadratically as the number of logic cells increases. Hence, it is only practical for a small number of logic cells.

These types of networks are commonly seen in PAL devices, where the programmable-AND plane plays the dual role of forming the interconnections as well as the logic functions.

2.1.2 Row Networks

Row-based networks are organised as horizontal rows of logic cells separated by routing channels. Horizontal wires of various lengths are placed end-to-end in a track, with multiple tracks comprising a channel. Switches within a track may connect several wires so they behave as one longer one. Additional switches connect track wires to either logic cell inputs or outputs. Connections between wiring tracks of different rows are made by vertical wires known as *feedthroughs*. Each feedthrough can be connected with any of the row wires. Since feedthrough connections can always be made, the routing problem is reduced to a one-dimensional *channel routing* problem [24, 25, 26].

This row-based organisation is very similar to the ASIC standard cell design methodology. In this approach, logic cells or standard cells of similar height are placed end-to-end in rows. The routing problem focuses on placing custom lengths of horizontal metal in tracks between two rows to connect the input and output pins of the cells. Vertical connections are made by adding custom feedthrough wires across as many rows as needed. Sometimes, adjacent cells in a row must be spaced apart to accommodate the feedthroughs.

The Actel ACT 1, 2, and 3 series PLDs [27] are based on row networks. In these architectures, the feedthroughs span the entire length of the device. The large number of switches required by these architectures are made possible because the anti-fuse switch used to program these devices is quite small.

There are numerous studies on the routing algorithms, the distribution of wire segment lengths and connection structures necessary to make row-based PLDs efficient [25, 24, 28, 26, 29]. However, these networks are no longer used in modern, commercial PLDs. One likely reason for their shrinking market share arises from the limitations of the anti-fuse switch. It permits a device to be programmable only once, making it difficult to fully verify a device before selling it to a customer. Also, anti-fuse switches are based on a specially developed manufacturing process. This makes it more difficult for these devices to track the economy of technology scaling trends as quickly as SRAM-based devices.

2.1.3 Hierarchical Networks

A hierarchical network is formed by partitioning a group of logic cells into subgroups, usually of equal size, and providing a set of wires to connect these subgroups. This is applied recursively, once for each level in the hierarchy. Figure 2.2 shows a routing network containing three levels of hierarchy. A distinguishing feature of a hierarchical network is that two groups of wires at the same level in the hierarchy can only be connected together through wires in higher levels. Hence, connections between two different groups at the same level are forbidden, even if the two groups are in close physical proximity.

Hierarchical networks are commonly used in Complex PLDs, or CPLDs, by adding one level of hierarchy to a collection of PALs. For example, CPLDs such as the Altera MAX7000 [30], the Xilinx XC9500, XC7200, and XC7300 [31], the Lattice ispMACH[1] [32] and ispLSI [33] families are all based on this two-level hierarchy concept. Academic studies of hierarchical routing networks have also been done by Aggarwal and Lewis [34, 35], Chan and Lewis [36], and Lai and Wang [37].

Altera uses hierarchy in many of their PLDs, including the FLEX, APEX, and ACEX families [30]. These products are not strictly hierarchical, however, because wires at the top-most level (the column wires) can connect to each other through wires one level below (the row wires). As well, in some of the architectures such as the FLEX6000, direct connections can be made at the lowest level between neighbouring groups [21, 22].

2.1.4 Mesh Networks

A mesh network is based on a two-dimensional array of logic cells with routing channels located between the rows and columns. Typically, the wires in a channel are a mixture of lengths, from short (the length of one or two logic cells), to medium (four to eight logic cells), and long (half to full length of the die). A signal can be propagate along multiple wires in any direction, horizontally or vertically, by using the switches at the channel intersections. This organisation encourages connections between wires of adjacent groups, a feature which clearly distinguishes it from a hierarchy.

[1] Lattice acquired the ispMACH products through the acquisition of Vantis, a former subsidiary of AMD.

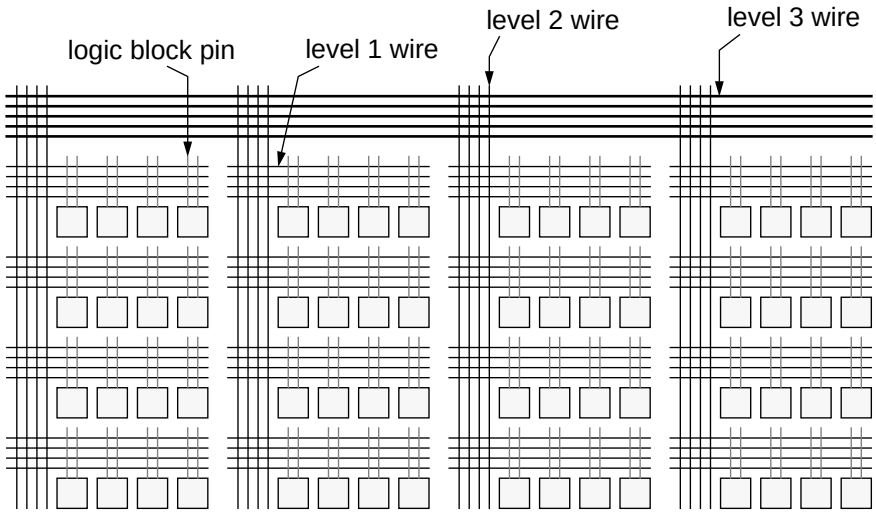


Figure 2.2: A hierarchical PLD routing network.

Mesh networks are commonly used in PLDs by Xilinx, including the XC4000, Spartan, and Virtex families. As well, the Lattice ORCA PLDs[2] and the latest Altera Stratix PLDs are mesh-based.

2.1.5 Selected Commercial PLDs

To gain a better understanding of the current state-of-the art, this section provides a review of the routing architecture used in the latest PLDs offered by the two largest PLD vendors, Altera and Xilinx.

Xilinx Virtex

The latest family of PLDs available from Xilinx is the Virtex series. The logic block architecture of the original Virtex, including the Virtex-E and Virtex-EM variations, is based 4-input lookup tables (4-LUTs). Four 4-LUTs are arranged in a group, called a clustered logic block (CLB) here. The largest device is constructed from an array of 104×156 CLBs. Between each row (or column) of CLBs is a horizontal (or vertical) routing channel. This architecture will be referred to as Virtex I. The more recent Virtex II and Virtex II Pro architectures are constructed from CLBs containing eight 4-LUTs, in an array

[2]Lattice acquired the ORCA products from Agere Systems, formerly a part of Lucent Technologies.

size up to 112×104 CLBs.

The Virtex I and Virtex II architectures are based on a similar interconnect fabric. Both architectures contain a large switch matrix at every intersection of the horizontal and vertical routing channels. This switch matrix is used to connect wires between the two channels as well as connect the channels to the input and output pins of one adjacent CLB. Details about the precise connections made in the switch matrix are not published, but it is possible to examine them using Xilinx's FPGA Editor tool.

The Virtex I routing channel contains wires that span one CLB ($L_{wire} = 1$), six CLBs ($L_{wire} = 6$), and the entire device ($L_{wire} = \infty$). Four direct connections between horizontally-adjacent CLBs are also available, but only on specific inputs. In contrast, the Virtex II routing channel adds wires spanning two CLBs ($L_{wire} = 2$) as well as more longer wires. The wires spanning one CLB are replaced with 16 direct connections between all eight immediate neighbours. The number of wires of each length found in these Virtex architectures is shown in Table 2.1. For comparison, this table also includes data for some other Xilinx PLDs.

The two Virtex families also differ in the way wires are buffered at the routing switches. In both families, the $L_{wire} = 6$ wires (and the $L_{wire} = 2$ wires in Virtex II) can only be driven from the wire ends. A signal can be received as an input by CLBs at the end or at the midpoint. Consequently, the wires in different tracks are staggered so they do not all end (or begin) at the same routing channel. In Virtex I, one-third of the $L_{wire} = 6$ wires are bidirectional, meaning they may be driven from either end. The remaining wires are directional: one-third are driven from one end, and one-third are driven from the opposite end. In Virtex II, all of the $L_{wire} = 6$ wires are directional. The directional or bi-directional nature of the wiring is indicated with the 'd' and 'b' suffixes in Table 2.1. This directional arrangement reduces the number of buffered switches, since half of all bidirectional buffers are guaranteed to remain unused. This suggests that layout area is dominated by transistor area and not wire density.

In both architectures, additional special-purpose routing resources are present for carry chains, tri-state lines, and global high-fanout signals.

Architecture	CLB Contents 3-LUT	CLB Contents 4-LUT	CLB I/O Pin Locations	CLB Array Size	Number of Tracks of Length L_{wire} Direct	1	2	4	6	8	∞
XC4000	1	2	distrib.	32×32	-	8	4	-	-	-	6
Spartan I	1	2	on four	28×28	-	8	4	-	-	-	6
XC4000X	1	2	sides	56×56	2	8	4	12	-	-	6h+10v
XC4000XV				92×92						8	
Virtex I	0	4	intersection	104×156	4h	24	-	-	48d+24b	-	12
Spartan II	0	4	of horiz. &	48×72	4h	24	-	-	64d+32b	-	12
Virtex II	0	8	vert. channels	112×104	16d	-	40	-	120d	-	24
Key: ‘b’ bidirectional, ‘d’ directional, ‘h’ horizontal, ‘v’ vertical.											

Table 2.1: CLB configuration and number of tracks in some recent Xilinx PLDs.

Altera Stratix

The latest PLD architecture from Altera is known as Stratix [30, 38]. This architecture is based on a CLB containing ten 4-LUTs connected in a mesh array up to 101×130 CLBs. Each CLB, called a LAB by Altera, has input and output pins distributed on two side columns and one (top) row; no pins are connected to the bottom channel. These LAB pins connect to wires of length $L_{wire} = 4$ and $L_{wire} = 8$ in the rows and columns. The starting points of these wires are staggered along a channel. The LAB input pins can also connect to the output pins of adjacent LABs directly without going through the interconnect. To cover longer distances, the interconnect can connect $L_{wire} = 4$ wires together or it can connect $L_{wire} = 8$ wires together. As well, the $L_{wire} = 4$ wires can connect to long wires of length $L_{wire} = 16$ vertically or $L_{wire} = 24$ horizontally. These long wires can also connect to each other, allowing the quick transport of signals across long distances.

The Stratix publications [30, 38] do not describe details about the amount of wiring or the precise connection patterns between the wires. However, it is clear from [38] that Stratix employs a directional routing scheme where wires are driven by a buffer located at only one end. This paper also suggests there are roughly 80 to 100 wiring tracks in a Stratix routing channel, but no firm number is given. Without additional information, it is difficult to compare the interconnect with that of the Xilinx PLDs.

Summary and Comparison

The latest PLD devices from Xilinx and Altera both use a mesh-based interconnect containing a variety of wire lengths. Both use clusters of 4-LUTs, with Stratix using ten to Virtex II's eight. Stratix uses lengths 4, 8, and 16 or 24 while Virtex II uses lengths 1, 2, 6, and ∞. Virtex I uses some unidirectional wires to reduce routing area. This is extended to all tracks in Virtex II, and it is also employed in all Stratix wiring tracks. Connections can always be formed between wires of similar length. Both architectures also permit connections between wires of different lengths, but the precise connection patterns are not published. The lack of detailed information about these connection patterns, including how to design them, provides motivation for this dissertation.

2.2 Mesh Architecture Studies

There has been a significant amount of interest in PLD routing architectures and algorithms, as demonstrated by the large number of papers on the topic. Mesh-based architectures have emerged as the most popular in all modern, commercial PLDs. These are also commonly known as island-style FPGAs. This section describes a number of academic studies about the design of effective mesh-style PLDs.

The most recent comprehensive collection of mesh architecture studies is the text from Betz, Rose and Marquardt [19]. Much of the information from this section can be found in this text and as well as a number of other papers cited herein.

2.2.1 Mesh Model

A mesh-based PLD can be described in terms of the architectural model shown in Figure 2.3. In this model, logic is implemented in blocks called CLBs, or clustered logic blocks. The CLB has inputs and outputs which must be connected by the routing network. Between the rows and columns of an array of CLBs are the routing channels, containing S blocks and C blocks. The C block connects the input and output pins of a CLB to the routing channel, and the S block connects the wires of two intersecting (orthogonal) chan-

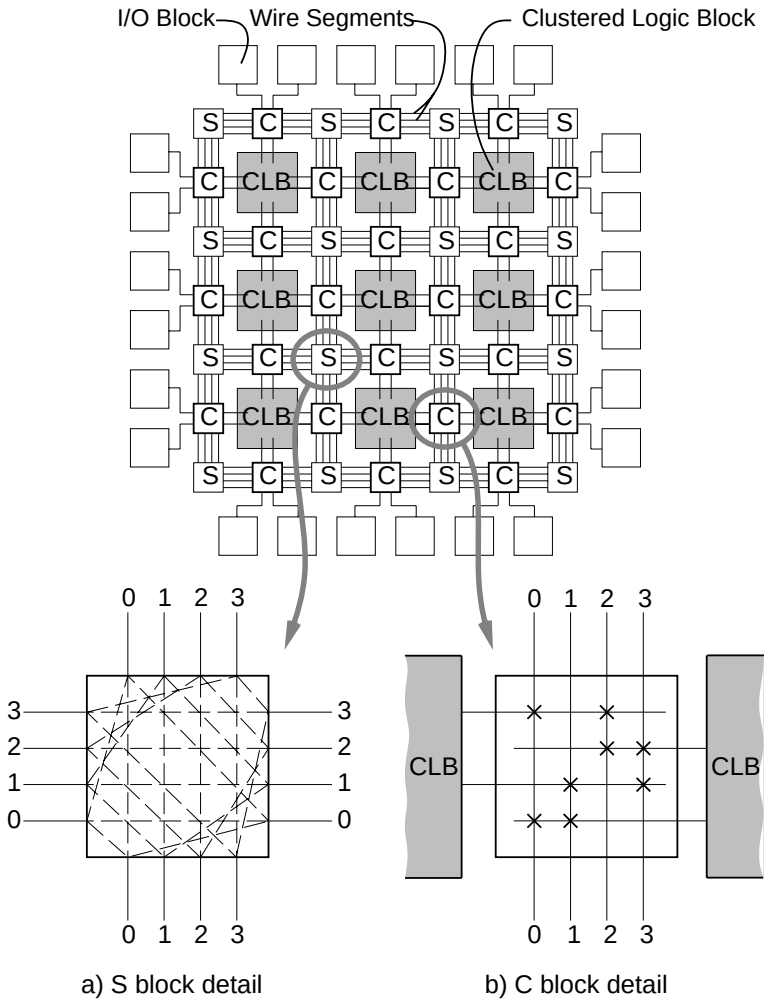


Figure 2.3: A mesh-style PLD routing network.

nels. Figure 2.3a) numbers the wiring tracks of an S block and uses a dashed line to show a possible connection between the ends of two wires. It is also possible to have longer wire segments passing through the S block, but these are not shown in this figure. Figure 2.3b) shows which of the numbered wiring tracks in the C block can be connected to a CLB input or output pin by drawing an $\times$ at their intersection. For input pins, only one wiring track can be selected as the source. For output pins, however, multiple tracks may be driven simultaneously. These assumptions affect how the C block can be implemented and the subsequent area cost.

The remaining sections below summarize important research results in the design of a CLB, S block and C block for mesh-style PLD architectures.

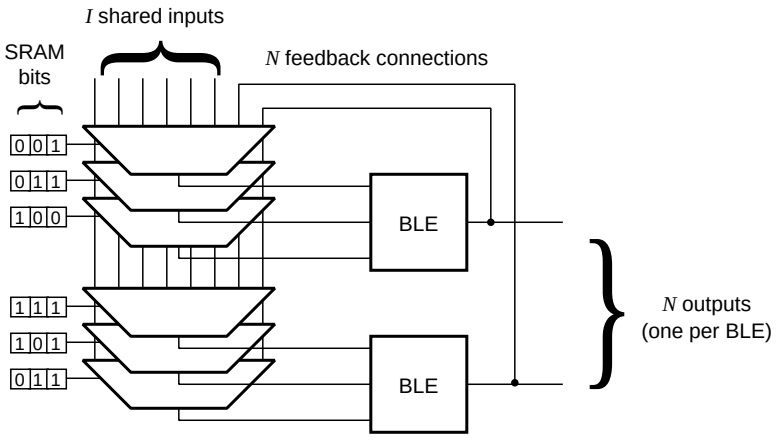


Figure 2.4: A clustered logic block (CLB).

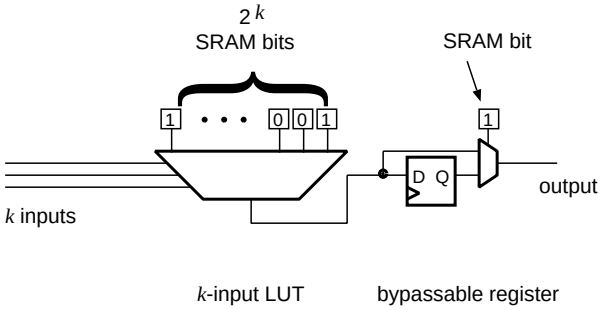


Figure 2.5: A basic logic element (BLE).

2.2.2 CLB Architecture Studies

The idea of clustering a group of N basic logic elements, or BLEs, into a CLB is explored in [39]. The contents of such a CLB are shown in Figure 2.4. Each BLE contains a k-input lookup table (a k-LUT) and bypassable register, as shown in Figure 2.5. Using a heuristic algorithm to pack lookup tables into fixed-size clusters, this paper [39] determines that clusters of N 4-input lookup tables use less area than an unclustered architecture. The savings is accomplished by sharing $I = 2N + 2$ inputs among all BLEs in the CLB. After accounting for the change in the number of routing tracks required, clusters of size $N = 4$ are shown to require 10% fewer transistors. Work by Ahmed [40, 41] determines that the more general expression $I = \lfloor k \cdot (N+1)/2 \rfloor$ is suitable for a range of LUT sizes from $k = 2$ to 7 and cluster sizes from $N = 1$ to 10.

2.2.3 Routing Architecture Studies

Pioneering work by Rose and Brown [42] examined the number of connections needed between wires in S blocks and C blocks. To minimise switch counts, they determined that S blocks should contain 3 connections per wire and C blocks should have a switch at 70–90% of all possible switch locations. These details are summarized by the flexibility parameters $F_s = 3$ and $F_c = 0.7$ to 0.9. For simplicity, $F_c = 1.0$ (a full crossbar) is often assumed by later research.

Subsequent work by Brown, Lemieux, and Khellah [43] determined that nearly 50% of the average net delay can be eliminated using long wire segments in the architecture.

In [44, 45], Betz determines that mesh PLDs should be square, that logic block pins should be evenly distributed on all four sides (confirming the results of previous work [46]), and that all routing channels should have the same number of wiring tracks.

Numerous routing experiments by Betz *et al* [47, 48] explored the best wire lengths and transistor sizes for a mesh routing network. The findings can be summarized as follows: wires which span 4 or 8 logic cells provide the lowest average critical-path delay and area·delay product, about half of the routing tracks should use pass transistor switches (with the remainder using buffered switches), buffers should use a drive stage which is 5 times minimum size (width), and pass transistors should be 10 times minimum width. In this work, Betz *et al* found that the most area-economical way to build a tri-state buffer for mesh interconnect is based on one shared inverter chain plus one wide pass transistor for each switched connection or tri-state output. That work also found that increasing the spacing between the metal routing wires is more effective at reducing delay than widening them. More recent work by Roopchansingh [49, 50] uses even shorter connections, specifically between nearest-neighbour CLBs, to improve delay by roughly 7%.

2.3 Switch Blocks

Considerable interest has been shown in switch block (S block) design. Initial work has examined the number of switches necessary to achieve good routability, while more recent research has sought the best organisation, or topology, of those switches.

2.3.1 Switch Block Flexibility

In [42], Rose and Brown define the *flexibility* of an S block, F_s, as the number of switches connected to the end of a wire. An early PLD architecture, the Xilinx XC3000, uses an F_s value ranging from 4 to 6, with $F_s = 5.4$ on average. In comparison, Rose and Brown concluded that $F_s = 3$ is the lowest value that is still flexible enough to route a suite of benchmark circuits. For a routing channel width of W tracks, this requires $6W$ switches. In the same paper, the authors introduce the notion that the *topology* of an S block, or the precise organisation of switches for a given value of F_s, is important — particularly when there are few switches.

2.3.2 Xilinx or *disjoint* Switch Block

In the same year that [42] was published, Xilinx released a new architecture, the XC4000, which contains an S block with $F_s = 3$. The design is based on a six-switch clique that can connect four distinct wire segments, one from each side. The connection pattern can be concisely described by numbering the wiring tracks on each side from 0 to $W-1$ and connecting wires with the same track number by a clique of 6 switches. This topology is commonly known as the *Xilinx switch block*, but other names have been used in other publications: *disjoint* [51], *clique-based* [52], *completely disjoint* [53], *subset* [19], and *planar* [54]. Within this dissertation, the term *disjoint* shall be used to describe this organisation. This term arises from the observation in [51] that the wiring tracks are independent (disconnected) when all the switches in the switch block are closed.

Two examples of a *disjoint* switch block are shown on the left of Figure 2.6. In the top left switch block, wires from $W = 4$ routing tracks are shown approaching from all four sides and ending at the open dots. Switches between wires are shown as lines connecting the open dots within the switch block. To illustrate the clique, the six switches connecting track 0 are drawn with thicker lines. The bottom left switch block in the figure is also *disjoint* but for $W = 5$. It is drawn in a way that better illustrates how the tracks are disjoint and how VLSI layout might be based on the repetition of clique structures.

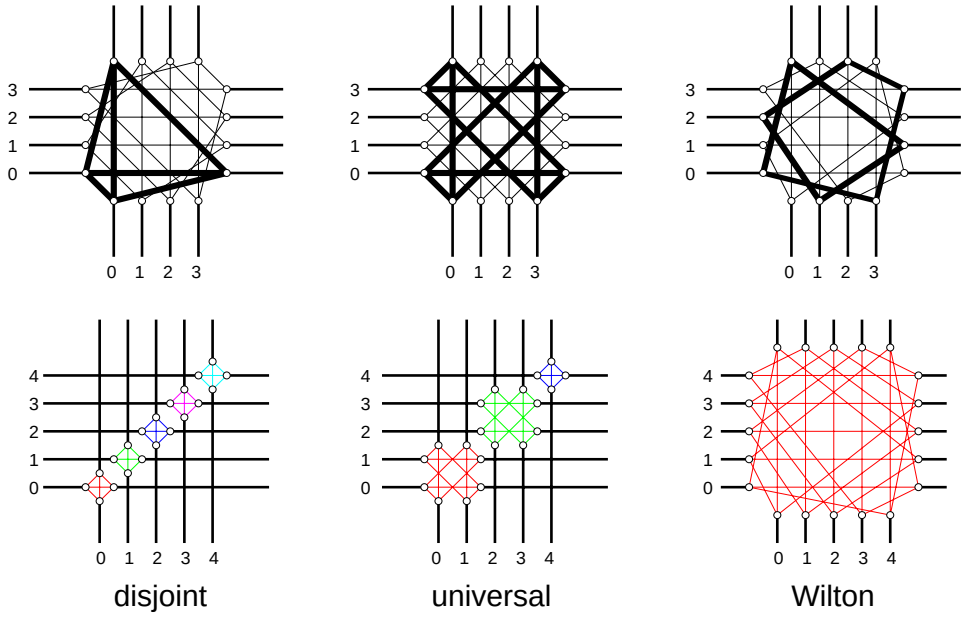


Figure 2.6: *Disjoint*, *universal* and *Wilton* switch block styles.

2.3.3 *universal* Switch Block

Another switch block, the *universal* switch block or USB, is introduced by Chang *et al* in [52, 55]. Like the *disjoint* block, this topology also uses $6W$ switches. However, it is called *universal* because the precise switch placements are locally optimal for two-terminal nets: any set of two-terminal nets can be routed, provided the total number of nets on any one side does not exceed W. This is referred to as a *bandwidth* or *dimensional constraint*. The *disjoint* pattern is considered inferior because it cannot route all cases that satisfy the bandwidth constraint. Experiments in [52, 55] demonstrate that the USB requires fewer routing tracks to route different benchmark circuits.

Unfortunately, the *universal* switch block definition assumes a routing architecture based on short routing wires which span only a single logic block. All modern PLDs employ longer routing wires, and it is not immediately obvious how the *universal* switch block design can be extended to include longer wires and remain universal. Chapter 7 considers one possible way of using long wire segments by using a *universal* switch block only at the endpoints. This new design is called *universal-L*, but it is not universal.

Two examples of a universal switch block are shown in Figure 2.6. The switch pattern

can be decomposed into two types of connections. The top example, drawn with $W = 4$, highlights one connection type between a pair of tracks. The bottom example, drawn with $W = 5$, shows that tracks are pairwise-disjoint, except for any odd leftover track which must use a 6-switch clique. Again, the lower example shows a structure which may be more conducive to VLSI layout.

Although the properties of the *universal* switch block sound promising, there are two practical limitations. First, it is optimal for only two-terminal nets. Second, it assumes that nets can be freely reordered on all four sides. The latter assumption shows how the locally-optimal property may not be exploitable in a global interconnect fabric which cannot guarantee any input ordering.

2.3.4 *generic universal* and *hyperuniversal* Switch Blocks

There are two interesting extensions to the *universal* switch block work. The first extension by Shyu *et al*, called a generic USB [56, 57] or GUSB, generalizes the block from only four sides to N sides, $N \geq 2$. Work by Fan *et al* [58] revealed that the GUSB construction in [57] is not universal for odd W, but a correction has been found [59]. The correction uses $\binom{N}{2} \cdot W$ switches if W is even, or slightly more if W is odd. Physical decompositions of the GUSB into smaller connected components are also described in [59], a feature which may aid VLSI layout. The second extension by Fan [60, 61, 62], called a *hyperuniversal* switch block, creates switch blocks which are locally optimal for multi-terminal nets. This *hyperuniversal* switch block removes the two-terminal net restriction of the *universal* design and requires approximately $6.7W$ switches (with 4 sides).

2.3.5 *Wilton* and *Imran* Switch Blocks

Work by Wilton [63] found that the *disjoint* and *universal* switch blocks restrict a net to stay within the same track or pair of tracks, respectively. These switch blocks did not perform well in routing experiments with very sparsely connected memories because memory pins often connected to only a few specific tracks. To alleviate this restriction, Wilton designed a new switch block which changes the track assignment on connections that turn. Hence,

overall track changes can be accomplished by permuting the global route. This allows greater flexibility in the initial track selection near the source. Routing experiments with single-length wires found this *Wilton* switch block to require about 5% (14%) fewer tracks than the *disjoint* (*universal*) switch block.[3]

Two examples of a *Wilton* switch block are shown in Figure 2.6. The top example shows a $W = 4$ example with a few switches traced in bold. These bold switches show how it is difficult to isolate a few tracks to create regular, decomposable structures. For odd W, such as that shown in the lower example, the results are similar. It is possible to reach any track from any other track by cycling through switches in this block.

In [19], Betz *et al* shows how long wire segments can lead to lower efficiency in the *Wilton* and *universal* switch blocks compared to a *disjoint* switch block. Although wire ends always connect with $F_s = 3$ switches, each long wire passing straight through a *disjoint* switch block connects to only one switch (*i.e.*, $F_s = 1$). However, in one of the other switch blocks, this same long wire may contain up to four distinct switches! The resulting increase in transistor area per track outweighs the savings obtained through a track count reduction.

This switch-per-track inefficiency of the *Wilton* switch block is addressed by Masud [64, 65]. The solution is simple: separate the tracks with wire endpoints from the tracks which contain the interior region (or midpoints) of long wires. The endpoints are connected with a *Wilton* switch block, and the wire midpoints are connected with a *disjoint* switch block. When only buffered switches are used, this *Imran* switch block requires less routing area than *disjoint*, *universal*, and *Wilton* switch blocks [65]. However, later work [66] suggests that the *disjoint* switch block has lower area·delay product than the *Imran* switch block when half of the tracks use pass transistors. This latter result is questionable because delay and area per track is sensitive to the precise way in which the two switch types are mixed. The *Imran* switch block will form connections between pass transistor tracks and buffered tracks. Clearly, this will affect delay, but it also negatively affects area because pass transistors are used in only one direction. In contrast, a *disjoint* switch

[3]Notice that Wilton found the *universal* switch blocks to be inferior to *disjoint*. This will be further discussed below.

block segregates the two switch types entirely to fully utilise each pass transistor in both directions. Hence, the choice of switch block topology must be done carefully lest it have a negative impact on the switch-per-track area efficiency and on the delay of connections.

The experiments run by Wilton suggest that the *disjoint* switch block requires fewer routing tracks than the *universal* one. This is contrary to the claims of Chang *et al* in [52, 55]. These results may differ because a number of experimental conditions are different between these investigations, including the routing tool and benchmark circuits used. However, the large difference and opposite conclusions are surprising.

New experiments are performed in Chapter 7 to compare these two switch block styles in a new context: a modification of the *universal* switch block is made so it is more efficient with longer wire segments. These new results suggest that the new *universal* switch block uses fewer tracks than the *disjoint* switch block, contradicting Wilton's results.

2.3.6 Switch Matrix

Another type of switch block is known as a *switch matrix*. A switch matrix is a special subclass of switch blocks which restrict the allowable switch locations based on the switch matrix model. An example of this model is shown in Figure 2.7a). It contains *separating switches* (shown as hollow circles) to break a wiring track into segments and *crossing switches* (shown as dark circles) to join orthogonal wires.

The switch matrix model is less flexible than the switch block model because a horizontal wire on the left side, for example, can only connect with one track on the opposite side or to those vertical wires to the left of its separating switch. However, it is an interesting model to consider when area constraints require fewer than the $6W$ switches needed by the *universal* switch block, for example. Using fewer switches within a more restricted model means a switch matrix is incapable of satisfying the bandwidth constraints obtained with *universal* switch blocks. As a result, switch matrix research has sought the best organisation to maximise the number of two-point connections that can be simultaneously realised.

In [15], Zhu *et al* give a switch matrix construction technique which is suitable when only a few switches are required. However, the switch locations computed are very irreg-

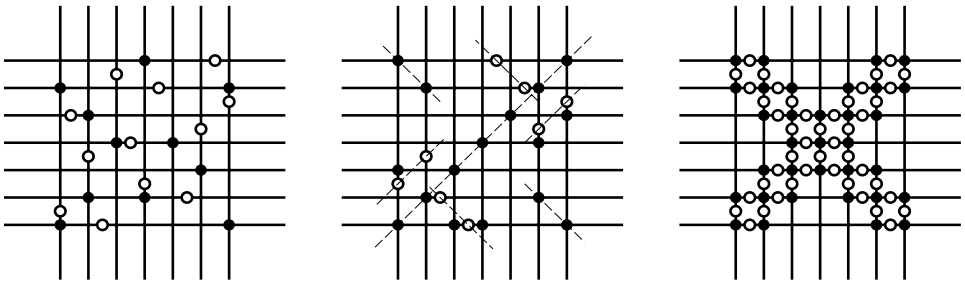


Figure 2.7: Examples of switch matrices a) by Zhu, b) by Chang, and c) the QUSM by Wu.

ular and there is no uniformity in the number of switches per wire. A better construction technique is given by Chang *et al* [67]. This produces more routable switch matrices by placing switches along various diagonals. As well, this distributes the number of switches per wire more uniformly. Figure 2.7a) and b) exemplify the type of switch matrix constructed by these two approaches.

Efforts by Wu and Chang [68, 69, 70] have attempted to extend the switch matrix model to be universal, *i.e.*, to always satisfy the full bandwidth constraint for two-terminal nets. With this extension, the switch matrix model is no longer a subclass of the more traditional switch block model. The extension they propose is the structure shown in Figure 2.7c), called a quasi-universal switch matrix or QUSM. The name is derived from the fact it is not quite universal, except in the limit as $W \rightarrow \infty$. In fact, within this model the QUSM is shown to reach the highest possible bandwidth capacities with the fewest number of switches. However, the QUSM is impractical because it contains up to four separating switches per switch block and it requires a total of $14W - 20$ switches (for even W), roughly twice as many as the *hyperuniversal* switch block.

2.3.7 Other Switch Blocks

Work by Wu *et al* [71, 72, 73, 74] considered the routing problem as divided into a global routing phase, where nets are assigned to routing channels only, followed by a detailed routing phase, where specific channel resources (wire segments and switches) are allocated for each net. That work shows that the detailed routing problem for a mesh-based PLD with *disjoint* switch blocks can be restated as a graph colouring problem. The global route for

a netlist determines an underlying graph to be coloured, H. This led to the use of graph colouring and clique detection algorithms to analyse detailed routing problems [53].

Although finding a minimal colouring of a general graph is NP-complete [75], efficient algorithms may exist if some graph structure on H can be assured. With this in mind, Wu *et al* suggest a new concept: a greedy routing architecture or GRA [74]. This is a radical change to switch block topology which allows greedy routing algorithms to be used to extend a partial routing solution into a full routing solution. The first GRA by Wu makes switch blocks fully connected between all sides, except between the left and right sides where switches are only placed straight across the block. This reduces the detailed routing problem to a one-dimensional packing problem. Takashima *et al* [76] further reduces the number of switches by using only one fully-connected side. The second GRA by Wu uses only three sides of a switch block and $3W$ switches. This roughly transforms the routing network into a hierarchical network with an H-tree structure. The H-tree guarantees that only one global route exists for a placed netlist. More importantly, the underlying graph structure leads to a linear-time detailed routing algorithm which uses no more than $3/2$ more routing tracks than required by the global route. Despite having good bounds, the rigid H-tree structure is impractical because all channels must have the same width, from the lowest level to the highest level. Hence, increasing interconnect demand at the top levels creates an excessive surplus of wires at the lowest level. The GRA concept is an interesting theoretical structure but many practical issues must still be addressed.

Another switch block explored by Sun *et al* in [77, 78] uses a model similar to the switch matrix model, except that a permanent wire cut is used instead of separating switches. Unfortunately, the algorithm used to place crossing switches is not described; it appears to be done randomly except for the constraint of keeping the number of switches per wire constant. Routing experiments with a few small benchmark circuits indicate that about half of the crossing locations must have switches to preserve routability. This is still a very large number of switches which is impractical for large PLD architectures.

Hallschmid and Wilton [79] explores the design of switch blocks used in rectangular mesh arrays. They found that more routing tracks are needed in the channels spanning the longer dimension, resulting in non-square switch blocks. To improve area efficiency, many

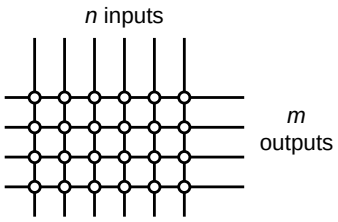


Figure 2.8: A 6×4 full crossbar.

of the switches in these non-square blocks can be removed, achieving a total area savings of roughly 9%.

2.4 Crossbars and Connection Blocks

An $n \times m$ crossbar connects n different input wires to m output wires, generally with $n \geq m$. An example of a few crossbars are shown in Figures 2.8 and 2.9. At the locations where an input (vertical wire) crosses an output (horizontal wire), a programmable switch or *crosspoint* may be present. These are indicated by small, open dots in the figure. The *capacity* of a crossbar, c, is the largest number of signals which are always routable for any assignment of signals to inputs. Clearly, $0 \leq c \leq m$. The term *population* refers to the number of switches in the crossbar, p, such that $0 \leq p \leq n \cdot m$. This can also be expressed as a density. The capacity c is determined by p as well as the precise location of switches within the crossbar.

The number of switches connecting to an input (or output) wire is its *fan-out* (*fan-in*). If all outputs have the same fan-in, they are said to be *balanced*. If the largest difference in fan-in across the output wires is k, they are said to have nearly balanced fan-in *within* k. The same terms also describe the fan-outs of the input wires.

The connection block, or C block, of a mesh style PLD can be implemented with a crossbar switch. This block connects the routing channel to a group of logic block pins. Since the channel width can be quite large, it is necessary to reduce area by using crossbars with significantly fewer than the maximum $n \times m$ switches. However, reducing the number of switches may have a negative impact on routability. For example, having fewer than $n \times m$ switches creates an immediate restriction that some inputs cannot connect to some

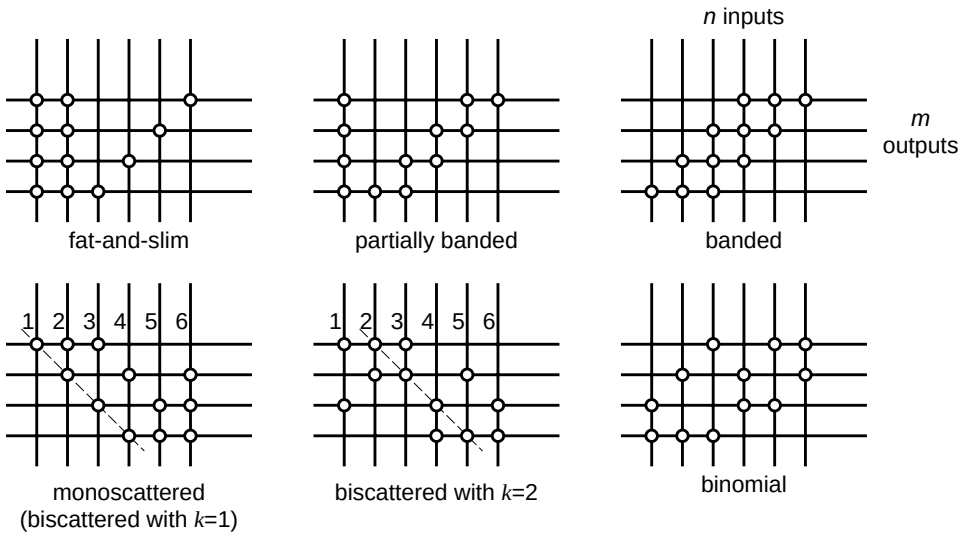


Figure 2.9: Examples of 6×4 minimal full-capacity crossbars.

outputs. If the order of the outputs is unimportant, the crossbar can tolerate the removal of a significant fraction[4] of the switches. This is often the case in PLD routing, where it is usually more important to select a specific subset of the inputs than to rearrange their order.[5] Hence, it is important to determine the right number of switches as well as the best location for them. It is also important to have a metric which measures the routability of these crossbars.

Below, a summary is given of prior research concerning the construction of crossbars. These studies are primarily concerned with the design of minimal and guaranteed-capacity crossbars. However, one practical element that is missing from these publications is a general method for constructing a sparse crossbar that contains very few switches and yet is still very routable. This omission is addressed in Chapter 4.

2.4.1 Full Crossbars

In most literature, the sole term *crossbar* usually implies a fully-connected crossbar with a total of $p = n \cdot m$ switches. In this dissertation, the terms *full crossbar* or *fully-populated*

[4]Depending upon the precise values of n and m, this could be 50% or more.

[5]PLDs often compensate for this eventually by using small lookup-tables, for example, which can freely permute the order.

crossbar will be used for this case to distinguish it from other types of crossbars, such as a minimal or sparse crossbars (defined below). An example of a full crossbar is shown in Figure 2.8.

Full crossbars are very flexible because they can connect *any* wire on the input side to connect to *any* wire on the output side, *i.e.*, they select any of the inputs and can freely reorder the outputs. Additionally, full crossbars can be used at *full capacity*: they can connect as many signals as the number of outputs in the crossbar, so $c = m$.

2.4.2 Minimal Full-Capacity Crossbars

One alternative to using a full crossbar is a *minimal full-capacity crossbar*. For convenience, these will be referred to as *minimal crossbars*. These well-known constructions use fewer switches than a full crossbar, and consequently lose the ability to arbitrarily permute their outputs. However, like a full crossbar, they can also route any $c = m$ inputs to m outputs. In a PLD, such a restriction on output permutations is often acceptable because downstream resources do not usually assume any particular input order.

A minimal crossbar always contains

$$p = (n - m + 1) \cdot m$$

switches. Nakamura and Masson [80] proves that no switches can be removed from a minimal crossbar without also removing the full-capacity property (hence, proving they are minimal). Minimal crossbars do not save many switches when $n \gg m$, but the number of switches is reduced from a quadratic expression to a roughly linear one when $n \simeq m$.

A few minimal crossbar constructions are shown in Figure 2.9. Note that none of the crossbars in this figure are isomorphic with respect to a permutation of the inputs or outputs. Except for the binomial and biscattered examples, this can easily be verified by the unique switch counts per input or output. These other two examples are both balanced, but they differ in another way: the biscattered one has two pairs of inputs which connect to the same outputs, but each of the inputs in the binomial crossbar connects to unique subset of outputs.

The simplest minimal topology, called a *fat-and-slim crossbar* [81], uses a full crossbar

between the first $n-m$ input wires and all m output wires. Each of the remaining m input wires are connected to m different output wires using one switch each. This is drawn in the figure as a diagonal line of switches. This topology results in balanced *fan-in* for the output wires, but largely unbalanced *fan-out* for the inputs. Other simple topologies are known as *partially banded* and *banded* crossbars [80, 81].

Some minimal crossbar topologies have balanced fan-in and nearly balanced fan-out. Fujiyoshi *et al* [82, 83] define a class of minimal crossbars called *monoscattered* and a more general class called *biscattered*. Both of these classes have balanced fan-in, but biscattered ones can be designed to have balanced fan-out as well. To construct these crossbars, begin with an empty grid of n vertical input wires and m horizontal output wires. Draw a single diagonal line of m switches at the intersection points, beginning at the top-left corner of the grid and placing one switch per input and one per output. The remaining switches, evenly divided per output, are placed anywhere to the right of this diagonal line. This creates a monoscattered crossbar. A biscattered crossbar begins with a similar diagonal, but its origin is offset to the right, starting at input wire number k. The remaining switches are evenly divided per output, with $k-1$ switches placed to the left of the diagonal and the remainder placed to the right. Hence, a monoscattered crossbar is actually biscattered with $k=1$. In [82, 83], a switch placement algorithm is given which balances the fan-out of biscattered crossbars. It is worthwhile to note that the authors provide a counter-example to demonstrate that not all minimal crossbars are biscattered.

Oruç and Huang [81] describes additional minimal crossbar topologies that have nearly balanced fan-in (within 2) provided there are $n=2m$ inputs. Guo and Oruç [84] extends this to an arbitrary number of inputs and outputs. More importantly, Guo and Oruç [84] also suggests a switch-move transformation that modifies one crossbar instance, such as the fat-and-slim topology, to create other minimal crossbar topologies. These transformations, which will be described in the next section, preserve full capacity and the fan-in profile of the outputs. Guo and Oruç [84] also proves that a series of these transformations can be used to obtain topologies with nearly-balanced fan-outs (within 2).

2.4.3 Sparse and Guaranteed-Capacity Crossbars

A *sparse crossbar* refers to a crossbar which is sparsely populated, meaning it has few switches. The demarcation point of when a crossbar becomes "sparse" is debatable: for example, nearly square crossbars can be sparsely populated yet support full capacity. Although arbitrary, this dissertation assumes that a crossbar is sparse if it contains

$$p < (n - m + 1) \cdot m$$

switches. Hence, no matter how well it is designed, a sparse crossbar cannot guarantee full routing capacity.

If a sparse crossbar is operating at less than full capacity, it is natural to ask whether the switches can be arranged so that a capacity of c can be guaranteed. Alternatively, one can ask what is the fewest number of switches required to construct a crossbar with capacity c. Previous work has addressed both of these questions. However, no previous work has asked the following question: *What is the best way to arrange a given number of switches such that routability is maximised?* Here, routability refers to the ability of a sparse crossbar to route any arbitrarily-chosen subset of its inputs to the outputs. This raises another interesting question: *How can routability be measured?* While Chapter 4 explores a solution for these last two questions, the remainder of this section describes the previous work which answers the first two.

In [85], Masson defines a *binomial crossbar* with parameter v. Such a crossbar contains $n \cdot v$ switches, $n = \binom{m}{v}$ inputs, m outputs, and has guaranteed capacity $c = v + 2$. Every input of a binomial crossbar contains exactly v switches arranged such that every input connects to a unique subset of the outputs.[6] The binomial crossbar shown in Figure 2.9 is an example of a minimal full-capacity crossbar where $v = 2$. The binomial crossbar is limited in practical use because it is a very specific construction which gives no freedom to choose the number of inputs.

In [80], Nakamura and Masson explore a lower bound on the number of switches. They prove that a sparse crossbar with capacity c must contain $p \geq n \cdot x$ switches, where x

[6] Notice the number of inputs is chosen so all possible subsets of outputs are covered by the inputs.

satisfies the expression

$$\frac{\binom{c}{x}}{\binom{m}{x}} n(c-x) - c^2 + c = 0.$$

It should be noted that x is not restricted to being an integer[7] and that $0 \leq x \leq c$. As $n \to \infty$, this bound asymptotically approaches $p = n \cdot c$ switches (for fixed m and c). This bound is tight for $c = 1$, $c = m$, and binomial crossbars, but it is not necessarily tight in general. The authors of [80] acknowledge two other limitations: a) except for the tight constructions mentioned, creating a sparse crossbar of capacity c with $n \cdot x$ switches is an open problem which may not be achievable, and b) they did not know of any systematic procedure for discarding certain inputs of a binomial crossbar to create a sparse crossbar with increased routing capacity.

For a given n, m, and c, the Nakamura-Masson lower bound is difficult to compute. Instead, Oruç and Huang [86] provides another lower bound which is very easy to compute:

$$p \geq \left\lceil \frac{m \cdot (n-c+1)}{(m-c+1)} \right\rceil.$$

Like the previous bound, this new bound is also tight when $c = 1$ or $c = m$. However, under other conditions it is less tight.

Oruç and Huang [86] also suggests an explicit construction of a sparse crossbar with guaranteed capacity c, provided that $n - m \leq c \leq \lfloor m/c \rfloor$. Figure 2.10 gives an example and illustrates the general structure of this pattern. This crossbar is of limited practical use because it is only applicable when the capacity is small ($c \leq \sqrt{m}$) and the crossbar is nearly square ($n \leq m + c$). Notice also that this construction may leave some outputs without any switches at all.

The Guo and Oruç transformation [84] mentioned in the previous subsection can also be applied to sparse crossbars. The crux of this transformation relies upon finding two inputs, I_i and I_j, where the set of outputs connected to I_i, or $O(I_i)$, is a superset of $O(I_j)$. When this occurs, it is said that I_i *covers* I_j and any number of switches can be moved from I_i to I_j provided this doesn't change the total number of switches or $O(I_i) \cup O(I_j)$. Guo and Oruç prove that this transformation creates a new switch pattern with at least the same capacity as the original pattern. This is a powerful switch-movement transformation,

[7] The gamma function can be used, resulting in $\binom{c}{x} = \frac{\Gamma(c+1)}{\Gamma(x+1) \cdot \Gamma(c-x+1)}$.

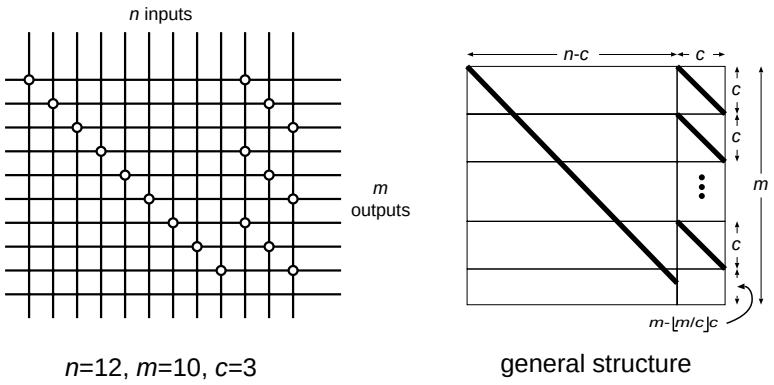


Figure 2.10: Oruç-Huang guaranteed-capacity sparse crossbar construction.

but its practical use depends upon finding I_i and I_j. If switches are placed randomly, for example, it is unlikely that many such pairs of inputs exists.

In [87], Azegami describes a sparse crossbar construction with guaranteed capacity c, for any n and m, and two switch-movement transformations which preserve capacity. The construction, shown in Figure 2.11, uses $p = m + (n-m)c = nc - m(c-1)$ switches. For fixed m and c, the number of switches approaches $n \cdot c$ as $n \to \infty$. This is comparable to the Nakamura-Masson lower bound. The two switch-movement transformations are a vertical switch move followed by a horizontal switch move. The vertical transform preserves c switches per input, but moves them from the dense $(n-m) \times c$ region on the right side of Figure 2.11 to the $(n-m) \times m$ region below it. This can nearly balance fan-in of the outputs. The horizontal transform is a special case of the Guo and Oruç transformation [84] because each input in the entire $(n-m)$ region on the right now covers many of the inputs in the m region on the left. Hence, there are many input pairs which support horizontal switch moves from the $(n-m)$ region to the m region. This can nearly balance fan-out of the inputs. Azegami has given an important sparse crossbar construction with guaranteed capacity and identified transformations that can nearly balance fan-in and fan-out.

The construction and transformation techniques discussed in this section provide ways to guaranteed a specific routing capacity. In practice, however, it is usually sufficient to provide a high probability that a routing solution exists; a few unroutable cases can be avoided by making other choices elsewhere in the network. Surprisingly, there have been no construction techniques published which attempt to maximize the probability of routing.

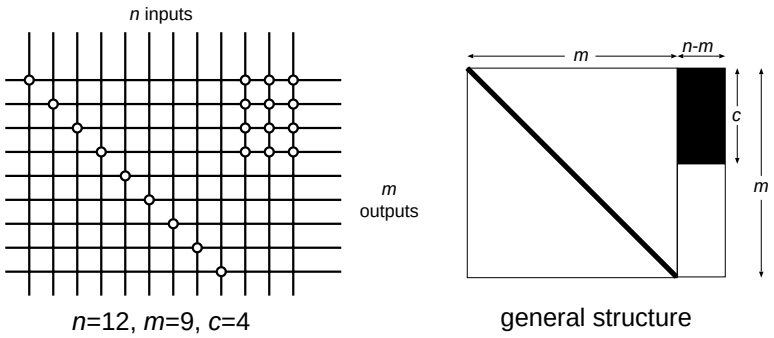


Figure 2.11: Azegami guaranteed-capacity sparse crossbar construction.

Chapter 4 addresses this by introducing a method to measure routability of sparse crossbars and a method to construct sparse crossbars which are highly routable.

2.5 Multi-Stage Networks

PLD routing networks impose many connectivity requirements that seem to parallel those previously encountered during the development of multistage networks for telephony and computer communications networks. Considering the amount of research directed at these networks, it is important to understand the impact they may have on PLD routing networks. However, the unique requirements of PLDs make direct application of previous network results difficult. This is because traditional multistage network research has focused on the design of networks: a) which are perfectly non-blocking, hence guaranteeing the existence of a routing solution, and b) which have simple routing algorithms, often ones which can form connections on-line in a continuously-updated fashion and/or with only local decisions at each step. In contrast, complex routing algorithms are often used for PLDs, and routing solutions are seldom guaranteed. As well, PLD interconnect resources are statically allocated whereas many modern communications networks employ time-sharing and queueing systems. The fan-out of connections also differs: in PLDs, a significant proportion of nets are high fan-out, whereas conference calls and internet multicasts are comparatively rare and have low fanout. Despite the vast differences, it is still insightful for a PLD designer to understand multistage networks. The area provides fresh ideas and gives rough bounds on the amount of switching resources needed.

This remainder of this section summarises important contributions in multistage network literature which are likely of interest to PLD network designers. A survey paper by Yavuz Oruç [88] contains a more comprehensive list of references.

2.5.1 Network Types

Networks can be broadly characterised as either blocking or non-blocking. A network is said to be *blocking* if there is at least one set of connection requirements, of inputs to outputs, which cannot be simultaneously satisfied. If there is a solution for every possible set of connection requirements, it is *non-blocking*.

There are various degrees of non-blocking behaviour, described as rearrangeable, non-blocking in the wide sense, and non-blocking in the strict sense. These differ in flexibility, from the least flexible (rearrangeable) to the most flexible (strict sense non-blocking). A *rearrangeable* network has a non-blocking solution, but routing the connection requirements in a serial fashion may require the rearrangement of previously made connections to realise a solution. A non-blocking network in the *wide sense* can make all of the connection assignments serially by following a well-defined routing rule or rule set without rearranging any previously made connections. The routing rules allow the network to avoid *blocking states*, wherein a partial solution is incapable of being extended to add a new arbitrary connection. A non-blocking network in the *strict sense* can always make all of the connection assignments serially without following any routing rules. If a network is simply said to be non-blocking, the strict sense is usually implied because there are no blocking states.

From the perspective of a PLD, a rearrangeable network might be desirable because a routing solution exists for all possible netlists. However, this is not adequate for PLDs in two different aspects. First, a rearrangeable network may contain more flexibility than what is actually needed. Rearrangeable networks do not exploit locality to save area, a characteristic which circuits are known to exhibit [6]. Second, there is no guarantee that a rearrangeable network will achieve good delay performance.

Most multistage network research has focused on the creation of non-blocking networks. Despite the potential drawbacks of using these in PLDs, it is useful to know how these networks are built. The following subsections describe a number of non-blocking

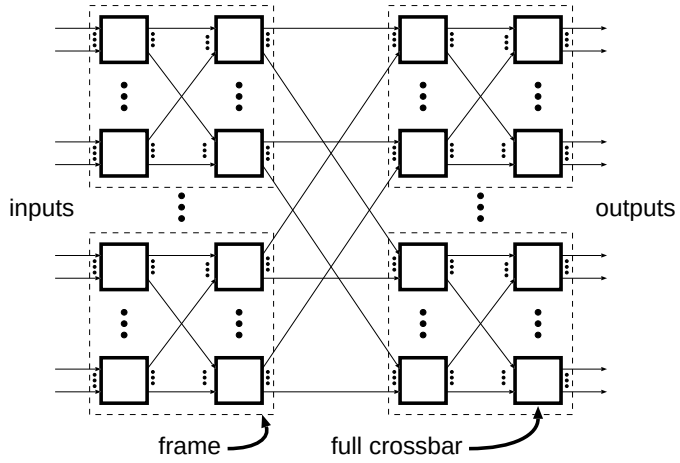


Figure 2.12: No. 5 crossbar switching network.

networks from the literature. It is interesting to note the importance of full crossbars and how they are used in the recursive decomposition of network structures.

2.5.2 No. 5 Crossbar

In the 1950s and 1960s, the No. 5 crossbar system was commonly used for telephone switching in urban areas. A typical size of this system would connect 1,000 inputs (or customer lines) to 1,000 outputs (or trunk lines), but typical use would seldom involve all 1,000 lines simultaneously.

The organisation of the No. 5 crossbar system is shown in Figure 2.12. In the figure, the boxes represent square full crossbars with a multitude of inputs (and outputs). Two levels of crossbars are connected in a group known as a frame, and two levels of frames are similarly connected. For a 1,000 line system, the repeating elements (shown with ellipses) might appear in groups of 10, for a total of 40,000 switches or crosspoints in all of the crossbars.

Despite its widespread use, the No. 5 crossbar is not very efficient. A 1,000 line system can only realize a tiny fraction ($\leq 10^{-64}$) of the 1000! permutations [89]. Hence, it is a blocking network. In comparison, the Beneš network described later can realize all permutations of 1,024 lines with only 32,768 switches.

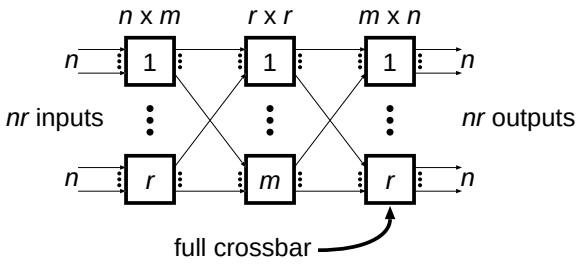


Figure 2.13: Clos network.

2.5.3 Clos Network

In 1953, Clos described a way of decomposing a full crossbar into three stages to use fewer switches, creating what is now known as a Clos network [90]. Figure 2.13 illustrates this approach for an $N \times N$ crossbar with $N = n \cdot r$ inputs and $N = n \cdot r$ outputs. Clos proved that this network is strictly non-blocking if $m \geq 2n-1$. For $m = 2n-1$ and $n = r = \sqrt{N}$, the Clos network uses only $6N^{3/2} - 3N$ switches compared to the N^2 required by a full crossbar. For example, an $N = 1,000$ Clos network uses 186,737 switches instead of 1,000,000. For $n \leq m < 2n-1$, the Clos network is no longer strictly non-blocking, but it is still rearrangeable and it uses fewer switches [89]. In particular, the $n = m$ case requires $3N^{3/2}$ switches. Clos noted that this decomposition can be applied recursively by replacing each of the middle $r \times r$ crossbars with another 3-stage network. Increasing the number of stages in this way reduces the $N^{3/2}$ exponent and this decreases the overall switch count (for sufficiently large N). This is the approach followed by Beneš in the next section.

2.5.4 Beneš Network

A network design by Beneš [91] is rearrangeable using $4N(\log_2 N - 2)$ switches. Figure 2.14 shows a Beneš network for $N = 16$ inputs. Such a network can be constructed from a Clos network by using 2×2 crossbars on the first and third stages, then recursively decomposing each of the inner two $N/2 \times N/2$ crossbars in the same way. This recursive construction is illustrated in Figure 2.15. The depth of such a network is $2\log_2 N - 1$ stages.

Over the years, a number of similar networks have been discovered and re-discovered in the course of telephony and computer network research. The banyan, baseline, butterfly,

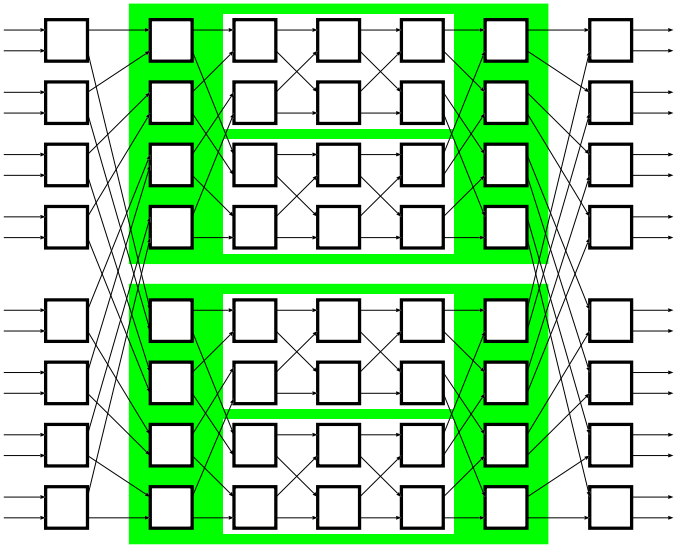


Figure 2.14: Beneš network for 16 inputs and 16 outputs.

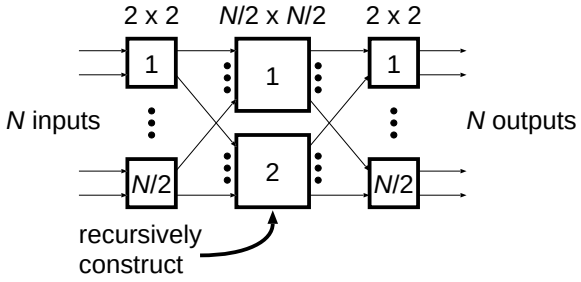


Figure 2.15: Recursive construction of a Beneš network from a Clos network.

delta, and omega networks, for example, are variations of the first $\log_2 N$ stages of the Beneš network. Each of these are defined by their specific shuffle steps, or wiring patterns, used after every stage. For example, the shuffle step of an omega network uses the same pattern as the last shuffle step of the Beneš network in Figure 2.14. However, the omega network uses this same shuffle step at every stage of the network. This makes it possible to save hardware resources by time-sharing a single stage and re-circulating the outputs back into the inputs.

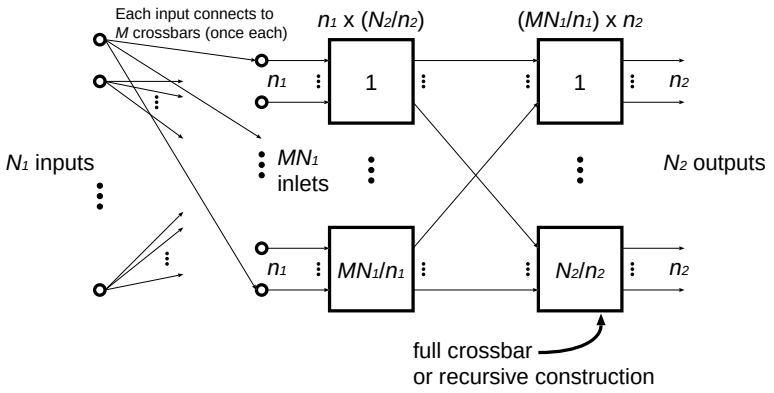


Figure 2.16: Non-blocking Richards-Hwang network with full broadcast ability.

2.5.5 Rearrangeable Networks with Fanout

Richards and Hwang [92] describe a novel two-stage network which is rearrangeable and supports arbitrary broadcasting of the inputs. The network is based on removing the first stage of a Clos network and fanning out M copies of each primary input to the inlets of M different crossbars in the newly created first stage.

Figure 2.16 shows the overall structure of a Richards-Hwang network. The precise fanout pattern is governed by an equation which selects a different group of inputs for each crossbar. The number and size of the crossbars within the network are as follows. For N_1 total inputs and N_2 total outputs, each first stage crossbar has n_1 inputs (n_1 is arbitrary, but $n_1 = \sqrt{N_1}$ works well) and each second stage crossbar has n_2 outputs.[8] Given these parameters, there are $M \cdot N_1/n_2$ first stage crossbars and N_2/n_2 second stage crossbars. These parameters are all shown in Figure 2.16.

A two-stage Richards-Hwang network contains $O(N_2 \cdot N_1^{2/3})$ switches. This can be recursively applied once to create a three-stage switch with $O(N_2 \cdot N_1^{1/2})$ switches. After further recursive applications, this asymptotically approaches $O(N_2 \cdot \log^2 N_1)$ for an arbitrarily large N_1 with many stages.

[8]The value of n_2 is also arbitrary, but an upper bound is determined by the fanout pattern used. The upper bound for n_2 is at least $M(M+1)-1$.

2.5.6 Connecting Multiple PLDs: Partial Crossbar Structures

A partial crossbar is a connection pattern proposed by Butts, Batcheller, and Vargese [93, 94] for use in large-scale logic emulation systems. Such a system is composed of a large number of PLDs which are connected together to emulate a much larger circuit than can be implemented in any single PLD. The partial crossbar network is, in fact, a one-sided Clos network. This can be obtained from the original Clos network by folding the network about the middle crossbar stages and merging the first and third stages so that a separate input and output pin now becomes a single I/O pin. A PLD is connected to the first stage of the partial crossbar. However, since PLDs are designed to freely permute their I/O pins, the first crossbar is unnecessary and can be collapsed into the PLD itself. This process is illustrated in Figure 2.17a) through e).

One problem with a Clos network is coping with rewiring as a small system is scaled up in size. Butts *et al* solve this by using a hierarchy of partial crossbars. In this application, some of the crossbar pins in Figure 2.17e) are not connected to the PLD. Instead, they are reserved to connect to another level of hierarchy formed by another set of crossbars. In this architecture, the number of pins reserved at each level can be calculated according to the wiring locality predicted by Rent's rule [6], for example.

Lewis *et al* [95, 96] makes two observations that reduce the wiring and delay of building these scalable, hierarchical folded-Clos systems. The first observation is that a partial crossbar hierarchy can be flattened to a single physical level while retaining a logical hierarchy. This improves delay significantly. The second observation reduces the wiring in a scalable system by increasing the number of wires within local clusters of the logical hierarchy. The increase in local wiring is shown in Figure 2.17f) with bold arrows. Since there are a different number of wires going to each crossbar, this is called a non-uniform partial crossbar architecture. This reduces the number of crossing wires if crossbar *A* and PLD 1 are partitioned from the remaining system, for example.

Khalid and Rose have proposed a modification to a partial crossbar structure, called a Hybrid Complete Graph Partial Crossbar (HCGP) [97, 98]. The HCGP organisation starts with a hierarchical partial crossbar organisation, but some pins on each PLD are not connected to the crossbars. Instead, they are used to make direct connections to other PLDs

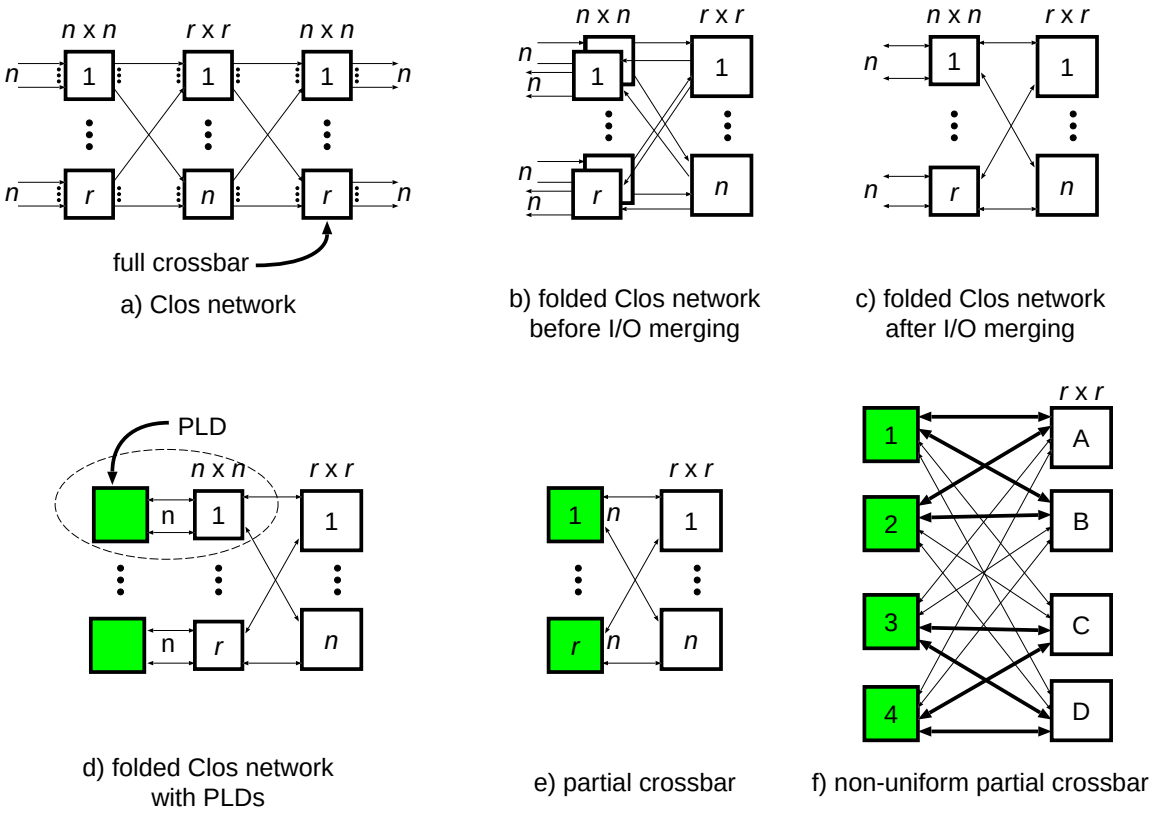


Figure 2.17: Partial crossbar network derived by folding a Clos network.

organised into cliques of four, for example. These direct connections are faster and reduce cost by saving partial crossbar pins, particularly when used for nets without fanout.

Due to its roots in a Clos network, a single-level partial crossbar system is completely routable when all nets contain only two terminals. For this case, Chan and Schlag [99] and Mak and Wong [100, 101] have independently developed optimal $O(n \cdot \log n)$ and $O(n^2)$ routing algorithms, respectively. The multiterminal case can be routed by decomposing nets into two-terminal nets [100, 101], but this heuristic procedure can lead to sub-optimal solutions. Song *et al* route multiterminal nets directly by transforming this routing problem into an SAT problem [102]. Similarly, Ejnioui and Ranganathan [103] and Lin, Lin and Hwang [104] transform the routing problem into an integer linear programming problem. Routing experiments with both heuristic and exhaustive solvers show exhaustive approaches to be practical for moderate size netlists in terms of runtime (taking only seconds to minutes) and the ability to determine if a netlist is unroutable.

2.5.7 Other Network Structures

The study of graphs and connection networks have also intersected to produce structures known as concentrators, superconcentrators, generalizers, and non-blocking graphs. These different network structures are defined by their varying ability to route disjoint paths of connections. Of these, concentrators and non-blocking graphs are most closely related to what is needed for PLD routing. Below, a brief description of these structures and comments on their construction is given.

An (n,m) concentrator is a graph with n inputs and m outputs that can always route any given subset of inputs of size $k \leq m$ to the outputs with disjoint paths. The concentrator is given the freedom to choose which output is reached by each input.[9] An (n,m,c) concentrator is similar, but it only guarantees disjoint routes for input subsets of size $k \leq c$. Concentrators perform the essential function of reducing a large number of candidates down to a few. These structures can be useful to select inputs for a logic function, for example. Sparse crossbars perform concentrator-like functions.

An n-nonblocking graph is a graph with n inputs and n outputs which can realize a disjoint path between each (input,output) pair, where up to n such pairs may be specified. In network nomenclature, it is a rearrangeably non-blocking network.

The best results in the literature for concentrators and non-blocking graphs use $O(n)$ and $O(n \cdot \log n)$ switches, respectively. Effort to reduce these bounds has focused on reducing the constant factor [105]. Generally, constructions of these networks are proven by counting arguments and/or recursive decomposition into other network types (superconcentrators and others). This recursion usually stops at a family of sparse bipartite graphs known as expanders, of which only a few constructions are known [17, 18]. Partial constructions of linear size concentrators are also known [106], but they rely on randomness to complete the construction.

Bipartite expander graphs have the useful property that any subset of the vertices on one side is connected with a greater number of neighbours. This property, known as expansion, guarantees that Hall's condition (defined later in Chapter 4) will always be satisfied. This

[9] A superconcentrator is similar to a concentrator except that an arbitrary subset of the outputs to be used (but not the output order) can be specified.

can be used directly to produce an (n, m, c) concentrator for some $k \leq c$. However, the expander graph constructions in [17, 18] suffer from two restrictions: the number of outputs must be a perfect square, and the number of inputs must be a multiple of the number of outputs. This makes these approaches impractical for general use in PLDs, where the number of inputs and outputs must remain flexible during architecture exploration.

Very recent advances in the explicit construction of expanders have been made by Capalbo *et al* [107]. The construction method works for any number of inputs or outputs and a fixed number of connections. If the method proves to be practical, it will be extremely useful for the creation of sparse crossbars given in Chapter 4.

2.5.8 Summary

Except for the design of concentrators and nonblocking graphs using expanders, all of the network constructions shown in this section have been constructed from full crossbars. The guaranteed full capacity and ability to fully permute the crossbar outputs are essential for the design of these networks. In contrast, PLD routing networks can often tolerate operating at reduced capacity or with relaxed signal order restrictions. One common trend across the many network designs is that the number of switches can be reduced by using more interconnect stages. In PLDs, this is undesirable because it increases delay. Another trend among the structures is they tend to spread their connections across numerous other sub-systems (crossbars). Although the No. 5 crossbar system also uses this approach, it is not done in a sufficiently careful way to build a fully rearrangeable network. This is a good example of why it is important to carefully design the precise connections to create a highly routable network.

Chapter 3

Models, Methodology and CAD Tools

This chapter describes the models, methodology and tools used in this dissertation to evaluate PLD routing networks. The PLD architectural model is described first, including a detailed account of the area and delay models used as performance metrics. Next, the general experimental methodology and CAD tool flow for evaluating PLD performance is given. Last, a description is given of the modifications made to the CAD tool used in the routing experiments.

3.1 PLD Models

This section describes a general model for the PLD architecture being studied as well as the area and delay models used to compute performance metrics.

3.1.1 Architecture Model

A mesh-based network is the primary routing architecture studied in this dissertation. This architecture can be represented by a model that organises the device into clustered logic blocks (CLBs), switch blocks (S), connection blocks (C), and I/O blocks as shown in Figure 3.1. The major architectural parameters for PLDs created with this model are given in Table 3.1. These parameters will be described below.

Each CLB, often called a *cluster*, contains N basic logic elements (BLEs) grouped together. The contents of an $N = 2$ CLB are shown in Figure 3.2. Each BLE contains a

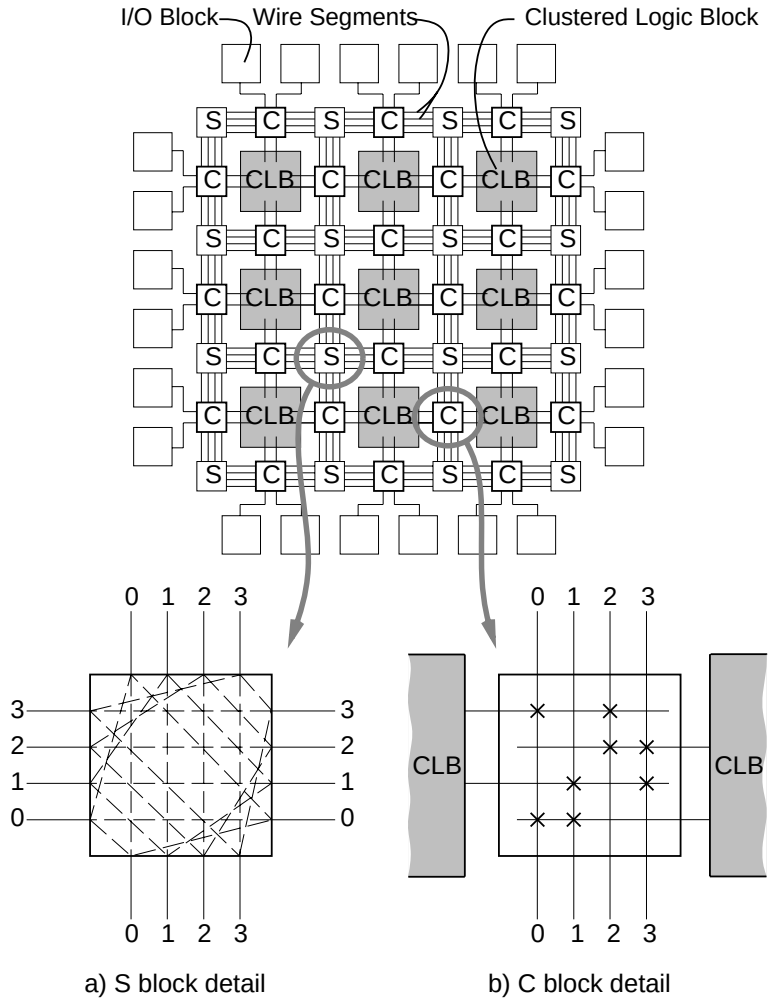


Figure 3.1: PLD architecture model.

Parameter	Description	Typical Values
k	LUT size (number of inputs)	3 to 7 inputs
N	cluster size (number of LUTs)	1 to 10 LUTs
I	number of cluster inputs	4 to 40 inputs
W	routing channel width	10 to 60 tracks
L_{wire}	logical wire length	4 CLBs
F_c	routing channel to cluster input switch density	0.1 to 1.0
$F_{c_{out}}$	cluster output to the routing channel switch density	0.1 to 1.0

Table 3.1: Architectural parameters.

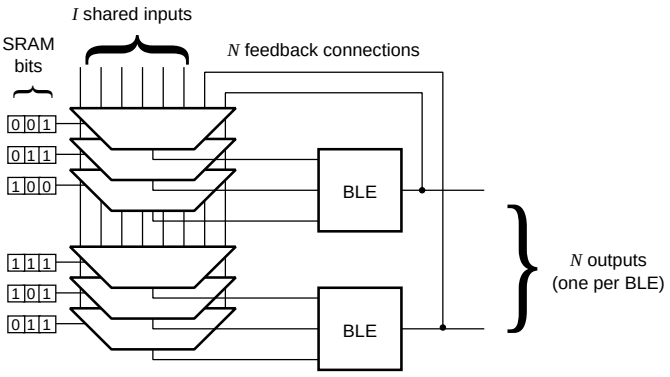


Figure 3.2: A clustered logic block (CLB).

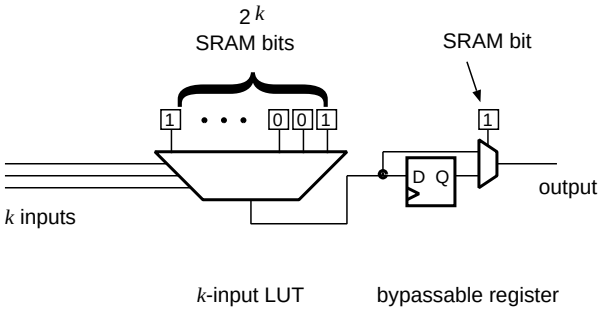


Figure 3.3: A basic logic element (BLE).

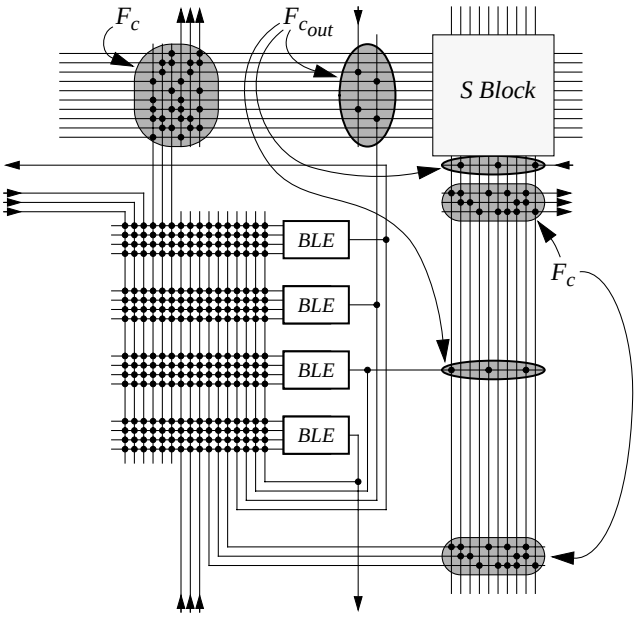


Figure 3.4: Layout tile of a clustered logic block (CLB) and interconnect.

k-input lookup table (a *k*-LUT) followed by a bypassable flip-flop. For example, a BLE with a 3-LUT is shown in Figure 3.3. The LUT inputs are chosen from among a set of I shared cluster inputs. The value of $I = \lfloor k \cdot (N+1)/2 \rfloor$ inputs per CLB is chosen for good packing efficiency [40].

In addition to the I cluster inputs, each LUT may also choose from the N outputs of the BLEs in this cluster. These are called *feedback connections*. The multiplexers that select from these cluster inputs and feedback connections form the *local* or *cluster-level interconnect*. In many cases, this interconnect is *fully connected* or *fully populated*, meaning each LUT input can select from any of these sources.

The interconnect between clusters is formed by the C and S blocks, comprising the horizontal and vertical *routing channels* between the CLBs. The C block is the region where the CLB input and output pins connect to the routing channels. The S block is where connections are made between the horizontal and vertical routing channels, allowing nets to turn corners or extend farther along the channel. Each routing channel contains W parallel tracks of wires, where W is called the *channel width*. The same width is used for all channels, since this was found to produce the lowest-area interconnect [19, 79]. Each track contains a series of *wire segments* placed end-to-end. The logical length of a wire segment, L_{wire}, is equal to the number of CLBs it spans. The portion of a routing channel that spans a single CLB is called a *channel segment*.

Although mesh-based PLD architectures are the primary target, many of the results are not limited to only this style of interconnect. For example, the sparse crossbars designed in Chapter 4 are very important components of hierarchical networks found in CPLDs and some PLDs. Similarly, the circuit design work in Chapter 6 can be applied to other network designs.

Cluster Layout Tile

The architectural model just described can represent a number of different implementations. One important implementation restriction is that the entire PLD array must be constructed by repeating a single cluster layout tile. Figure 3.4 shows the contents of such a layout tile, including the CLB and the interconnect along the top and right edges. The

interconnect on the bottom and left edges is formed by top and right edges of adjacent tiles, so the wiring within a tile must be planned so it will be properly aligned across tile boundaries.

Since all real PLD layouts are created with this tiled concept, most of the architectures evaluated in this dissertation use a single tile. The only exception is some work in Chapter 7, which uses two different layout tiles arranged in a checkered fashion. These two tiles differ only in their S block topology. Considerable effort has been undertaken to explore only architectures that can be built using a tiled layout strategy.

Cluster Connectivity Details

In [19], Betz has shown that it is best to evenly distribute the input and output pins on the four sides of the CLB. Sparse crossbars in the C blocks determine which pins connect to which tracks of the routing channels. These matrices use switch densities of F_c and $F_{c_{out}}$ for the cluster inputs and outputs, respectively. These switch density parameters are listed along with the other architectural parameters in Table 3.1.

The cluster inputs are treated as logically equivalent and hence freely permutable. The LUT inputs are also assumed to be freely permutable. In addition, the LUT inputs are assumed to be fully connected to the cluster inputs and feedback connections in all chapters except Chapter 5.

Each BLE output directly drives a cluster output and a local feedback connection. The BLE outputs are assumed to be logically equivalent, allowing any function to be placed in any of the BLEs of the cluster.

To improve routability, the routing tool takes advantage of the input and output equivalences just described. It may also replicate logic onto multiple BLEs in the same cluster, provided there are empty BLEs available.

Routing Architecture Details

The *baseline routing architecture* used throughout this dissertation is one developed by Betz *et al* in [19]. This architecture assumes that global wires are used to route clock and set/reset signals to the flip-flops. For other types of signals, the general interconnect

contains wires of length $L_{wire} = 4$ CLBs. Half of the routing tracks use buffered routing switches, and the remainder use only pass transistors.[1]

In the general interconnect, the starting points of these wires are staggered such that one-quarter of the tracks start and end in each tile. However, staggering can create a layout difficulty when the number of tracks with the same switch type is not a perfect multiple of L_{wire}. In this case, some additional effort is required to lay out the remaining tracks. Rather than sacrifice area resolution in the experimental results by keeping W a multiple of L_{wire}, this work assumes that such a layout effort is feasible. A commercial implementation might consider padding the channel with a few additional tracks instead.

Unless otherwise stated, the *disjoint* S block is used. This means that signals entering the routing on track number i must remain on that track number until the destination is reached. For C blocks, $F_c = 0.5$ is used when $N = 6$ and $F_c = 0.366$ is used when $N = 10$. As well, $F_{c_{out}} = 1/N$ is used throughout. These switch density choices are made to be consistent with previous work [19, 40]. However, as will be shown in Chapter 5, the precise F_c selection is not too critical.

The baseline routing architecture just described has been shown to achieve good area·delay performance in [19]. It has also been used in other studies [40, 108, 49].

I/O Block Details

The I/O block connects bond pads on the die periphery to the routing channel. Each pad contains a distinct input pin and output pin which are both fully connected to the routing channel. The pitch of the number of I/O pads per CLB tile is set to 5 for clusters with $N = 6$, and to 7 for clusters with $N = 10$. These values are consistent with previous work [19, 40]. Usually, this permits the core area to dictate the PLD array size for each benchmark circuit. However, there are some cases which are pad-limited, such as those involving the *bigkey* and *des* circuits, where the I/O density dictates the array size.

The area results in Chapter 5 were generated earlier and include the area of the I/O pin switches. However, Chapters 6 and 7 specifically exclude these switches from area measurements to be more realistic and more conservative. The reasoning is that the small

[1] The sizes of these routing switches will be considered separately in the individual chapters.

benchmark circuits used here require relatively small PLD array sizes. In larger, modern PLDs, the I/O pin switch area along the periphery is amortized among significantly more CLBs. Hence, the small benchmarks over-emphasize the importance of switches in the periphery. Excluding I/O pin switch area produces more conservative results in Chapter 6 because the proposed new switch designs save more area at I/O pin switches than in the CLB core. Consequently, I/O pin switch area is omitted from these later chapters.

3.1.2 Area Model

As mentioned in the work by Betz *et al* [19], PLD vendors have admitted that transistor area, and not wiring density, is the area-limiting factor. The use of directional wires in Virtex I also suggests that routing area is transistor-dominant and must be reduced. This has led to an area model which estimates the layout area as the sum of area required for every transistor in a PLD. The goal of this model is not to compute the final layout area, but to provide a reasonable way to rank the implementation costs of different architectures.

This model has been used in previous architecture research [19, 40].

Area Units

The area unit is based on the size of one *minimum-size, contactable transistor*, or one T. Figure 3.5 and Table 3.2 indicates the appropriate layout design rules for measuring one T in absolute physical units.[2] By contactable, this means the transistor is sufficiently wide to allow it to be contacted; this may be larger than the minimum diffusion width allowed by the design rules.

This area of one T includes the diffusion area, $W \times L$, plus the separating space to an adjacent transistor, X, in both dimensions. Hence, the area of one T is:

$$T = (W_{min} + X) \times (Z + X).$$

This area unit represents the smallest possible transistor and neglects other types of overhead such as gate contacts or n-well spacing. As will be discussed shortly, this overhead

[2] Due to confidentiality agreements with TSMC, specific values for these design rules cannot be listed here.

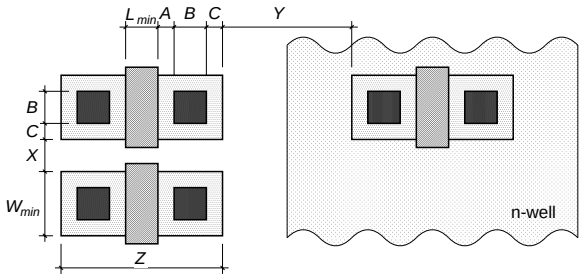


Figure 3.5: Layout design rules for a minimum-size transistor used in the area model.

Parameter	Description
A	minimum poly gate to contact separation
B	minimum contact size
C	minimum diffusion extent past contact
L_{min}	minimum transistor gate length
W_{min}	minimum contactable transistor width
X	minimum separation between similar transistor types
Y	minimum separation between complementary transistor types
Z	minimum overall transistor length

Table 3.2: Layout design rules for a minimum-size transistor area model.

will be treated separately when layout area is estimated.

Counting Transistor and Logic Area

By summing the area used by each transistor in all of the logic structures, the total transistor area of the PLD can be estimated. This is the metric used to compare the area of different PLD architectures.

The *size* of a single transistor, M, is defined to be its width relative to the width of one T. For example, an NMOS device of width W_n would have size $M = W_n/W_{min}$. The *area* of this transistor is computed relative to one T as:

$$TransistorArea = M/2 + 0.5.$$

This calculation assumes that the width of the transistor, $B + 2 \cdot C$, is nearly equal to the separation distance between two transistors of similar type, X.

The area for a logic structure is computed by adding up the transistor area of every transistor t in the logic:

$$LogicArea(\text{logic}) = \sum_{t \in \text{logic}} (M_t/2 + 0.5)$$

where M_t is the size of transistor t. These area metrics are process-independent because the values of *TransistorArea* and *LogicArea* are constant as technology scales.

All of the logic structures in the PLD core are modeled, including the BLEs, the LUT input multiplexers, and the cluster routing, but not the padframe containing the bond pads. The area contribution of a pass transistor, for example, depends on the transistor width, and a buffer chain depends on the number of inverter stages as well as the drive strength of each stage. The area of an SRAM configuration bit is held fixed at 6 T.

Layout Area

In the timing model (discussed below), the absolute physical length of a wire must be estimated to calculate its RC properties. This requires the physical size of one T in the target semiconductor process, as well as an estimate of additional overhead, such as connections between transistors or the separation distance between NMOS and PMOS devices, Y. The amount of overhead can also vary depending upon the experience of the layout artist. To account for these factors, overhead is encapsulated in the following parameter:

$$LayoutEfficiency = 60\%.$$

This value, which is based upon the area estimates of actual PLDs, is also used in [19].

The unit transistor area and layout efficiency values are used to compute an estimate of layout area as follows:

$$LayoutArea(\text{logic}) = LogicArea(\text{logic}) \times \frac{T}{LayoutEfficiency}.$$

As well, the length of a CLB layout tile is computed using the formula:

$$L_{tile} = \sqrt{LayoutArea(\text{CLB})}.$$

3.1.3 Delay Model Calculations

The delay calculations used in the CAD flow involve two separate steps. First, the delay for each net is computed using an Elmore delay model. Second, an analysis phase determines the overall critical path by summing the net delays and tracing signal paths. The same delay model calculations are used in [19]. These two steps are described in greater detail below.

Interconnect delays for each net are computed using the Elmore delay method [109], which works well for RC trees. In this computation, wires and unbuffered switches are modeled as RC elements. In addition, the Elmore delay computation is augmented to allow buffers inside the RC tree [110]. This involves replacing each buffer with a constant delay element, a voltage source, an output resistance, and input and output capacitors.

At the inputs and outputs of a CLB, the Elmore delay computation ends and a constant delay model is used instead. The delay of each component within the CLB is represented by a constant value. For example, the delay from a cluster input to a LUT input, and the delay through a LUT are set to different constant values. The delay of a path within a CLB is computed by summing the individual component delays — this may include multiple LUT delays and the setup time for a register, for example.

For the critical-path computation, the delay of all paths from all primary inputs or register outputs to the primary outputs or register inputs is traced. During this forward-scan, each path delay is determined by an accumulation of the interconnect and CLB delays. Of these paths, the one with the longest delay is termed the *critical path*.

The next section describes the way in which the various delay model parameters are computed.

3.1.4 Delay Model Parameters

The delay model requires resistance and capacitance values for the interconnect wires, as well as constant delays for individual CLB components. The CLB delay values are taken from the worst-case propagation delays simulated using HSPICE. The technology targeted is a 0.18μm TSMC semiconductor process. Due to confidentiality agreements with TSMC,

precise timing or process characteristics cannot be disclosed.

The metal interconnect wires are assumed to be drawn in the metal 3 layer at minimum-width and at twice the minimum spacing. Betz *et al* [19] has shown that these feature sizes produce good delay and area·delay results in 0.35μm TSMC technology, so they are also employed here.

The CLB timing and interconnect RC values used in Chapter 5 are the same as those used by Ahmed [41], which is based on the same 0.18μm process technology used in this dissertation. These results, generated using the procedure described in [19], also make the following circuit implementation assumptions. First, the gate voltage of pass transistors is boosted to 2.1V (from the regular 1.8V) to reduce static power. Second, interconnect wire RC parameters are set to fixed values for a given k and N. These are calculated *a priori* according to the average cluster tile length across the benchmark suite. Third, the routing switch sizes are scaled linearly in size according to the average tile length. This scaling technique, also employed by Marquardt *et al* in [111], increases the delay and area of routing switches used in larger tiles. More information about switch scaling is given in Chapter 5.

In Chapters 6 and 7, some of the circuit implementation assumptions just described are abandoned and this changes the delay model parameters. The work in Chapter 6 redesigns the interconnect switches with level-restoring instead of gate-boosting to improve long-term semiconductor reliability. This lowers overall interconnect performance, partly due to the level-restoring circuits and partly due an increase in pass transistor resistance. The chapter also investigates the optimum switch size for different RC loads and shows that the switch scaling proposed by Marquardt is unnecessary. Hence, these chapters use a fixed switch size for all cluster sizes. Furthermore, better CAD tool integration of the area model allows the interconnect RC parameters to be estimated for every generated architecture. This means that changes to the architecture, such as channel width or switch density, may result in a change to the resistance and capacitance of interconnect wires. The work in Chapter 7 incorporates all of these changes.

3.2 Experimentation and CAD Flow

The evaluation of different PLD architectures in Chapters 5, 6, and 7 is based on an experimental process of mapping benchmark circuits into the architecture. After a successful mapping, the area and delay models are used to estimate area of the architecture and the critical-path delay of the mapped benchmark circuit. Details of the flow, the routing step, and a procedure used to determine the amount of routing effort are given below.

3.2.1 Overall Flow

The CAD flow for the experimental procedure is shown in Figure 3.6. The process begins with a benchmark circuit and a logic synthesis step which optimises and transforms the circuit into an appropriate netlist form. This netlist is then placed and routed in the target architecture. More details about each of the steps are given below. In general, all steps are performed with the goals of reducing both area and delay.

The benchmark circuits selected are the twenty largest circuits from the LGSynth93 benchmark suite available from the Collaborative Benchmarking Laboratory [112]. This benchmark suite is commonly known as the MCNC suite.

The circuits are prepared for numerous routing experiments by performing the logic synthesis and placement steps ahead of time. First, technology-independent optimization using SIS [113] is done. Then, the technology mapping tools FlowMap and Flowpack [114] convert the logic netlist into k-input lookup tables and registers. Next, the T-VPACK algorithm [115] groups these LUTs and registers into CLBs containing N BLEs. The resulting netlist of CLBs is placed using VPR version 4.30 [116, 19]. This version includes the latest timing-driven enhancements from [117]. The placement step fixes a location for each CLB in the smallest square PLD array size possible (as limited by the benchmark's CLB or I/O pin count).

The above netlist optimization and placement steps are done only once per circuit for a given combination of k and N values. Experiments use several lookup table sizes, with k varying from four to seven, but usually the cluster size is fixed at $N = 6$. Results by Ahmed [40] show that this cluster size produces good area·delay results for all of these

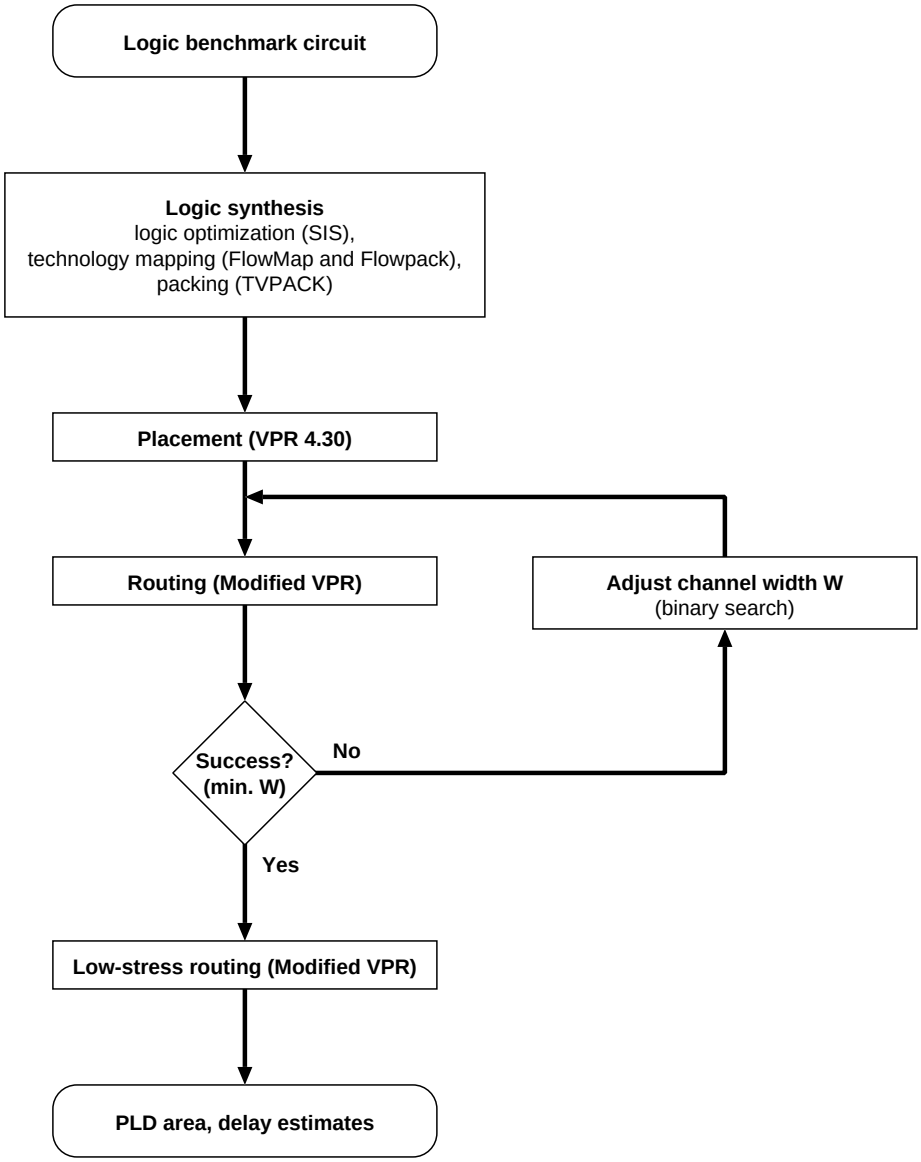


Figure 3.6: Experimental process used to evaluate PLD architectures.

lookup table sizes.

The last steps of the CAD flow involve routing the placed netlist so that area and delay estimates can be computed. The routing step is discussed next.

3.2.2 Routing Step

Routing a netlist involves assigning a series of wires and switches to every net so that all of the connection requirements are satisfied. The router software used is a extended version

of the VPR 4.30 tool called **VPRx**. The modifications are described later in section 3.3.

The routing step is an iterative process. First, a netlist is routed in a PLD architecture with a given channel width. If the channel width is too small, the router will fail to find a solution. After each iteration, the channel width is adjusted in a binary search fashion according to routing failure or success. Increasing the channel width makes the PLD larger and easier to route. Eventually, this procedure determines the minimum number of tracks required to route the benchmark circuit, W_{min}.

To ensure this is a true minimum channel width, *i.e.*, there is not simply an isolated failure at $W_{min} - 1$ tracks, channels with 2 and 3 fewer tracks than W_{min} are also routed to verify that they are unroutable. Should one of these be routable, W_{min} is repeatedly adjusted down by one until there are 3 failures in a row. In general, this procedure sometimes lowers W_{min} by two or three tracks in only a few routing cases (perhaps fewer than 5%). However, the importance of this procedure is not to reduce W_{min} for these uncommon cases. Instead, it is designed to prevent anomalous cases where a very large channel width is accepted as W_{min} because the router or architecture encounter a pathologically bad case. Such anomalies were encountered by experiments performed in Chapter 5.

Since a channel width of W_{min} places the PLD architecture at the brink of being unroutable, a low-stress route is performed with 20 or 30% additional tracks to produce the area and delay results. This is akin to logic designers (and PLD architects) following conservative design practices by making allowance for future design changes. From this low-stress routing solution, area, delay, and area·delay metrics are computed.

Nearly all results are reported as geometric averages for the 20 benchmark circuits. The specific area and delay numbers for each benchmark are computed as follows. The delay of a benchmark circuit is computed according to the critical-path calculation presented earlier. The area is computed as the *LogicArea* of a CLB tile (*i.e.*, including the area of the routing channels) times the number of CLBs required to implement the benchmark's logic. We call this the *active area* of the benchmark because it excludes CLBs in the array that remain unused. Ahmed [40, 41] reports that this method improves area resolution and better reflects changes to the packing algorithm.

3.2.3 Determination of Router Effort

In general, the packing, placement and routing tools are executed in timing-driven mode using their default parameters. However, experience gained from early experiments suggested that the default amount of routing effort is too low. This is usually observed as noise in the results when an architectural parameter is varied.

Two key router parameters control the amount of effort: the maximum number of router iterations, and the penalty cost applied when nets share wires. The maximum iteration count is directly controlled with the `max_router_iterations` parameter. The penalty cost geometrically increases from iteration to iteration by a factor provided with the `pres_fac_mult` parameter.

To determine appropriate values for these two parameters, a number of routes were conducted with the maximum number of router iterations fixed at 300 as `pres_fac_mult` was varied between 1.02 and 2.0 (the default value). All benchmarks were routed multiple times as one architectural parameter was varied.[3]

The results from the low-stress routing with 30% more tracks than W_{min} are shown in Figure 3.7. This figure shows that average router runtime increases rapidly as the value of `pres_fac_mult` (shown along the *x*-axis) is made smaller. The average critical path delay across all benchmarks is also shown. Error bars indicate the minimum and maximum values in the **average** delay[4] as the architectural parameter is varied. This variation is primarily noise: it is roughly 10% for large `pres_fac_mult` values, but it diminishes significantly when the `pres_fac_mult` value drops below 1.4. A value of 1.3 generates fairly consistent results without a large runtime increase.

Figure 3.8 plots the same average runtime data along with the number of router iterations needed to find a solution. On average, the number of iterations increases from 23 to 160 in a manner that closely follows the increase in runtime. This suggests that the default VPR value of 30 iterations is far too low. More importantly, the *maximum* number of iterations does not go significantly above 100 except when `pres_fac_mult` is small. Hence,

[3]The architecture used here is a cluster of ten 7-input LUTs, with a 43% switch density in the local cluster routing. The number of spare inputs is varied from 0 to 20.

[4]The average is taken across all circuits.

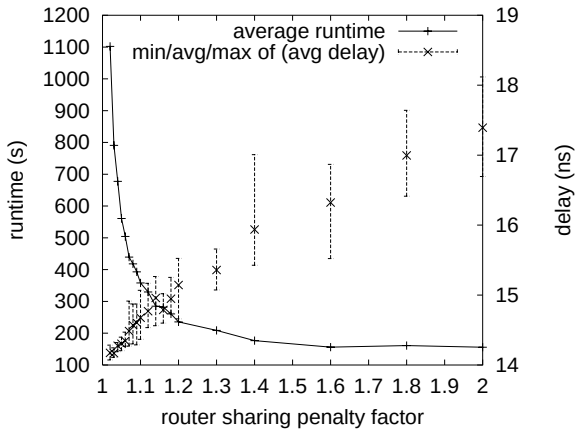


Figure 3.7: Runtime and variation in critical path delay.

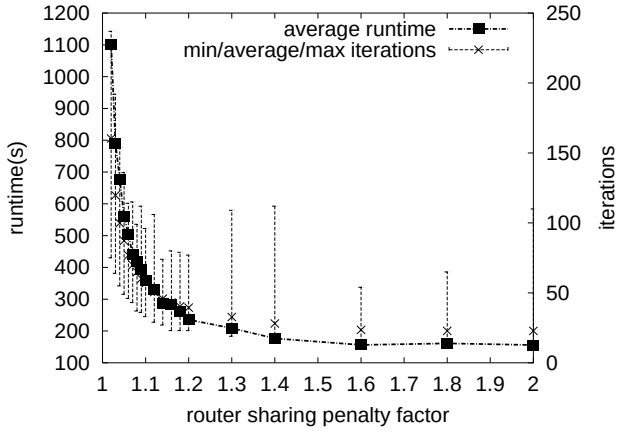


Figure 3.8: Runtime and number of router iterations.

a reasonable value for `max_router_iterations` is 100 iterations.

The amount of effort used for the routing experiments in this dissertation has been selected from these graphs and is summarized in Table 3.3. Experiments in Chapter 5 use a large value of 250 for `max_router_iterations` during the binary search to produce very high quality routing solutions. However, this makes the binary search runtime too large, so a smaller value of 100 is used in Chapters 6 and 7. Hence, the router computes slightly larger W_{min} values in these later chapters. For all binary searches, a `pres_fac_mult` value of 1.3 is used. For the low-stress routing, a `pres_fac_mult` value of 1.05 is used in Chapter 5. This is increased to 1.1 for the later chapters to further reduce runtime. It should be noted that the maximum iteration count used for low-stress

Chapter	Routing Step	Value of `max_router_iterations`	Value of `pres_fac_mult`
5	binary search	250	1.3
5	final route	300	1.05
6 and 7	binary search	100	1.3
6 and 7	final route	300	1.1

Table 3.3: Amount of routing effort used for all experiments.

routing must be high enough that the routing doesn't fail, so a value of 300 is used. Since a solution must be found, this choice does not affect runtime. The final values chosen for use in Chapters 6 and 7 are suitable default values for most future VPR and VPRx experiments.

3.3 VPR Extensions (VPRx)

This section describes a number of extensions that have been made to the VPR router to create VPRx. The first section describes changes which are used throughout the entire dissertation. The remaining sections describe changes that only apply to Chapters 6 and 7.

3.3.1 Routing Graph and Netlist Changes for Sparse Clusters

The routing graph used by VPR to represent the routing architecture has been extended so that sparse crossbars can be used within the local cluster interconnect. The original graph represented all of the interconnect wires and switches between CLBs, ending at the CLB inputs and outputs. The VPRx graph includes all wires and switches up to the BLE inputs and outputs. A similar change to the timing graph and netlist structure has also been made. The netlist change is significant because sinks are now recorded for each BLE input pin rather than for each cluster input pin.

Since routability inside a cluster is no longer guaranteed, additional netlist changes are necessary for local feedback connections. The original VPR removes feedback connections that remain within a cluster. Thus, when the net is entirely local to the cluster, the entire net is removed. In contrast, VPRx must keep these connections in the netlist and explicitly route them.

To improve routability, the restriction that a net use only one cluster input pin has been removed. This is sometimes necessary to avoid blockage when a net must connect to multiple BLEs in the same cluster. Similarly, local feedback connections might become blocked while routing, so they are allowed to leave the cluster and re-enter through a cluster input pin. This increases the demand for cluster input pins.

The above modifications require significant changes to the VPR code base which are necessary for Chapter 5. The experiments in Chapters 6 and 7 also use these code modifications, even though the cluster interconnect is kept fully populated there.

The effectiveness of the VPRx router has been validated against VPR 4.30. Both routers obtained similar delays, channel widths and area results for fully populated clusters using a variety of cluster and LUT sizes.

3.3.2 Architecture Model Change

The work in Chapter 6 introduces an architecture feature called *output pin merging* which moves some output pin connections directly into S block switches.[5] By connecting CLB outputs directly to the S block switches (according to the C block connection pattern), a small amount of area (about 1%) is saved.[6] VPRx automatically calculates this savings whenever it can be applied.

3.3.3 Area Model Changes

The area model calculations in VPRx have been upgraded to compute the area of logic structures inside the cluster, including the LUTs, registers, and even sparse local cluster interconnect. Previously, this computation was performed using a separate tool. The integration of these two tools allows VPRx to directly estimate the size of the cluster layout tile and recalculate interconnect parasitics as the channel width changes.

Another change to the area model calculations involves the scaling of buffers inside the CLB to match load conditions. Previously, fixed buffer sizes were used. Instead, the loads

[5] Output pin merging requires the use of *fan-in based routing switches* which are introduced in Chapter 6.

[6] This savings is obtained by replacing the wide transistor used on the output of a tristate driver with a small transistor on the input multiplexer of a routing switch.

are estimated and buffers are scaled appropriately. The remainder of this section describes these buffer scaling rules which have been developed from numerous HSPICE simulations [Ahmed and Wilton, *private communication*].

Buffers that drive multiplexer select lines, such as the buffer driving a LUT input, are sized according the total gate width they drive. A buffer of size B is used to drive a total gate width of $8B$. Since multiplexers are assumed to be constructed from a tree of pass transistors, the select lines controlling the leafs of the tree are more heavily loaded and require larger buffers. These buffers are now scaled appropriately.

Buffers that drive multiplexer data inputs, such as CLB input buffers, typically have more than one multiplexer as a load. These buffers are sized according to the total diffusion width they drive. This type of load is larger (per unit width) due to the depth of the multiplexer tree. To simplify calculations, the actual depth of this tree is ignored. Instead, a size B buffer is used when the first level fan-out of the buffer is loaded by a total diffusion width of $2B$. However, the buffer size is limited to be at least size $B = 7$ and at most size $B = 25$.

The buffer sizing rules just described result in the following changes to CLB area compared with previous work [19]. The new LUT area is slightly smaller because the previous work uses the largest buffer size for all LUT inputs. Also, the new CLB input buffer area is slightly larger because the previous work uses only fixed-size buffers of size $B = 4$. Overall, the new calculations produce a slightly larger total CLB tile area than before.

3.3.4 Delay Model Improvements

Two important extensions have been included in VPRx to improve the accuracy and fidelity of timing calculations. First, the area model is used to compute the cluster tile length and scale interconnect wire RC values accordingly. Hence, architectures with wider channel widths or more local cluster interconnect will encounter higher routing delays.

Second, the Elmore delay computation now accounts for fan-out delay at a buffered routing switch. This was not done previously because each fan-out was assumed to have its own private buffer. In this work, the switch fan-outs are assumed to share a buffer but have independent pass transistors for tristate control. To compute the delay when multiple

pass transistors are enabled, an RC node is added immediately between the buffer and the pass transistors. Experiments show that this increases benchmark critical-path delay by 5% on average or 16% in the worst case.

These changes to the Elmore delay computation are made in two places: during timing-driven wavefront expansion and during post-routing static timing analysis. The change during routing expansion involves two separate calculations: a) degrading the delay of sinks that have already been routed and b) degrading the delay of the sink that is about to be routed. The calculation in step a) is important because the routing of subsequent sinks may join-in at any previously routed portion of the net. The calculation in step b) is important to make correct routing trade-off decisions (in particular, fanout effects) while routing the current sink.

3.3.5 Runtime Improvements

To improve the router runtime, a number of other changes have been made to the router heuristics: the nets are routed in order of decreasing fan-out, sinks in the same cluster are routed consecutively, and significantly fewer iterations are made when the circuit is very difficult to route. Further details about these runtime improvements are as follows.

The netlist is now sorted so that high fan-out nets are routed first. Swartz [118, 119] found this reduces runtime by 23% and delay by 11%. The fan-out runtime problem is exacerbated in VPRx because there are more sinks per net.

To further improve runtime, sinks in the same cluster are now grouped and routed consecutively. After the first sink in a group is routed (the most critical), the heap responsible for wavefront expansion is re-seeded with only three wires, namely those found while tracing back from this sink to the source. Wavefront expansion then proceeds for the next sink. This is repeated again for each remaining sink in the group. By seeding the heap with precisely three wires, expansion uses the cluster input pin and two (possibly perpendicular) channel wires. Runtime is reduced in two ways: a) re-seeding the heap is faster (only three wires are added instead of the entire route tree), and b) wavefront searches are limited to a small, local region. This speeds the search for the remaining grouped sinks.

Another router change detects very hard to route cases and significantly improves bi-

```
isUnroutable( iteration i, num_shared_wires[1..i], max_iterations )
{
   // trivial checks, ensures enough effort is expended
   if( num_of_shared_wires[i] <= 50 )
      return false;     // few shared wires, keep trying
   if( i < 20 )
      return false;     // too few iterations, keep trying

   // estimate number of iterations required
   // using averaged linear extrapolation from the last 20 iterations
   // averages over 10 iterations reduces noise & hill-climbing effects
   shared_wire_count_recent = average( num_shared_wires[i-9 to i]     );
   shared_wire_count_past   = average( num_shared_wires[i-19 to i-10] );
   shared_wire_decrease_rate = ( shared_wire_count_past -
                                 shared_wire_count_recent ) / 10;

   iterations_remaining = num_shared_wires / shared_wire_decrease_rate;
   if( i + iterations_remaining <= max_iterations )
      return false;     // prediction that sharing will be resolved

   // too many iterations required, declare unroutable
   return true;
}
```

Figure 3.9: Early detection of unroutable architectures.

nary search runtime. When the channel width is far too small, the router wastes time by executing the maximum number of iterations before declaring it unroutable. By tracking the number of shared routing wires (*i.e.*, routing violations) after each iteration, these unroutable cases can be detected earlier. To do this, the unroutable detection procedure described in Figure 3.9 is executed after every iteration. If it returns true, the remaining iterations are skipped so the binary search can try a wider channel width. The parameters of this procedure have been chosen after plotting the number of shared wires per iteration for numerous routing experiments. These specific values prune cases that are clearly unroutable, but continue routing cases that are marginally routable.

3.3.6 Experimental Noise Reduction

The original VPR router exits as soon as a valid routing solution is found, but there is sometimes a significant increase in delay in the last few iterations when the router is focused on eliminating congestion. This is a source of noise in the delay results. The noise can be pronounced enough to obscure the effect of varying an architectural parameter.

To reduce delay noise, VPRx finds up to five valid routing solutions by making addi-

tional iterations (up to the maximum allowed). The first routing solution which is within 5% of the average delay for all previous iterations is kept. Otherwise, the lowest delay of the five solutions is used.

This noise-reduction technique is a simplification of the one employed by Roopchansingh [49]. That work further reduced noise using two additional techniques: averaging results from five different circuit placements, and restarting the router with a lower router sharing penalty cost, `pres_fac_mult`. The experiments here use a very low `pres_fac_mult` already, so lowering it further or using numerous placements would increase runtime impractically. Likewise, using multiple placements would also increase runtime impractically.

3.3.7 Correctness Changes

There are three minor changes made to the VPR 4.30 code to solve correctness problems. First, when adding a new sink to the routing tree, VPR would sometimes add a new branch that includes a node already found elsewhere in the tree. This creates a reconvergent path in the routing solution. To avoid this problem, VPRx checks for these reconvergent nodes and does not add the redundant portion to the tree.

Second, VPR does not generate a routing architecture which can be constructed with a single layout tile when $F_c < 1.0$. The problem arises because a fixed sparse switch pattern is used for each C block, but different S blocks are used to stagger the wire endpoints. This is corrected in VPRx by adjusting the C block switch pattern according to wire endpoint locations in the adjacent S block. Unfortunately, due to the complexity involved, this change is presently hard-coded for length four wires only.

Third, VPR will abort with an error if too many iterations are executed and an overflow occurs in the sharing penalty cost. In this case, VPR incorrectly declares a problem with the routing architecture. To handle this more correctly and gracefully, the maximum number of iterations is automatically lowered in VPRx. This way, the overflow is completely avoided. Instead, the architecture is found to be unroutable and routing can continue with a wider channel width.

Chapter 4

Sparse Crossbars

This chapter presents a method for evaluating and constructing sparse crossbars so they are both area efficient and highly routable. The evaluation method uses a Monte Carlo technique to estimate the percentage of random test vectors that can be routed. The routability of each test vector is determined perfectly using a network flow algorithm. The construction method attempts to maximise the spread of the switch locations, such that any given subset of input wires can connect to as many output wires as possible.

The hardest test vectors to route are those which attempt to use all of the crossbar outputs. Results indicate that area-efficient sparse crossbars can be constructed by using more outputs than required and a sufficient number of switches. Using a few specific case studies, it is shown that sparse crossbars with about 90% fewer switches than a full crossbar can be constructed, and these crossbars are capable of routing over 95% of randomly chosen test vectors. In one case, a new switch matrix which can replace the one in the Altera FLEX8000 family is shown. This new switch matrix uses approximately 14% more transistors, yet can increase the routability of the most difficult test vectors from 1% to over 96%.

4.1 Introduction

Programmable logic devices commonly use full crossbars and sparse crossbars as building blocks in routing networks. A full crossbar is often chosen when a highly-routable cross-

bar is desired, and a sparse crossbar is selected when area use is most important. Sparse crossbars have significantly fewer crosspoints or switches than a full crossbar, but their routability is not well understood. This naturally brings up the question, “Is it possible to get the best of both worlds?” There are many instances where a highly routable crossbar would be preferred, but the area cost of a full crossbar is prohibitive. This will be illustrated with the following examples.

In large-scale logic emulation research systems like Teramac from Hewlett-Packard Labs [120], or in commercial systems such as those by Quickturn [94], circuits are partitioned across a large number of PLDs. Each of the generated subcircuits must be successfully placed-and-routed in the PLD, otherwise the entire circuit must be re-partitioned, re-placed and re-routed. It is important that each of the PLDs in these large emulation systems are highly routable to avoid this time-consuming iteration. For example, there are 1728 Plasma PLDs [121] used in Teramac. The design goal was to completely place and route each PLD within 3 seconds. Plasma would have used full crossbars to guarantee this routability, but to save area it was necessary to use only 1/4 of the switches. Results in this chapter will show that sparse crossbars with a switch density of only 1/28 achieve better routability than the switch pattern chosen for Teramac.

Using highly routable components in a single PLD may also lead to the benefits of reduced compute time and memory use. The latest PLDs by Altera and Xilinx have a large number of LUTs and wiring resources. However, to route these devices, CAD tools usually build complex data structures to represent all of the wires and switches as well as the nets that must be routed. This leads to considerable memory use. For example, Altera recommends using 1GB of RAM to route designs for the APEX 20K1000E device [122]. It may be possible to make the CAD tools more efficient by following the Teramac and logic emulation system model: partition a circuit into smaller subcircuits, then place and route each piece independently. To do this effectively without rip-up and re-partitioning, there must be confidence that each subcircuit is likely to route.

As another example, CPLDs are required to be highly routable because they are often close to 100% utilised. However, full crossbars are not normally used in the global interconnect of CPLDs due to the area overhead involved. Instead, an area-efficient sparse

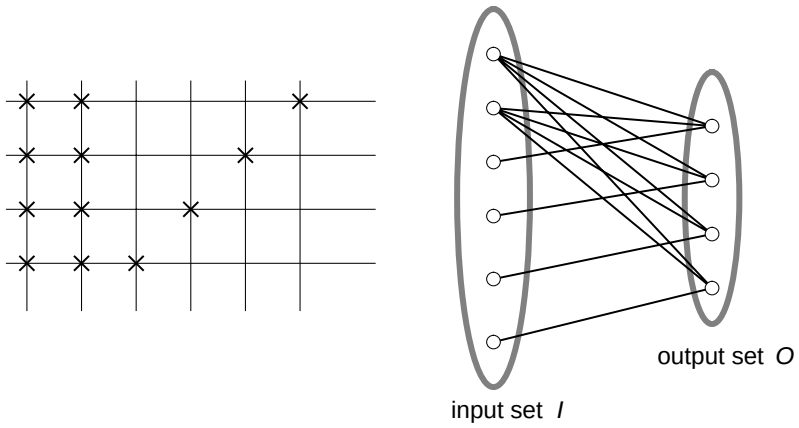


Figure 4.1: A 6×4 minimal crossbar and its graph representation.

pattern is required.

The above scenarios indicate that highly routable, sparsely populated crossbars would be useful, yet there is little published work about how to build them. This chapter addresses this issue by describing conditions necessary for routability (Hall's Theorem), a method for evaluating routability without resorting to place-and-route experiments, and a construction algorithm that achieves good performance. Results for a few design cases will exemplify the area requirements and routability obtainable from these sparse crossbars. In general, however, the construction algorithm and evaluation method will work for any number of inputs, outputs, or switches.

4.1.1 Graph Representation

Crossbars are easily modeled as a graph when wires are represented by nodes and switches are represented by edges. A crossbar forms a bipartite graph G composed of two sets of nodes, input wires I and output wires O, and a set of edges. There are no edges connecting the nodes within each set, but an edge can exist between any node in set I and any node in set O. An example of a 6×4 minimal crossbar and its graph is shown in Figure 4.1.

4.2 Evaluating Routability

The traditional approach to evaluate the routability of a PLD, and hence evaluate the sparse crossbars contained therein, is to run place and route experiments with a suite of benchmark circuits. This is an effective method to design a PLD and its CAD tools in concert, but it can be a lengthy process. As well, the routing performance of the crossbars in the PLD relies upon the effectiveness of the CAD tools and the benchmarks to exercise the architecture.

It is desirable to quickly test the routability of sparse crossbars independently of the CAD tool or benchmark circuits used. As well, a test should provide a more sensitive, yet still practical, measurement of routability than obtainable with traditional experiments. This approach also helps avoid the problem of "training" a PLD architecture or CAD tool to a particular benchmark suite.

One possible routability metric is the maximum guaranteed capacity of a crossbar, c. With this metric, the largest value c is found such that any subset $I' \subseteq I$ of size $|I'| \leq c$ is guaranteed to be routable. The main problem with this metric is that it is very difficult to compute: an exhaustive search will have exponential complexity because it must examine all subsets of I with cardinality c or smaller. A branch-and-bound algorithm to search for this value was implemented, but it is only practical for the smallest of crossbars. A greedy heuristic search was also implemented to find a group of c' inputs which are unroutable, hence proving that $c \leq c'$. However, the results of this greedy search are not consistent enough to compare the quality of two different switch patterns.

Instead, the routability of a crossbar is measured using a Monte Carlo test. For this test, a number of random test vectors are generated, each of which is routed on the crossbar using a network flow algorithm. The routability of the crossbar is then computed as the percentage of test vectors which can be successfully routed.

A test vector is a set of k inputs that must be routed to the crossbar outputs. More specifically, it is a subset of the input wires, $I' \subset I$, where $|I'| = k$. In terms of actual PLD routing, a test vector resembles the case where logic signals have already been assigned to specific wires due to previous routing restrictions.

Routability can be evaluated as a function of the test vector size, k, to distinguish

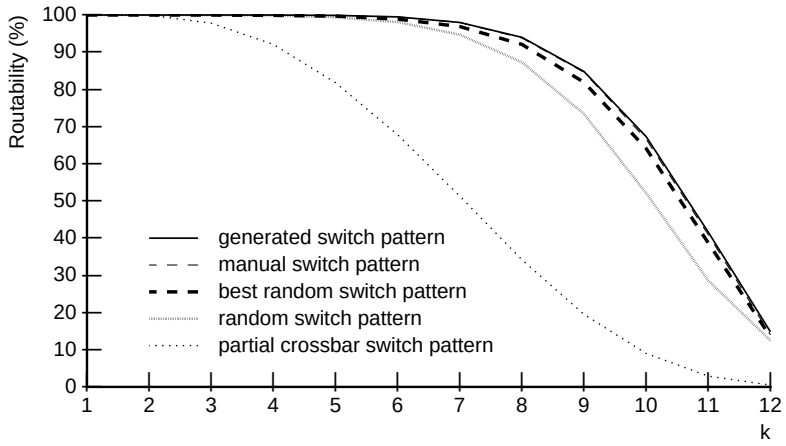


Figure 4.2: Routability of different switch patterns in a 80×12 sparse crossbar.

the easily routed vectors, *i.e.*, when k is small, from the difficult ones. For example, the routability of various 80×12 sparse crossbars with 160 switches is shown in Figure 4.2. In this graph, the x-axis is the test vector size, and the y-axis is the routability. Five curves are drawn representing each of five different switch patterns. A description of these specific patterns is given Section 4.4.

A *highly-routable* sparse crossbar is one which can route as many of these precon-strained test vectors as possible. In this dissertation, the *highly-routable* point is arbitrarily defined as being able to route at least 95% of the hardest test vectors, *i.e.*, those containing the maximum number of signals intended to be carried by the crossbar. The assumption is that if a sparse crossbar can route nearly any configuration that it is given, then it will perform nearly as well as a full crossbar when used in a PLD.

A network flow algorithm [123] is used to route the test vectors because it is *guaranteed* to find a routing solution if one exists. The process for setting up the flow network and solving it is as follows. Figure 4.3 gives an example of the flow network that must be constructed around a 6×4 minimal crossbar to be tested. In addition to the crossbar nodes and edges, a source S and a sink T are added. Edges are added connecting S to the inputs I and the outputs O to T. Each edge is labelled with '1/0' to denote it has unit capacity and initially carries zero flow. For clarity, not all labels are shown in the figure. To solve the flow network from S to T for maximum flow, integral amounts of flow are added to

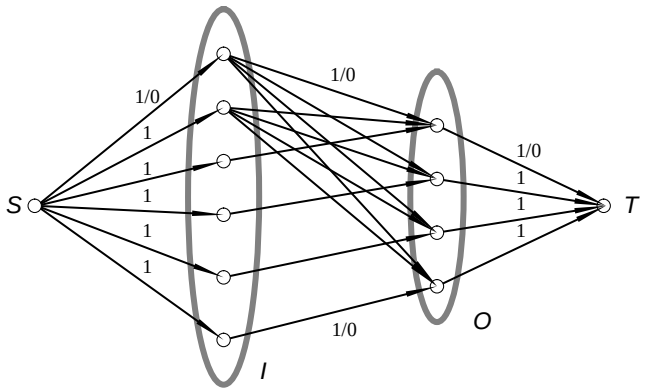


Figure 4.3: Flow network used to test the routability of a 6×4 minimal crossbar.

each edge in a systematic way such that each capacity is never violated. The final solution includes a number of edges that carry unit flow — these are the switches that must be turned on to form a routing solution.

Clearly, the maximum flow that can be achieved from S to T is restricted by the number of inputs, outputs, and the arrangement of the edges in the sparse crossbar. To test the routability of a given test vector, the input set I can be restricted to include only those nodes specified in the test vector. This is done by assigning a zero capacity to those edges leading from S to I that are not specified in the vector. Hence, a test vector of size k creates a total of k unit-capacity edges leading from S to I. An overall flow of size k must be found in this network to produce a solution containing k switches. If a lower flow value is found, it indicates the largest number of inputs (and outputs) that are routable. This guarantee of finding a solution makes it an ideal routing tool.

4.3 Routable Switch Patterns

This section examines the following basic problem: given p switches, how should they be placed in a sparse crossbar to make it as routable as possible? The foundations for the switch-placement algorithm presented in the next section are based on the following theorem and observations.

4.3.1 Hall's Theorem

Hall's Theorem [16] is a result that can be applied to bipartite graphs to determine whether a matching that saturates the smaller partition can be found. A matching is a subset of the edges in a graph such that no two edges share a node. Hence, every pair of edges in a matching involve 4 distinct nodes. Let X be any subset of vertices in a graph G. A matching saturates X if it contains an edge to every node of X. Hall's Theorem gives the precise conditions under which such a matching can exist.

Hall's Theorem. If G is a bipartite graph with bipartition X and Y, $|X| \leq |Y|$, then G has a matching of X into Y which saturates X if and only if

$$\forall S \subseteq X, \quad |S| \leq |N(S)|$$

where $|S|$ denotes the cardinality of subset S, and $N(S)$ is the set of neighbours of S in Y.

4.3.2 Application of Hall's Theorem

Hall's theorem can be applied to sparse crossbars because a matching actually forms a routing solution. The Y set represents the output wire set O, and X is a specific test vector or subset of the input wires, $X = I' \subseteq I$. A test vector is fully routable if and only if Hall's condition is satisfied and a saturating matching can be found. A saturating matching is desired because every test vector node (input wire) must match a unique output. The edges in the matching are the switches which must be turned on to form the connections.

To design a routable sparse crossbar, switches should be placed so that Hall's condition is satisfied for as many test vectors as possible. For test vectors of size k, it is a necessary condition that at least k distinct output wires are reachable by switches.

The switch placement algorithm described in the next section assumes that the switches connected to any specific subset of input wires should be spread out to connect with as many output wires as possible. This is equivalent to making the neighbour set $N(S)$ as large as possible so that Hall's condition is likely to be satisfied.

Switch placement is not trivial because the switch pattern chosen for each input wire interacts with the other input wires. Given one subset of input wires, the objective is to

spread the switches to as many different outputs as possible. This may cause a different subset of inputs (containing a few inputs from the original subset) to reach too few outputs.

The above example also suggests that each input wire, having equal likelihood of being a part of any particular subset, should have an equal number of switches. This is supported by further reasoning as follows. If one input has fewer switches, it would not be able to "spread out" to as many different neighbours. As a result, subsets which included this input may be less routable. To get around this, all inputs should have a similar number of switches so the fan-outs of the input wires are roughly equal.

A similar argument implies that the fan-ins of the output wires should also be balanced. For this reason, the switch matrices constructed in this paper all have balanced (or nearly-balanced) fan-in and fan-out.

4.3.3 Hamming Distance and Coding Theory

The switch placement problem requires that subsets of the input wires span as many output wires as possible. Doing this for every possible subset of input wires is a difficult task. Instead, this can be simplified by spreading out the switches between every pair of input wires. In this form, the switch placement problem is equivalent to a problem in the design of communication codes. Hence, code-design techniques such as those from [124] can be used to solve the switch placement problem.

The location where switches are placed on an input wire can be represented by a bit vector of length m, where a 1 in the bit vector indicates that a switch is present. There are n such bit vectors, one for each input, forming the codewords of a binary code.

The number of neighbours of an input wire subset is the number of ones in the bitwise-OR of their bit vectors. Given two bit vectors, the increase in the number of neighbours (output wires) reached by the combination of the two is related to the Hamming distance[1] between them. Spreading out the switches between a pair of input wires i and j represented by bit vectors bv_i and bv_j is the same as maximising $HammingDistance(bv_i, bv_j)$. Code design techniques often attempt to maximise the minimum Hamming distance between all of the codewords.

[1]The Hamming distance is the number of bit positions that differ between the two bit vectors.

One interesting branch of coding theory which relates to sparse crossbar design is the study of constant-weight codes. These are codes in which every codeword has the same number of set bits, w. This is similar to a sparse crossbar with a constant fan-out of w at every input. For a given codeword length m (number of outputs) and weight w, there is a maximal number of codewords (number of inputs), defined as $A(m,d,w)$, which are separated by some minimum Hamming distance d.[2] There has been significant effort to tighten the lower bound for $A(m,d,w)$, as described in a comprehensive paper by Brouwer *et al* [125]. Some bounds are determined by mathematical counting arguments [126], while others are tightened by explicitly constructing codes and counting the number of codewords. For example, stochastic search heuristics have been successfully combined with traditional constructive approaches [127].

The sparse crossbar construction algorithm may be used as a tool to help find new lower bounds on $A(m,d,w)$ as well. To see this, consider that the algorithm fixes n, m and w and generates a code. Intuitively, the algorithm finds widely-spaced codewords and places them closer together, using the resulting slack to increase the minimum d between closely-spaced codewords. Hence, after code generation it produces a code with some large minimum d. By increasing n as large as possible (until a further increase would reduce the minimum d), a maximal-sized code is obtained. This value of n is from a real code, hence it may represent a new lower bound for $A(m,d,w)$.

Some early efforts were made to use the algorithm in this context. The algorithm was often able to approach or reach some previously published bounds, but it did not exceed any of them. This was not vigourously pursued further because the switch placement algorithm is not tuned for this type of application. Instead of packing codewords tightly together, the algorithm is designed spread out a fixed number of codewords. It also places constraints on fan-in profiles and it supports non-regular fan-in and fan-out profiles. As a result, the algorithm given here may be more helpful where the bounds are not yet known (*e.g.*, they are not tabulated in [125]). A modification of the algorithm to pursue improved bounds in coding theory, or vice versa, would be an interesting avenue of further research.

[2] The information theory literature refers to the maximal number of codewords as $A(n,d,w)$; the variable m is used here instead to remain notationally consistent with the number of crossbar outputs.

4.3.4 Expander Graphs

Our interpretation of Hall's theorem is that a routable sparse crossbar is formed if all subsets of the inputs have large enough neighbourhoods. In expander graphs, a similar property is found where the neighbourhood of any input subset is larger (by a constant amount, the expansion factor) than the input subset itself if the input subset is sufficiently small (*i.e.*, it is no more than a fixed fraction of the input set). Explicit constructions of expander graphs, such as those described by Margulis [17] and Gabber and Galil [18], cannot cope with an arbitrary number of inputs or outputs.

Very recent advances in the explicit construction of expanders have been made by Capalbo *et al* [107]. The construction method works for any number of inputs or outputs and a fixed number of connections. If this new method proves to be practical, it will be extremely useful and it may be used to replace the switch placement algorithm described in this chapter.

Until a suitable comparison with this new approach is made, the following algorithm is the only known general method that can generate routable sparse crossbars for an *arbitrary* number of inputs, outputs, switches, as well as *arbitrarily prespecified* fan-in and fan-out profiles.

4.4 Switch Placement Algorithm

Using insight gained in the previous section, a two-stage switch placement algorithm has been developed. In the first stage, an initial switch pattern is chosen subject to given constraints. The second stage performs iterative optimisation without violating the constraints. The inputs required for the algorithm are the matrix size, $n \times m$, and the number of switches p. Optionally, fan-in and/or fan-out profiles can be specified for the outputs or inputs. An overview of the algorithm is given in Figure 4.4.

To motivate the switch placement, consider again the routability of the different patterns in Figure 4.2. The pattern with the best routability is obtained with the switch pattern generator described in this section. The second best routability is obtained with a pattern that was hand-generated by a PLD designer prior to this work. The next two are random

```
MainSwitchPlacementAlgorithm()
{
   input:  n, m, p
         // n = # of inputs, m = # of outputs, p = # of switches
   optional input:  fanin[m], fanout[n]
         // fanin or fanout profiles

   output: sw[n][m]
      // switch placement (n x m adjacency matrix)

   if no fanin/fanout constraints specified
      generateFaninFanoutConstraints( n, m, p, fanin, fanout )

   generateInitialSwitchPlacement( n, m, p, fanin, fanout )
   optimisePlacement( n, m )
   print final switch placement
}
```

Figure 4.4: Overview of switch placement algorithm.

```
   generateFaninFanoutConstraints( n, m, p, fanin, fanout )
   {
      // create uniform fanout constraints for inputs
      for i = 0 to n-1
         fanout[i] = floor(p / n)
      for i = 0 to (p%n) - 1
         fanout[i]++

      // create uniform fanin constraints for outputs
      for j = 0 to m-1
         fanin[j] = floor(p / m)
      for j = 0 to (p%m) - 1
         fanin[j]++
   }
```

Figure 4.5: Algorithm to generate uniform fanin/fanout constraints.

patterns, one is completely random and the other is chosen as the best out of a large number of random patterns. The last one is a partial crossbar pattern.

4.4.1 Initial Switch Pattern Generation

The goal of the initial switch pattern generator is to place switches according to given fan-in and fan-out specifications. These specifications limit the number of switches that will be placed on each wire. The user may provide any valid fan-in and/or fan-out distribution. If no specification is provided, one is created by the algorithm given in Figure 4.5. This algorithm uniformly divides the number of switches, p, among all n inputs and all m

```
generateInitialSwitchPlacement( n, m, p, fanin, fanout )
{
   const  TRY_LIMIT =  p * p
   const PASS_LIMIT = 100

   // try randomly, up to PASS_LIMIT times
   do {

      initialise sw[i][j] = 0 for all i,j
      num_placed_switches = 0

      do {
         tries=0

         while(1) {
            randomly choose input i, output j
            tries++
            if( tries > TRY_LIMIT ) break
            if new switch at i,j obeys fanin/fanout constraints {
               sw[i][j] = 1
               num_placed_switches++
               break
            }
         }

         if( tries > TRY_LIMIT ) break
      } while( num_placed_switches < p )

      passes++
   } while passes < PASS_LIMIT and tries > TRY_LIMIT

   if passes > PASS_LIMIT and tries > TRY_LIMIT {
      // random failed, use network flow
      initialNetworkFlowPlacement( n, m, p, fanin, fanout )
   }

}
```

Figure 4.6: Random initial switch placement algorithm.

outputs.

A switch pattern which obeys the fan-in/out specifications is generated in one of two ways: either randomly, or by a network flow computation. The random method given in Figure 4.6 repeatedly places a switch at a random location in the crossbar provided it doesn't violate the fan-in/out specifications. If it cannot find a valid location for the next switch after a certain number of tries, it erases all switches and starts over. Unless there are a large number of switches to place, an initial pattern is usually found the first time. If it still fails after restarting a few times, the network flow method is used instead.

The network flow method given in Figure 4.7 finds a precise solution to the fan-in/out specifications. The procedure constructs a flow network very similar to that shown in

```
initialNetworkFlowPlacement( n, m, p, fanin, fanout )
{
   add n input nodes to I
   add m output nodes to O

   add supersource node S
   for i = 0 to n-1
      add edge from S to input i with capacity fanout[i]

   add supersink node T
   for j = 0 to m-1
      add edge from output j to T with capacity fanin[j]

   for i = 0 to n-1
      for j = 0 to m-1
         add edge from input i to output j with capacity 1

   solve maximum network flow from S to T
   if maximum flow < p
       exit "no switch placement satisfies fanin/fanout constraints"

   // switch locations are copied from the maximum flow solution
   for i = 0 to n-1
      for j = 0 to m-1
         if edge from input i to output j has flow==1
            sw[i][j] = 1  // place a switch
         else
            sw[i][j] = 0  // do not place a switch

}
```

Figure 4.7: Initial switch placement algorithm using a maximum network flow algorithm.

Figure 4.3. First, start with a source S, a sink T, input nodes I, and output nodes O. Add edges from S to I and O to T and assign flow capacities equal to the fanout and fanin capacity specifications, respectively. Then, temporarily place a switch at every location in the crossbar (fully connect from I to O) assign each a unit capacity. Find the maximum flow from S to T. If a switch pattern can be generated to obey the given constraints, it will be found as solution with a total flow of p. Keep all of the edges used by the flow solver (carrying flow) as the locations for the initial switch pattern and discard the others which carry no flow. This network flow method is used as a backup method because it is usually slower than the random method.

4.4.2 Switch Placement Optimiser

The routability of an initial switch pattern is improved by the optimisation phase. A number of switches can be moved into a more “spread out” pattern through a number of “switch

location swaps". A swap involves moving two switches at the same time so the fan-in and fan-out profiles remain unchanged. The optimising algorithm given in Figure 4.8 generates random swap candidates and accepts the swap if the following cost function is reduced:

$$\sum_{\forall x,y|x\neq y} \frac{1}{HammingDistance(bv_x, bv_y)^2}.$$

This is the same cost function used by El Gamal *et al* [128] to design codes. It encourages switches to spread because two inputs that reach nearly the same set of outputs will have a higher cost due to their smaller Hamming distance.[3]

A more efficient $O(n)$ version of this cost calculation is given in Figure 4.9. The efficiency arises because only the part of the cost which has changed, namely the distances from the two changed inputs i_1 and i_2, needs to be calculated.

The optimisation algorithm works in a greedy fashion: it generates random swap candidates, but it only accepts the swaps if the cost function improves. The algorithm stops after the number of iterations specified by the user parameter `MAX_ITERATIONS`. A simulated-annealing approach was initially used but this did not result in better switch placements. It was noticed that any hill-climbing moves which raised the cost function were nearly always found again and undone. Similar results were obtained by Leventis [129]. Instead, the greedy algorithm lowers the cost function much more quickly. Another algorithm that exhaustively tried all possible swap candidates ran considerably slower and also did not improve results.

Further details about the cost function, generating swap candidates, and limitations of this algorithm are discussed below.

4.4.3 Cost Function Pitfalls

This section describes a few of the alternative cost functions that were considered and rejected for the optimisation algorithm. Each of these alternatives has a drawback which should be considered if a new cost function is desired.

One alternative cost function maximises the minimum Hamming distance between all inputs. As noted in [128], this is not very useful because not all switch swaps would lead

[3] In the event that the Hamming distance between two bit vectors is zero, an arbitrary cost of 100 is used.

```
optimisePlacement( n, m )
{
   const MAX_ITERATIONS = 10000
   while iteration < MAX_ITERATIONS {

      // find random swap candidate pair
      (i1, j1, i2, j2) = findRandomSwapCandidates()

      originalCost = cost( i1, i2 )

      // perform switch swap
      move switch from location i1,j1 to i1,j2
      move switch from location i2,j2 to i2,j1

      newCost = cost( i1, i2 )

      if newCost > originalCost
         restore original pattern
      iteration++
   }
}
```

Figure 4.8: Iterative optimisation of switch placement.

```
cost( i1, i2 )
{
   // compute cost relative to inputs i1, i2
   cost = 0.0

   // NOTE:  the function HammingDistance(x,y) below returns
   //        0.1 rather than 0.0 when x and y are identical.

   // compute cost relative to i1
   for i = 0 to n-1
      if( i != i1 )
         cost += 1.0 / HammingDistance( sw[i1][*], sw[i][*] )^2

   // compute cost relative to i2
   // don't double-count the cost relative to i1
   for i = 0 to n-1
      if( i != i2 AND i != i1 )
         cost += 1.0 / HammingDistance( sw[i2][*], sw[i][*] )^2

   return cost
}
```

Figure 4.9: Cost computation.

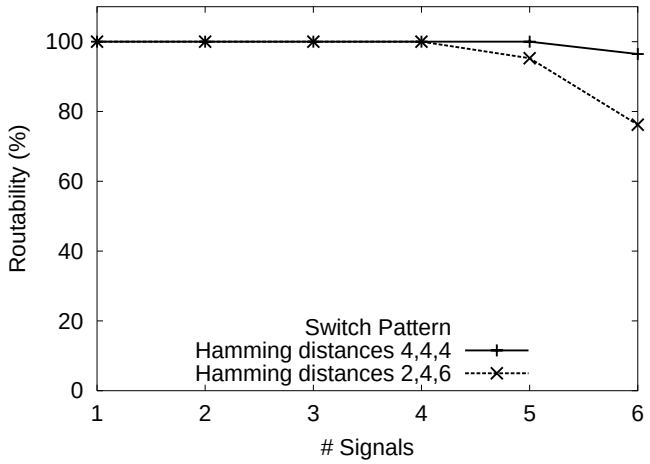


Figure 4.10: Routability of 9×6 sparse crossbars with different Hamming distance profiles.

to an observable change in the cost function.

Another possible cost function is to maximise the total Hamming distance between all pairs:

$$\sum_{\forall x,y|x\neq y} HammingDistance(bv_x, bv_y).$$

Unfortunately, this does not sufficiently penalise close bit vectors. For example, consider the 3 bit vectors 111000, 011100, 000111 with Hamming distances of 2, 4, and 6 and the alternative switch topology 111000, 001110, 010011 with distances of 4, 4, and 4. Two 9×6 sparse crossbars can be created with these topologies by replicating each input bit vector three times. Figure 4.10 shows that the routability of the latter topology (with distances of 4, 4 and 4) is better than the former. However, no difference between these bit vector sets is found if only the total Hamming distance is examined.

Another, more direct cost function is the routability estimate from the Monte Carlo tests. This accomplishes the goal of improving routability. However, it also leads to two problems. First, a Monte Carlo simulation is much slower to compute. Second, the results of a single swap are not always readily discernible by the simulation.

In comparison to the above alternatives, the Hamming distance cost selected is relatively quick to compute and it can distinguish many types of changes to the switch pattern.

4.4.4 Generating Swap Candidates

Swap candidates are determined by the random selection of two input wires. Using the placement of switches on these two wires, two output wires are randomly selected to form a swap candidate. An example of a swap candidate between inputs 1 and 4 and two outputs is given on the left of Figure 4.12. To preserve the fan-in/out distribution profiles, there must be only two switches between these four nodes. Exchanging the outputs reached by the two inputs constitutes a swap. If no valid swap candidate can be found, a new pair of input wires is chosen. The simplified version of the algorithm to find swap candidates is given in Figure 4.11. A more complex version which detects cases when no swap candidates exist is actually implemented.

Rather than swapping, it is also possible to move a single switch at a time. An example of a single-switch move is given on the right of Figure 4.12. This can be done while preserving the fan-in and fan-out distribution profiles if the following conditions are met. For a given input wire, a switch can be moved from one output to another if the original output's fan-in is one greater than the new output's fan-in before the move. After the move, the outputs exchange their fan-in specifications. Similarly, a switch fixed to an output wire can switch input wires provided the fan-outs of the old and new inputs differ by one.

During early experiments, single switch moves were done after all of the two-switch swaps were completed. The generation of specific single moves was tried two ways: randomly for speed, and exhaustively for thoroughness. Unfortunately, single switch moves did not improve routability in these experiments. To speed the generation of results for this chapter, single switch moves are not performed.

4.4.5 Limitations of the Algorithm

When $p \leq 2n$, the switch matrix is very sparse and suboptimal switch placements are sometimes generated. Under these conditions, small disconnected components may be present in the bipartite graph, yet they are indistinguishable by the cost function.

For example, consider the leftmost matrix in Figure 4.12. The cost function is unable to determine whether the swap shown is good, since it has the same cost before and after.

```
findRandomSwapCandidates()
{
   const TRY_LIMIT = 1000

   // find random swap candidate pair
   loop

      // randomly choose two inputs
      randomly choose input i1
      loop
         randomly choose input i2
      until i2 != i1
      let o12 = outputs reached by i1 + i2

      // randomly choose outputs reached by inputs
      try = 0
      loop
         randomly choose output j1 from outputs o12
         randomly choose output j2 from outputs o12
         try++
      until ( sw[i1][j1] == 1 AND  // exit when: switch at i1,j1 AND
              sw[i2][j2] == 1 AND  //             switch at i2,j2 AND
              sw[i1][j2] == 0 AND  //             no switch at i1,j2 AND
              sw[i2][j1] == 0 AND  //             no switch at i2,j1 AND
              j1 != j2             //             reaches different outputs
           OR  try >= TRY_LIMIT )   // OR exit if all tries are exhausted

   until try < TRY_LIMIT

   // return valid swap candidate
   return i1, j1, i2, j2
}
```

Figure 4.11: Find eligible switch moves.

Figure 4.12: Example of a bad move (left) and a good move (right).

However, it is a bad move. The original matrix routes any test vector of size 3, but the new matrix cannot route the input subset $\{1,5,6\}$.

Despite this limitation, the switch placement algorithm often makes intelligent decisions. Consider the switch matrix on the right of Figure 4.12 which cannot route the subset $\{1,2,3\}$. To lower the cost, the algorithm will find the single switch move indicated. The new switch pattern can now route *all* groups of 3.

4.5 Results

The switch placement algorithm and evaluation method described above have been implemented in a tool using C++ to construct and test sparse crossbars. In this section, a number of experiments are run with a sparse crossbar containing 168 inputs and 24 outputs. This size is chosen for two reasons: it is small enough to run experiments quickly, and it is the same size as the C block used in Altera's FLEX8000 family [130] to choose 24 CLB input signals from the 168 row wires. To assist in comparisons, Altera has confirmed that the FLEX8000 sparse crossbar contains two switches for every crossbar input, for total of 336 switches [Heile, *private communication*]. However, the exact switch locations are unknown.

In the following design examples, an assumption will be made that the Altera FLEX8000 CLB is intended to be highly routable for 24 signals. However, the actual design goal may have been for the average case or for the most common cases of perhaps 16 or 20 outputs, for example. In these cases, a similar methodology can be followed with this new design objective, and similar results would be obtained.

It should also be noted that the following analysis is simplified by examining the routability of an isolated sparse crossbar. That is, the results assume that there is no "upstream flexibility" available which could be used to rearrange the signals to different inputs, hence making some of the random test vectors irrelevant. Accounting for this flexibility makes the design problem very complex and would be a very interesting extension of this work. However, before performing such an extension, it is first necessary to understand the performance of a sparse crossbar in isolation as described below.

4.5.1 Adding Extra Switches

This first set of experiments investigates how sensitive the routability of a sparse crossbar is to the addition of switches. For example, it is not clear how quickly routability increases as the number of switches is increased. Does it increase at the same rate, or is there a region where it increases more rapidly?

The leftmost graph in Figure 4.13 shows a routability curve for each fixed switch count.

The *x*-axis indicates the number of signals to be routed, *i.e.*, the test vector size. At each test vector size, 10,000 random vectors are generated and the percentage of vectors that can be routed is given on the *y*-axis. Clearly, large test vectors are more difficult to route, and sometimes the drop-off is very rapid. Each curve shows the routability obtained with a fixed number of switches arranged in a fixed pattern. The number of switches ranges from 175 to 700, in steps of 25. Because the switch counts differ, a unique switch pattern is created for each curve.

As switches are added in the left graph of Figure 4.13, the entire routability curve shifts upward. When only a few switches are present, the amount of the shift can be quite large. For example, going from 200 to 300 switches improves the routability of size 12 test vectors from 15% to 80%. As more switches are added, the amount of the increase diminishes, implying less utility is gained from each additional switch. For example, the improvement going from 600 to 700 switches is barely noticeable except for the largest test vectors.

Using the same data, the rightmost graph in Figure 4.13 better illustrates this sensitivity of routability on switch count. In this graph, the number of switches varies along the *x*-axis. Each curve represents a fixed test vector size. A given test vector size of 20 signals, for example, shows the greatest improvement in routability as the number of switches is increased from 300 to 500. The largest test vector size (24 signals) shows the slowest improvement rate. Consequently, a large number of switches are required for that vector size to reach 95% routability.

There are two key observations to be made from Figure 4.13. First, there is a range where routability is very sensitive to the number of switches. Outside of this range, routability is either very poor or very good. When reducing the number of switches, designers should stay outside of this highly sensitive region to ensure good routability. Second, the routability of the largest test vectors is the least sensitive to the addition of switches. If a designer is trying to make these large vectors highly routable, a large number of switches are required. In contrast, significantly fewer switches are required if the number of signals routed is less than the number outputs.

Another way to present the efficiency of the switch pattern is shown in Figure 4.14.

This graph replots the data from Figure 4.13 as follows. Each curve in the new graph represents a fixed routability level, say 90%. As before, the *x*-axis is the number of signals to be routed. However, the *y*-axis is the smallest number of switches *per output signal* required to achieve the 90% routability level. The curves show that most, but not all, of the crossbar outputs should be used to make efficient use of the switches. If nearly all of the outputs are used, *i.e.*, more than 20, significantly more switches per signal are needed to reach high levels of routability. Hence, the value obtained by adding each additional switch in this region is small; many switches are needed to make a significant contribution to routability.

Two additional curves of interest are shown in Figure 4.14: the entropy curve and the 100% lower bound curve. The latter curve illustrates the minimum number of switches needed to reach perfect routability. This is determined using the formula from [86]:

$$p = \lceil (n-k+1)m/(m-k+1) \rceil$$

where k is the number of signals on the *x*-axis. The lower bound demonstrates that a large number of additional switches per signal is needed to go from 99.9% to 100% routability for large test vector sizes. When the number of signals is small (≤ 16), the bound might also suggest that the sparse crossbars are using too many switches to reach $\geq 90\%$ routability. However, it is known that the lower bound is not tight [86].

The entropy curve shows the absolute minimum number of SRAM bits that would be needed to program the switch matrix. As shown by DeHon [131], the number of bits required is $\lceil \log_2 \binom{n}{k} \rceil$. This shows that significantly more switches are needed than what a simple counting calculation would predict.

4.5.2 Adding Extra Output Wires

In the previous subsection, the switch matrix contains exactly 24 outputs and carries up to 24 signals. Here, the number of crossbar output wires is gradually increased from 24 to 48, but the crossbar routability is only tested with 24 signals. The results of widening the output stage like this are shown in Figure 4.15 for a variety of switch counts. When the number of switches is low, the routability increase due to the additional output wires is not

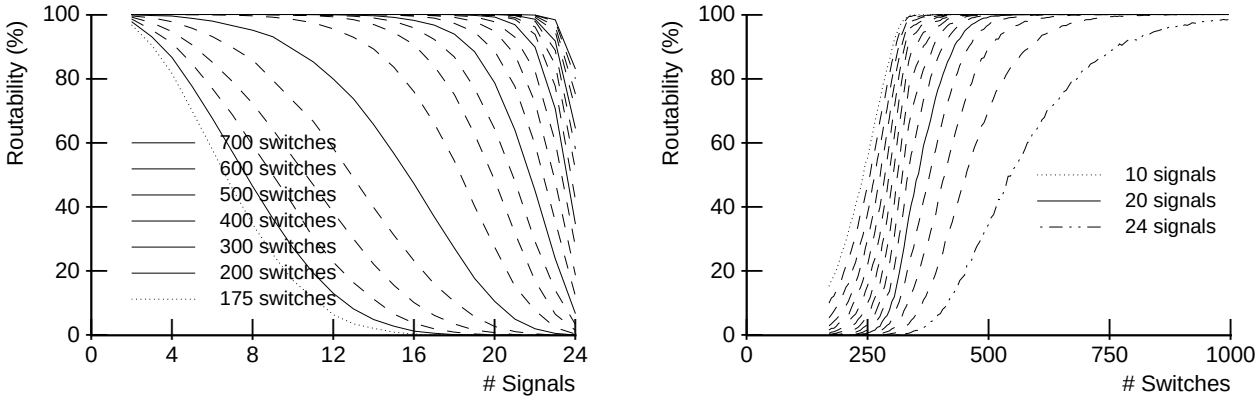


Figure 4.13: The effect of adding extra switches on routability of a 168 × 24 crossbar.

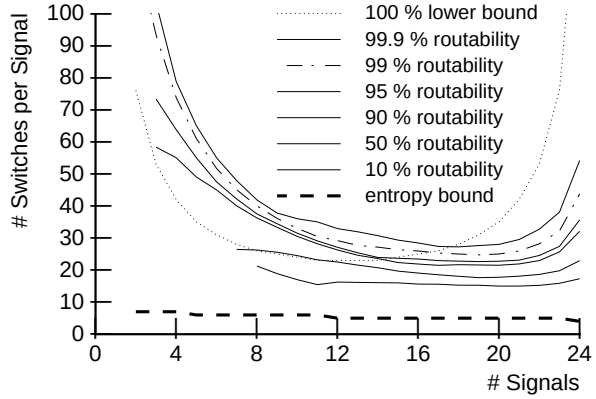


Figure 4.14: Efficiency of switches in a 168 × 24 crossbar.

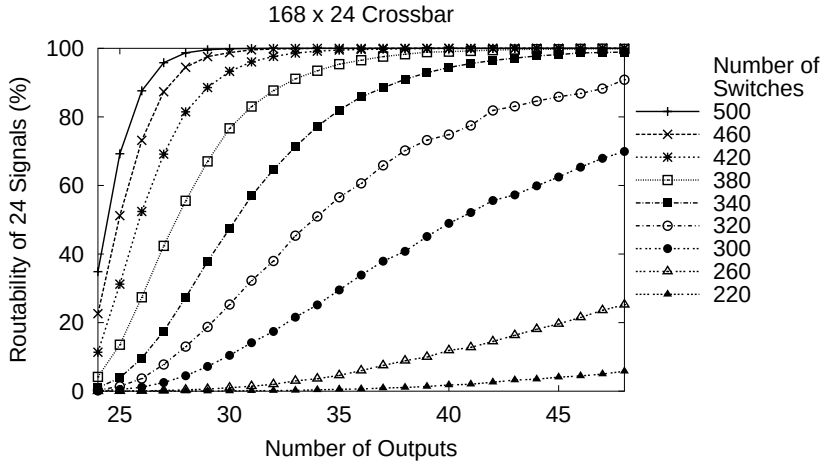


Figure 4.15: Routability of 24 signals in a 168 × 24 crossbar as output wires are added.

very significant. When 340 or more switches are used, however, improvements of up to 100% can be seen with just a few additional outputs.

From this data, it is evident that widening the output stage is an easy way to improve the routability of a sparse crossbar. However, a certain minimum number of switches must be present to take advantage of the extra output wires. Producing more outputs than required can also place higher area demands on downstream routing resources. This will be considered more fully in the next section.

4.5.3 Adding Both Switches and Wires

There is a cost associated with having more output wires. Namely, each additional output must be considered as an additional input to the downstream interconnect it feeds. In the Altera FLEX8000 architecture, for example, the sparse crossbar feeds the local interconnect of the CLB, or clustered logic block. The CLB contains eight 4-LUTs sharing 24 inputs. Each LUT input is completely connected to all of these 24 inputs and to all of the eight local LUT outputs.

Figure 4.16 shows the combined effect of adding switches and widening the output stage of a sparse matrix design that can be used in the FLEX8000. In this graph, three key curves are shown:

1. architecture A (baseline) with 336 switches and 24 output wires (dotted curve),
2. architecture B with 336 switches and 30 output wires (lower solid curve), and
3. architecture E with 510 switches and 30 output wires (upper solid curve).

The baseline architecture is similar to the FLEX8000. The addition of output wires (architecture B) and then more switches (architecture E) dramatically improves routability to nearly 100% for all test vector sizes.

It is interesting to compare the above results with the simple addition of more switches. Architecture C also uses the same number of switches as E, but it is not as routable because it has only 24 outputs. However, architecture E has an advantage: it contains more switches inside the CLB (from the additional sparse crossbar outputs). To make a fair comparison

by keeping total switch count the same, the same number of additional switches should be added to architecture C as well. This is shown as architecture D with 702 switches. Even with the same number of switches, architecture E is still more routable.

A more complete picture of the trade-off between number of output wires and total switch count is shown in Figure 4.17. Routability on the *y*-axis is plotted as a function of the number of outputs on the *x*-axis. Each curve represents a fixed total number of switches contained in two locations: in the sparse crossbar and in a fully populated Altera FLEX8000-style cluster.[4] As more output wires are added, routability of the sparse crossbar ultimately degrades because more of the switches must be located inside the cluster.

The data used to create Figure 4.17 indicates that the fewest switches to reach 99.95% routability is 1470 switches along with 30 output wires. This total is divided as 510 switches in the sparse crossbar and 960 inside the CLB. In comparison, the FLEX8000 has a total of 1104 switches and reaches less than 20% routability. By using a few more switches and output wires, a significant boost to routability can be obtained.

Each of the routability curves in Figure 4.17 peaks at a particular number of output wires. This best number of output wires is the point which produces the highest routability level for a fixed number of switches. Figure 4.18 plots this highest routability level (produced with the best number of outputs) against the number of switches. For comparison, the routability level reached with only 24 outputs is also plotted. It is clear that the additional outputs raises the sensitivity of routability to the number of switches, and reaches near-100% levels with significantly fewer switches.

4.5.4 Summary

The results from this section indicate that, to be area-efficient, a sparse crossbar should be intentionally under-utilised. In particular, the number of outputs should be more than the number of signals to be routed through it. The number of switches can be reduced, but care must be taken not to make a crossbar very sparse and push it into a highly-sensitive region of routability. As well, it is important to choose the number of switches and wires together because a minimum number of switches are needed to benefit from the extra output wires.

[4]The 256 switches for local LUT feedback connections have been excluded from these totals.

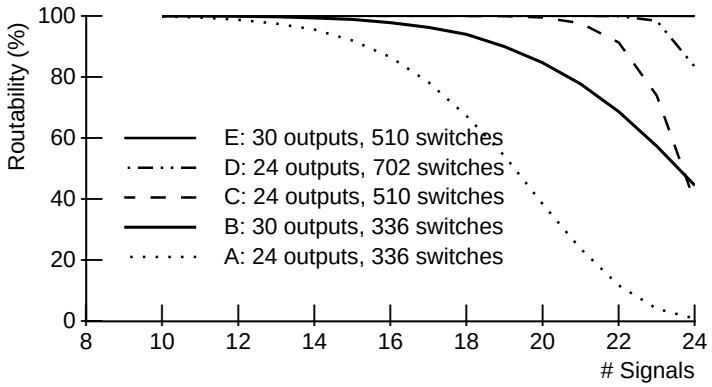


Figure 4.16: Routability of a 168 $\times$ 24 crossbar after adding output wires and switches.

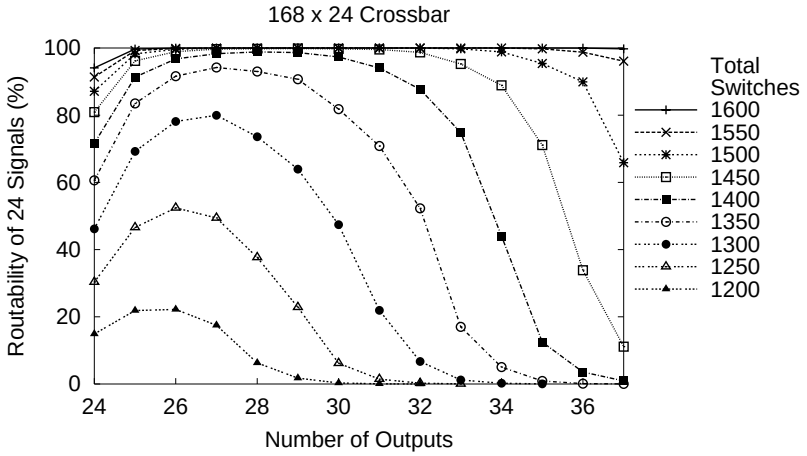


Figure 4.17: Routability of 24 signals with a fixed number of total switches.

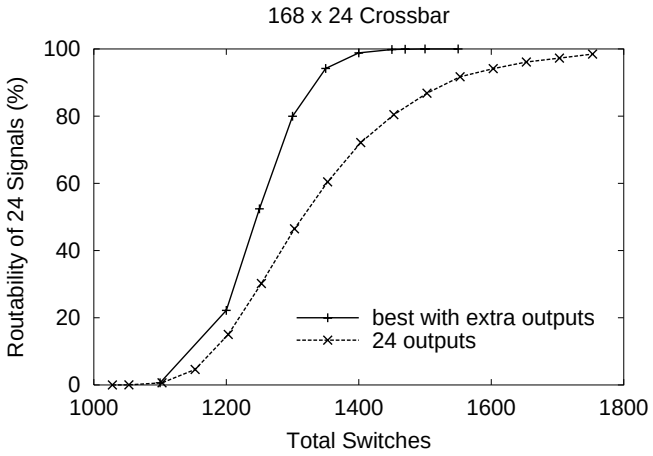


Figure 4.18: Routability of 24 signals while varying total switch count.

4.6 Design Examples

The previous section investigated the design space around a sparse crossbar of a particular size. In this section, a number of design examples are explored which would make good use of a sparse crossbar. For each example, a search is done for sparse crossbar configurations which have low transistor counts and which reach 95% or better routability at the largest test vector size.

In general, most of these designs use a *two-level interconnect* formed by a top-level sparse crossbar followed by a fully-connected lower level. In some cases, a mid-level minimal crossbar is inserted to reduce the number of inputs seen at the lower level. The search involves varying the number of sparse crossbar switches and outputs.

Area Model

The area model used here is simply the total transistor count used by the two or three switching levels. The transistor count calculation assumes that each level's output is implemented with an n-input multiplexer of $2n-2$ transistors. Such a multiplexer would be constructed from a tree of 2:1 multiplexers using pass-transistor logic. As well, each multiplexer is controlled by $\lceil \log_2 n \rceil$ SRAM bits containing 6 transistors each. To keep things simple, this model ignores the buffering of signals or transistor sizing that would accompany a real design.

Search Methodology

To find the lowest-area configuration, a thorough sweep across a number of switch counts and output wires is done. Those designs which use the fewest transistors are tested to see if they route at least 95% of the largest test vectors. Further details of this search methodology will be described below.

During the search, the number of sparse crossbar outputs is increased one by one, beginning with the minimum required and ending with approximately 50% more than required. The number of switches is initially determined so each output connects with 1/2 of the inputs. Suppose this requires an r-input multiplexer for each output. Afterward, the switch

density is reduced by shrinking r by one until $r = 2$ is reached.

During the sweep, any designs with roughly 40% or more transistors than a standard baseline architecture are immediately rejected. This prunes a large number of organisations from routability testing. The standard baseline architecture, defined to have the minimum number of outputs and two switches per input wire, is similar to the connectivity found in the Altera FLEX8000 devices. The threshold of 40% additional transistors is arbitrary and is sometimes adjusted depending on the number of design matches found.

For the remaining architectures, a routability test is done using 1000 test vectors. If less than 95% routability is obtained, the remaining less-dense organisations are assumed to have less routability and abandoned. At this point, the search resumes with one additional output at the 1/2 density level. Otherwise, if more than 95% routability is obtained, the architecture is accepted as one area-efficient and highly routable design solution and the search continues with a lower switch density. In early experiments, some of these architectures were re-tested with more test vectors, but this did not usually change routability by more than 1%.

The end result is the generation of a number of designs which have excellent routability (95% or better) and require low area (roughly, no more than 40% above the spartan baseline).

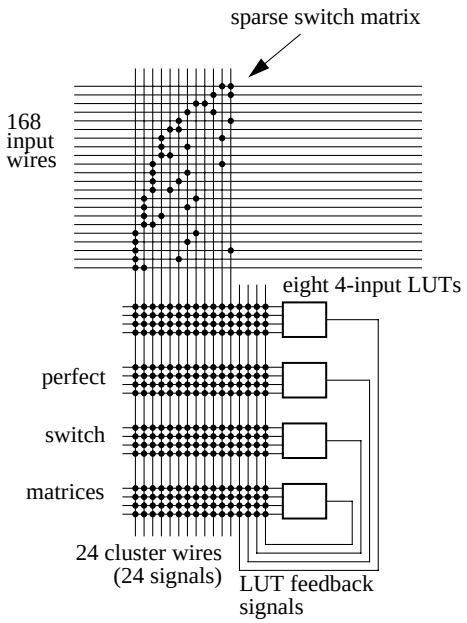


Figure 4.19: Interconnect model of the Altera FLEX8000 architecture.

4.6.1 Design Example: Altera FLEX8000 PLD

The Altera FLEX8000 device uses a 168×24 sparse crossbar to connect long row wires into the CLB clusters. This sparse crossbar is 1/12 populated, *i.e.*, each row wire has two "opportunities" to connect into a cluster. This sparse crossbar organisation, which will be referred to as the *FLEX8000 baseline*, uses a total of 336 switches. Within the cluster, the inputs of eight 4-input LUTs are fully connected to the sparse crossbar outputs and local LUT output signals using 1024 switches. When both levels are counted together, there are 1360 switches which require 4144 transistors.

The routability of the baseline organisation is shown in Figure 4.20. The sparse crossbar is highly routable when fewer than 10 signals are required. However, the routability of 15 or more signals is below 90%. At 24 signals, it is only 1%.

To increase routability of 24 signals beyond 95%, the number of switches must be increased to more than 888 (1912 switches in total). This requires 5536 total transistors, or 34% more than the baseline. Further details about these two organisations are in the first two of rows of Table 4.1. In this table, the 'L1' columns represent the sparse crossbar and the 'In Cluster' columns represent the downstream portion, a fully-connected cluster. In

the text below, a number of new organisations will be compared to this new highly-routable version.

The second group of rows in Table 4.1 considers increasing the number of outputs. As a consequence, this increases the number of cluster inputs and size of the internal cluster interconnect. The table shows the impact on switch and transistor counts within the cluster, as well as the totals. For example, there is a 168×29 sparse crossbar with 464 switches that uses a total of 5022 transistors. This saves 10% in transistors compared to the highly routable version with only 24 outputs.

Rather than add more cluster inputs, the additional outputs can be reduced to precisely 24 outputs with a slight change to the architecture. This is done by inserting an 'L2' minimal crossbar between the sparse crossbar and the cluster inputs. The third group of rows in Table 4.1 give the switch and transistor counts when using an 'L2' crossbar.

Figure 4.21 illustrates the total number of switches and transistors used in these different organisations. The total includes the 'L1', 'In Cluster' and, where appropriate, the 'L2' portions. In these graphs, the number of sparse crossbar outputs is varied along the *x*-axis. This produces a savings of roughly 10% switches and transistors at 26 to 29 outputs.

The curves labelled '2 levels' in Figure 4.21 include the 'L2' minimal crossbar. Compared to the '1 level' curves, there is seldom any switch count reduction and most organisations require more transistors. There would also be an increase in delay compared with the '1 level' architecture. Hence, no advantage is obtained by using two levels here; it is better to have a few more outputs in the sparse crossbar instead.

The curves labelled 'min. cluster' in Figure 4.21 show that some additional savings can be obtained if the clusters internally use minimal crossbars instead of full crossbars. When used in combination with the additional outputs, there is an overall reduction of 15% in transistors and 20% in switch count. (These results are not shown in Table 4.1.) Hence, minimal crossbars in clusters save a significant amount of area here.

Compared to the original baseline, the best highly-routable organisations (excluding the 'min. cluster' results) use approximately 20% more switches and 23% more transistors. Figure 4.20 shows how this additional area makes a significant improvement to routability. There is no advantage to using an additional 'L2' minimal crossbar.

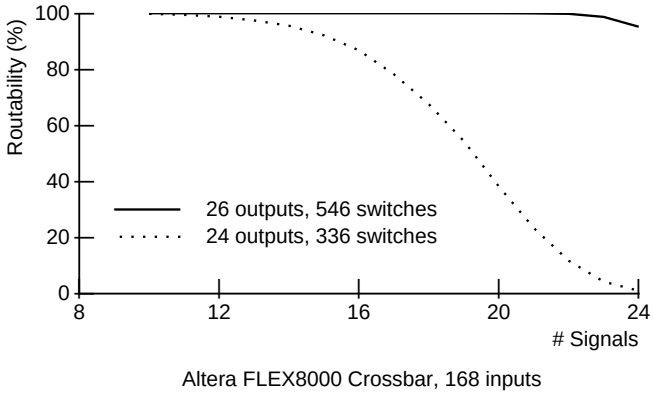


Figure 4.20: Routability improvements made to the FLEX8000 architecture.

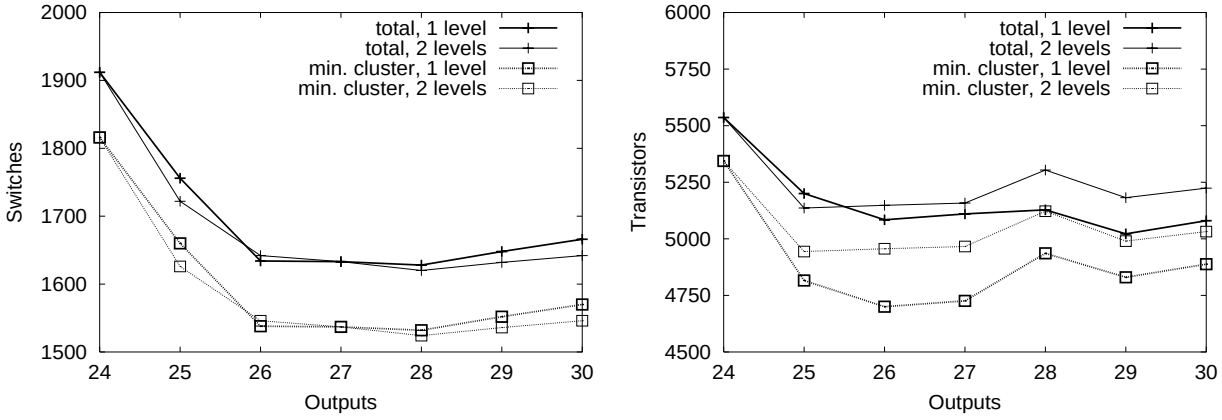


Figure 4.21: Sizes of highly routable Altera FLEX8000 organisations.

Sparse Crossbar Size	Number of Switches L1	L2	In Cluster	Total	Number of Transistors L1	L2	In Cluster	Total	Routability
Provide exactly 24 outputs from the sparse crossbar.									
168 × 24	336	–	1024	1360	1200	–	2944	4144	1.0%
168 × 24	888	–	1024	1912	2592	–	2944	5536	95.8%
Provide more than 24 outputs, increase cluster interconnect (1 level).									
168 × 25	650	–	1056	1756	2000	–	3200	5200	96.9%
168 × 26	546	–	1088	1634	1820	–	3264	5084	96.1%
168 × 27	513	–	1120	1633	1782	–	3328	5110	96.9%
168 × 28	476	–	1152	1628	1736	–	3392	5128	96.2%
168 × 29	464	–	1184	1648	1566	–	3456	5022	98.6%
168 × 30	450	–	1216	1666	1560	–	3520	5080	98.2%
Reduce to exactly 24 outputs using an additional minimal crossbar (2 levels).									
168 × 25 × 24	650	48	1024	1722	2000	192	2944	5136	96.9%
168 × 26 × 24	546	72	1024	1642	1820	384	2944	5148	96.1%
168 × 27 × 24	513	96	1024	1633	1782	432	2944	5158	96.9%
168 × 28 × 24	476	120	1024	1620	1736	624	2944	5304	96.2%
168 × 29 × 24	464	144	1024	1632	1566	672	2944	5182	98.6%
168 × 30 × 24	450	168	1024	1642	1560	720	2944	5224	98.2%

Table 4.1: Highly routable sparse crossbars designed for the Altera FLEX8000 PLD.

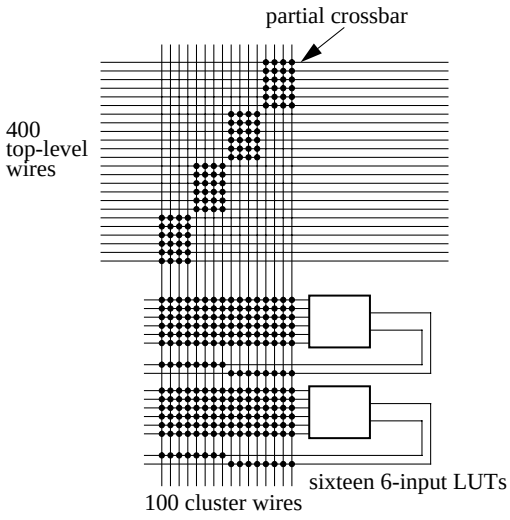


Figure 4.22: Interconnect model of the HP Plasma architecture.

4.6.2 Design Example: HP Teramac Plasma PLD

Teramac [120], from HP Labs, is a large reconfigurable system made up of custom-designed Plasma PLDs [121]. A full Teramac system is designed to have the capacity of about one million gates distributed over 1728 PLD chips. An important goal in the Plasma design was to design a highly routable PLD: to limit compile times to about an hour, placing and routing each PLD must be done quickly (within 3 seconds). This approach meant each PLD should be nearly 100% routable so that almost no time would be spent in rip-up or repartitioning the mapped circuit.

Plasma is unique among PLDs in that it uses 6-input, 2-output LUTs. The Plasma 2-level hierarchy comprises sixteen clusters, called *hextants*, of sixteen LUTs each. Each hextant, shown in Figure 4.22 has 100 inputs from the 400 top-level signals. These inputs are chosen using a 400×100 partial crossbar which is 1/4 populated with $10,000$ cross-points. This is equivalent to four diagonally-placed 100×25 full crossbars. Within each hextant, the 100 local cluster wires are fully connected to all of the LUT inputs. As well, each of the LUT outputs connect to half of these wires.

The Plasma chip is easy to route because the partial crossbars make it predictable: as long as fewer than 25 signals enter each full crossbar, it can be routed. The router only needs to consider which crossbar it routes to, and not the precise detailed route. Despite

this advantage, there are few routable signal assignments when the partial crossbar contains more than 75 input signals. This is shown in the routability curve of Figure 4.23.

By replacing the partial crossbar with a sparse crossbar, a large number of switches can be saved **and** routability can be improved. The sparse crossbar search found a 400×104 sparse crossbar with $1,456$ switching points, for a switch density of roughly 1/28. Even though this sparse crossbar contains about 1/7 the number of switches in Plasma, it can route over 95% of test vectors containing 100 input signals. In comparison, the partial crossbar pattern used in Plasma can route less than 1% of the hardest vectors. Given this new switch pattern, a router would have even higher assurances it can successfully route each Plasma chip.

There are a number of other design solutions as well. A sparse crossbar of size 400×105 with $1,680$ switches routes over 99.9% of the largest test vectors. Alternatively, with only $1,365$ switches this same size crossbar routes over 95.1% of the test vectors.

Additional sparse crossbars solutions are listed in Table 4.2. The total number of switches and transistors of these organisations, and a few others, are plotted in Figure 4.24.

The curves labelled as '2 levels' in Figure 4.24 add an 'L2' crossbar in the same way that 3-level organisations are considered in Section 4.6.1. These architectures use an additional minimal crossbar to reduce the number of sparse crossbar outputs to exactly 100. Like the FLEX8000 results, these results indicate that the 'L2' minimal crossbar increases both switch and transistor counts. Rather than using an 'L2' crossbar, it is better to increase the size of the cluster interconnect by a small amount.

From measurements taken of the Plasma die photograph in [121], the partial crossbar appears to consume 32% of the core area (excluding the I/O padframe). The Plasma implementation uses one pass transistor and one SRAM bit for every switch, so switch count accurately reflects area. Since a 104-output sparse crossbar eliminates 85% of the switches, total core area would be reduced by as much as 27%.

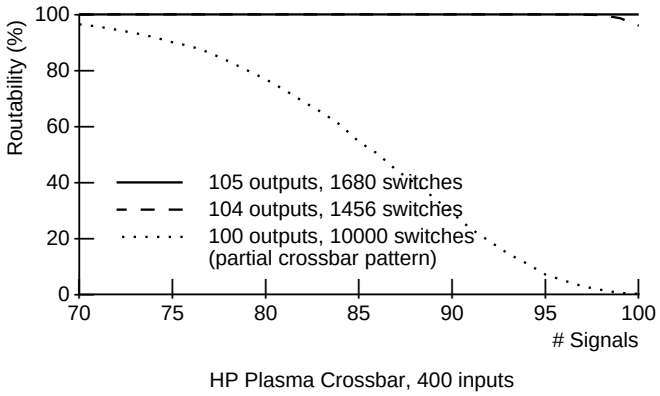


Figure 4.23: Routability improvements made to the HP Plasma architecture.

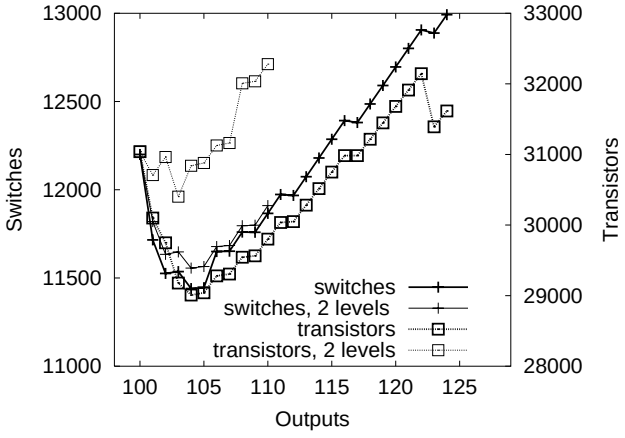


Figure 4.24: Sizes of highly routable HP Plasma organisations.

Crossbar Size	Number of Switches			Number of Transistors			Routability
	L1	In Cluster	Total	L1	In Cluster	Total	
Partial crossbar switch pattern used by HP.							
400 × 100	10000	9600	19600	24000	23040	47040	0.4%
Sparse crossbar switch patterns.							
400 × 100	2600	9600	12200	8000	23040	31040	96.3%
400 × 101	2020	9696	11716	6868	23232	30100	96.6%
400 × 102	1734	9792	11526	6324	23424	29748	96.6%
400 × 103	1648	9888	11536	5562	23616	29178	98.8%
400 × 104	1456	9984	11440	5200	23808	29008	95.9%
400 × 105	1365	10080	11445	5040	24000	29040	95.1%
400 × 106	1378	10272	11650	5088	24192	29280	98.6%
400 × 107	1284	10368	11652	4922	24394	29316	96.9%
400 × 108	1296	10464	11760	4968	24576	29544	99.3%
400 × 109	1199	10560	11759	4796	24768	29564	97.1%
400 × 110	1210	10656	11866	4840	24960	29800	98.8%

Table 4.2: Highly routable sparse crossbars designed for the HP Plasma PLD.

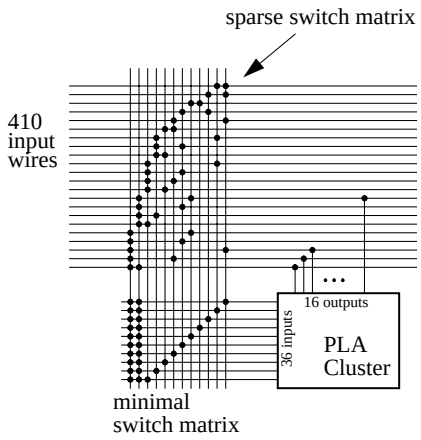


Figure 4.25: Interconnect model of the Altera MAX7256 architecture.

4.6.3 Design Example: Altera MAX7256 CPLD

The Altera MAX7256 CPLD family has two levels of hierarchy, where the top level contains multiple sparse $n \times 36$ crossbars. The lower level consists of a 36-input product term array. The number of top-level wires, n, is not known, but it is probably not more than 410 in the largest device, the MAX7256.[5] Hence, $n = 410$ is assumed for this design. A model for this architecture is shown in Figure 4.25. Since the number of inputs to the product term array should not be increased, this architecture always uses an additional 'L2' minimal crossbar.

The same SRAM-based model used throughout this chapter is used to count transistors here, except that the product term array is excluded. Since the MAX7256 implementations use one EEPROM cell and one gate per switch [30], the switch count metric is more representative of area than transistor count.

The highly-routable designs found during the search are shown in Figure 4.26 and Table 4.3. A sparse crossbar implementation with exactly 36 outputs needs 2412 switches to reach 95% routability. Without compromising routability, a 40% savings in total switch count can be obtained by widening the output stage to 40 or more and using the 'L2' crossbar. Clearly, under-utilising the outputs of the sparse crossbar is very important for area economy.

[5]This assumes one wire for each macrocell output and each I/O pin. A macrocell is a logic cell containing five product terms.

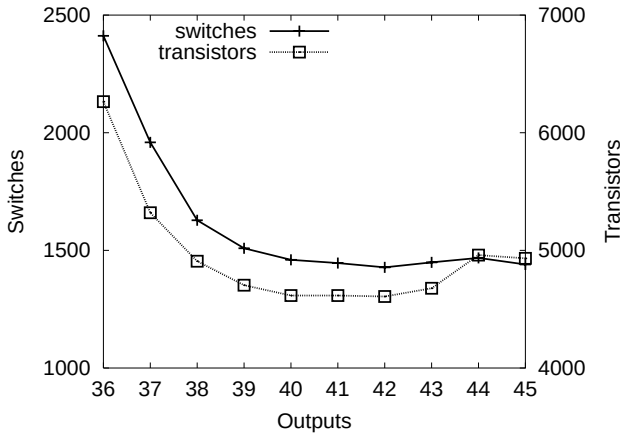


Figure 4.26: Sizes of highly routable Altera MAX7256 organisations.

Crossbar Size	Number of Switches L1	L2	Total	Number of Transistors L1	L2	Total	Routability
Provide exactly 36 outputs from the sparse crossbar.							
410×36	2412	–	2412	6264	–	6264	95.70%
Reduce to exactly 36 outputs using a minimal crossbar.							
$410 \times 37 \times 36$	1887	72	1959	5032	288	5320	96.9%
$410 \times 38 \times 36$	1520	108	1628	4332	576	4908	95.1%
$410 \times 39 \times 36$	1365	144	1509	4056	648	4704	95.6%
$410 \times 40 \times 36$	1280	180	1460	3680	936	4616	95.3%
$410 \times 41 \times 36$	1230	216	1446	3608	1008	4616	96.5%
$410 \times 42 \times 36$	1176	252	1428	3528	1080	4608	95.2%
$410 \times 43 \times 36$	1161	288	1449	3526	1152	4678	97.2%
$410 \times 44 \times 36$	1144	324	1468	3520	1440	4960	97.5%
$410 \times 45 \times 36$	1080	360	1440	3420	1512	4932	96.2%

Table 4.3: Highly routable sparse crossbars designed for the Altera MAX7256 CPLD.

4.6.4 Design Example: Varying the Sparse Crossbar Aspect Ratio

The previous sections have found efficient sparse cluster designs by reducing the number of switches and adding more outputs. However, basic size of the sparse crossbar was held fixed: the number of inputs and the required number of outputs did not change. In this section, the impact of varying these two parameters within the FLEX8000 architectural model from Section 4.6.1 is investigated. This is done by increasing the number of row wires (representing the crossbar inputs) and the cluster size (representing the number of required outputs).

Details of the experiment are as follows. The number of inputs is varied from 168 to 995. The number of required outputs is varied by adjusting the number of LUTs in a cluster, N, from 2 to 12. For each cluster size, two different methods are used to determine the number of required output signals (*i.e.*, cluster inputs): $3N$ and $2N+2$. The former is an assumption used to mimic FLEX8000 case of $N = 8$ which requires 24 outputs, while the latter is based upon input-sharing results from Betz [19]. For each sparse crossbar size, the search used in Section 4.6 is repeated here to find the lowest transistor-count configuration which has over 95% routability. The transistor count presented is the total of the sparse crossbar and local interconnect.[6] To fairly compare the efficiency of different cluster sizes, the transistor count is divided by the total number of LUT inputs in the cluster.

The results of this experiment are shown in Figures 4.27 and 4.28. The graphs on the left require $3N$ outputs, while those on the right require $2N+2$ outputs. In some cases, the search prunes away all designs that would meet the 95% routability target because they contain more transistors than the baseline architecture + 40%. Rather than increase the 40% threshold, no datapoint is plotted at that particular input and cluster size.

The impact of cluster size on interconnect area can be seen in Figure 4.27. For a sparse crossbar with only 168 inputs, interconnect area varies by roughly 25%. A cluster size from 3 to 8 gives the best efficiency for $3N$ outputs, and from 4 to 9 for $2N+2$ outputs. In both cases, a minimum is reached at $N = 6$. It is interesting to note that the FLEX8000 architecture, with $N = 8$, is among these efficient sparse crossbar sizes. As the number of inputs is increased, the best cluster size also increases. However, for small cluster sizes

[6] Note that an 'L2' minimal crossbar is not being used here.

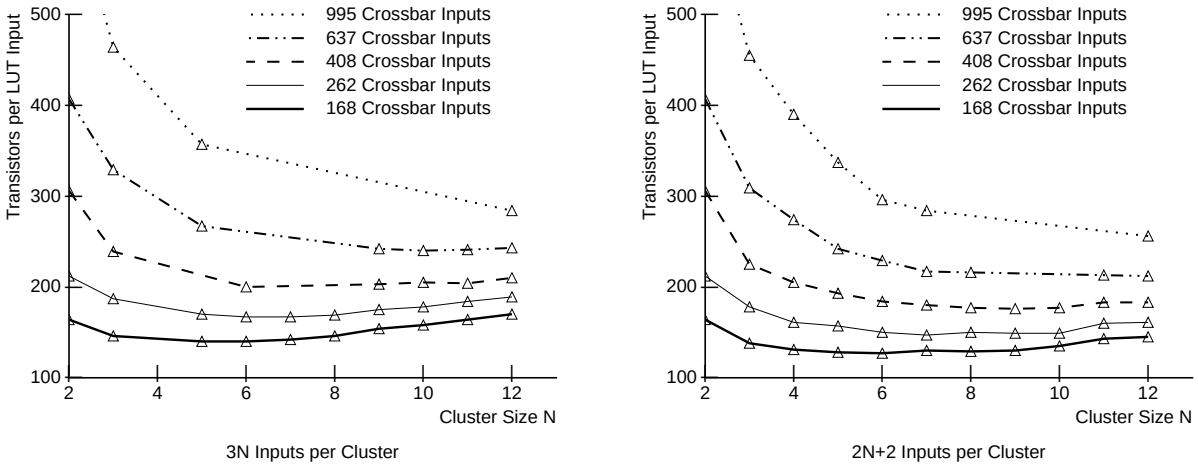


Figure 4.27: Effect of varying the cluster size N on interconnect size.

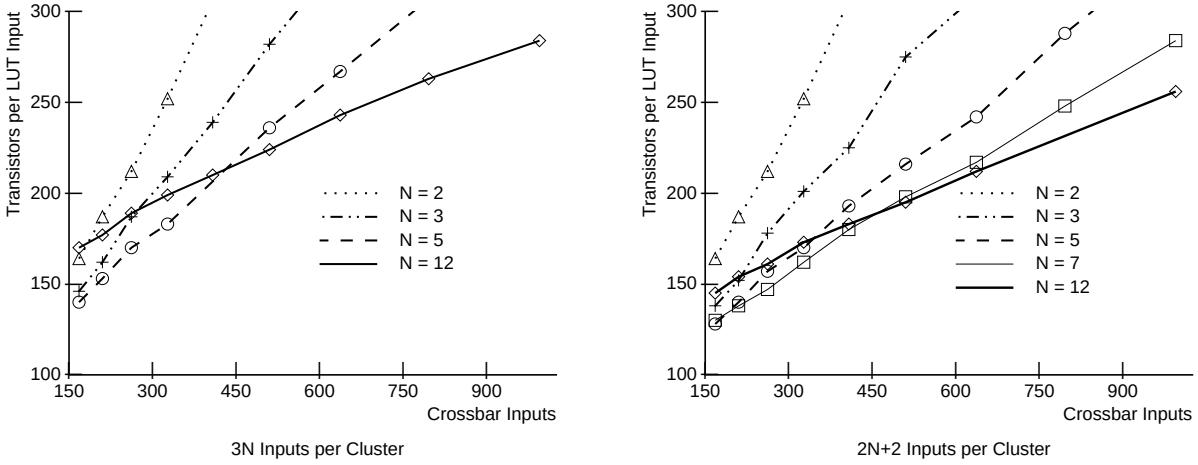


Figure 4.28: Effect of varying the number of top-level inputs on interconnect size.

$N < 6$, the area required to support a large number of inputs is much larger than for large cluster sizes. Letting the input:output aspect ratio of the sparse crossbar get too large hinders the efficiency.

To better illustrate the effect of the number of sparse crossbar inputs, Figure 4.28 plots the same data for a few select cluster sizes. This figure shows a roughly linear relationship between transistor count and input count, suggesting that the best switch density is constant for a given cluster size. The curves for large cluster sizes have a lower slope, an indication that they have a lower switch density. This explains why the most efficient cluster size increases as the number of inputs is increased.

4.7 Conclusions

This chapter has developed a method for evaluating and constructing sparse crossbars. The evaluation method consists of a Monte Carlo test of routing numerous random test vectors using a network flow algorithm. The construction technique, which is based upon observations from Hall's Theorem, uses a greedy iterative algorithm to spread apart the switch locations.

The routability of sparse crossbars can be improved by adding more switches and by widening the output stage of the crossbar. The latter method is the most effective once there are enough switches to be used. In the case of a 168×24 crossbar, two switches per input are required so that widening the output stage can reach a highly routable state. Careful evaluation using both methods is necessary to obtain optimum routability at minimum area.

A number of design examples further demonstrate that it is beneficial to underutilise the output stage of a sparse crossbar and use the correct number of switches. In the HP Plasma example, the use of 4 additional output wires and a switch density of 1/28 reaches a highly routable state. This organisation uses 85% fewer switches than the HP Plasma, and obtains superior results. In the Altera FLEX8000 example, 5 additional output wires and a slight increase in switch density (from 1/12 to about 1/10) improves routability from 1% to over 95%. In the Altera MAX7256 example, widening the output stage and using a second level minimal crossbar reduces switch count by 40%. These specific examples all demonstrate that high routability can be obtained with a reasonable amount of resources.

Exploration of different sparse crossbar sizes with the FLEX8000 model provides interesting results. First, clusters roughly 4 to 8 LUTs requires the fewest interconnect transistors per LUT. Second, a larger number of crossbar inputs becomes more efficient with more outputs, *i.e.*, larger cluster sizes. Third, the best switch density depends on the number of outputs but not the number of inputs. Fourth, the best switch density decreases as more outputs are needed.

4.8 Future Work

A number of aspects of this work deserve further attention. The switch placement algorithm has been developed to work on a single switching stage. The design of a two-stage network is a difficult but important problem that needs to be addressed. This becomes an even more difficult problem if a single stage is used to deliver signals to a number of parallel-connected stages. Such a configuration would be found in PLD that is designed using a hierarchy of clusters, *i.e.*, clusters of clusters.

Additional work should be done to examine the impact of inputs and outputs on switch density. For example, can a general equation be found to relate the number of inputs, outputs and switch density to a desired level of routability?

Additional work should make a more through comparison of different switch placement algorithms. Also, a more direct way of placing switches is desirable.

Chapter 5

Sparse Clusters

In PLDs, the connections from the CLB inputs and LUT outputs to the LUT inputs are often formed with a full crossbar. Such a high degree of connectivity makes routing easier, but it has significant area overhead. This chapter explores the use of sparse crossbars as an alternative switch matrix inside the clusters. This organisation is called a *sparse cluster* architecture. The experimental results show that switch densities can be reduced by 50% or more to save 10–18% in area. This switch reduction does not degrade critical-path delay, but some spare cluster inputs are required to compensate for the decrease in routability. Although not explored in this dissertation, it may be possible to achieve further improvements to area and delay by using CLBs with more LUTs and by calculating delay more accurately.

5.1 Introduction

Modern PLD architectures are based on a *clustered architecture*, where a number of lookup tables (LUTs) are grouped together to act as the configurable logic block. The motivation for using clusters is manifold: to reduce area, to reduce critical-path delay, and to reduce CAD tool runtime [41, 40, 115, 117]. This trend is followed by the most recent PLDs from Xilinx (the Virtex-II and Spartan-II families) and Altera (the Stratix and Cyclone families). All of these PLDs are based on clusters of 4-input lookup tables.

In a clustered architecture, the LUT inputs are chosen from two sources: either *cluster*

Architecture		Fully Populated Cluster Tile Area (Number of Minimum-Width Transistor Areas)						
LUT size, k	Cluster size, N	LUT+FF		Routing		LUT Input Mux		Total
4	6	990	(10.6%)	6050	(65.0%)	2267	(24.4%)	9307
5	6	1840	(16.4%)	6321	(56.2%)	3080	(27.4%)	11241
6	6	3496	(24.4%)	6713	(46.9%)	4109	(28.7%)	14318
7	6	6831	(34.8%)	7645	(39.0%)	5146	(26.2%)	19622
7	10	11358	(32.3%)	12022	(34.2%)	11765	(33.5%)	35145

Table 5.1: Breakdown of cluster tile area. Routing and total area are arithmetic averages.

inputs or *feedback connections*. The cluster inputs are connections from the external routing, carrying signals from other clusters into this one. The feedback connections are from the outputs of LUTs in the local cluster. In some PLDs, such as Altera's APEX family, these internal cluster connections are *fully populated* or *fully connected*. This is equivalent to employing a full crossbar: a crosspoint switch exists at the intersection point of every LUT input and every cluster input or feedback connection.

The area required to implement this full crossbar is significant. A typical implementation uses a multiplexer to select each output. Each of these multiplexers, which form the LUT inputs, select one of the crossbar inputs according to the state of a few SRAM bits. As a result, each of the multiplexers typically contains a few tens of inputs. Since there are a number of these multiplexers which must be driven, the crossbar inputs must be buffered. These buffers, multiplexers, and SRAM bits contribute 24 to 33% of the total area in a CLB. A breakdown of transistor area estimates for various LUT sizes and CLB sizes is provided in Table 5.1.

The significant amount of area required by the LUT input multiplexers motivates the idea of removing switching points from the full crossbar, or *depopulating* it, to result in a sparse crossbar. Naturally, depopulating raises the following questions:

1. What amount of depopulation is reasonable?

2. Will depopulation create unroutable architectures?

3. Does depopulation save total area, or does an increase in routing area overwhelm it?

4. Will depopulation reduce or increase routing delays?

5. How much area or delay reduction can be attained?

6. What is the impact on routing run-time?

This chapter addresses these questions using an experimental process of mapping benchmark circuits to clustered PLD architectures and measuring the resulting area and delay characteristics.

5.2 Methodology

To answer the questions raised in the preceding section, the twenty largest MCNC benchmark circuits are mapped into a variety of clustered PLD architectures following the same procedure described in Chapter 3. The effect of sparsely populating the internal logic clusters is examined using the area and delay estimates computed by an extended version of the VPR routing tool, called **VPRx**.

5.2.1 Routing Experiments

Routing experiments are used to determine the minimum channel width required to route a benchmark circuit, W_{min}. Afterward, a final low-stress routing is done with $W = 1.3 \cdot W_{min}$ tracks to compute area and delay information. Using more than the minimum number of tracks mimics the way PLDs are actually used; designers are seldom comfortable working on the edge of capacity or routability.

In some rare situations, the final low-stress routing fails even though the binary search finds a solution with a smaller channel width. This occurs in less than 1% of the circuit/architecture combinations explored in this chapter. All of these failures appear in routing architectures with very low routing flexibility.[1] The routing failures might be caused by slow convergence in a difficult-to-route architecture, or by an unroutable circuit/architecture combination. In the latter case, the automatically generated routing archi-

[1] Of the failed combinations, roughly half of them had $I_{spare} = 0$ and the remainder had $F_{c_{int}} \leq 0.25$.

tecture might contain flaws that would normally be avoided by a PLD architect through extensive verification and hand-crafting of the routing network.

An ad hoc approach is taken to resolve these routing failures and mimic architecture corrections made by an PLD architect. One, two, then three additional tracks are added to the channel and routing is attempted again. This strategy is sufficient to route all but four of the troublesome cases — the three underlying architectures for these cases are deemed unroutable and abandoned.

Also, the architecture is deemed unroutable and abandoned if the binary search is unable to find a reasonable minimum channel width ($W_{min} \leq 240$) for any of the circuits.

5.2.2 Conservative Delay Results

Delay results in this chapter ignore two possible sources of delay improvement. First, the interconnect RC values are overestimated because they ignore the cluster tile size shrink as clusters are made more sparse. Second, delay reduction inside the cluster is overlooked because the impact of smaller multiplexers, reduced loading and smaller cluster input buffers is not considered. Consequently, the delay model produces pessimistic results for both inside the cluster and outside the cluster.[2] According to Sheng [108], these represent roughly 1/3 and 2/3 of the average critical-path delay, respectively. More accurate modelling may produce a savings in either component and lead to a noticeable delay reduction (perhaps as large as 10%).

5.3 Architecture Parameters

The general PLD architecture considered is the mesh-based network model described in Chapter 3. Each clustered logic block (CLB) and the surrounding interconnect forms a layout tile similar to that shown in Figure 5.1. This tile includes the switch block, connection block, and local cluster routing. Further details are described below.

[2]These two conservative delay assumptions are imposed due to limitations of an early version of the VPRx code. All subsequent chapters use the mature version of VPRx which removes this restriction.

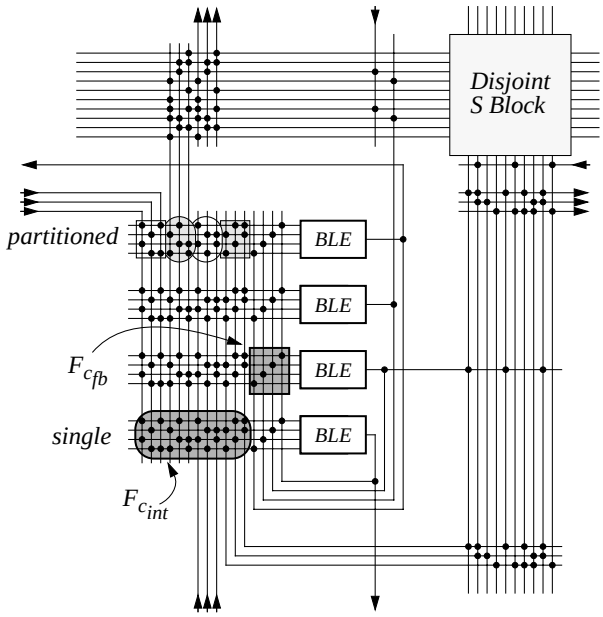


Figure 5.1: Details of the cluster tile architecture.

5.3.1 Routing Architecture

The switch block uses the *disjoint* switch pattern with length four wires. Half of the routing tracks use buffered routing switches, and the remaining half use pass transistors.

Routing Switch Sizes

This section describes how the sizes of the routing switches used in this chapter have been determined from the results of previous work. Betz *et al* [19] has shown that the best area·delay product results are obtained with buffers and pass transistors sized at 5 and 10 times the minimum-width, respectively. This was computed for a cluster based on $k = 4$ and $N = 4$.

Previous work by Betz *et al* [19, 111] suggests that these switch sizes should be increased linearly as the cluster tile length, hence the wiring load, increases. This linear fit is forced through the origin so that switch sizes are doubled if the tile length doubles. For cluster organisations with larger k and N, the tile size increase was calculated by Ahmed in [40] to produce the switch sizes given in Table 5.2. These same switch sizes are used in this chapter.

k	N	**Buffer Size**	**Pass Transistor Size**
4	4	5.0	10.0
4	6	6.1	12.2
5	6	6.6	13.2
6	6	7.1	14.2
7	6	8.9	17.8
7	10	11.8	23.6

Table 5.2: Routing switch sizes ($\times$ minimum size) used for different cluster organisations.

5.3.2 Cluster Architecture

The cluster inputs, LUT inputs, and cluster outputs have a number of design parameters and equivalence relationships that are important during routing. These are described below.

Spare Cluster Inputs

The clustered logic block has I primary inputs which are considered during packing. As well, an additional I_{spare} spare cluster inputs are used during routing only. Hence, each cluster has a total of $I + I_{spare}$ inputs from the routing architecture.

Reserving these spare inputs for routing is useful in two ways. First, some sparse switch patterns are unable to directly connect all possible local feedback netlists. This problem can be solved by allowing these local feedback connections to leave the cluster and then re-enter from the general routing. Since the primary inputs may already be 100% utilised by the packing algorithm, this requires spare cluster inputs. Second, spare inputs can improve routability by providing more alternatives. Routes blocked by sparse switch patterns or a congested routing channel may be able to use a spare input instead. As observed in Chapter 4, planning to under-utilise the cluster inputs also improves the routability of the C block. Other publications [132, 133] have also indicated that a surplus of wiring resources is useful to improve routability.

The number of spare inputs, I_{spare}, is varied as an architectural parameter and specified prior to routing. For convenience in implementation, the packing tool adds these as part of the netlist and the router automatically uses them. This arrangement allows the packer to use the spare inputs in the future (but then they wouldn't be spare!).

Parameter	Description
$F_{c_{int}}$	cluster input to LUT input density
$F_{c_{fb}}$	LUT feedback to LUT input density
F_c	routing channel to cluster input density
$F_{c_{out}}$	cluster output to routing channel density

Table 5.3: Switch density parameters.

LUT Inputs, Cluster Inputs and Outputs

The LUT inputs can select signals from two independent sources: either cluster inputs or feedback connections. The density of switches for these two regions are independently controlled by the parameters $F_{c_{int}}$ and $F_{c_{fb}}$, respectively. These parameters determine the sparseness of switches inside the cluster.

The cluster inputs and outputs use sparse switch matrices to connect to the general routing (in the C block). These matrices use switch densities of F_c and $F_{c_{out}}$ for the cluster inputs and outputs, respectively.

The parameters controlling switch densities inside and outside of the cluster are summarized in Table 5.3. As described later in Section 5.4.3, the cluster input region is partitioned into four groups based on which side the input is placed. This also influences the design of the switch pattern used within the cluster.

Pin Equivalences

Within a BLE, the LUT inputs are assumed to be logically equivalent and hence freely permutable. Additionally, BLE outputs within a cluster are assumed to be logically equivalent. This allows any function to be placed in any of the BLEs of the cluster.

Each BLE output directly drives a cluster output and a local feedback connection. Hence, to achieve the output equivalence, every BLE is given the exact same input switch pattern. While there may be a small routability disadvantage to having the same switch pattern on every BLE, an advantage is gained by making every BLE output equivalent. It would be interesting to make a more thorough evaluation of this tradeoff.

Architectures with different input switch patterns for each LUT can also be built. This

would require a full permutation stage to reorder the BLE outputs as they connect to the cluster outputs and feedback connections. This can be done by fixing $F_{c_{fb}} = F_{c_{out}} = 1.0$, for example, or by using N additional N-input multiplexers. Due to the additional area involved, these alternatives are not considered here.

The cluster inputs are also treated as logically equivalent, but due to reduced connectivity they may connect to only some of the LUT inputs. This may cause difficulty during routing, but it is relieved somewhat by having spare cluster inputs to choose from.

To improve routability, the routing tool takes advantage of the input and output equivalences just described. It may also replicate logic onto multiple BLEs within the same cluster, provided there are empty BLEs available.

5.3.3 Sparse Cluster Switch Patterns

The switch pattern generator from Chapter 4 has been integrated into VPRx to create the switch patterns for the LUT input multiplexers. This generator first distributes switches to balance the fan-in and fan-out of each wire, usually in a random pattern. A greedy improvement strategy is then followed which tries to maximise the number of distinct output wires (LUT inputs) reached by every pair of input wires (cluster inputs). Using this technique, the switch patterns within a cluster are individually well-designed.

Other switch patterns in the routing fabric, namely the cluster C block patterns, use the original VPR switch placement generators. Additionally, no attempt has been made to optimise the cascading of the the cluster input multiplexers and LUT input multiplexers, except as noted below in Section 5.4.3.

5.4 Results

This section presents the area and delay results of routing the benchmark circuits. In all cases, the results are presented as the geometric average of the benchmark set. Initial experiments determine good values for the routing architecture parameters, then the performance of sparse cluster architectures is investigated.

5.4.1 Key to Curve Labels

In the following graphs, each curve represents a family of architectures parameterised along the *x*-axis. Each curve label describes the specific architecture parameters in the following order:

$$k \quad N \quad I_{spare} \quad F_{c_{int}} \quad F_{c_{fb}}$$

Where the value of a parameter is given as 'X', that simply means the parameter is being varied along the *x*-axis. For convenience, this key is repeated in the top-right corner of each set of graphs.

5.4.2 Routing Architecture Selection

To explore the sparse population of switches inside the cluster, it is first necessary to establish a good routing architecture outside of the cluster. Hence, the best values for F_c and $F_{c_{out}}$ are selected below to find switch density values that will result in using near-minimum area. To expediently generate the following results, the number of router iterations is limited to 75. All other parameters remain at their default values.

Selecting F_c for Minimum Area

The density of switches connecting channel wires to cluster inputs in the C blocks is represented by F_c. The choice of F_c depends on the effectiveness of the CAD tools as well as the size of the C block. The C block inputs are formed by the channel wires (W inputs), and outputs are the cluster inputs ($I + I_{spare}$ outputs). Results in Section 4.6.4 indicate that the number of outputs ($I + I_{spare}$) is the most important factor influencing the choice of F_c.

Routing experiments are done with a $k = 7$ LUT size, varying N from 2 to 9. This large LUT size is unlikely to be chosen for a commercial PLD, but using it makes it easier to explore a wide range in I without changing LUT sizes in the middle. Both full (100%) and sparse (50%) population levels inside the cluster are tested. The 50% density is used because this is almost always routable without adding spare inputs, hence $I_{spare} = 0$ here.

The average low-stress channel width required to route the benchmark circuits, $W = 1.3 \cdot W_{min}$, is presented in Figure 5.2 for a variety of F_c values. Only three cluster sizes are

illustrated, but the other results are similar. From these results, it can be seen that choosing $F_c > 0.4$ has little impact on channel width. This is particularly true for larger values of N.

Comparing the sparse and fully-populated cluster results, it is interesting note the channel width requirements are very similar. Sparse clusters typically require only 2 to 4 more tracks than the corresponding fully populated ones. Hence, the sparse architecture is still quite routable at the 50% population level.

Although channel width is not hindered by a large value of F_c, having more switches than necessary contributes to an area increase. To show this, Figure 5.3 plots PLD area versus F_c for cluster sizes 2 and 9.[3] Similar results are obtained using cluster sizes 3 through 8.

One unexpected result shown in this data is that it is better to sparsely populate the inside cluster than outside in the routing channel. This can be seen in Figure 5.3 where point B is lower than A, and D is lower than C. This trend holds for the other cluster sizes as well. Hence, for these benchmarks, it is better to use a 50% switch density inside cluster than any switch density in general routing. It is not clear if this observation will continue to hold with larger benchmark circuits that require wider routing channels.

One explanation of why sparse clusters are better is that the LUT input multiplexers outnumber the cluster input multiplexers. Using the equation $I = \lfloor k(N+1)/2 \rfloor$ to determine the number of cluster inputs results in twice as many LUT input multiplexers. Even though the cluster input multiplexers may have twice as many inputs (or more, depending on the channel width), the number of multiplexers is more significant. Reducing the width of the LUT input multiplexers saves more area due to the number of SRAM bits saved than reducing the width of the cluster input multiplexers.

Figure 5.3 also suggests that sparse clusters can shift the optimum design point toward larger clusters containing more LUTs. This is shown by a significantly larger area reduction for $N = 9$ than $N = 2$. The reduction is large enough that the $N = 9$ architecture goes from using more area (curve C) than the corresponding $N = 2$ architecture (curve A) to using less area (curves D and B). Although the results are not shown here, this $k = 7$ architecture is

[3]Notice that the sparse $F_c = 1.0$ result is missing for $N = 9$ in Figures 5.2 and 5.3. This is because slow convergence did not allow VPRx to route the *clma* circuit under low-stress conditions.

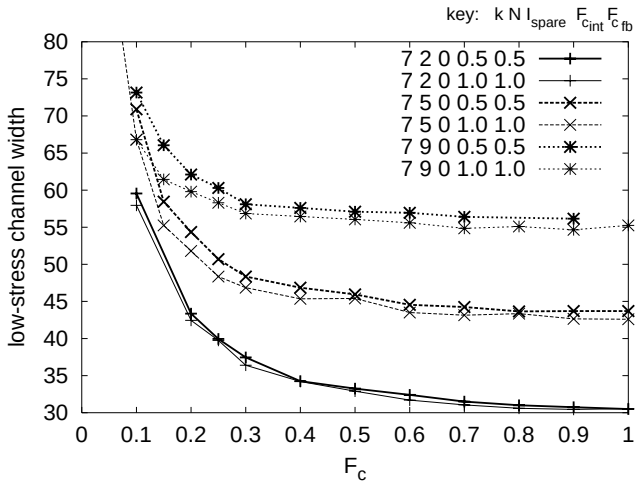


Figure 5.2: F_c impact on channel width.

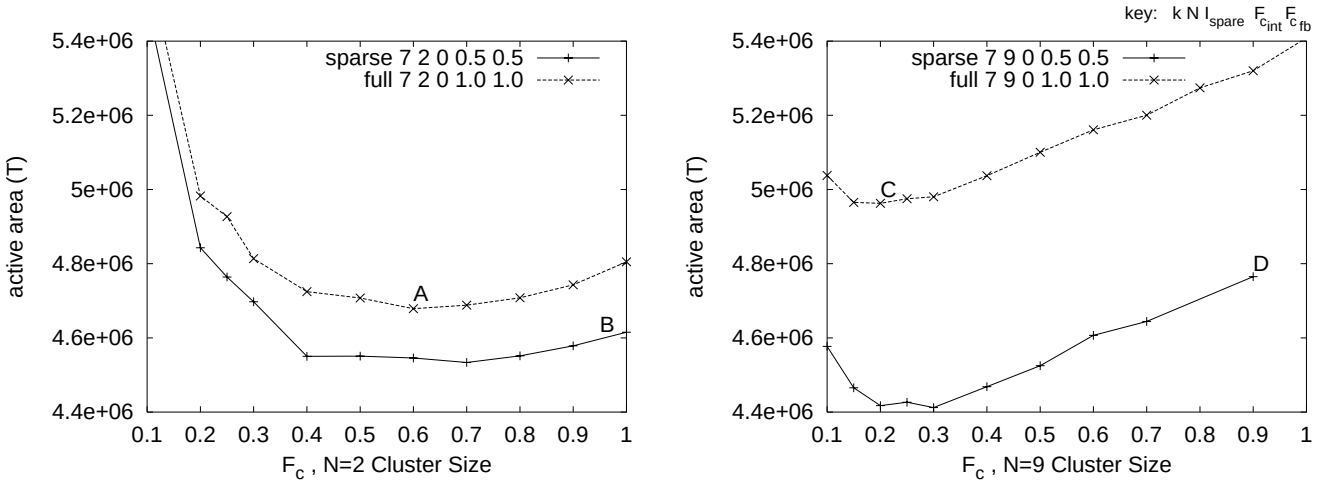


Figure 5.3: F_c impact on area for cluster sizes of $N = 2$ and 9.

most efficient with clusters containing $N = 4$ to 6 LUTs if the cluster interconnect is fully-populated, or 4 to 9 LUTs if only 50% populated. Most of the work in this dissertation assumes $N = 6$. This particular size is good across a wide range of LUT sizes and cluster interconnect population levels.

There is significant motivation for using larger clusters because it helps reduce placement runtime [19]. Since previous work has always assumed a fully-populated cluster interconnect, further investigation of the optimum number of LUTs per cluster is warranted.

The value of F_c that produces the lowest area for each value of N is shown in Figure 5.4. The x-axis in this graph is labelled with the number of cluster inputs, I, that corresponds to

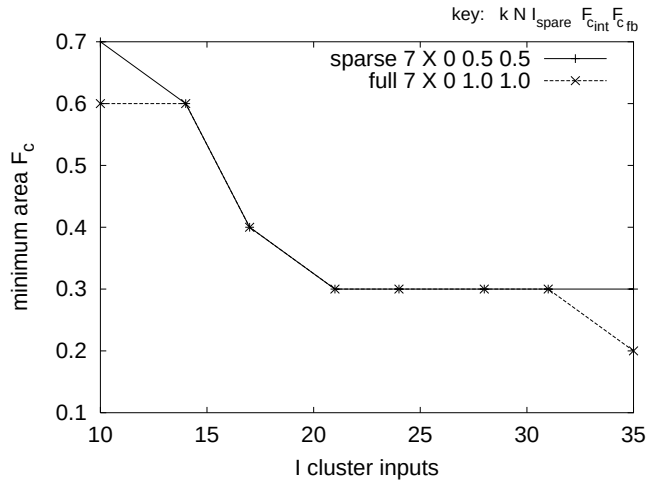


Figure 5.4: Best F_c for minimum area with $I = \lfloor 7(N+1)/2 \rfloor$ cluster inputs.

each cluster size. It is worth noting that the best F_c results with sparse and fully populated clusters are very similar. This can be partly attributed to the relative flatness near the minimum area. For $N = 2$, both area curves are nearly flat for $F_c = 0.4$ to 0.8. For $N = 9$, varying F_c from 0.1 to 0.5 causes less than 5% change in area.

The flatness just described implies that *precise F_c selection is not critical*. This is true as long as F_c is large enough to be routable, yet not wastefully large. It also suggests that a fixed value of F_c will not hinder area results, and that choosing a slightly larger F_c than suggested by Figure 5.4 will not cause a significant area penalty. Instead, a larger F_c may help routability as clusters are made sparse. This assistance is important to mitigate effects such as interference patterns which may arise from cascading the C block and internal switch patterns.

The selection of F_c for all experiments in this dissertation is based on the information just presented and historical values used in previous literature. The value of $F_c = 0.5$ is used for all of the $N = 6$ architectures, and $F_c = 0.366$ is used for the $k = 7, N = 10$ architecture. By selecting these values to be the same as those used in previous work [40], direct area comparisons can be made.

Selecting $F_{c_{out}}$

Previous work has shown that $F_{c_{out}} = 1/N$ is adequate for routing in fully populated architectures [19]. This gives a clustered logic block access to every wire in the routing channel. Considering the similarity of the F_c area results between sparse and fully populated architectures, it is unlikely that modifying this relationship for $F_{c_{out}}$ would have significant impact in a sparsely connected architecture. Hence, $F_{c_{out}} = 1/N$ is used for all experiments.

5.4.3 Partitioning of Cluster Inputs

Additional net delay may occur in sparsely populated clusters because some LUT inputs are not reachable from some sides of the cluster. For example, consider the case where some LUT input connections have already been formed and the last remaining input signal is being made. A lack of switches inside the cluster may cause that net to enter the cluster from a more distant side, resulting in greater delay.

To investigate this problem, two different switch matrix designs were evaluated within the cluster: one is a *single* switch matrix for all cluster inputs, while another is *partitioned* into four smaller switch matrices (one for the inputs from each side). Both of these matrices are illustrated in Figure 5.1. The partitioned matrix addresses the above problem by ensuring that all of the cluster inputs from any particular side can reach all of the LUT inputs. It also has a weakness though: these smaller switch matrices are not carefully co-designed to couple together.

The procedure for producing the partitioned switch matrix is as follows. Each individual submatrix uses the same base switch pattern, but each has its own permutation of the rows (or outputs) to balance the fan-in of the LUT inputs. The base switch pattern is generated in the same way that a single switch matrix would be generated using the techniques from Chapter 4.

To compare the two switch designs, the benchmark circuits are routed in the $k = 7$, $N = 10$, $F_{c_{int}} = F_{c_{fb}} = 0.43 \approx 3/7$ architecture. Both designs require identical transistor area (channel width), but the *partitioned* matrix is about 1% faster on average. Although this is not significantly faster, the partitioned matrix is used for all subsequent experiments

because it may help with some pathological cases.

5.4.4 Sparse Cluster Area Results

The primary motivation for depopulating clusters is to reduce the area of a PLD. Section 5.4.2 has shown that simply depopulating the cluster to 50% is more effective at reducing area than choosing the best value of F_c. In this section, depopulation of the cluster below the 50% level is explored.

To reduce the number of routing experiments, the cluster size is fixed at $N = 6$ and the LUT size is varied from 4 through 7. The larger LUT sizes are especially interesting because their switch matrices have more outputs, hence they offer more potential for depopulation. One additional architecture with $k = 7, N = 10$ is used to study a situation with an even larger number of cluster inputs.

To evaluate sparse clusters, a number of preliminary routing experiments were run with a wide range of values for $F_{c_{int}}$ and $F_{c_{fb}}$. From these results, which are not shown here, it was confirmed that $F_{c_{fb}}$ has less influence on area. However, as $F_{c_{fb}}$ was reduced below 50%, it was observed that a number of circuits could no longer be routed. It was determined that $F_{c_{fb}} = 50\%$ (or $= 43\%$ for $k = 7$) is as low a value that can be tolerated. Similarly, these preliminary sweeps indicated that $F_{c_{int}} \geq 0.5$ is nearly always routable, so experiments did not need to consider higher switch densities.

The area results from routing the four LUT sizes are shown in Figure 5.6. In these graphs, each curve represents the geometric average of PLD area for a fixed value of $F_{c_{int}}$. The number of spare inputs is varied along the x-axis. The sparse cluster results should be compared against the bold curve representing the fully-populated cluster area.

The most apparent trend in these curves is a gentle dip, then a general upward climb in area as I_{spare} is increased. The upward trend is an expected result, since each spare input requires an additional cluster input multiplexer. The dip is caused by a rapid initial decline in average channel width. The channel width then gradually reaches a 5% to 20% reduction (10% is typical). To illustrate this, the average low-stress channel width for all 20 circuits is shown as a function of I_{spare} in Figure 5.5. These results are given for a fully-populated cluster, but sparse cluster results are similar. Spare inputs are an effective way to reduce

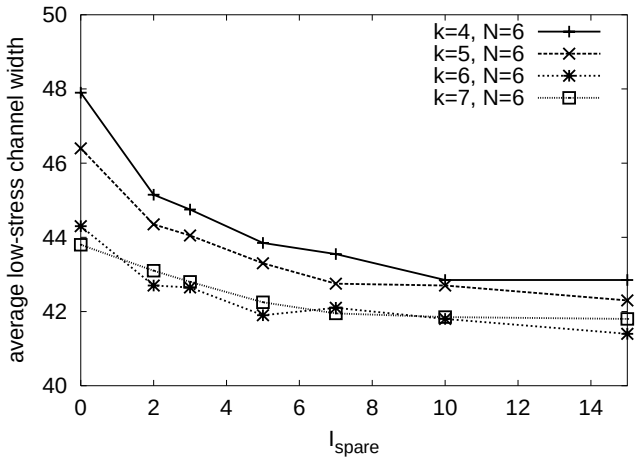


Figure 5.5: Spare inputs reduce channel width in fully populated clusters.

channel width demands.

A number of data points are missing in Figure 5.6, specifically for small I_{spare} values. This is because one or more benchmark circuits could not be routed on the architecture. Hence, although they contribute to area reduction in a few cases, it is more important to have these spare inputs to make sparse clusters routable. Typically between two and five spare inputs are required to make the architecture routable and attain the lowest area. Hence, providing one spare input per side is a reasonable guideline for PLD architects.

The lowest-area architectures from Figure 5.6 are summarized in Table 5.4. As well, the large $N = 10$ cluster architecture is included. This table compares the baseline case, a fully-populated architecture (with no spare inputs), to the lowest-area sparsely populated architecture (with spare inputs). The table also illustrates the channel width savings that can be achieved if the I_{spare} spare inputs from the sparse architecture are given to the fully populated architecture (note, however, that this is not the lowest channel width that can be achieved). With these architectures, a 10 to 18% area savings is achieved using sparse clusters and spare inputs.

When moving from a fully populated cluster to a sparsely populated one, the best-area architecture shifts from a $k = 4$ input LUT to a $k = 6$ one. This produces a 13% area savings. As will be shown below, a $k = 6$ architecture also results in faster circuits — this win-win combination gives compelling evidence to reconsider $k = 6$ input LUTs in

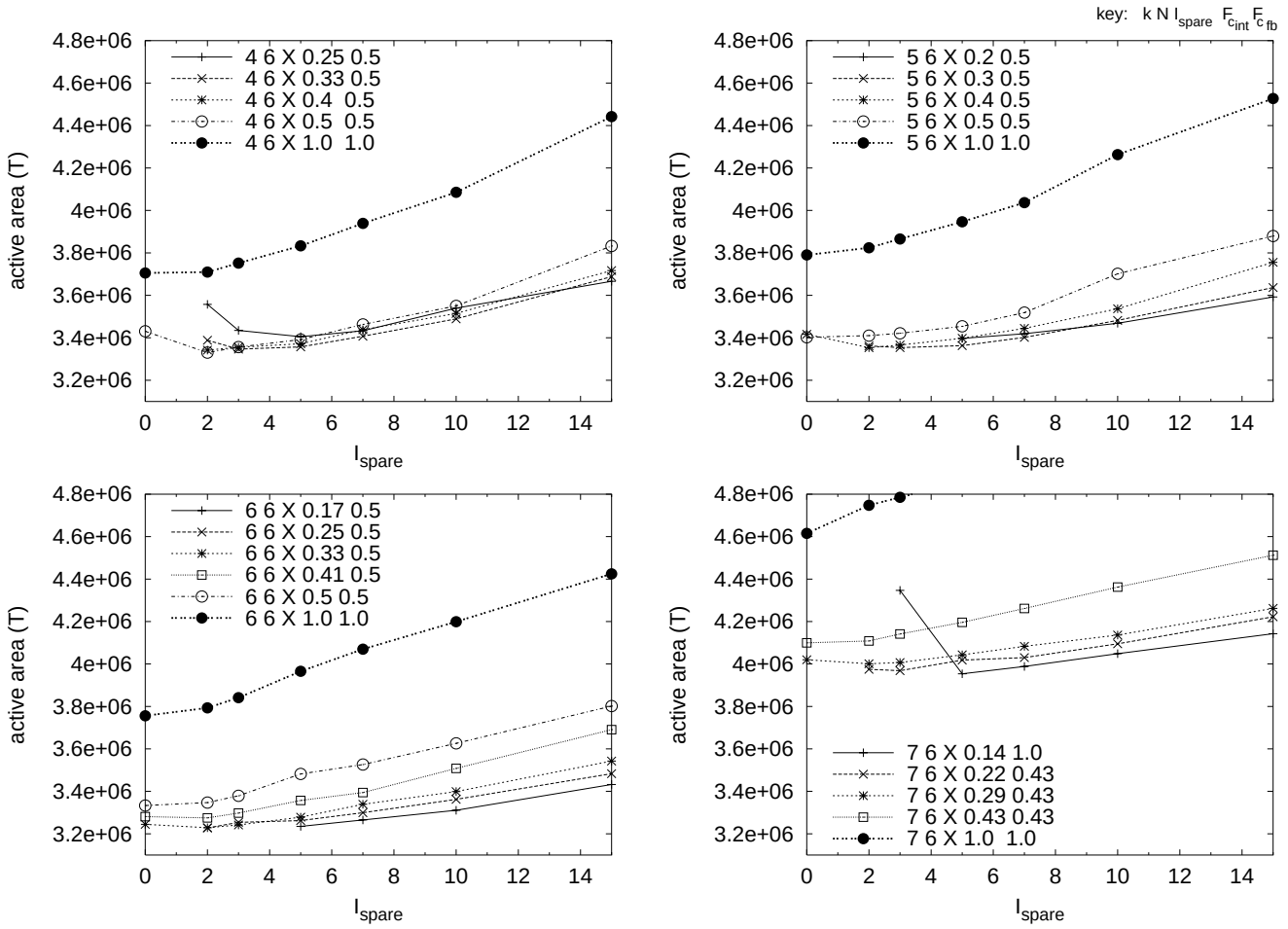


Figure 5.6: PLD area of fully and sparsely populated clusters.

Architecture				Best Sparse Parameters			Channel Width (arith. avg.)			PLD Area ($\times 10^6$ T)		
k	N	I	F_c	I_{spare}	$F_{c_{int}}$	$F_{c_{fb}}$	Fully Populated		Best Sparse	Fully Populated	Best Sparse	Savings
							no spares	spares	spares	no spares	spares	
4	6	14	0.5	2	0.5	0.5	47.9	45.2	45.9	3.71	3.33	10.1%
5	6	17	0.5	2	0.4	0.5	46.4	44.4	45.6	3.79	3.35	11.5%
6	6	21	0.5	2	0.33	0.5	44.3	42.7	43.5	3.76	3.23	14.0%
7	6	24	0.5	5	0.143	0.43	43.8	42.3	44.6	4.62	3.95	14.3%
7	10	38	0.366	10	0.143	0.43	53.7	n/a	55.1	4.96	4.03	18.8%

Table 5.4: PLD area savings obtained by depopulating switches inside the cluster.

commercial PLDs.

A breakdown of the cluster tile area is given in Table 5.5. These results show that fully populated LUT input multiplexers consume 24–33% of the total tile area. After the cluster interconnect is made sparse, these consume only 12–18% of the tile area. For 4-

Architecture		Tile Area (Number of Minimum-Width Transistor Areas)									
		Fully Populated Cluster					Best-Area Sparse Cluster				
k	*N*	Total	LUT+FF	Routing	LUT Input Mux		Total	LUT+FF	Routing	LUT Input Mux	
4	6	9307	990	6050	2267	(24.4%)	8380	990	5960	1430	(17.1%)
5	6	11241	1840	6321	3080	(27.4%)	9964	1840	6371	1753	(17.6%)
6	6	14318	3496	6713	4109	(28.7%)	12343	3496	6732	2115	(17.1%)
7	6	19622	6831	7645	5146	(26.2%)	16879	6831	8120	1928	(11.4%)
7	10	35145	11358	12022	11765	(33.5%)	28646	11358	12990	4298	(15.0%)

Table 5.5: Breakdown of cluster tile area. The routing area is an arithmetic average.

input LUTs, there is also a slight decrease in routing area because the spare inputs helped reduce average channel width. The 5- and 6-input LUTs cases do not reduce routing area because the spare inputs contribute more to area than the amount saved from channel width reduction. The two 7-LUT architectures also have an increase in routing area, but this is due to having both spare inputs and a channel width increase. The net effect, however, is that the sparse clusters produce a net area savings of 14% and 18%, with the larger LUT and cluster sizes benefitting more.

One very interesting result with sparse clusters is that the 6-input LUT architecture is 3% more area-efficient than the 4-input LUT architecture. This is a departure from previous work which has consistently shown that 4-LUTs achieve lower area with fully populated clusters. The reason for this difference is simple: larger LUTs provide more opportunity for depopulation. Despite the small area advantage shown here, 6-input LUTs are not used in any current commercial PLD.[4] This is possibly because PLD vendors have a larger benchmark set containing larger circuits, and because they have additional features such as carry chains which can be exploited. However, these results motivate more effort to be placed in researching architectures and CAD tools for 6-input LUTs.

5.4.5 Sparse Cluster Delay Results

As mentioned earlier, reduced switch densities may cause an increase in delay due to an increase in bends or wire use to achieve routability. Although delay may decrease for other reasons such as reduced loading, the experimental conditions here are conservative and

[4] A 6-input, 2-output LUT is used in HP Teramac [120], but it is not commercially available.

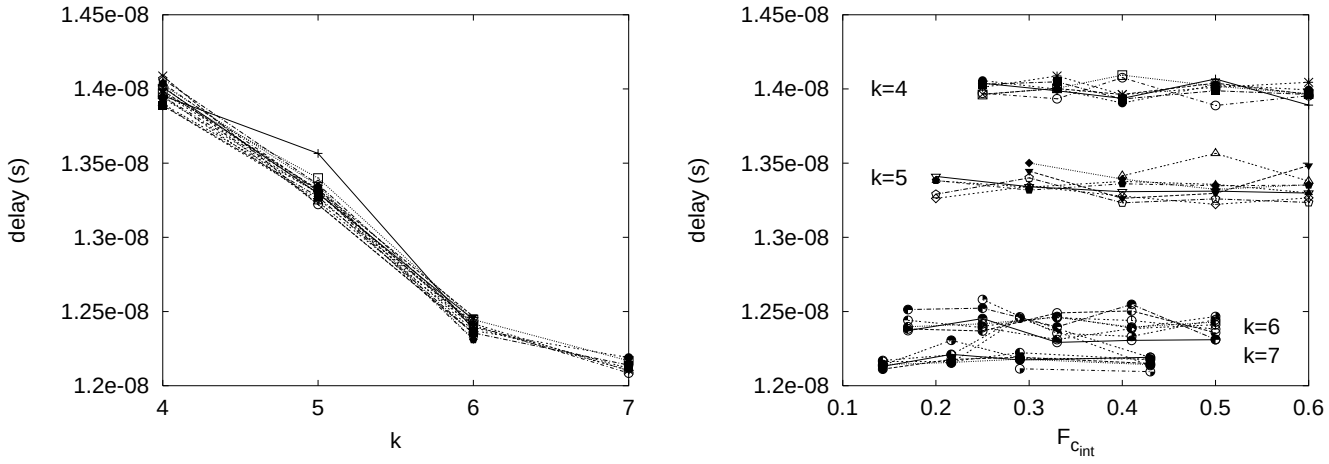


Figure 5.7: Delay depends on LUT size (left), but not on switch density (right).

ignore these possible benefits.

The curves in Figure 5.7a) show the impact that varying the LUT size has on delay for a variety of the $N = 6$ architectures with different switch densities and spare inputs. The curve labels identifying the architectures have been omitted for clarity, since only trends need to be observed. The important thing to notice is that, for all architectures, delay goes down as k increases.

Similarly, Figure 5.7b) shows the change in delay as the switch density $F_{c_{int}}$ is varied. It is apparent in the graph that curves for the same LUT size are all grouped together. In particular, the 4- and 5-LUT data is easily distinguished from the 6- and 7-LUT data. The flatness of all of these curves illustrates how little impact $F_{c_{int}}$ has on delay.

Analysis of delay while varying I_{spare} or $F_{c_{fb}}$ shows the same result: delay is independent of these parameters. Even though sparse clusters remove many choices, and even though some local feedback connections must use the general-purpose routing, the router can still assign the fastest paths to the critical sinks.

5.4.6 Sparse Cluster Area·Delay Product

The previous two sections presented results indicating the 6-LUT has the lowest area and the 7-LUT has the lowest delay. When the area·delay product is formed, the 6-LUT emerges as the superior choice. This metric is important because it indicates when the

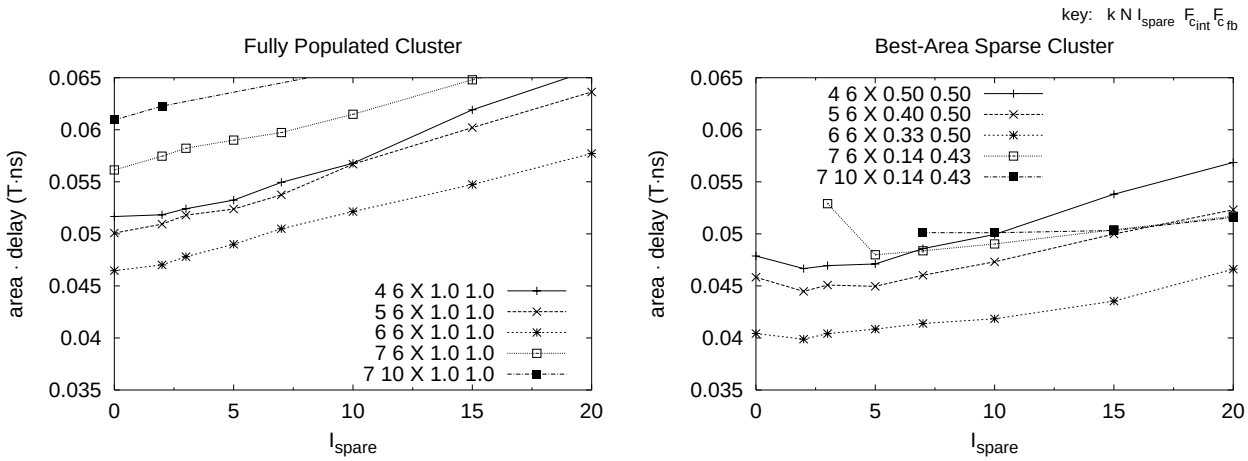


Figure 5.8: Area·delay for fully-populated (left) and best-area sparse (right) clusters.

best tradeoff is being made between using an additional amount of area for a similar relative gain in clock rate (or vice versa).

The best organisations for area·delay product are compared in Figure 5.8. Sparse clusters improve the area·delay product at every LUT size due to the area savings. The overall best sparse architecture containing 6-LUTs is about 14% more efficient than best sparse one containing 4-LUTs, and about 22% more efficient than the traditional fully-populated 4-LUT cluster.

5.4.7 Routing Runtime with Sparse Clusters

The removal of switches inside the cluster also removes the guarantee of routability within the cluster. Consequently, the router must pay attention to all of the wires and switches within the cluster. This should result in an increase in the routing runtime.

The average runtime and average number of iterations required to route the different architectures are shown in Table 5.6. These data are obtained using an 866 MHz Intel Pentium III computer with 512MB of SDRAM, and represent the arithmetic averages of routing the 20 benchmark circuits. Results are presented for fully populated clusters to compare the original VPR 4.30 to the modified one, VPRx. As well, VPRx is compared against itself to study the additional impact of routing the best-area sparse clusters.

Generally, VPRx currently runs about three to four times slower than the original ver-

sion when fully populated clusters are used. Even though runtime has increased, the number of router iterations used is practically unchanged. More work is done each iteration because there are more wires and switches to be considered in a sparse cluster architecture. Also, nets are allowed to enter a cluster more than once, so more routing paths are evaluated before making a decision.

Despite the increased number of options, the main reason for the slowdown comes from the increase in the number of sinks that must be routed. This can be explained and runtime can be reduced as follows. Routing with sparse clusters requires each LUT input, rather than each cluster input, to be regarded as a sink. This is particularly expensive for high-fanout nets, since timing-driven routing rebuilds the search heap for each sink.

However, sinks within the same cluster can be grouped and routed successively. When doing this, the search heap only needs to be re-seeded with a few nodes (this work uses 3) while backtracing from the previously routed sink to the source. This modification produces similar quality results, yet cuts runtime almost in half. The routing data (and runtimes) presented in this chapter use the slower algorithm, but the other chapters use the faster algorithm.

It is worthwhile to note that having larger LUT sizes and cluster sizes reduces the amount of work that VPR 4.30 must do, so runtime decreases. This benefit is not realised with VPRx because the amount of wiring inside the cluster also increases, keeping runtime relatively flat.

The additional runtime needed to route the best-area sparse architectures is also shown in Table 5.6. For $k = 4, 5, 6$ the runtime and the number of iterations is similar. However, the $k = 7$ runtime nearly doubles and the number of iterations increases by 25–30%.[5] This increase in the average is caused by a large increase for four of the normally difficult-to-route circuits. The need for more router iterations suggests these architectures are barely routable, probably because $F_{c_{int}}$ is so low.

[5]The search space usually increases with each iteration, so the runtime of each iteration tends to increase as well.

		Average Runtime (seconds)			Average # Routing Iterations		
		VPR 4.30	VPRx		VPR 4.30	VPRx	
k	*N*	Fully Populated		Best Sparse	Fully Populated		Best Sparse
4	6	70	153	150	84	86	86
5	6	72	183	205	93	91	93
6	6	57	177	178	84	86	88
7	6	53	188	350	88	83	109
7	10	43	177	275	96	94	116

Table 5.6: Average runtime and number of routing iterations for the low-stress route.

5.5 Comparison to Previous Work

The use of fully-connected clusters likely stems from previous work by Rose and Brown [42] which suggested that inputs of a 4-LUT be fully connected to the routing channel. They indicated that this is required to obtain minimum channel widths, the area metric in use at that time. Since then, clustered architectures have become prevalent, CAD tools have improved, and area metrics have become more detailed.

Reducing the amount of connectivity within the cluster was more recently explored using a simple striped switch layout [64]. Rather than modify the router, the T-VPACK packing algorithm was altered in such a way that routability of the cluster was still guaranteed. Unfortunately, the area improvement obtained using this technique was limited to 5% and delays increased up to 30%.

In this work, the packing algorithm is left unchanged. Instead, improved switch patterns are used, spare cluster inputs are added to the cluster, and modifications to the router are made to support these architectural changes. Although these spare inputs contribute to additional area, they also improve routability and reduce channel width requirements. Overall, a net area reduction of up to 18% is obtained with no degradation to critical-path delay.

Comparisons with existing commercial PLDs can be made as follows. Altera's products have traditionally used a fully connected local cluster interconnect [30]. However, the Altera Stratix [38] architecture, which was designed in parallel with or after this work, uses local cluster interconnect populated at 50% to save 7% in area. This is less than the 10%

savings estimated in this chapter, probably because the Stratix devices have wider routing channels. The Xilinx Virtex architectures [31] use a modified switch block that flattens the traditional switch block, connection block, and internal cluster connection structures. This effectively creates sparse connections within the cluster but it also adds connectivity to the traditional switch block role.

5.6 Conclusions

This chapter has studied the area and delay impact of sparsely populating the internal connections of a clustered architecture. At the expense of three to four times the compute time, an area savings of 10 to over 14% is realised by sparsely populating the cluster internals of 4-, 5-, 6-, and 7-input LUT architectures containing 6 LUTs per cluster. A larger cluster size of ten 7-LUTs obtained an 18% area savings. The additional router effort and reduced routing flexibility does not come at the expense of critical-path delay. It remains unchanged.

The increase in routability and the decreases in channel width and area indicate that it is best to force the packing algorithm to leave a few spare inputs (two or three) for the router. By adding up to 15 spare inputs, the channel width decreases by about 10% in most architectures, whether full or sparsely populated. These inputs are used only by routing, and are not used or required for packing. Although sparse clusters require a small increase in channel width, the spare inputs have the opposite effect. The net effect is a small savings in channel width.

There are two other noteworthy observations made in this chapter which run contrary to popular belief. First, it may be more area-efficient to depopulate only the LUT input multiplexers than the cluster input multiplexers. Of course, depopulating both regions provides even more savings. Second, 6-input LUTs may be more area efficient than 4-input LUTs when sparse clusters are employed. However, these observations may change if better benchmark circuits and more realistic CAD tools (with carry chains) are available.

The area and delay results in this chapter use conservative estimates and ignore secondary effects which may improve results further. In particular, the tile size reduction from

employing sparse clusters should improve delay by a small amount. Delay improvement may also come from reduced loading inside the cluster and by generally employing larger cluster sizes, which are more area-efficient when using sparse clusters.

5.7 Future Work

Future work in this area will include effort to jointly design the LUT input switch matrices with the cluster input multiplexers to avoid switch pattern interference. Additional constraints such as carry chains or other local routing may impact sparse cluster design and should be considered. A wider variety of cluster sizes, particularly the effectiveness of large clusters, should also be explored. The area savings from sparsely populated clusters will reduce tile size, but the subsequent area and delay reduction from using smaller routing switches should also be quantified. The delay improvements arising from reduced loading and larger cluster sizes should be investigated. Also, efforts should be made to improve the runtime of the router while still retaining the area savings.

An interesting extension of this work is to involve tighter coupling between sparse clusters and the packing tool. For example, under special circumstances, it may be reasonable to have the packing tool use the spare inputs reserved for routing. Before doing this, it could first do a routability test to verify whether the potential cluster of logic blocks is routable. Since this shouldn't be a common situation, it can be done with reasonable CPU effort. This may increase the usefulness of the PLD architecture for subcircuits which have wide fan-in (or poor input sharing), such as finite state machines.

Chapter 6

Routing Switch Circuit Design

In commercial PLDs, buffers have recently replaced pass transistors as the preferred type of routing switch. This chapter investigates interconnect which mixes these two switch types together. The goals of this study are twofold: to reduce area by replacing a number of buffered switches with pass transistors and to reduce delay by allowing a signal to alternate between a buffer and a pass transistor switch. This is accomplished with three evolutionary steps. The first step is detailed transistor-level design of routing switches in 0.18μm technology. The second step is the evaluation of three new switch types which combine the advantages of two previously-used switch designs: they are nearly as fast under fanout as the fastest design, and they use less area than the smallest design. The third step is the evaluation of PLD architectures which mix buffers and pass transistors by replacing existing buffers with pass transistors.

6.1 Introduction

It is well known among VLSI designers that the propagation delay through one pass transistor is less than the corresponding delay through one buffer. However, it is also known that placing many pass transistors in series is much slower than a similar chain of buffers because delay grows quadratically with the former, but linearly with the latter. The rule of thumb for the equivalent delay point is usually three or four series connections. This section motivates the detailed, transistor-level design of buffered switches and the need for

mixing both buffered and pass transistor switches to reduce area and delay.

6.1.1 Transistor-Level Switch Design

Before architectures which contain both pass transistors and buffers can be explored, extensive transistor-level design work must be done to ensure the switches considered are as small and fast as possible. To do this, a modern 0.18μm technology is used. Full HSPICE-level design is necessary to ensure that practical considerations are not overlooked when doing architectural work. For example, at this 0.18μm technology node and beyond, new solutions to the leakage current problem must be found. Level-restoring is pursued as a solution here.

A number of other discoveries have been made from this detailed transistor-level design work. The first discovery challenges a design practice used in previous work [19, 40, 111] where routing switches are scaled in size linearly according to the logic block tile length to achieve the best delay results. The linear fit employed in that work is forced through the origin, so switch sizes are doubled if the tile length doubles. Although it is clear that longer wire loads require stronger buffers, it is not clear whether this is an appropriate scaling mechanism. This chapter shows that a fit through the origin is inappropriate for optimal delay, and that a *fixed* switch size is sufficient to obtain within 5% of optimal delay or area·delay for a wide range of tile lengths.

Second, this chapter also determines that the constant delay model used internally by VPRx to model buffers is sufficiently accurate for routing. This is important because the number of pass transistors on the signal path prior to a buffer may not be easy to determine while a sink is being routed. Also, the routing of subsequent sinks for a net may further alter the slew rate for previously-routed sinks. It is found that a total path delay increase of less than 10% is caused by the poor slew rates obtained after a signal travels through eight wires connected in series by pass transistor switches. Hence, the constant buffer delay model used within VPRx is reasonably well-founded. However, detailed timing analysers which are used after routing should probably account for input slew rate for accurate results.

A third outcome of the detailed transistor-level design work is the design of three new routing switch circuits. Previous circuits from [19] either use a large amount of area (when

switches don't share buffers) or they suffer from large delay increases (up to 100%) when a buffer must drive multiple fan-outs. This work proposes using fanin-based switches which virtually eliminate large fanout delays. Although the new switches increase delay slightly (by up to 20%) when there is no fanout, the presence of fanout is common enough to achieve a net performance advantage. The Alexander architecture [134], a research project from Xilinx, also uses fanin-based switches which can directly accept logic outputs. The development of the Stratix architecture also advocates the use of fanin-based switches [38]. The new switches investigated here appear to be similar, but the area and delay advantages are better quantified. The new switch types can improve delay by 7% and area·delay by 9%.

6.1.2 Mixing Buffers and Pass Transistors

PLD interconnect is often based on tristate routing switches where only one driver per wire is programmed to be active. Older PLDs such as the Xilinx XC4000 use pass transistors as unbuffered switches, but modern PLDs such as the Xilinx Virtex II and Altera Stratix now use buffered switches. Buffers have become necessary to avoid the quadratic delay growth associated with the long connections that must be formed as the number of CLBs increases. However, the use of buffers significantly increases area, and they are slower for short connections.

One commercial approach that combines buffers with pass transistors is the Xilinx XC4000X architecture [135], where every group of six pass transistor switches also includes one buffer switch to break long RC chains. Academic studies have considered the idea of mixing buffers and pass transistors in another way. Betz *et al* [19] creates two separate types of routing tracks: one set of tracks use buffers while the other set uses pass transistors. This way, short connections can use the pass transistor tracks while longer ones can use the buffered tracks. The Xilinx Virtex architecture also follows this approach by using pass transistor switches in tracks containing length-1 wires and buffers for longer wires.

Later academic work by Sheng *et al* [108, 136] obtains better delay results by placing additional routing switches between the buffered tracks and the unbuffered tracks. This

allows a signal to cross over from one track type to the other. However, it adds to area because the number of switches or transistors per track increases. Despite the area increase, that work shows reductions of 10% to delay and 6% to area·delay. In comparison, the work presented here does not add new switches, it merely replaces buffers found in the buffered tracks with pass transistors. It is shown that alternating between buffers and pass transistors can hypothetically improve connection delay by up to 25%. After routing benchmark circuits with VPRx, real critical-path delay is reduced by 4–6% and area·delay is reduced by 11–14%. Alternatively, area can be reduced up to 13% while increasing delay only 1%.

6.1.3 Related Work

There is little published work on the circuit design of PLD routing switches. Dobbelaere [137] proposes a novel, self-timed circuit that speeds signals using pass transistors, but it has metastability implications. Circuit design issues for building the LEGO FPGA are described in [138]. Work by Khellah [139] touches on pass transistor sizing. Betz [19] has shown that buffers at 5 times minimum size and pass transistors at 10 times minimum size make low area·delay interconnect in 0.35μm technology. Some of that work is extended here in greater detail using 0.18μm technology.

Other circuit design work such as [140] has focused on the production of a PLD which uses low energy, particularly in the interconnect. For this purpose, a low voltage swing driver with level-restoring is used in [141]. Later work further reduces energy by using a lower-voltage driver and a more complex receiver circuit [142]. However, the area required by the receiver (about ten transistors each) may be too large for PLD use. A comparison of various low-swing techniques suitable for PLD use can be found in [143].

6.2 Methodology

This section describes the methodology used for the circuit simulations in Section 6.3 and details about the routing procedure used in the later sections.

6.2.1 Circuit Simulation

All circuit design work is conducted with the HSPICE simulator using "typical" process corner models of TSMC's 0.18μm technology. Delay measurements are taken when the signal passes through $(V_{dd} - V_t)/2 = 660$mV at both ends. Only the worst-case of the rising or falling delay times are used, and voltage swings are always begun at full rail voltage. To account for slow input slew-rate effects, step inputs are conditioned by passing the signal through two identical circuits. Delay measurements are taken from the second of these circuits.

It is assumed that metal wires are implemented in metal-3 using minimum-width, at twice the minimum spacing. Betz [19] found that this is the most effective way to reduce wiring capacitance and improve delay.

Based on the logic and average routing area required by a cluster of four 4-input LUTs, an estimate of 116μm is used as the CLB tile length for most HSPICE modelling. As will be shown later, the precise tile length is not critical in determining the best buffer construction.

6.2.2 Routing Experiments

First, the minimum channel width required to route each circuit, W_{min}, is determined using the baseline routing architecture presented below. Then, the performance of each specific architecture is evaluated by *rerouting* each circuit with $1.2 \cdot W_{min}$ routing tracks. All area and delay results are reported from the completion of this low-stress route. The rerouting is necessary for two reasons. First, some architectures differ slightly in switch block topology. Second, the timing-driven router should reroute to make appropriate delay-oriented decisions when forming connections with the different switch types.

Due to extensive circuit design changes, such as removing gate boosting, VPRx-computed delay times in this chapter are slower than the delays reported in Chapter 5 using the same technology. Adopting this new circuit design work into that chapter would not change the fundamental conclusions therein because all of the results would be equally affected.

6.2.3 Baseline Routing Architecture

The baseline routing architecture uses only length 4 wires, with half the tracks connected by size 16 pass transistors, and half connected by size 6 buffered switches. This choice of switch sizes will be explained later, in Section 6.3.4.

The routing architectures considered in this chapter will use three new types of buffered switches and replace some of these buffers with size 6 pass transistors. It should be emphasized that these changes only affect the half of the tracks containing buffers; the other half always use size 16 pass transistors.

6.3 Detailed Circuit Design

This section investigates circuit design issues relevant to routing switches. First, it presents a case for mixing buffers and pass transistors. Then, it addresses leakage current, buffer construction, and transistor sizing. Next, it determines optimal switch sizes for PLD interconnect. Lastly, it verifies two assumptions about PLD switches: whether it is important to scale the switch sizes as the tile length (or wire length) grows, and whether a constant delay timing model for buffers is sufficient.

6.3.1 A Case for Mixing Buffers and Pass Transistors

The linear delay growth of buffered routing switches makes them essential for use in large PLDs where quadratic delays would be intolerable. Unfortunately, buffers are slower for short connections and require approximately 2–4 times more area than pass transistors.

The advantages of both switch types can be gained by placing a buffer after every N pass transistors, a concept called *buffer/N-pass* switching. Some routing architectures which support buffer/pass switching will be examined in Section 6.5, but this section performs the transistor-level circuit design of the buffers and pass transistors working alone and also as buffer/N-pass switches working in tandem.

Typical performance results from using a few switch types and sizes are shown in Figure 6.1. The graph plots end-to-end connection delay through one to eight routing wires

connected in series. In the figure, the delay for one wire (at the far left) always includes the delay of an initial buffer (of the same size as the other switches).

The results in Figure 6.1 show the quadratic delay growth of pass transistor switches compared to the linear delay growth obtained with buffers. For size 6 switches, the two switch types have similar delay after four routing wires are connected. The use of wider pass transistors (size 16) greatly improves performance, making it equivalent in delay to size 6 buffers at roughly seven interconnect wires. However, the use of a larger buffer improves delay only slightly – the best performance is at size 9, but this is not shown in the figure. Instead, an equivalent delay improvement of roughly 9% can be obtained after eight routing wires by replacing every other size 6 buffer with a size 6 pass transistor. Hence, it is possible to reduce delay **and** area. This delay improvement increases to 25% by increasing the size of these buffer/1-pass switches to 16. Note, however, that these improvements are for only one point-to-point connection. The amount of this improvement that can be seen in actual netlists will depend upon a number of other factors, such as logic depth and fanout.

From these results, it can be seen that the buffer/1-pass switches are able to capture the best delay characteristics of both buffers and pass transistors. Due to rebuffering, the delay growth after a number of buffer/1-pass switches is linear. However, the use of pass transistors lowers the slope of this linear growth below that of using buffers only. This presents a strong case to consider the effect of mixing buffers and pass transistors in this chapter.

Throughout this chapter, circuit design decisions are made for buffers and pass transistors to optimise the separate cases of when they are used alone and when they are combined in buffer/1-pass switching. Later, routing experiments will be performed with new architecture topologies that implement buffer/N-pass switching as well as other ad hoc schemes.

6.3.2 Leakage Current Problem and Solution

With the extensive use of pass transistors as routing switches, the problem of excessive *leakage current* arises. One solution to this problem is the use of *level-restoring pullups* to restore logic high signals. The problem, the solution, and its side effects are described

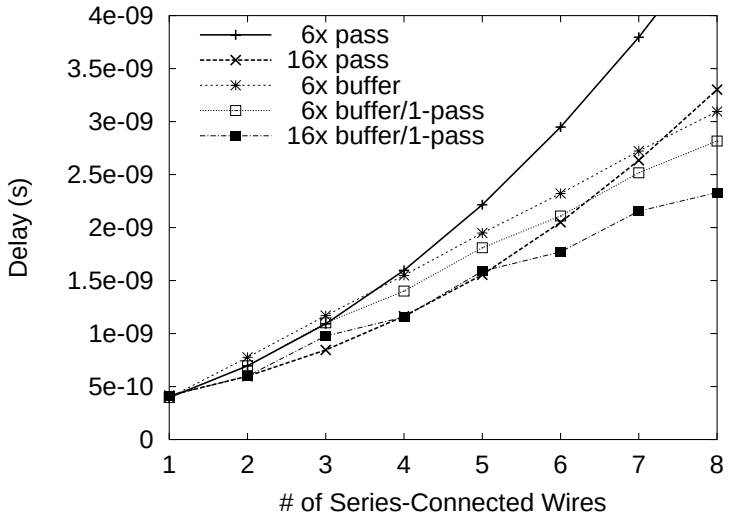


Figure 6.1: End-to-end connection delay using different switch types.

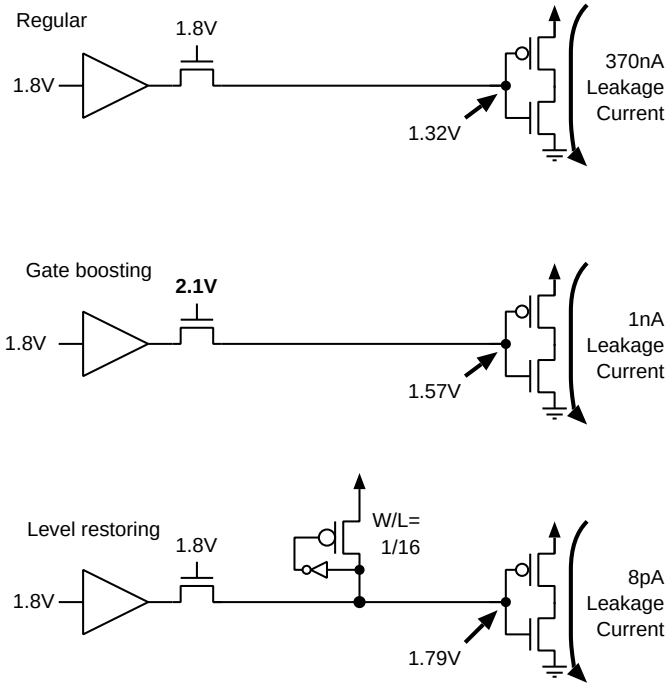


Figure 6.2: Level-restoring circuit to reduce leakage current.

further below.

Leakage Current Problem

One drawback of using NMOS pass transistors is that they cause *leakage current* in downstream buffers when passing a logic-high voltage. The steady-state output voltage for such devices is approximately $V_g - V_t$, where V_g is the gate voltage and V_t is the threshold voltage of the device. This produces a *weak-1* instead of a *strong-1*, causing both NMOS and PMOS transistors of downstream buffers to be partially on. The high source voltage magnifies this problem, since the effective value of V_t increases due to the body effect. With a very large number of downstream buffers, significant leakage current and static power dissipation results.

An example of the output voltages and the resulting static power in 0.18μm is shown in Figure 6.2. A pass transistor reduces a 1.8V input to 1.32V, resulting in 370nA of leakage current in each downstream buffer it connects to. A typical routing wire will have many (tens) of these downstream buffers. Even though most of these buffers will not be actively used by the netlist, they will all consume static power due to leakage.

Gate Boosting Solution

One solution to the leakage problem employs a boosted gate voltage on the pass transistor. A gate voltage of 2.1V reduces leakage to 1nA. Gate boosting has been used in previous work [19, 144, 40], but device reliability problems with this technique arise in 0.18μm and below due to thin gate oxides. These problems result in the physical deterioration of the device, eventually rendering it inoperable.

Level-Restoring Solution

As an alternative to gate boosting, the level-restoring circuit [145, 146] shown in Figure 6.2 can be used to pull a weak-1 signal into a strong-1. This circuit involves positive feedback of a sense inverter driving the gate of a weak PMOS pullup. When a weak-1 is present, the sense inverter begins to turn on the pullup by driving a low signal on its gate. In turn, this increases the voltage of the weak-1 until the pullup has restored the voltage to

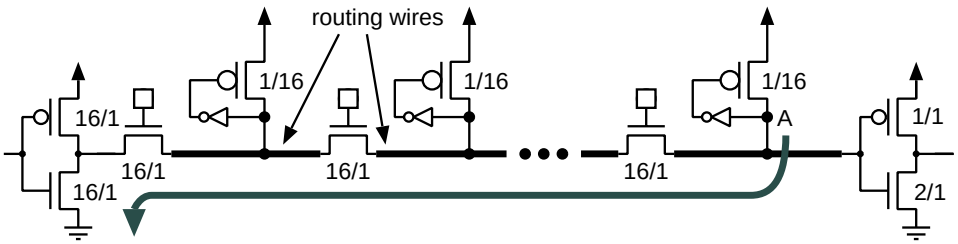


Figure 6.3: The level-restoring pulldown problem.

V_{dd}. One level-restoring circuit is needed on every wire that: a) is likely to be driven by a pass transistor, and b) drives a significant amount of regular CMOS logic. All HSPICE simulations in this chapter include one level-restoring circuit on every interconnect wire.

Level-restoring circuits have been used in at least one field-programmable device, CHESS [147]. Designed by HP Labs for reconfigurable computing, it is implemented in 0.35μm and uses only NMOS pass transistors in the interconnect.

Level-Restoring Circuit Size

Sizing the pullup used in level-restoring involves tradeoffs. A very weak pullup is desirable, but increasing the channel length consumes more area. On the other hand, a very strong pullup is small and can quickly restore full voltage to the wire, but this creates new problems. Since the pullup is always on while there is a high voltage on the wire, it hinders the ability of another driver to pull the signal back to V_{ss}. This increases the fall time of the wire. PLD interconnect exacerbates this problem, and may lead to a stuck high state.

The stuck high state can be described as the *level-restoring pulldown problem*. This problem is best illustrated through the example shown in Figure 6.3. In this example, the distant node *A* is presently high and must be pulled low through a long chain of pass transistors. Since the pass transistors and wires have significant resistance, a voltage divider is formed at node *A*. If the voltage there cannot be pulled below the switching threshold of the sense inverter, the wire will be stuck high.

The pulldown problem can be avoided by simply preventing connections that would exhibit it. For example, a PLD router might prohibit the formation of very long connections or those with significant fanout. Alternatively, the PLD architecture might be de-

signed to make such connections impossible to form or easily detected (and avoided) by the router. For example, carefully placing buffers after every eight pass transistors may provide enough isolation that the worst-case can always be pulled down. For the benchmark circuits used here, the required PLD sizes are small enough that using weak pullups appears to be sufficient. However, a more robust solution is required in the future.

A number of HSPICE simulations with long chains of N pass transistors and level-restoring circuits helped to determine the best pullup channel length, L_p. These simulations use pass transistors of size $W_n/W_{minT} = 10$ connecting $N+1$ wires.[1] Very long chains ($N > 32$) cannot be pulled down when the pullup length is sized $L_p < 10 \cdot L_{min}$. Chains with $N \geq 16$ have their fall times become critical when $L_p \leq N \cdot L_{min}$. This can be quite severe, *e.g.*, at $N = 32$ with $L_p = 10 \cdot L_{min}$, the fall time is 2.5 times the rise time. Pullups with $L_p = 16 \cdot L_{min}$ are a good compromise between area and pulldown complications and this is used for all level-restoring circuits herein. At this length, it takes roughly 50ns to fully restore the routing wire to V_{dd}, and more than 40 series-connected wires can be pulled down. The area required by the pullup transistor is only about three times bigger than a minimum-size transistor, and only one pullup is required *per wire*.

If no pullup is used, a modern-sized architecture would dissipate 1.85A of DC current or 3.33W of static power.[2]

Alternative Leakage Solutions

There are alternative solutions to the leakage current problem which do not involve gate boosting or level restoring. Some of these alternatives are:

1. using pass transistors with reduced voltage thresholds (*e.g.*, native devices),
2. using PMOS devices with increased voltage thresholds on the inputs of downstream buffers and logic gates, and/or
3. using full transmission gates rather than pass transistors.

[1] Throughout the circuit design sections, L_p is the length of a PMOS device, $L_{min} = 0.18\mu$m, W_n is the width of an NMOS device, and W_{minT} is the minimum *contactable* diffusion width.

[2] Calculation assumes an architecture with length 4 wires, an array of 100×100 CLBs, and a channel width of 100, 10 downstream buffers per wire, and 370nA per buffer.

These options are briefly described below.

In the first two alternatives, altering the threshold voltage is possible during device manufacturing and may require additional processing steps. In the first case, lowering V_t creates pass transistors that are difficult to turn off completely. To use these as routing switches, they must adequately isolate different wires or electrical performance will suffer. Although lower V_t devices are available from the TSMC 0.18μm process using so-called *native devices*, the lack of isolation makes this solution unattractive.

In the second alternative, using pullup devices with an increased threshold voltage makes it difficult to turn on the pullup device. This would slow the response time to a falling input. There is also reduced gate over-drive, further increasing delay [4, 148]. Since increased V_t devices are not available from TSMC, this option has not been explored.

The third alternative involves using a full transmission gate, but this adds to device area and capacitance. A preliminary investigation was performed to compare the performance of transmission gates to NMOS pass transistors. Transmission gates were found to increase area·delay by 18–33%, depending on the switch size. Consequently, transmission gates are not considered any further.

It is worthwhile to note that it is not uncommon for a foundry to specialize some manufacturing steps for PLDs. This can make other solutions to the leakage problem viable. For example, the latest Altera PLD, implemented in a 0.13μm technology, reportedly uses a combination of the normal thin gate oxide, a higher core V_{dd} for increased gate over-drive, and increased V_t devices [149].

6.3.3 Transistor-Level Buffer Design

Large buffers are commonly formed by tapering, or connecting multiple inverter stages of increasing size, as shown in Figure 6.4. The input drives the first inverter, or sense stage, and the drive stage produces the final output. Intermediate stages, if any, scale up in size by a fixed factor to reach the drive stage.

For the routing switches used in this dissertation, a tristate output is formed by adding an NMOS pass transistor to the drive stage. Other tristate buffer designs with improved drive ability, such as those in [19, 146], have not been considered because they require

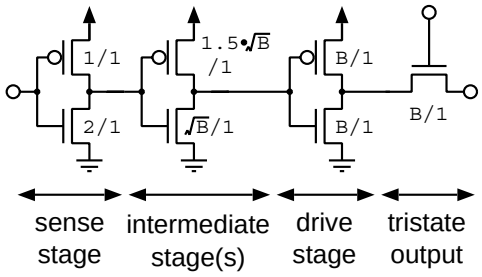


Figure 6.4: Multistage buffer with (optional) tristate output.

more area.

For example, suppose a three-stage buffer of size B is constructed. The intermediate stage is size $\sqrt{B}$ and the drive stage and tristate output transistor are both of size B. A size B stage uses an NMOS device of width $W_n = B \cdot W_{minT}$, and a PMOS device of width $W_p = B \cdot W_{minT} \cdot (W_p/W_n)$. The W_p/W_n ratio controls the relative size of the PMOS device relative to its NMOS counterpart.

While simulating tapered buffer circuits, a ratio of $W_p/W_n = 1.5$ produced the lowest delay through a loaded (fanout of 3) inverter chain. Hence, this W_p/W_n ratio is used in the intermediate stages of all buffers. For the sense and drive stages, the W_p/W_n ratios are carefully selected in tandem as part of the overall design approach to minimize delay. The determination of these ratios is discussed later.

The remainder of this buffer design section is organised as follows. First, the circuit model used to represent an interconnect wire is presented. This is followed by a general description of a broad search which guided the buffer design away from local minima. Last, the method used to determine buffer sense and drive stage ratios is presented.

Wire Model

The circuit shown in Figure 6.5 is used to model an interconnect wire. Its intrinsic resistance and capacitance is divided up as shown into L_{wire} equal portions of physical length L_{tile} each. Each portion contains two resistors and a lumped capacitance, the values of which are determined from L_{tile} and the process characteristics. Also included in the circuit is an active driver and a number of inactive pass transistors, representing the loads

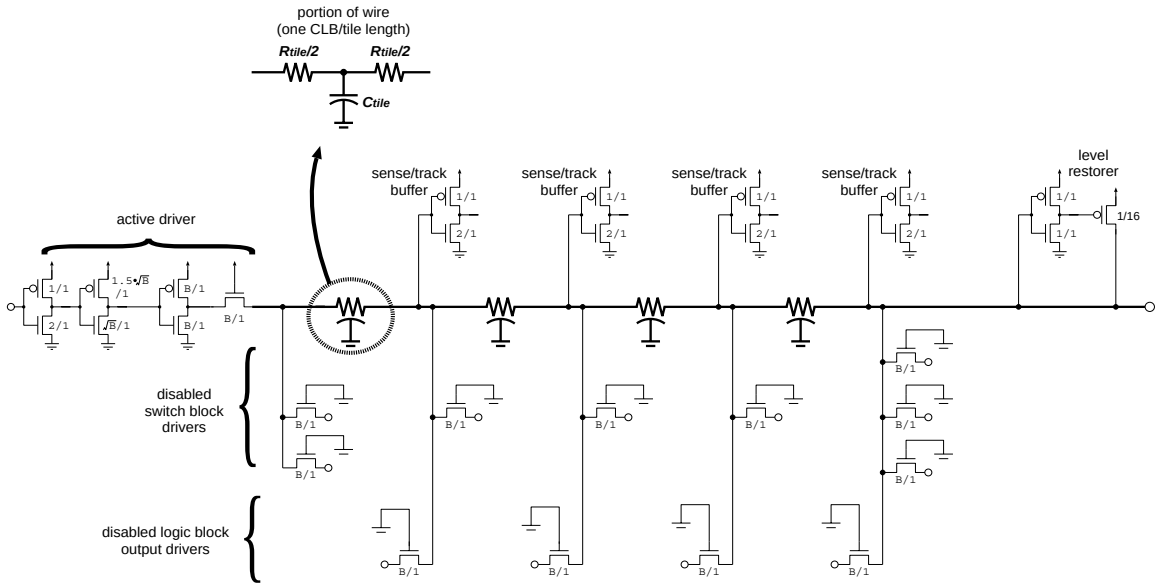


Figure 6.5: HSPICE circuit of a length-4 wire segment and all drivers.

from unused drivers (both routing switches and CLB outputs). A number of sense/track buffers are placed along the length of the wire as well. These are the first stage of the track buffers (which drive the CLB inputs) and other buffered routing switches.

In Figure 6.5, the transistor labels indicate the W/L ratios. A transistor shown as $B/1$ is B times the minimum contactable width and 1 times minimum length. Appropriate values of B, the number of stages in the active driver, and the sizes of the sense/track buffers will be varied during experimentation. The number of wires connected in series, using either routing buffers or pass transistors, will also be varied.

General Search

The first step in buffer design involves examining the impact of a number of design parameters on delay:

- drive stage W_p/W_n between 1 and 2,
- size B from 2 to 64, and
- the number of stages from 2 to 4

- driving 1 to 16 series-connected pass transistors and wires.

The results of this search, which are not shown here, indicate that the best results for total delay are obtained with buffers containing 3 inverter stages. The best delay-per-wire results use switches of size 14–16 and use buffer/2-pass switching with $W_p/W_n = 1.0$. The characteristics for the best area·delay per wire results are similar but with a smaller size (7–9). These results are considered preliminary and guided the results of the subsequent local optimisations. This initial result provides some assurance that the local optimisations steer toward a global minimum rather than local minima.

Adjusting Buffer Sense and Drive Stages

Careful buffer design for minimum delay requires choosing the right number of buffer stages as well sizing of the W_p/W_n ratios used in the sense and drive stages. These two stages are coupled in that transitions from the drive stage must be sensed by the sense stage of other downstream buffers. Between them, interconnect wires form the principle load. The sizing of these ratios and choosing the number of buffer stages for minimum delay is performed here using an iterative optimisation process.

The optimisation process begins by fixing the drive stage width ratio at $W_p/W_n = 1.5$ and varying the sense stage W_p/W_n from 0.1 to 2. When $W_p/W_n < 1$, the PMOS transistor is fixed at minimum width and the NMOS transistor is widened. Delay-per-wire curves similar to Figure 6.6a indicate that minimum delay is reached when W_p/W_n roughly equals 0.5, 0.3, and 0.7 for 2-, 3-, and 4-stage buffers, respectively. In all cases, a minimum size PMOS transistor is used. This differs from typical CMOS guidelines which suggest $W_p/W_n = 2$, but it reflects the need to sense a lower voltage swing caused by the pass transistor V_t loss.

Repeating this process for buffer/1-pass switching gives the lower set of three delay-per-wire curves in Figure 6.6a. The best W_p/W_n ratio here is similar to before for 3-stage buffers, but larger ratios must be used for the 2- and 4-stage buffers.

Next, the sense buffer sizes are fixed at their best values and the drive stage W_p/W_n is varied from 0.3 to 2. Results such as those in Figure 6.6b indicate that 2- and 4-stage buffers require a stronger PMOS driver with $W_p/W_n \approx 1.0$, but $W_p/W_n \approx 0.9$ is sufficient

for 3-stage buffers. This is again repeated for buffer/1-pass switching, where the best ratios lower to $W_p/W_n \approx 0.7$ to 0.8.

After determining the best drive stage transistor sizes, they are fixed to these values and another pass is made to re-adjust the sense stage and then the drive stage. After repeating this for a third iteration, the best ratios do not change significantly, so these values are accepted as stable. The graphs in Figures 6.6 and 6.7 are the final results of this effort for size 6 and size 16 switches, respectively.

For all switch configurations, it is evident from the figures that 3-stage buffers provide a lower delay-per-wire. There are two explanations why the 3-stage buffer is faster. First, the 3-stage buffer has slightly lower intrinsic delay than the 2- or 4-stage versions. Inspection of the intrinsic buffer delays (not shown) indicate that a 3-stage buffer is only 4% faster than a 2-stage buffer.[3] However, the overall delay improvement using 3 stages is nearly 10%. This suggests that the inverting property must also improve delay (particularly with the size 6 switch).

Further examination with 2- and 4-stage buffers (of size 6) reveals that their low-to-high transitions are usually slower. This is also observable in Figure 6.6b by the need for a slightly larger PMOS drive transistor to minimize delay. In contrast, a 3-stage buffer (of size 6) does not benefit from a strong pullup because it converts a slow-rising input into a fast-falling output (or vice versa). A stronger pullup would impede the falling output transition more than it aids a rising output transition.[4] Figure 6.6b shows that a weaker pullup with $W_p/W_n \approx 0.9$ is better at reducing delay for 3-stage buffers than the $W_p/W_n \approx 1.0$ for 2- and 4-stage buffers. Hence, the falling-output delay of these non-inverting buffers becomes critical at a larger W_p/W_n than the inverting buffer.

The size 16 switch results in Figure 6.7 do not exhibit the same trends because rise and fall times are more equal near $W_p/W_n \approx 1.0$ for the three stage sizes. This is understandable because self-loading of the drive stage becomes more important as switch size increases.

In summary, adjusting the number of stages and the sense- and drive-stage transistor widths is an important step to reduce interconnect delay. The best design choices for size

[3]This is simulated using buffers with unloaded outputs, size 6 switches, with driver $W_p/W_n = 1.0$, and properly conditioned inputs.

[4]The effectiveness of a larger PMOS device is reduced by the NMOS transistor used to tristate the output.

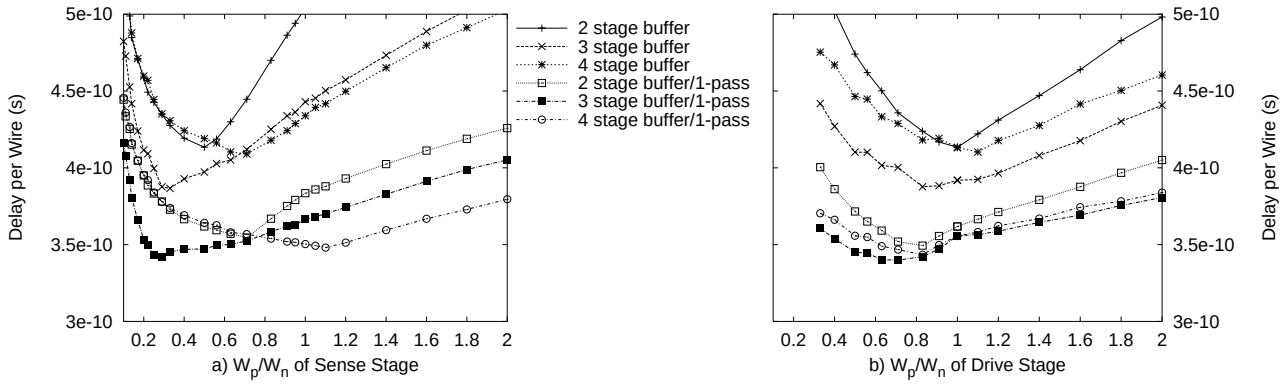


Figure 6.6: Adjusting the sense and drive stages of a size 6 switch.

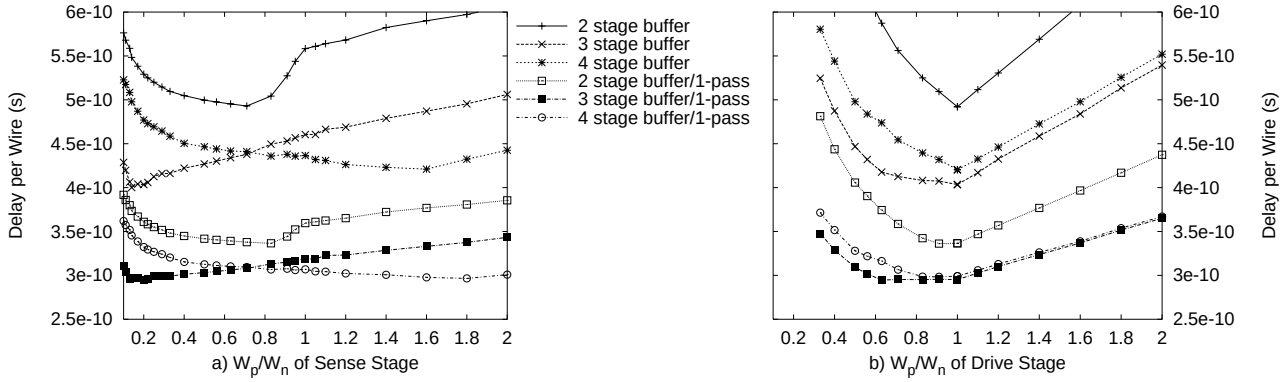


Figure 6.7: Adjusting the sense and drive stages of a size 16 switch.

6 buffered switches are 3-stage buffers, a sense stage $W_p/W_n \approx 0.3$, and a drive stage $W_p/W_n \approx 1.0$. The best design choices for size 16 buffered switches are 3-stage buffers, a sense stage $W_p/W_n \approx 0.2$, and a drive stage $W_p/W_n \approx 1.0$. For buffer/1-pass switching, the same sense stage ratio can be used with a drive stage $W_p/W_n \approx 0.7$ for both switch sizes. These drive stage ratios are also effective for keeping area requirements low; traditional transistor sizing would suggest the driver PMOS transistor to be twice as large.

To simplify the remaining design work across a number of switch sizes and different buffer/N-pass switching combinations, only 3-stage buffers will be used. All sense stage ratios will use $W_p/W_n = 0.5$. This compromise does not greatly impede performance, and it keeps the area of the sense stage small. All drive stage ratios will use $W_p/W_n = 1.0$. This final buffer design is shown earlier in Figure 6.4. This design will be used for all subsequent results in this chapter.

6.3.4 Best Switch Sizes

Selecting the proper switch size is an important step in designing interconnect for low delay and area. To investigate this, the end-to-end delay for various switch sizes is simulated using a buffer driving one to eight wires connected in series by pass transistors. This represents a wide range of conditions, from only-buffered wires to primarily pass-transistor connected wires. The delay and area·delay results *per wire* are presented in Figures 6.8 and 6.9.

Best Delay Results

In terms of delay, superior results are obtained with buffer/N-pass switches. Using buffers only, the best delays of approximately 360ps per wire are obtained with switch sizes of 8–10. For pass-transistor dominated interconnect, buffer/6-pass switches reach a similar delay of 370ps per wire with size 16 switches. In contrast, the lowest delays are obtained with buffer/1-pass and buffer/2-pass switches: 300ps and 290ps per wire with switch sizes 9–11 and sizes 12–16+, respectively. These are about 25% faster than buffer or pass-transistor dominated interconnect.

Best Area·Delay Results

The best area·delay results are obtained again with buffer/N-pass switches. Using buffers only, the best sizes are 5–7. The lowest buffer/1-pass switch designs use sizes 6–8. However, the best results are obtained with buffer/2-pass and buffer/3-pass switches of size 7–9. Area·delay with these switches is 43% lower than with buffers alone, and 10% lower than buffer/1-pass switches.

Switch Sizes Chosen

To get the best area·delay results possible, a PLD architect would likely select buffers of size 6 and pass transistors of size 8 from Figure 6.9. This differs from previous work [19], where the pass transistor is chosen to be twice the size of the buffer. One possible reason for this difference is the shift from a 0.35μm process to a 0.18μm process. This illustrates

the need to re-check basic assumptions about the circuit design for each technology node used.

Recall that the goal of this chapter is to assess the area and delay benefits of buffer/pass switching. To do this conservatively, the switch sizes should be chosen which make it as difficult as possible for buffer/pass switching to reduce area or delay. To create fast pass transistor interconnect, Figure 6.8 suggests that size 16 switches are fastest when two to seven pass transistors are used in series. To create the best area·delay buffered-only interconnect, Figure 6.9 suggests using size 6 switches. The pass transistor portion of the interconnect will remain fixed, so it will be difficult to improve the delay or area·delay of the PLD by modifying only the buffered portion. Using only these two switch sizes also greatly simplifies experimental conditions.

To see why these experimental conditions are very conservative, consider that half of all interconnect tracks will be based on size 6 buffers. In the routing experiments, some of these buffers will be replaced with size 6 pass transistors, hence forming buffer/pass switches which are smaller than the ideal sizes. This choice will make it difficult to show improvement with buffer/pass switches.

The remaining half of the interconnect tracks will be based on size 16 pass transistors only. These will be driven from the CLBs using size 16 buffers. The choice of using size 16 pass transistors is probably larger than one would normally choose from Figure 6.9. However, this choice will result in a very fast and slightly larger interconnect. This also makes area and delay improvements obtained by altering the buffered tracks more conservative. To see this, consider that the fixed area from the pass-transistor portion will be larger than required. Hence, any savings from reducing the area of the buffered portion will be understated. Similarly, a fast pass transistor portion makes it difficult for buffer/pass connections to be faster. This will reduce the need for the delay improvement from buffer/pass switches, again understating any gains achieved.

6.3.5 Verifying Assumptions

The assumption that a fixed switch size can be used in the interconnect, regardless of the wire loading conditions, greatly reduces the architectural search space. Furthermore, the

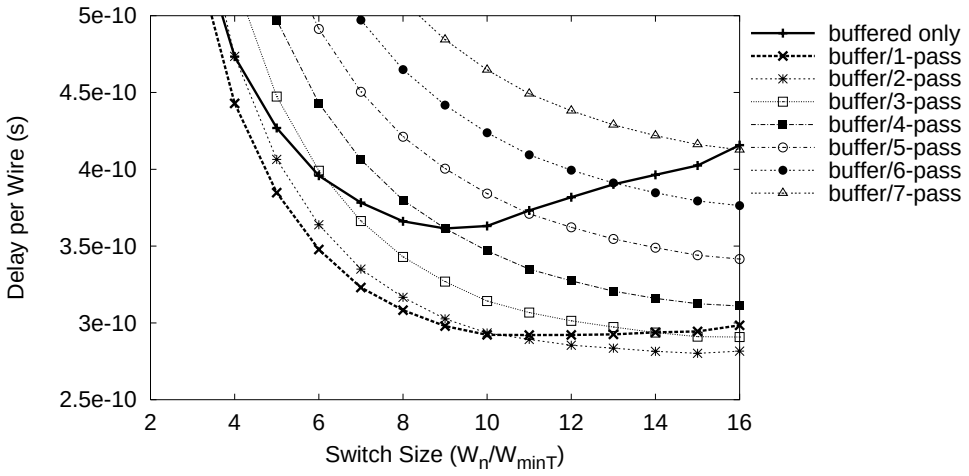


Figure 6.8: Delay per wire for various switch sizes.

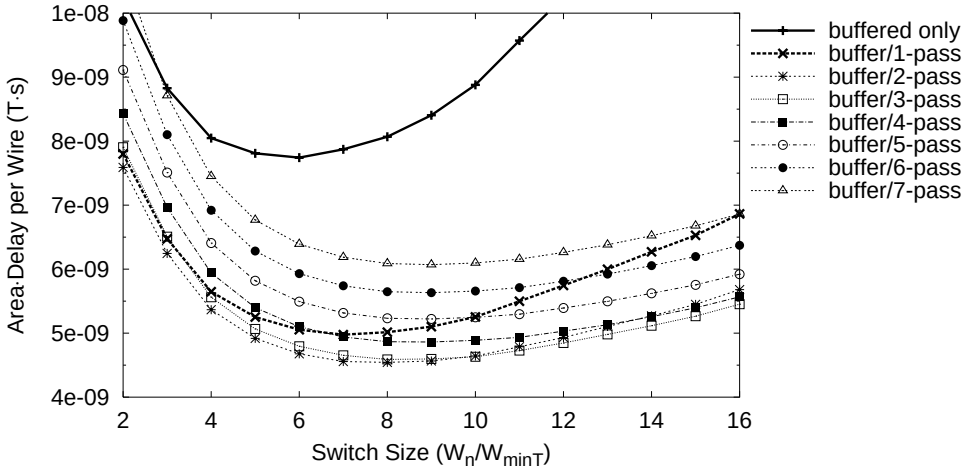


Figure 6.9: Area-delay per wire for various switch sizes.

mixing of buffers and pass transistors creates an environment where the constant-delay buffer timing model, which is based on fixed input slew rates, is not as accurate. VPRx assumes that this constant-delay model is acceptable. This section verifies that these two simplifying assumptions do hold true.

Assumption 1: Constant Switch Sizes With Varying Tile Length

An estimated tile length of $L_{tile} = 116\mu$m has been used for most of the HSPICE design work. However, the actual tile size fluctuates as k, N, and the other architectural parameters

are varied. Hence, it is necessary to verify the sensitivity of the best switch size on L_{tile}.

This sensitivity is computed using HSPICE delay simulations of a buffered switch followed by a wire for a variety of tile lengths and switch sizes. The delay and area·delay product for each tile length is shown in Figure 6.10. In these graphs, the switch size B is varied along the x-axis to generate one curve per L_{tile}. The lowest points for each curve (each tile length) are connected together by the bold curve labelled **best**. Two additional bold curves trace the smallest switch sizes that would result in being **within 5%** and **within 10%** of the best delay or area·delay.

In Figure 6.11, the same best and within 5% data from Figure 6.10 are replotted as a function of tile length. Additional curves are shown which correspond to equations below. These equations are determined by best-fit calculations to the experimental datapoints.

The best possible delays in Figures 6.10 and 6.11 are achieved by scaling the switch size B with tile length. Previous work [40, 111] forced a linear fit of B through the origin and a single datapoint (size 5 buffer at 116μm), resulting in the following equation:

$$B_{\text{best buffered area·delay, linear}} = 0.0431 \cdot L_{tile} \tag{6.1}$$

where L_{tile} is given in μm.[5] Although Equation 6.1 was fit to a best area·delay datapoint, the scaling was done for purposes of improving delay. It is a very poor fit to the area·delay experimental results determined here, which are roughly flat.

Better fits to the experimental results can be made. The following two equations correspond to a linear fit and a sub-linear fit, respectively:

$$B_{\text{best buffer/1-pass delay, linear}} = 0.0154 \cdot L_{tile} + 7.47 \tag{6.2}$$

$$B_{\text{best buffer/1-pass delay, sub-linear}} = 0.644 \cdot (L_{tile})^{0.5} + 1.90 \tag{6.3}$$

where L_{tile} is given in μm. In this fit, the exponent value of 0.5 was determined manually and the other parameters were calculated by regression analysis. For both fits, the linear regression correlation coefficients, R^2, are slightly greater than 0.95. Hence, both equations are good fits to the data.

Although scaling gives the best delay, it is apparent from the flatness of each delay curve in Figure 6.10 that delay is RC limited beyond a certain tile length and thus insen-

[5] Note that all equations in this section assume that wires span 4 tiles.

sitive to the switch size chosen. This is also apparent from the sub-linear growth of the **within 5%** and **within 10%** curves in Figure 6.11. These curves suggest that a switch size of 15 and 12, respectively, is probably large enough for best delay with tile lengths beyond 1000μm.

A similar analysis of the area·delay curves shows that a fixed switch size of 5 or 6 is effective for nearly all tile lengths.

To further illustrate the lack of sensitivity to tile size, the delay and area·delay of size 6 and 16 switches, respectively, is given in Figure 6.12. The vertical axis shows the increase due to these fixed sizes relative to the minimum value obtained with the best switch size. The increase in delay is less than 10% for tile lengths of 174μm or longer. To keep the increase less than 10% with smaller tile lengths, smaller switch sizes should be used. Similarly, the increase in area·delay is less than 8% throughout the range. Hence, it is effective to use a fixed switch size with buffered interconnect when optimising for area·delay.

The results just presented were produced by a buffered switch driving a wire. The same work was repeated for a buffer/1-pass connection, *i.e.*, a buffered switch followed by a wire, a pass transistor switch, and second wire. The results are shown in Figures 6.13 to 6.15.

The buffer size equations fit to give best delay for buffer/1-pass connections are:

$$B_{\text{best buffered delay, linear}} = 0.0145 \cdot L_{tile} + 11.35 \quad (6.4)$$

$$B_{\text{best buffered delay, sub-linear}} = 6.74 \cdot (L_{tile})^{0.23} - 7.81 \quad (6.5)$$

where L_{tile} is given in μm. The exponent in Equation 6.5 and the data both clearly indicate the best switch size for delay grows sub-linearly with tile length. The sub-linear equation more closely fits the data than the linear equation, with correlation coefficients of $R^2 = 0.98$ and $R^2 = 0.91$, respectively. The **within 5%** and **within 10%** curves flatten even more quickly than the buffered-wire results: the constant switch size of 14 and 11, respectively, is sufficient for all tile lengths beyond 230μm. Although smaller tiles may benefit from a small amount of switch scaling, it is unnecessary for larger tiles.

A similar analysis of the area·delay curves in Figure 6.14 shows that a fixed switch size of 4 to 6 is effective for all tile lengths. However, in this case the best size actually drops for

the longest tile lengths. This is because the longer tile lengths near a switch size of 5 have less delay sensitivity to switch size than the medium tile lengths. Hence, the area penalty of the larger switch is greater than the delay improvement, making the smaller switch size more attractive.

The sensitivity of delay and area·delay to a fixed switch size is shown in Figure 6.15. These results show that the increase is less than 5% across the range of tile lengths. This data very strongly supports the use of a fixed switch size for buffer/1-pass connections.

Data in previous work [47] also suggests that the best switch size is insensitive to *logical* wire lengths. That data shows the best switch sizes are consistent for lengths of 4, 8 and 16 tiles. This can be explained as follows. Although a longer logical length implies more switch loading, wire capacitance dominates. Hence, the effect of tile length and logical length should be similar: they both impact physical wire length and increase its RC.

The ability to use a single switch size for a wide range of logical wire and tile lengths greatly simplifies PLD research. For example, one may construct practical area and delay models based on layout experience of only a single size. It also suggests that previous research which scales switches, such as [19, 40, 111], may have used overly large routing switches. Scaling switches unfairly over-penalizes larger cluster sizes and leads to the (possibly incorrect) conclusion that smaller cluster sizes are better.

With the understanding that a fixed switch size is sufficient for a broad range of architectures, all routing experiments here are based on the two switch sizes determined earlier: size 6 (for buffered switches) and size 16 (for pass-transistor switches).

Assumption 2: Constant-delay Buffer Timing Model

In VPRx, delay through buffers is simplified to a constant value, the intrinsic buffer delay. This is adequate in strictly-buffered interconnect because the input slew rate can be easily determined in advance and included in this delay. However, the input slew rate is not known in advance because it depends on the topology of the routed net and the number of pass transistors used before rebuffering. Hence, an accurate timing-driven router must determine the slew rate based on the current net topology prior to forming a connection to

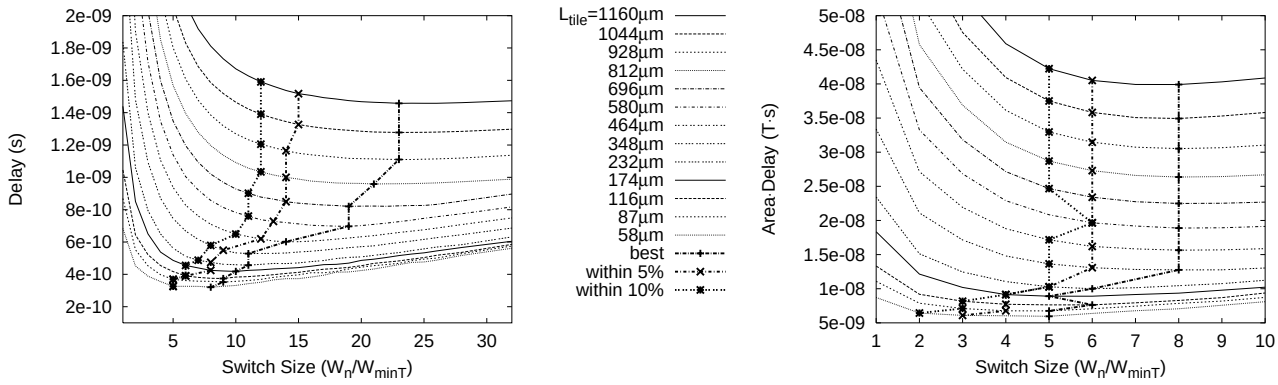


Figure 6.10: Effect of tile length on performance of a **buffer-wire** connection.

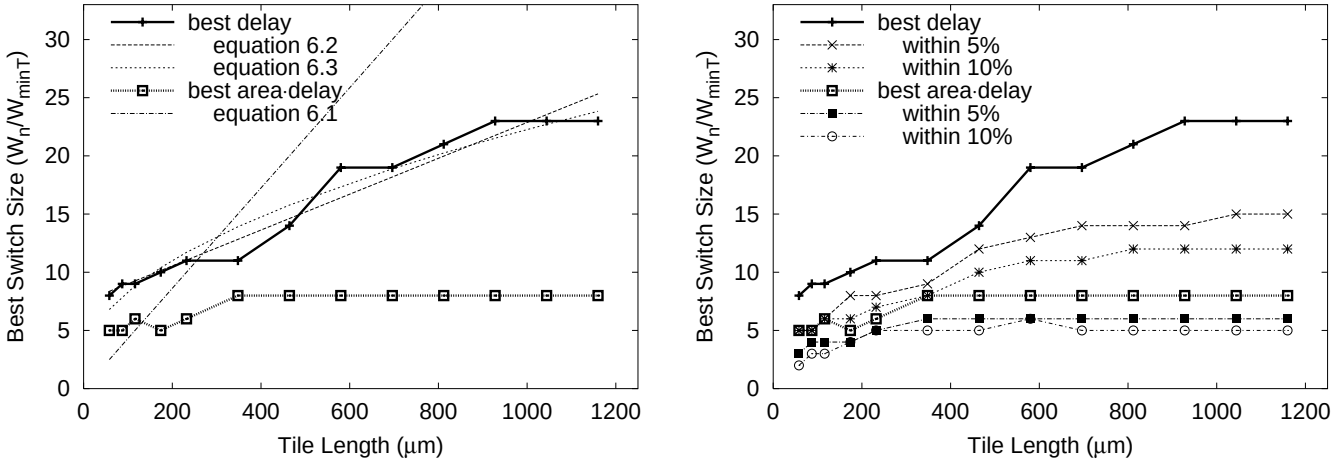


Figure 6.11: Best switch sizes as a function of tile length (replot of Figure 6.10 data).

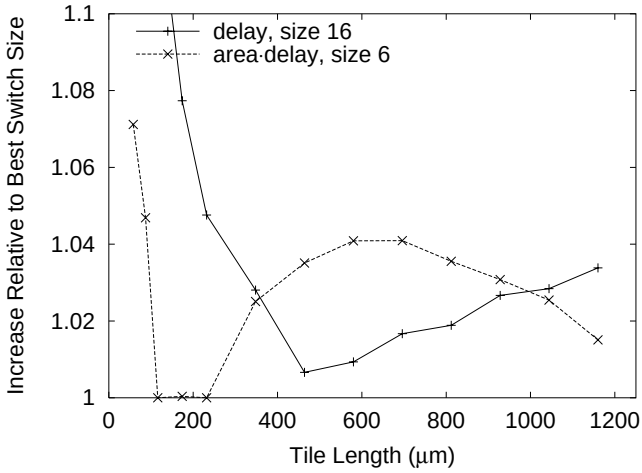


Figure 6.12: Increases from using a fixed switch size in a **buffer-wire** connection.

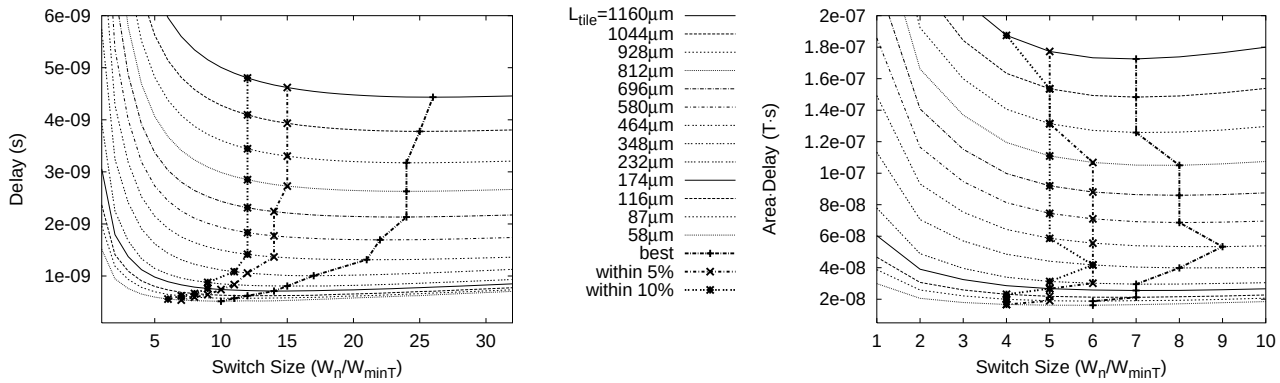


Figure 6.13: Effect of tile length on performance of a **buffer-wire-pass-wire** connection.

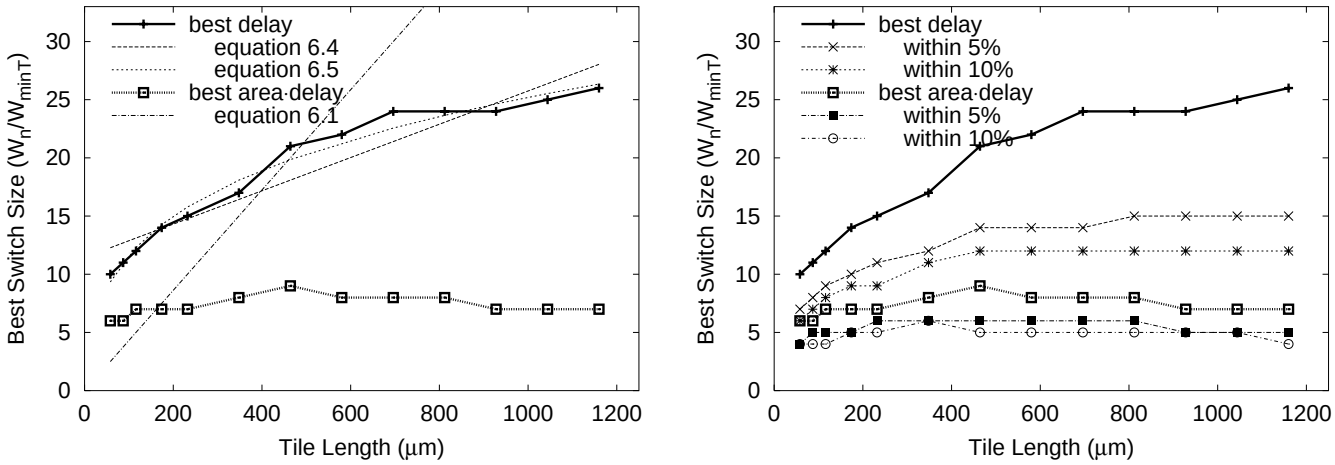


Figure 6.14: Best switch sizes as a function of tile length (replot of Figure 6.13 data).

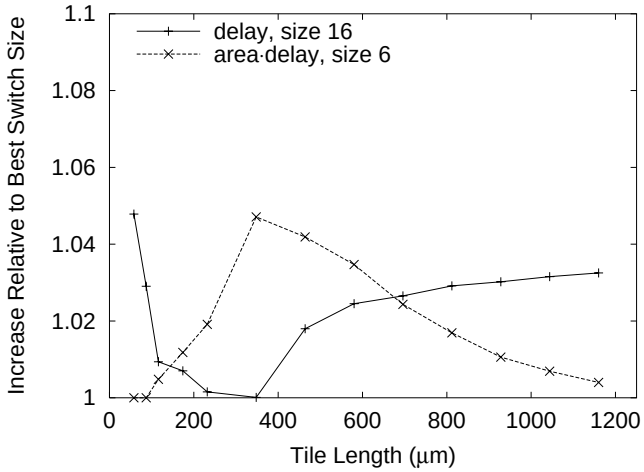


Figure 6.15: Increases using a fixed switch size in a **buffer-wire-pass-wire** connection.

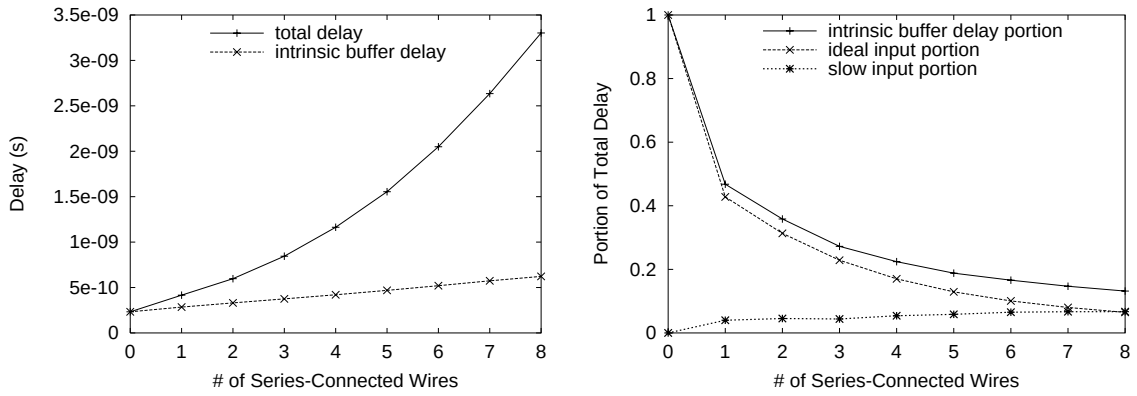


Figure 6.16: Impact of slow input slew rate on delay, size 16 switch.

the next sink.

The goal in this section is to determine whether this *slow input slew-rate effect* is significant enough to make buffer/pass switches too slow, and whether it is important enough to modify PLD router software and timing analyzers to explicitly compute it rather than assuming a typical or worst case.

To determine the impact of input slew rate on intrinsic buffer delays, delay of an unloaded buffer is measured with a conditioned input. The input is conditioned by a circuit containing between zero and eight pass-transistor connected wires which are driven by a similar buffer. For the situation with zero wires, an **ideal input** is formed by using two minimum-size inverters to condition a step input. The results are presented in the two graphs of Figure 6.16.

In the first graph, delay results are given for the **total delay** and the corresponding **intrinsic buffer delay**. Although the total delay through the pass transistors is quadratic, the intrinsic buffer delay increases linearly. For the size 16 buffers used in the figure, this delay roughly doubles across the range of inputs. Although not shown, the delay of size 6 buffers triples across the same range.

In the second graph, the delays are replotted as a portion of the total delay. The **intrinsic buffer delay portion** curve shows the fraction of **total delay** that is represented by the **intrinsic buffer delay**. When buffers are placed on every wire, this represents nearly 50% of the delay. As more pass transistors are used, this decreases to about 15% of the

total delay. The **ideal input portion** curve removes the slew-rate effect by plotting the **ideal input** delay (the delay obtained at zero wires, a constant) as a portion of total delay. The **slow input portion** curve shows the difference, *i.e.*, the portion of total delay directly caused by the slew-rate effect. Although it increases mildly as the input is degraded, it represents about 7% of the overall delay. For the similar results obtained with size 6 switches (not shown), the slew-rate effect reaches a maximum of 8% at three wires, and decreases to 6% at eight wires.

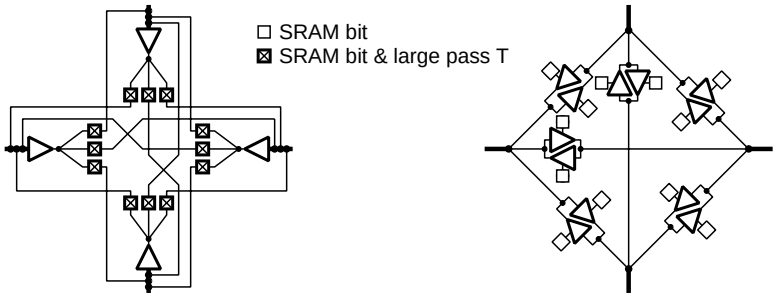
The observations here show that slow input slew rates can increase timing delays by about 8%. Such an increase is small enough that first-generation routing tools can ignore it and use worst-case delays instead. This is the approach taken in this work, where intrinsic buffer delays are taken as the constant worst-case delay of a buffer/1-pass combination. However, detailed timing analysis tools that strive for accurate delay estimates should take slew rate into account.

6.4 Three New Switch Types

Historically, the area and delay results computed by VPR and reported in previous literature have been based using two different switch types. Area reports are usually based on the smaller, buffer-sharing switch, *buf*, which is shown in Figure 6.17a. In contrast, delay reports are based on a larger switch with no buffer-sharing, *bufns* in Figure 6.17b because its delay is unaffected by fanout. Rather than presenting two best-case numbers obtained with two different circuit designs, what is needed is a single circuit design which combines the area advantage of *buf* with the delay advantage of *bufns*. Three new circuit designs which satisfy this criteria are described and evaluated below.

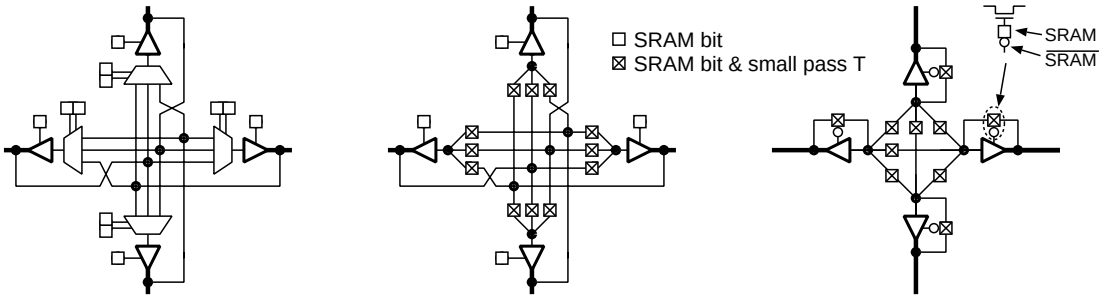
6.4.1 Fanin-Based Switches

The concept behind the new switch types is to pull a signal across the switch block rather than to push it across, resulting in a switch which is based upon fanin rather than fanout. Three new switch designs which illustrate this concept, *bufm*, *bufp*, and *bufp2*, are shown in Figure 6.18. By changing to a pull, the buffers avoid fanout entirely and large pass



a) '**buf**' switch, buffer-sharing b) '**bufns**' switch, no buffer-sharing

Figure 6.17: Two previous fanout-based switch types.



a) '**bufm**' mux switch b) '**bufp**' pass switch c) '**bufp2**' pass switch

Figure 6.18: Three new fanin-based switch types.

transistors on the buffer outputs can be replaced with smaller ones on the buffer inputs. Hence, fanin-based switches also offer potential area savings.

The differences between *bufm* and *bufp* are as follows. The *bufm* switch assumes a mux-tree structure on the input side, requiring only a few SRAM bits under high fan-in conditions. The *bufp* switch replaces the mux-tree with a flat layer of NMOS pass transistors and one SRAM bit per input. The *bufp2* switch is a novel, area-efficient variation of *bufp*. In *bufp2*, the input pass transistors are used in a bidirectional fashion to significantly reduce SRAM count and, consequently, area.

The implementations of *bufm*, *bufp* and *bufp2* have some similarities. All of the switches use similar driving structures, and they all use minimum-sized NMOS transistors on the input side. HSPICE simulations have shown that wider transistors on the inputs do not significantly improve delay. Also for performance reasons, level-restoring is not done on the internal points of these switches. If level-restoring is used, wider input transis-

Switch Type	Area		
	Profile	**size 6 (T)**	**size 16 (T)**
bufns	$12S + 12P + 12B$	276	480
buf	$12S + 12P + 4B$	168	276
pass	$6S + 6P$	57	87
bufm	$12S + 4P + 4B + 16p$	156	224
bufp	$16S + 4P + 4B + 12p$	176	244
bufp2	$10S + 4P + 4B + 10p$	138	206
Key: SRAM (S), buffer (B), large and small pass transistor (P, p). $S = 6T$, $p = 1T$. Size 6: $P = 3.5T$, $B = 13.5T$. Size 16: $P = 8.5T$, $B = 25.5T$.			

Table 6.1: Transistor area required to connect four wire endpoints.

tors are required to overpower the restoring pullup. However, the lack of level-restoring at this point contributes only a small amount to static leakage current because the majority of buffers are unused (*i.e.*, they are inactive switches and can be forced to have a low input). Of the remaining buffers which are actively used by the netlist, only about half of them will leak — since the buffers are inverting, the remaining half will contain a logic low.

Switch Area

To better illustrate the area overhead of each switch type, an area profile is shown in Table 6.1. The area of each switch type is divided into the number of SRAM bits, large buffers, and large and small pass transistors that are required to connect four wires at a switch block endpoint. This is converted into an area count for two switch sizes, B =6 and 16, using the transistor area model in Chapter 3. For $B > 3$, the *bufm* and *bufp2* switches are smaller than the previous *buf* switch. In comparison, the *bufp* switch becomes smaller than *buf* only when $B > 8$. Being larger than *buf* at size 6 makes *bufp* unsuitable for later routing experiments which attempt to reduce both area and delay.

Wire Delay Under Switch Fanout

For connections without fanout, the *bufm* and *bufp2* switches are slightly slower than the *buf* switch due to their input structures, but the *bufp* switch is slightly faster due to less load-

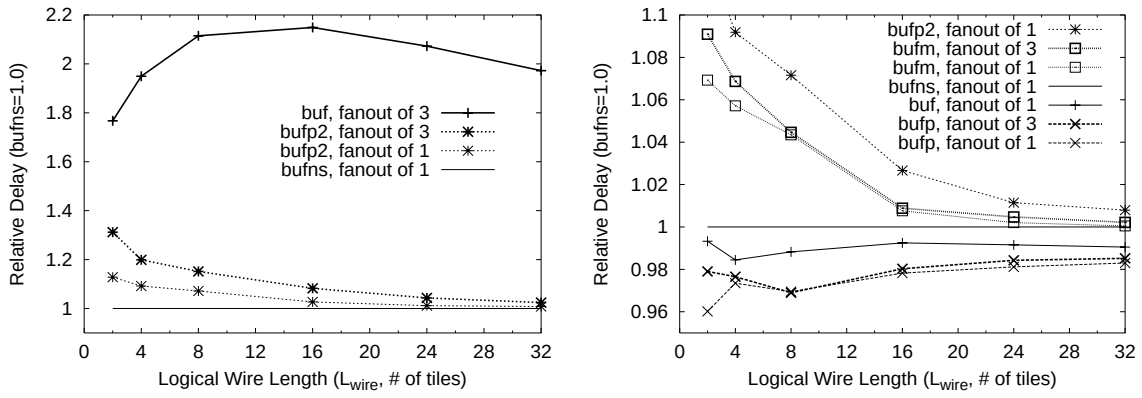


Figure 6.19: Delay per wire under fanout, normalized to *bufns*, size 6 switches.

ing. In the presence of fanout, however, the *buf* switch slows significantly while the others do not. Hence, the frequency that fanout occurs in a routed netlist will significantly affect the overall critical-path delay. Before assessing this impact, however, it is first prudent to examine the impact of switch fanout on the delay of a single wire.

First, consider the worst-case delay of the *bufns* switch. Under fanout conditions, different buffers share the same source wire and drive separate load wires, so the delay is independent of fanout. The *bufm* and *bufp* switches are also relatively unaffected by fanout because the load increase (from the worst-case single fanout cases) is limited. In contrast, the *bufp2* switch will be affected more by fanout because the entire fanout subtree is connected through the narrow pass transistor that is in parallel with the buffer.

The delay increases caused by fanout with a switch size $B = 6$ are displayed in Figure 6.19. Each delay result shown is the combined switch and wire delay after driving a fanout of one or three wire loads. These results have been normalized to the fanout-of-one *bufns* switch delays, which is the timing result normally reported by VPR 4.30. Although not shown, the fanout-of-three *bufns* switch delay is also 1.0. In the graph, the logical wire length, L_{wire}, is varied along the x-axis to simulate different loading conditions.

With a fanout of three and a wire length of four tiles, the *buf* switch is 95% slower than the *bufns* switch. This is a significant increase that is not calculated by VPR 4.30! Varying the wire length has little impact on the magnitude of this increase, it is consistently around 100%. In comparison, the new *bufm* and *bufp2* switches are only 7% and 10% slower,

respectively, under the same conditions. This amount **decreases** for longer wires, since a fanout of three produces a very small load increase compared to the wire capacitance itself. Interestingly, the performance of the *bufp* switch is **slightly faster** (2–3%) under fanout than the *bufns* switch due to the smaller transistor loads. Unfortunately, it is also 5% larger than a *bufns* switch at size 6.

With a fanout of one, some of the results are quite different. Due to less loading, the *buf* switch delay is much better than before, making it lower than the *bufns* switch by 2%. Of the new switches, the *bufm* and *bufp* switch delays are only slightly faster than before (2–3% for short wires). In contrast, the *bufp2* delay improves significantly, becoming nearly as fast (within 5%) as the *bufm* switch with low fanout.

From this data, it is clear that although it is among the fastest switches when there is a fanout of one, the *bufns* switch is inferior under fanout. The *bufp* switch is the fastest overall, under any fanout conditions, but the small delay improvement is offset by a larger area increase. Although the *bufm* and *bufp2* switches are always slower than *bufns* with a fanout of one, they are both significantly faster with a fanout of three. As well, Table 6.1 shows that these two switches are 36–57% smaller!

The best switch design to improve delay is *bufp*. To improve both area and delay, it is not immediately clear which is the overall best switch design. The ranking will strongly depend upon how much fanout is present in a typical netlist. Hence, the routing experiments performed at the end of this section will compare these different switches and assess their ultimate performance impact.

It is worth noting that preliminary investigations with the *bufm*, *bufp*, and *bufp2* circuits included a minimum-sized inverter to drive their input selection structures. This addition made the switches too slow to be useful.

6.4.2 Output Pin Merging

The input structure of the *bufm* and *bufp* switches is ideal to support a small increase to fanin without a significant increase in area or delay. To take advantage of this efficiency, logic block output pins are directly connected to fanin points of these routing switches. This feature, called *output pin merging*, has also been used in the mux-based switches of

Alexander routing architecture [134], part of a larger research project from Xilinx.

Output pin merging saves area as follows. Normally, a CLB output uses a shared-buffer switch, *buf*, where the connection to each track is made using an SRAM bit and a wide pass transistor. However, this wide pass transistor is not required on tracks that contain a fanin-based buffer switch.[6] Instead, the fanin portion can be extended to accommodate one more input. For example, a *bufm* switch requires two additional minimum-size pass transistors to extend the input mux from 3 to 4 inputs. If the total number of inputs does not pass 2^n, this will also save one SRAM bit for every output connection that is merged. Alternatively, a *bufp* switch requires one narrow pass transistor for each merged connection, although no SRAM bits are saved. The possibility of saving SRAM bits makes *bufm* more area-efficient than *bufp*.

The magnitude of the area savings produced by this technique is not very large; less than 100T area per tile is saved (about 1%). This savings is small because the routing architecture in this work contains as few buffers as possible. However, architectures that use many buffers or have very high-fanout at output pins (*e.g.*, at the I/O pads) will receive greater benefit from this technique.

Output pin merging does not significantly impact delay because a large buffer is required in both cases. As well, the additional delay of the fanin input structure is similar to the additional delay of a shared-buffer driving many unused output branches. In the experiments below, the delay difference is accounted for in the routing tool, and the overall impact on critical-path delay is included in the final results.

6.4.3 Experimental Results

The new *bufm* and *bufp2* switches offer significant potential to reduce area and delay, particularly in circuits which have many high fanout nets on the critical path. The improvement is reported here using the area and delay results calculated by VPRx. Although the *bufp* switch should also reduce delay, it is larger in area than the other switches so it will not be considered any further.

The area and intrinsic buffer delay of the new switch types depend upon the fanin

[6] It is still required on pass transistor switched tracks.

of each particular switch instance. For area, VPRx explicitly computes this fanin and the resulting area. For delay, the RC details of the *bufm* and *bufp2* input structures are abstracted into a constant worst-case intrinsic buffer delay. This abstraction involves the following two key simplifications.

- The intrinsic buffer delay used for the *bufm* switch has been selected to match the HSPICE simulation results obtained with a fanout of three. For a multiplexer of n inputs, this delay is represented by the equation $T(n) = \lceil C_1 \log_2 n \rceil^2 + C_2$, where C_1 and C_2 are curve fitting constants.

- The intrinsic buffer delay of the *bufp2* switch is reduced to a constant delay equivalent to the 4-input *bufm* delay. This nearly matches the *bufp2* fanout of one delay, representing its best possible performance.

Properly computing the *bufp2* switch delay under fanout is unnecessary because the experiments below will show that *bufm* performs better than this best-case *bufp2*.

To fairly estimate the influence of fanout on the delay of *buf* switches, the VPRx timing model adds an RC node at the drive stage output (before the tristate stage). This change sometimes results in delay increases of 200% or more to individual nets.[7] Despite this large delay increase to some individual nets, critical-path delay is not as strongly affected.

The geometric average critical-path delays of 20 MCNC circuits are presented in Table 6.2. The **ignoring fanout** columns contain the delays reported by VPR 4.30 with the original *unmodified* timing model. The **including fanout** columns give the new delay results produced with the new timing model in VPRx. These two different delay results are computed from the same routing solution, which is generated by the router with the new timing model. The change in delay performance, from the old timing model to the new one, is reported as a percentage in the **increase** columns. These results show that fanout at *buf* switches increases the average critical path by roughly 5% for all three LUT sizes. For individual circuits, this increase is as high as 16%. In contrast, the *bufm* switch limits

[7] The largest increases are seen on output pin nets, where a single high-fanout shared-buffer switch connects to many pass transistor tracks.

LUT size, k	*buf* Routing Switch			*bufp2* Routing Switch			*bufm* Routing Switch		
	ignoring fanout (ns)	including fanout (ns)	increase	ignoring fanout (ns)	including fanout (ns)	increase	ignoring fanout (ns)	including fanout (ns)	increase
4	16.9	17.8	5.2%	17.3	17.4	0.8%	16.5	16.6	0.6%
5	16.2	17.1	5.3%	16.5	16.7	0.8%	15.8	16.0	0.8%
6	15.4	16.2	5.3%	15.3	15.4	0.6%	14.9	15.0	0.8%

Table 6.2: Delay increases due to the improved modelling of buffer fanout within VPRx.

Buffer Type	Area ($\times 10^6 T$)			Delay (ns)			Area·Delay (T·s)		
	$k=4$	5	6	$k=4$	5	6	$k=4$	5	6
unnormalized	**3.25**	**3.34**	**3.28**	**17.8**	**17.1**	**16.2**	**0.058**	**0.057**	**0.053**
buf	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
bufp2	0.97	0.98	0.98	0.98	0.97	0.95	0.95	0.95	0.94
bufm	0.97	0.98	0.98	0.93	0.93	0.93	0.91	0.91	0.91

Table 6.3: Transistor area, delay, and area·delay results using different buffer types.

this increase to 1% on average, or a maximum of 5% for the worst-case individual circuit.[8] This increase would be zero, except that the CLB output pins still use the *buf* switch to connect with pass transistor tracks.

There is some variation among the **ignoring fanout** columns for the three different switch types. This variation comes from three sources. First, the switches have different intrinsic delays, so *bufm* and *bufp2* are expected to be slower. Second, the benchmark circuits are rerouted for each switch type. This change to the routing solution may significantly alter the delay result for each circuit; the necessity of rerouting is explained below. Third, the delay of the *bufm* switches is scaled according to the number of inputs, but this is not done for *bufp2* switches.

The need to reroute the benchmark circuits for each different switch type is based on a practical assumption. Since the router is timing-driven, it must use the delay parameters which match the switch type so it can make proper delay-oriented tradeoffs while routing. The use of the same routing solution would create a case where some solutions are generated with the wrong delay parameters. This is the same as inserting arbitrary 'fudge factors' in the heuristics.

[8] Although a similar result is shown for *bufp2*, recall that fanout at the *bufp2* switches is computed using the idealized fanout-of-one case and is not as accurate.

The normalized area and delay results in Table 6.3 show that the new switches save 2–3% in transistor area and up to 7% in delay. It is unexpected that both *bufp2* and *bufm* have similar transistor area costs because the area profile in Table 6.1 shows *bufp2* to be smaller. However, *bufm* saves more area from output pin merging, leading to similar area results. In terms of delay, *bufm* saves 7% compared to only 2–5% with *bufp2*. The *bufm* switch is better due to improved modelling: switches at wire midpoints are faster because they have a reduced maximum fanin. In comparison, the *bufp2* switch cannot improve as much because it is given a fixed delay for all switches. The different treatment for these switches is not entirely fair, but it is an approximation made to compensate for *bufp2* being slower under fanout. Overall, the *bufp2* and *bufm* switches improve the area·delay product by 5% and 9%, respectively.

Summary

The average delay increase caused by fanout is reduced from 5% with *buf* to 1% with *bufm*. In the worst-case circuit, this delay increase is reduced from 16% to only 5%. Compared to *buf*, the *bufm* switch improves both area and delay. On average, area is reduced by 2–3%, delay is reduced by 7%, and area·delay is reduced by 9%. Although equivalent in area, the *bufm* switch is superior to *bufp2* in terms of delay. For this reason, *bufm* will be used in the remainder of this chapter.

6.5 Buffer/Pass Architectures

In this section, numerous routing architectures that allow a signal to be switched by a combination of buffers and pass transistors are presented and evaluated. First, two switch schemes that strictly alternate between buffers and pass transistors are introduced. This concept is generalized to cycle among a collection of G different switch types. Then, some less-structured buffer/pass architectures are described. All architectures considered here are derived from the *baseline architecture* given in Section 6.2.3. by replacing some of the buffers with pass transistors, with the intention of reducing both area and delay.

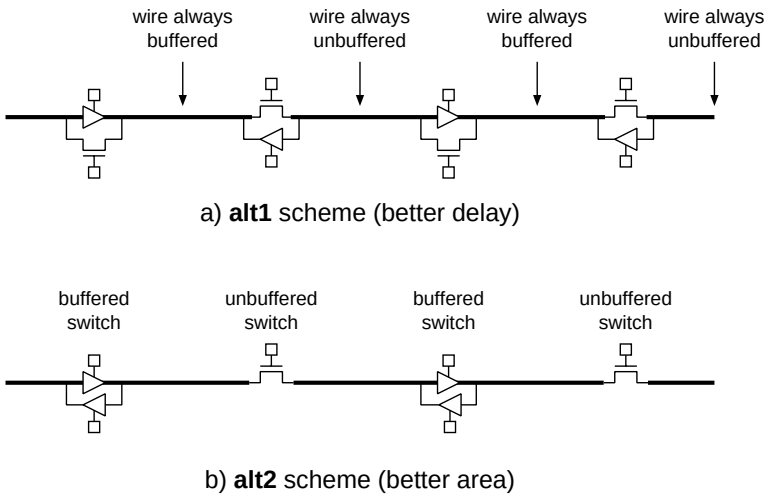


Figure 6.20: Key difference between two alternating schemes.

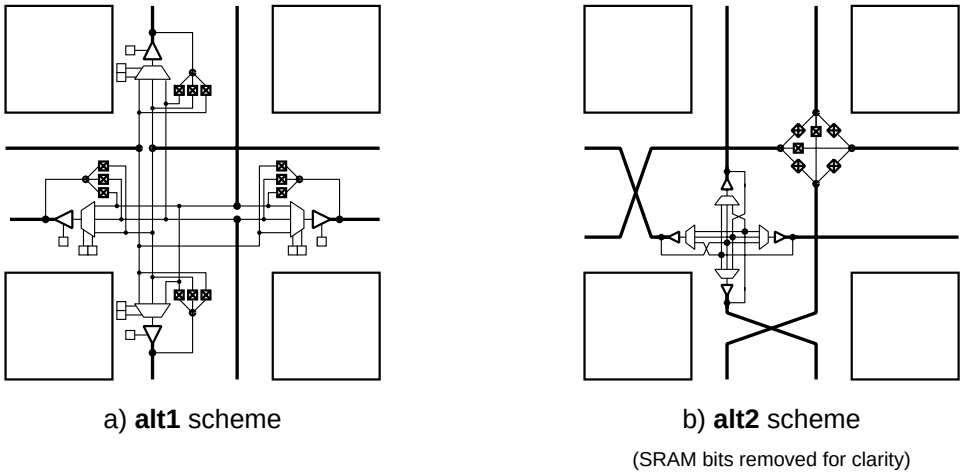


Figure 6.21: Tile and switch details of alternating schemes with length 1 wires.

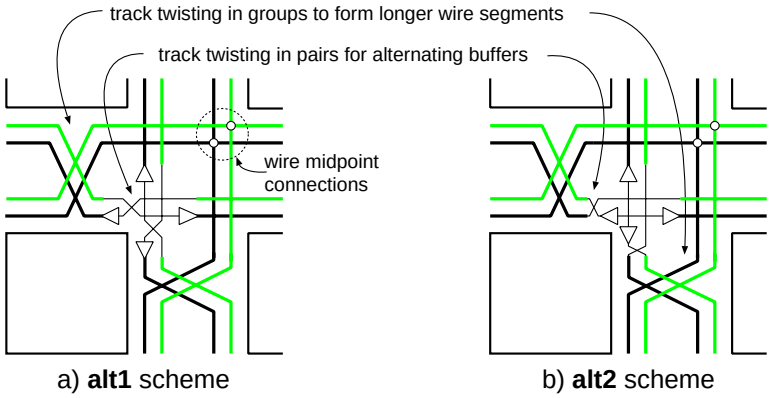


Figure 6.22: Extra track twisting is necessary to form longer wires (length 2 shown).

6.5.1 Alternating Buffer/Pass Schemes

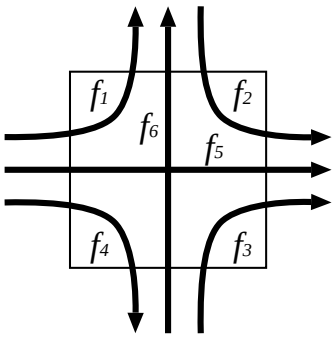
Two ways of replacing buffers with pass transistors are shown in Figure 6.20. Both of these schemes allow long connections to alternate between the two switch types. In the first scheme, **alt1**, two buffers which are normally in parallel but drive opposite directions will have one of the buffers replaced. In the second scheme, **alt2**, an entire group of buffers is replaced at every other switch block location.

With a small modification to switch block topology, both of these schemes can be implemented in a single layout tile. To produce **alt2**, start with two 4-wire cliques, such as those shown in Figure 6.18, and twist the straight connections in track pairs, as shown in Figure 6.21b.[9] The **alt1** scheme requires this track-pair twist plus a reorganisation of the turning connections. A detailed example of this is shown in Figure 6.21a, but a generalisation will be presented in more detail later. A more abstract representation of the differences between these two schemes is given in Figure 6.22. Notice that this figure shows how an additional type of twisting is required with both schemes to create longer wire segments.

The **alt1** scheme promises greater speed at the expense of slightly higher area. With this arrangement, some wires are always driven by only buffers, while others are always driven by only pass transistors. This way, a long connection will always strictly alternate between being on a buffered wire and an unbuffered wire. It is potentially faster because loading occurs close to the buffer source: *i)* all pass-transistor fanout occurs as a signal leaves the buffered wire, and *ii)* all of the diffusion capacitance from unused buffer switches is connected to the same buffered wire. Note that this scheme uses the pass transistor in only one direction, that which is opposite to the buffer. This is enforced by the router software.

In comparison, the **alt2** scheme uses less area because it takes advantage of both switching directions of a pass transistor. This reduces the number of pass transistors and SRAM bits required. It does not have the full delay advantages of **alt1** because of higher loads, some of which are located farther from the buffer.

[9] Interestingly, this twisting creates a universal switch pattern for **alt2**.



$$
\begin{aligned}
f_1(g) &= (g+1) \bmod G \\
f_2(g) &= (G-g+1) \bmod G \\
f_3(g) &= (g+1) \bmod G \\
f_4(g) &= (G-g-1) \bmod G \\
f_5(g) &= (g+1) \bmod G \\
f_6(g) &= (g+1) \bmod G
\end{aligned}
$$

Figure 6.23: Switch block to evenly cycle through a sequence of switches.

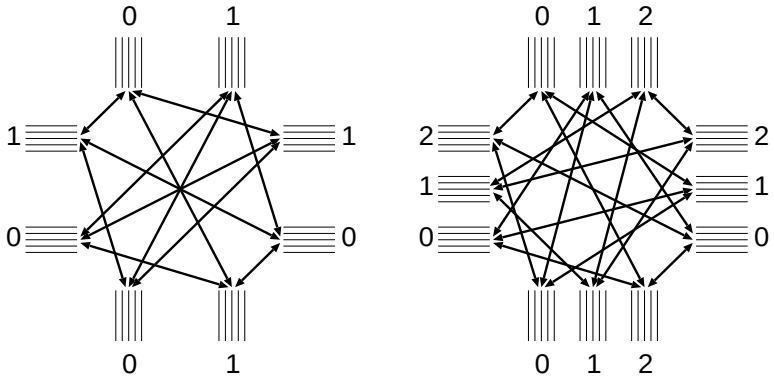


Figure 6.24: Switch block examples cycling among 2 or 3 switch types.

Wire Midpoint Details

The modifications described above apply to wire endpoints only. The connections at wire midpoints, such as those indicated in Figure 6.22, must be considered separately. For the **alt1** scheme, the switch type used at midpoint connections is easily determined by examining whether the wire is considered to be buffered or unbuffered. Hence, **alt1** strictly alternates between the two switch types no matter where turns are made. For the **alt2** scheme, such strict alternation is not possible. In this case, midpoint connections must alternate between using buffers and pass transistors along the length of a wire. In our implementation, the alternation is determined from the black and white checkered pattern described in Chapter 7.

Generalising: Any Switch Sequence

The switch block changes for **alt1** can be generalised so that long connections will cycle among any sequence of switches. For example, one possible sequence would be a buffer followed by two pass transistors. This strict cycling continues in the presence of turns, provided that the turns are made at wire endpoints only.[10]

To accomplish this cycling among a group of G switch types, the tracks in a channel are divided into G groups. These groups are then interconnected as follows. Tracks in group g are connected to tracks of group $f(g)$ across the S block. The $f(g)$ mapping functions differ depending on the turn direction. These turn directions and the corresponding mapping functions are shown in Figure 6.23. In this environment, switch type g is used for all connections *to* group g. Two examples of this switch block, for connecting two or three different switch types, are given in Figure 6.24. The specific assignment of which tracks are connected on different sides does not matter, provided the tracks are each from the proper group.

6.5.2 Other Possible Buffer/Pass Schemes

One alternative to the above two alternating schemes is to replace only midpoint connections with pass transistors. Even more area can be saved if only endpoint connections are replaced with pass transistors instead. Other combinations like this are possible as well, forming a list of numerous switching schemes that should all reduce area (by replacing buffers with pass transistors) and may reduce delay (by combining buffered-only with buffer/pass combinations).

6.5.3 Buffer/Pass Schemes Considered

The buffer/pass switching schemes evaluated in this chapter involve assigning all combinations of the following switch types to endpoint and midpoint connections. The specific combinations of schemes are:

- midpoints: **buffered**, **pass**, **alt1**, or **alt2**

[10] Strict cycling at wire midpoints with more than two switch types is an open problem.

- endpoints: **buffered**, **pass**, **alt1**, **alt2**, **strbuf_turnpass**, or **strpass_turnbuf**.

The last two schemes, **strbuf_turnpass** and **strpass_turnbuf**, keep buffers on only straight or only turning connections, respectively, while the other connections are replaced with pass transistors.

Many combinations of the above schemes do not strictly enforce buffer/pass alternation for long, straight connections. However, delay can still be reduced because most of them do create the opportunity for alternation as a connection executes turns.

6.5.4 Experimental Results

The results of routing experiments using the switch schemes listed in the previous section are presented in Table 6.4. The entries in this table are normalized to the baseline routing architecture using *bufm* switches at both midpoint and endpoint locations.

Comparing *buf* and *bufm* Switches

The first set of rows, numbered from 1 to 8, compare the performance of the *buf* and *bufm* switches when only midpoint switches are replaced with pass transistors. As expected, more area is saved as more buffers are replaced, from **alt1** to **alt2** to **pass**. There is greater savings with 4-input LUTs, since a greater proportion of its area is consumed by the routing. Most of these buffer/pass schemes have higher delay: the *buf* switch increases delay by 7–11%, while *bufm* increases it by only 0–3%. This consistent increase is unexpected because buffer/pass schemes should improve delay of long connections. In terms of area·delay, the delay increase of the *buf* switch dominates, making it less efficient (> 1) than the baseline. In contrast, the *bufm* switch is more efficient (≤ 1).

Delay of Remaining Buffer/Pass Schemes

The second set of rows, from 9 to 24, evaluate the remaining buffer/pass schemes. Due to its superior delay performance, only *bufm* switch results are included in this portion of the table. As well, endpoint switches using only pass transistors are excluded due to very poor

Row	Endpoint Switches	Midpoint Switches	Buffer Type	Area ($\times 10^6 T$) $k=4$	5	6	Delay (ns) $k=4$	5	6	Area·Delay (T·s) $k=4$	5	6
unnormalized				**3.16**	**3.27**	**3.23**	**16.6**	**16.0**	**15.0**	**0.0523**	**0.0522**	**0.0485**
normalized												
1	buffered	buffered	bufm	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	buffered	alt1	bufm	0.98	0.98	0.99	1.03	1.00	1.00	1.01	0.98	0.99
3	buffered	alt2	bufm	0.97	0.97	0.98	1.02	1.03	1.00	0.99	1.00	0.98
4	buffered	pass	bufm	0.93	0.94	0.96	1.02	1.03	1.02	0.95	0.97	0.98
5	buffered	buffered	buf	1.03	1.02	1.02	1.08	1.07	1.08	1.10	1.10	1.10
6	buffered	alt1	buf	0.99	0.99	1.00	1.08	1.10	1.10	1.07	1.09	1.10
7	buffered	alt2	buf	0.98	0.99	0.99	1.09	1.10	1.10	1.07	1.09	1.09
8	buffered	pass	buf	0.94	0.95	0.96	1.09	1.10	1.11	1.02	1.04	1.07
9	alt1	buffered	bufm	0.99	0.99	1.00	1.03	1.03	1.03	1.02	1.02	1.03
10	alt1	alt1	bufm	0.96	0.97	0.98	1.04	1.06	1.05	0.99	1.02	1.03
11	alt1	alt2	bufm	0.96	0.96	0.97	1.05	1.06	1.07	1.01	1.02	1.04
12	alt1	pass	bufm	0.91	0.93	0.95	1.12	1.12	1.12	1.03	1.04	1.06
13	alt2	buffered	bufm	0.98	0.98	0.99	1.03	1.02	1.02	1.01	1.00	1.00
14	alt2	alt1	bufm	0.95	0.96	0.97	1.06	1.07	1.07	1.00	1.02	1.03
15	alt2	alt2	bufm	0.94	0.95	0.96	1.05	1.06	1.06	0.99	1.01	1.02
16	alt2	pass	bufm	0.90	0.92	0.94	1.09	1.09	1.11	0.97	1.00	1.04
17	strbuf_turnpass	buffered	bufm	0.99	0.99	0.99	1.06	1.07	1.09	1.05	1.06	1.08
18	strbuf_turnpass	alt1	bufm	0.96	0.96	0.97	1.11	1.13	1.16	1.06	1.09	1.12
19	strbuf_turnpass	alt2	bufm	0.95	0.96	0.97	1.12	1.13	1.12	1.06	1.08	1.08
20	strbuf_turnpass	pass	bufm	0.91	0.92	0.94	1.18	1.18	1.19	1.07	1.09	1.12
21	strpass_turnbuf	buffered	bufm	1.00	1.00	1.00	1.05	1.05	1.05	1.04	1.05	1.04
22	strpass_turnbuf	alt1	bufm	0.97	0.98	0.98	1.10	1.10	1.12	1.07	1.08	1.10
23	strpass_turnbuf	alt2	bufm	0.96	0.97	0.98	1.09	1.10	1.09	1.05	1.07	1.07
24	strpass_turnbuf	pass	bufm	0.92	0.94	0.95	1.14	1.13	1.16	1.06	1.06	1.11

Table 6.4: Area, delay, and area·delay results using different switch schemes.

delay results (20–70% increases). In the following discussion, the terminology **alt1-pass** scheme refers to interconnect using **alt1** endpoint switches and **pass** midpoint switches.

All of the remaining buffer/pass schemes are slower than the **buffered-buffered** scheme. The **alt1-buffered** and **alt2-buffered** versions have the lowest delay increase of 2–3%. As expected, the **alt1-alt1** scheme is slightly faster (by 1%) than **alt2-alt2**. However, the **alt1-pass** scheme is up to 3% slower than **alt2-pass**. The **strbuf_turnpass** and **strpass_turnbuf** schemes perform the worst (except for the excluded **pass-other** cases) at about 5–19% slower.

Endpoint Switches	Midpoint Switches	Buffer Type	Delay (ns)
buffered	buffered	buf	8.4
buffered	buffered	bufm	8.6
alt1	alt1	bufm	7.8

Table 6.5: Delay of a long connection across a PLD with 24×24 tiles.

Understanding Delay Increases

A major motivation for considering buffer/pass switching is to reduce delay, yet none of the schemes explored here are faster than the **buffered-buffered** interconnect. Routing the benchmark circuits with significantly more tracks per channel (*i.e.*, $W_{min} + 40\%$) does not improve delay either. As an initial step to investigate this issue, the Elmore delay calculations in VPRx are first tested with a simple circuit to see whether any delay improvement can be seen from a buffer/pass scheme.

The test circuit contains a single long connection between opposite corners of a PLD with 24×24 CLBs. A typical routing solution for this circuit is shown in Figure 6.25. The pad-to-pad delays for this circuit are computed by VPRx and listed in Table 6.5. As expected for a netlist with no fanout, *bufm* is 2% slower than *buf*. Also, the buffer/pass switching version, **alt1-alt1**, is nearly 10% faster than the **buffered-buffered** version. This shows that the delay model is capturing the expected timing differences.

Further investigation with some of the benchmark circuits shows that they often contain a number of high-fanout nets on the critical path. Under fanout, buffer/pass connections slow down and this can swamp any expected gain. Keeping some purely-buffered tracks would be one way to route these high-fanout nets with lower delay. This option is not explored here but is left to future investigation.

Area and Area·Delay of Remaining Buffer/Pass Schemes

Of the buffer/pass schemes shown in rows 1–4 and 9–24, those using **pass** require the least area, followed by those using **alt2**, **alt1**, then **buffered**. Interconnect using **alt2-alt2** is

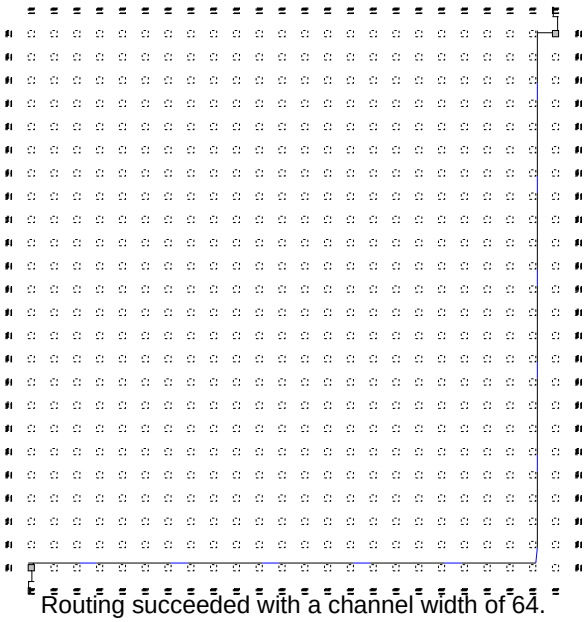


Figure 6.25: Test circuit and routing solution obtained using *buf* switches.

2% smaller than using **alt1-alt1**. The **strbuf_turnpass** and **strbuf_turnpass** schemes are unable to save more area than **alt1-pass** or **alt2-pass**, which use the least.

Only a few of the schemes in rows 9–24 are able to improve area·delay, and this occurs only for 4-input LUTs. The best are **alt2-pass**, **alt2-alt2** and **alt1-alt1**. In general, the **alt-alt** schemes keep area·delay increases within 6%, but the **strbuf** and **strpass** schemes increase it by up to 12%.

Summary

The best overall switching schemes are summarized in Table 6.6. Unlike the previous table, these results are normalized to the *buf* switch results to illustrate the total savings realized. The best delay scheme uses only buffered interconnect, but 7% is saved using the *bufm* switch. The best area·delay scheme uses midpoint pass transistors, achieving an 11–14% savings. The best area scheme saves 8–13% yet sacrifices only 1–2% in delay. The use of buffer/pass interconnect does not improve delay, but this is likely due to high fanout nodes that would be better mapped onto purely buffered tracks. This could be an avenue of future work.

Criterion	Endpoint Switches	Midpoint Switches	Buffer Type	Area ($\times 10^6 T$) $k=4$	5	6	Delay (ns) $k=4$	5	6	Area·Delay (T·s) $k=4$	5	6
unnormalized				**3.25**	**3.34**	**3.28**	**17.8**	**17.1**	**16.2**	**0.058**	**0.057**	**0.053**
normalized												
baseline	buffered	buffered	buf	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
best delay	buffered	buffered	bufm	0.97	0.98	0.98	0.93	0.93	0.93	0.91	0.91	0.91
best area·delay	buffered	pass	bufm	0.91	0.93	0.94	0.95	0.96	0.94	0.86	0.89	0.89
best area	alt2	pass	bufm	0.87	0.90	0.92	1.01	1.02	1.02	0.88	0.91	0.95

Table 6.6: Best buffer/pass schemes compared to the baseline.

6.6 Conclusions

Using a level-restoring circuit instead of gate-boosting is an effective way to solve the static power dissipation problem. When employing this circuit technique, signal falltime can become dominant. As well, dangerous stuck-high states may be reached when buffers cannot overpower a distant level-restoring circuit. This condition, described as the level-restoring pulldown problem, must be solved in large PLDs by carefully designing the router software and/or the architecture.

The fastest routing buffers use three buffer stages. This is more likely a result of the inverting property of the buffer, rather than lower intrinsic delay. A larger NMOS transistor on the sense stage helps speed signalling, since the full rail voltage of V_{dd} is not easily reached through NMOS pass transistors. Also, equal-size NMOS and PMOS drive-stage transistors are sufficient to reach low delay. Using a larger PMOS transistor, such as the traditional choice of twice the NMOS transistor width, consumes more area but does not effectively increase drive strength. The limiting factor during pullup is the NMOS pass transistor used as the tristate control.

Buffer intrinsic delays are sensitive to input slew rates. The slew rate degrades as a signal passes through more pass transistors (and wires). Consequently, the intrinsic delay of size 6 buffers triples and the intrinsic delay of size 16 buffers doubles as slew rates degrade by travelling through up to eight wires connected in series with pass transistors. The calculated error for these cases is up to 8% of total net delay. This is small enough to be ignored in the first-generation tools used here, but timing analyzers and second-generation routing tools should probably account for this effect,

A search for the optimum switch size, in terms of area and area·delay, with a wide range of tile lengths produces an unexpected result. It is found that a fixed buffer size obtains within 5% of the best possible delay and area·delay for all of these tile lengths. Even for small tile lengths of less than 150–200μm, the impact of using a fixed switch size on delay or area·delay is less than 5%. In general, however, the best delay is reached by scaling switches sub-linearly with tile size using an exponent of roughly 0.2.

Modelling buffer fanout in VPRx increases delay results by 5% on average, or 16% in the worst case. Three new switches, *bufm*, *bufp* and *bufp2*, have been presented which nearly eliminate this increase by avoiding fanout at the buffer source. The *bufp* switch is the fastest switch, but it is typically larger than *buf*. The *bufp2* is more a area-efficient version, but it is slower. The *bufm* switch is faster than *bufp2* and just as area-efficient, so it is the preferred choice. All of the new switches are more area efficient than the previous ones for large buffer sizes. The *bufm* switch is also more efficient when higher flexibility (fanin) is required. This produces an area savings if the CLB output pins are connected directly into the routing switches via *output pin merging*.

Replacing some buffers with pass transistors creates an interconnect capable of alternating signals between buffers and pass transistors. In doing this, area savings is guaranteed. Even though alternating between these switches produces up to 25% faster delays in HSPICE, no delay improvement is seen in the final routed circuits. Instead, a delay increase of up to 5% is typical, but increases up to 70% are observed in some poor architectures. Hence, proper buffer/pass design is essential to avoid these large increases. Although no direct proof is given, the increase is presumably caused by fanout loading; the delay improvement is only expected for long, single-fanout nets. Further investigation is required to realise delay improvement.

Overall, the architectures of choice are summarised in Table 6.6. To illustrate the total area and delay savings realised with the new switch designs, these results have been normalised to the baseline architecture results. The best delay scheme uses only buffered interconnect with the *bufm* switch and reduces delay 7%. The best area·delay scheme uses midpoint pass transistors, achieving an 11–14% savings. The best area scheme saves 8–13% yet sacrifices only 1–2% in delay. Due to the use of fast, wide pass transistors in the

unmodified portion of the interconnect, all of these results are deemed to be conservative estimates.

6.7 Future Work

The improvement from replacing buffered interconnect with a mixture of buffers and pass transistors is diluted due to fanout at the pass transistors. It would be interesting to observe whether delay increases can be prevented by keeping some purely-buffered tracks. Alternatively, the router could be altered to encourage fanout at buffered switches instead where it will have a minimum impact on delay. The present implementation hides this detail from the router when it is expanding the wavefront. These techniques may produce an overall delay savings up to 10% with the conservative architecture assumed here, or up to 25% if a more aggressive design is used.

Tristate buffers considered in this chapter use a single NMOS pass transistor to reach high impedance state. This reduces drive strength and increases delay. Other buffer designs with improved drive strength could be investigated to reduce delay (possibly at the expense of area). Unidirectional wires, such as those found in Virtex I, could also be examined.

The circuit design issues here do not consider dynamic power, yet power use is an ever-increasing problem as devices get larger and larger. The interconnect contains significant capacitance, so methods to reduce power should take this into consideration. For example, low-swing circuits such as those presented in [142] could be used to cut energy use in half.

Also, a robust solution to the level-restoring pulldown problem is required in pass-transistor dominated interconnect. One possible solution is simply creating buffered boundaries at regular intervals to limit maximum fanout of nets routed with pass transistors.

Chapter 7

Switch Block Design Framework

Previous switch block design has focused on the analysis of individual switch blocks or the use of ad hoc design with experimental evaluation. This chapter presents an analytical framework which considers the design of a continuous fabric of switch blocks containing any length of wire segments. The framework is used to design new switch blocks which are experimentally shown to be as effective as the best ones known to date. With this framework, we hope to inspire new ways of looking at switch block design.

7.1 Introduction

Over the past several years, a number of different switch block designs have been proposed such as those shown in Figure 7.1. PLDs such as the Xilinx XC4000-series [31] use a switch block style known as *disjoint*. Some alternatives to this style are commonly known as *universal* [52] and *Wilton* [63]. These latter two topologies require fewer routing tracks and use less transistor area than the *disjoint* pattern with interconnect containing single-length wires. However, with longer wire segments they use more switches per track and often require more transistor area overall [19]. The *Imran* block [64, 65] reduces this area overhead by modifying the *Wilton* pattern to use the same number of switches as the *disjoint* pattern.

One notable difference between these switch blocks is their design methodology. The *universal* switch block is analytically designed to be fully routable, when consid-

ered in isolation, for all two-point nets. For this reason, it is said to be locally optimal. Fan *et al* [60, 62] extends this local optimality to include multipoint nets to create a *hyperuniversal* switch block. Both of these blocks rely on reordering nets at every switch block, so their local optimality does not extend to the entire routing fabric. In comparison, the *Wilton* and *Imran* switch blocks are examples of ad hoc design with experimental validation. The *Wilton* switch pattern changes the track number assigned to a net as it turns. This way, two different global routes may reach two different routing tracks at the same destination channel. This forms two disjoint paths, a feature called the *diversity* of a network. The *Wilton* and *Imran* designs introduce the notion that a switch block must consider its role as part of a larger switching fabric, including the effects of long wire segments.

The above two methods have produced switch blocks which perform well, but there is no known formal method to design a switch block while considering the overall routing fabric. In pursuit of this goal, this chapter introduces an analytical framework to design switch blocks which considers both long wire segments and the interaction of many switch blocks connected together. This framework includes a restricted switch block model which allows one to analyse, measure and control the diversity of the network. This framework is used to design an ad hoc switch block named *shifty* and two analytic ones named *diverse* and *diverse-clique*. These new switch blocks are very diverse, and routing experiments show they are as effective as the others.

The main purpose of the framework is to simplify the organisation of the interconnect fabric. This leads to a better understanding of how long wire segments and different switch locations affect the global fabric, a feature often overlooked in the design of other switch blocks. This framework and the switch blocks designed with it are intentionally simple. By building upon the simple foundations of the framework, it may be possible to design more efficient and complex networks, such as those containing many different wire segment lengths.

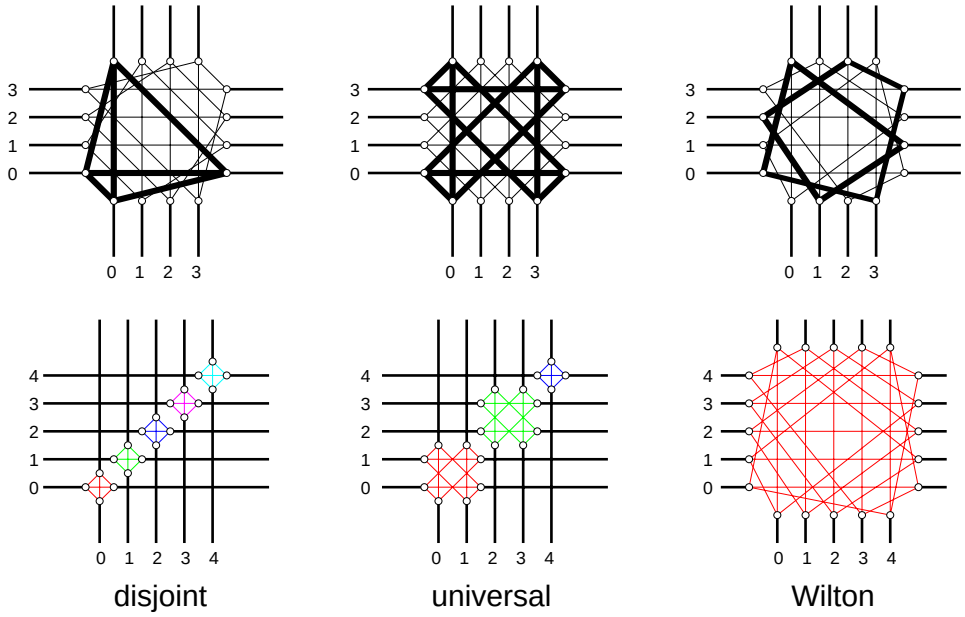


Figure 7.1: *Disjoint*, *universal* and *Wilton* switch block styles.

7.2 Design Framework

This section describes a new switch block framework, being composed of a new switch block model, permutation mapping functions, and a few basic assumptions. As well, an option to consider using only commutative permutation mapping functions is presented.

7.2.1 Switch Block Model

The traditional model of a switch block draws a large box around the intersection of a horizontal and vertical routing channel. Within the box, switches connect a wire on one side to any wires on the other three sides. Long wire segments pass straight across the switch block, but some track shifting is necessary to implement fixed length wires with one layout tile. Figure 7.2a) presents this model in a new way by partitioning the switch block into three types of subblocks: *endpoint* (f_e), *midpoint* (f_m), and *midpoint-endpoint* (f_{me}) subblocks. The endpoint (midpoint) subblock is the region where the ends (midpoints) of wire segments connect to the ends (midpoints) of other wire segments. The f_{me} subblock connects the middle regions of some wires to the ends of others. A switch placed between wires on two sides automatically falls into one of these three types of subblocks.

The traditional model in Figure 7.2a) is too general and makes diversity analysis quite complex. This can be simplified by restricting switch locations to smaller regions than what is allowed by the traditional model, producing the new models in Figure 7.2b) through d). The *crossing locations model* of Figure 7.2b) allows some midpoint-to-endpoint connections, but only if the wires naturally cross. The *midpoint/endpoint segregation model* of Figure 7.2c) removes the ability to connect midpoints to endpoints entirely. This model is used implicitly by the *Imran* block [64, 65] to reduce the number of switches used by the *Wilton* block with long wire segments. The last model, called the *track group model*, is the one recommended for use within this framework.

The track group model partitions wires into *track groups* according to their wire length and starting points. This can be seen by illustration at the top of Figure 7.3, where four switch blocks are shown with length-four wires. In this figure, the path of a horizontal wire is highlighted in green to show how it passes through a different subblock at every switch block. Notice that the wires all starting at the same location as the green wire connect in the same subblocks: this is the track group. Similarly, although only one switch block is shown, the red wire passes through different subblocks in the vertical direction to form vertical track groups. Due to the organisation of the switch locations, the different track groups always remain disconnected from each other, establishing their independence. Hence, it is sufficient to examine a single track group.

In Figure 7.3, the midpoint subblocks are labelled $f_{m,i}$, where i is a position between 1 and $L-1$ along the wires of length $L=4$. Hence, switch locations are restricted to $L-1$ midpoint subblocks and one endpoint subblock. The lower part of Figure 7.3 illustrates a single track group, containing the red and green wires, in a fabric of 2×4 logic blocks (CLBs). The other three track groups (not shown) are staggered to use the remaining three possible starting points. As mentioned earlier, there are no connections between the track groups.

This model is somewhat restrictive, but it can still represent many switch blocks, *e.g.*, *Imran*, and routing experiments (which will be fully described below) indicate it has good performance. As well, early experiments that did not restrict switches to only the $f_{m,i}$ subblocks did not produce better results. By disconnecting wires that have different starting

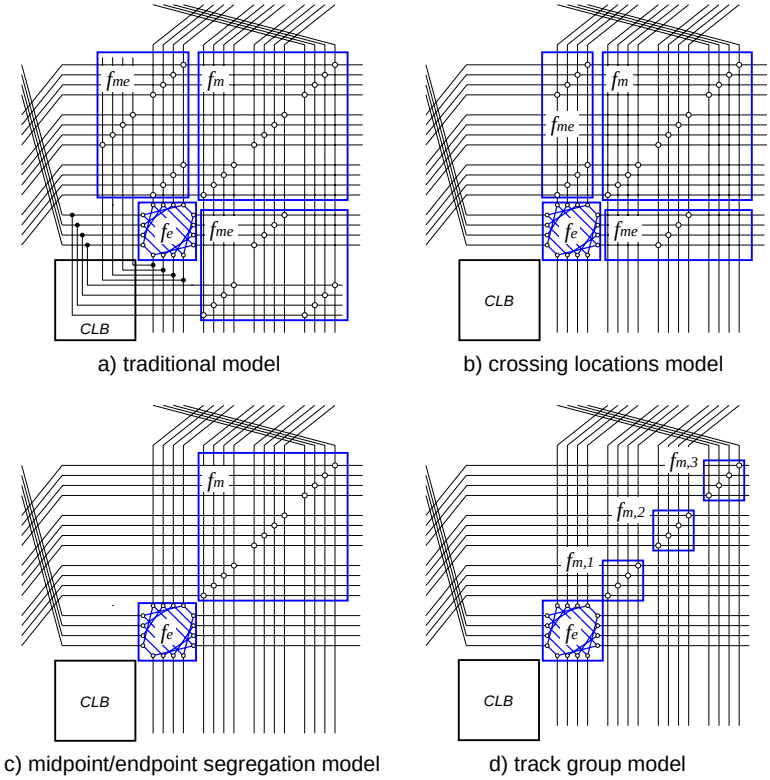


Figure 7.2: Switch block models containing subblocks.

points and different lengths, this model explicitly forces track groups to be in separate routing domains. This simplifies analysis and design because each track group can be treated separately.

7.2.2 Permutation Mapping Functions

Previous work suggests only a small number of switches need to be placed within a switch block. Pioneering work by Rose and Brown [42] defined switch block flexibility, or the parameter F_s, as the number of other wires connecting to each wire in a block. That work found that $F_s = 3$ is the lowest flexibility that remains routable with single-length wire segments. Other work by Betz *et al* [19] and Masud [65] with longer wire segments uses $F_s = 3$ at wire endpoints and $F_s = 1$ at wire midpoints. They also found that switch blocks which use slightly more switches than this (*i.e.*, *Wilton* and *universal*) result in higher transistor area overall. As well, our experience with $F_s < 3$ (see Appendix A) at endpoints

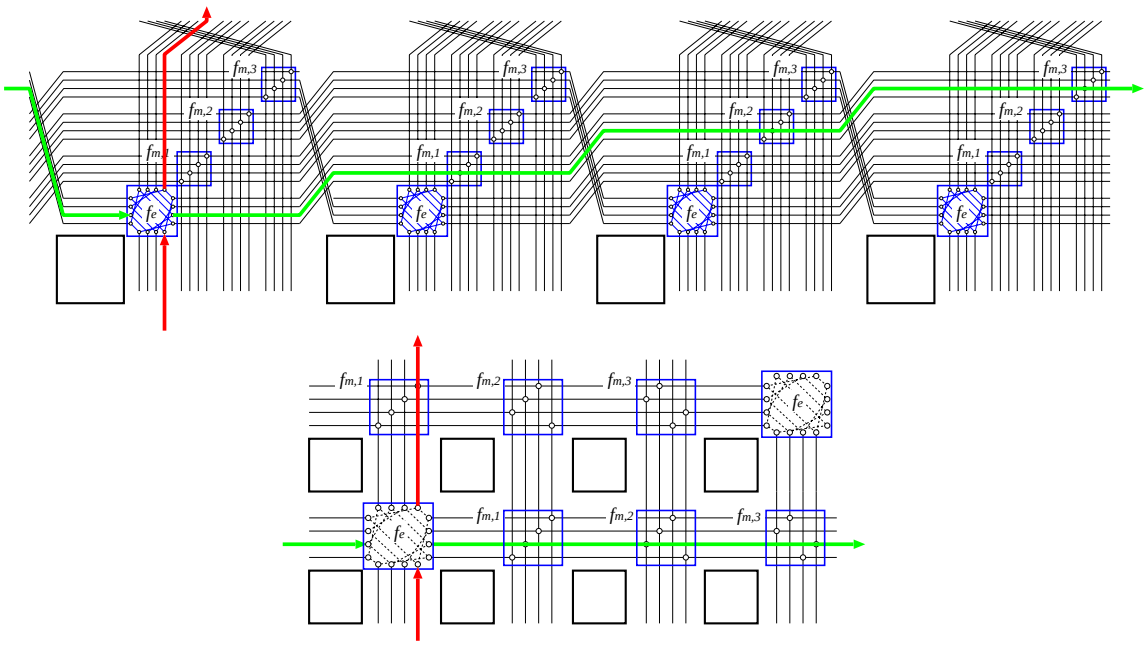


Figure 7.3: Four switch blocks (above) and a portion of the switching fabric created by one track group (below).

is that a few more tracks but less transistor area is needed. This suggests $6W$ and W are reasonable upper bounds for the number of switches in endpoint and midpoint subblocks, respectively.

Given these upper bounds, switch locations can be represented by a *permutation mapping function* between each pair of sides. The different mapping functions and their implied forward direction are shown in Figure 7.4. In this figure, $f_{e,i}(t)$, or simply $f_{e,i}$, represents the mapping function for an endpoint turn of type i. A switch connects the wire originating at track t on one side to track $f_{e,i}(t)$ on the destination side. Turns in the reverse direction to those indicated are represented as $f_{e,i}^{-1}$ such that $f^{-1}(f(t)) = t$.

Similarly, $f_{m,i}$ is a mapping function representing a midpoint turn at position i along the length of a wire, with the most South/West point being the origin at position $i = 0$. The right side of Figure 7.4 illustrates the different midpoint subblocks in a fabric of 2×4 CLBs with a single track group. The other three track groups would be similar, but they are independent and have staggered starting locations relative to the track group shown in the figure. Of course, there are no connections between the track groups.

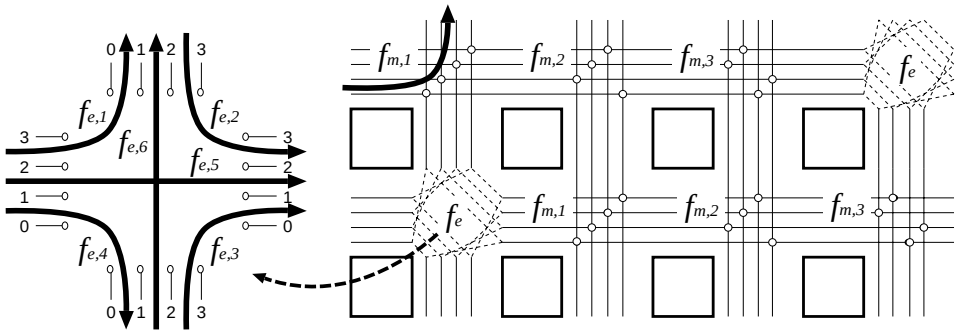


Figure 7.4: Mapping functions for endpoint and midpoint subblock turns.

Mapping		***universal***			
Function	***disjoint***	**standard**	**alternate 1**	**alternate 2**	***Wilton***
$f_{e,1}(t)$	t	$W-t-1$	t	t	$W-t$
$f_{e,2}(t)$	t	t	$W-t-1$	t	$t+1$
$f_{e,3}(t)$	t	$W-t-1$	t	t	$W-t-2$
$f_{e,4}(t)$	t	t	$W-t-1$	t	$t-1$
$f_{e,5}(t)$	t	t	t	$W-t-1$	t
$f_{e,6}(t)$	t	t	t	$W-t-1$	t

Table 7.1: Mapping functions for some switch block styles.

Examples of mapping functions for various switch blocks are shown in Table 7.1. Each of these functions are modulo W, where W is the track group width. Also, note that it is common for connections straight across a switch block (E–W or N–S) to stay in the same track, so it is usually assumed that $f_{e,5} = f_{e,6} = t$.

7.2.3 Additional Assumptions

In addition to the explicit assumptions stated above, there are a few implicit ones being made as well. It is assumed that the subblocks (and the switch blocks) are square with W tracks on each side. Although non-square blocks might be useful in some applications, square blocks form a more area-efficient interconnect fabric [19, 79].

As well, it is assumed that the number of switches are uniformly distributed over each wire and each subblock. This means there is a one-to-one correspondence between the originating track and the destination track. Since $f^{-1}(f(t)) = t$, it is also presumed that

each switch is bidirectional.

Finally, each track group is assumed to be homogeneous, consisting of only one wire length and one switch type. It is possible to relax these assumptions, but this option is not explored here. For example, one wire length can be used for vertical wires and another can be used for horizontal wires.

7.2.4 Commutative Switch Blocks

The mapping functions of the *universal* and *Imran* switch blocks involve *twists* where the function is of the form $f(t) = W - t + c \bmod W$. Due to the twist, or $-t$ portion, these functions are not commutative. For example, consider the two mapping functions $f_i(t) = 7 - t$ and $f_j(t) = 4 - t$ for a fixed $W = 8$. To see that the composition of these functions is not commutative, observe that $f_i(f_j(t)) \neq f_j(f_i(t))$:

$$\begin{aligned} f_i(f_j(t)) &= 7-(4-t) = 3+t \\ f_j(f_i(t)) &= 4-(7-t) = -3+t = 5+t(\bmod 8). \end{aligned}$$

Although some mapping functions may not be commutative, it is possible to select all of the functions so they are commutative with composition, leading to the definition below.

> *Definition.* A switch block is *commutative* if every pair of its mapping functions is commutative with composition. Hence, it is required that $f_i(f_j(t)) = f_j(f_i(t))$ for every pair of mapping functions, $f_i(t)$ and $f_j(t)$, in the switch block.

Commutative switch blocks are an optional part of the framework, but they will be used throughout this chapter.

To help distinguish between commutative and non-commutative switch blocks, consider the example shown in Figure 7.5. In this figure, two paths are compared in two different architectures: the left architecture uses commutative switch blocks, but the right one does not. In both cases, the destination track of the upper path is $f_{e,2}(f_{e,4}(f_{e,3}(f_{e,1}(t))))$, while the lower path is $f_{e,3}(f_{e,1}(f_{e,2}(f_{e,4}(t))))$. In a commutative architecture, both paths

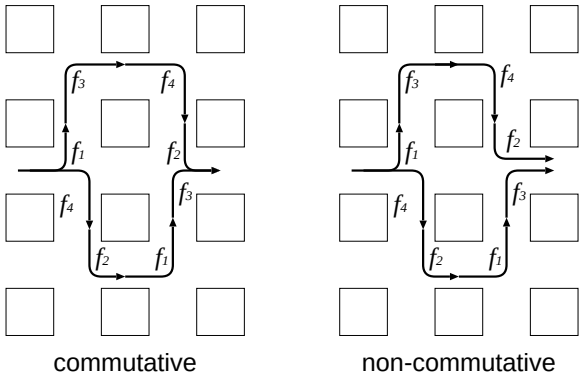


Figure 7.5: Turn order is not important in commutative switch blocks.

can be rewritten as $f_{e,1}(f_{e,2}(f_{e,3}(f_{e,4}(t))))$. These necessarily reach the same track. However, in a non-commutative architecture, the operations cannot be reordered and the paths may reach different tracks. This example suggests that commutative architectures are less diverse. However, results presented later in Section 7.4 will demonstrate that commutative switch blocks can be made quite diverse and are as routable as the non-commutative *Imran* block.

The ability to commute the permutation mapping functions is useful because it simplifies analysis of the network: the order in which turns are made is unimportant, so a number of alternative paths become equivalent. In Section 7.3.2, this will significantly reduce the search space for designing diverse switch blocks. Another use for commutative switch blocks is to reduce an arbitrary number or sequence of turns to a canonical form where only the number of turns of each type is important. This may assist the generation of alternative paths to reach a desired destination track without performing complex detailed routing. For example, this allows *domain negotiation* [54], a technique used to accelerate net routing by checking for available tracks near the input pins, to be applied in architectures with switch blocks other than *disjoint*.

7.3 Framework Applications

To illustrate the use of the framework just described, this section presents the design of various switch blocks, first using an ad hoc method and then using an analytical method.

Each design involves the selection of permutation mapping functions for a switch block. For simplicity, all examples assume length four wires are used in the interconnect.

7.3.1 Application: *shifty* and *universal-L* Switch Block Designs

The first application of the new framework is the design of a commutative switch block similar to *Imran* but without the non-commutative twists. The following mapping functions describe the new switch block: $f_{e,1}(t) = t-1$, $f_{e,2}(t) = t-3$, $f_{e,3}(t) = t-2$, $f_{e,4}(t) = t-4$, and $f_{m,i} = t \pmod{W}$. This block is named *shifty* because each turn involves a shift from one track number to another by a constant amount. The constant values are chosen to be small because the arithmetic is always done modulo W. This avoids $f_{e,1}$ from being equivalent to $f_{e,4}$, for example, except with certain small W values.

Other switch blocks can also be adopted within this framework. For example, the *disjoint* and *Imran* switch blocks naturally conform to the track group model already. As well, suppose the *universal* pattern is applied only at endpoint subblocks and the identity mapping $f_{m,i} = t$ is used at midpoint subblocks. With this new pattern, *universal-L*, *each subblock* can connect any set of two-point nets that obey basic bandwidth constraints. When long wire segments are used, the *universal-L* pattern requires less transistor area than the *universal* pattern used within VPR. This savings is achieved in the same way that *Imran* improves *Wilton*: the number of switches per track is reduced by enforcing $F_s = 3$ at endpoints and $F_s = 1$ at midpoints.

Checkered Layout for Increased Diversity

To create additional diversity, it is possible to use two different switch block designs (two layout tiles) in a checkered layout pattern. If one switch block design is assigned to the white square locations, a different one can be used at the black square locations. These black square switch blocks will be characterised by their own mapping functions, g.

To see the impact on diversity, consider the routing example in Figure 7.6 with single-length wire segments. Without checkering, the two paths shown always reach the same track, $f_3(f_1(t))$. This is always true, regardless of which switch block is used. With the checkering of two switch blocks, the two paths may reach two different tracks, $g_3(f_1(t))$

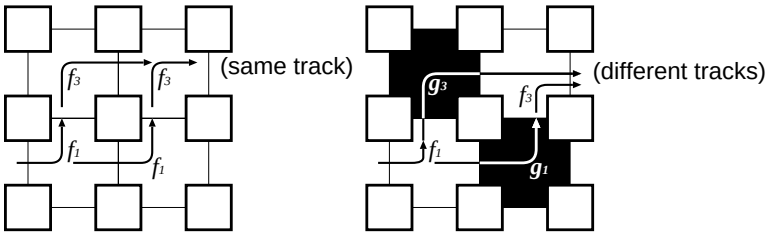
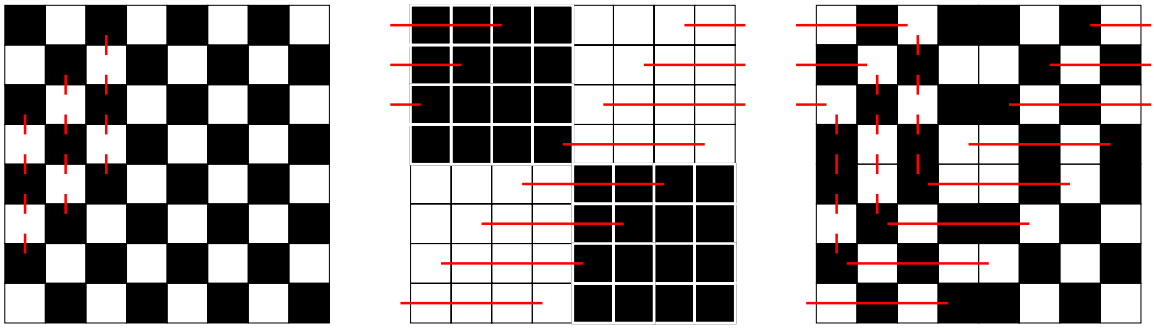


Figure 7.6: Checkering two switch blocks can increase diversity with single length wires.



a) checkering for length 1 wires: **all** wires end at both types of switch blocks.

b) checkering for length 4 wires: **all** wires end at both types of switch blocks.

c) checkering for length 1 & 4 wires: **all** length 4 and **most** length 1 wires end at both types of switch blocks.

Figure 7.7: Checkering with different wire lengths.

and $f_3(g_1(t))$. With checkering, the f and g functions can be chosen to provide diversity, even if no diversity exists with f alone (for example, the *disjoint* switch block).

The checkering pattern must be adjusted when it is used with longer wire segments. Figures 7.7 a) through c) show three checker patterns that can be used with length 1, length 4, or a mixture of length 1 and 4 wires, respectively. Mixing two different wire lengths requires the exclusive-or of the two constituent checker patterns. Lines drawn on top of the checker pattern show how wires if different lengths have their endpoints located in differently-coloured squares. The pattern shown in Figure 7.7c) is used in this dissertation, although Figure 7.7b) could have been used as well.

To experiment with checkering, ad hoc selection of g mapping functions for various switch blocks has been made. The g functions are designed to be slightly different from their f counterparts. The final f and g mapping functions are shown in Table 7.2. To preserve the layout sub-structures of the *disjoint* and *universal-L* blocks, the g_e functions are chosen so that only the horizontal tracks are re-ordered.

Mapping Function	Type of Switch Block			
	disjoint	***universal-L***	***Imran***	***shifty***
$f_{e,1}(t)$	t	$W-t-1$	$W-t$	$t-1$
$f_{e,2}(t)$	t	t	$t+1$	$t-3$
$f_{e,3}(t)$	t	$W-t-1$	$W-t-2$	$t-2$
$f_{e,4}(t)$	t	t	$t-1$	$t-4$
$f_{m,i}(t)$	t	t	t	t
$g_{e,1}(t)$	$t-1$	$W-t-2$	$W-t+3$	$t-8$
$g_{e,2}(t)$	$t+1$	$t+1$	$t+3$	$t-7$
$g_{e,3}(t)$	$t+1$	$W-t$	$W-t+2$	$t-9$
$g_{e,4}(t)$	$t-1$	$t-1$	$t+1$	$t-6$
$g_{m,i}(t)$	$t+1$	$t+1$	$t+1$	$t+1$

Table 7.2: Switch block mappings used for white (f) and black (g) squares.

7.3.2 Application: Diverse Switch Block Designs

This section will use the design framework to develop commutative switch blocks that are maximally diverse for all possible two-turn paths. Two different switch blocks will be designed, *diverse* and *diverse-clique*. The latter design is more restricted because its endpoint subblock uses the 4-wire clique layout structure of the *disjoint* switch block. This design is repeated for an architecture containing two layout tiles, f and g, arranged in a checkered pattern.

Design Space

Let each switch block mapping function be represented by the equations $f_i(t) = t + a_i \bmod W$ or $g_i(t) = t + b_i \bmod W$, where i represents one of the endpoint or midpoint turn types. Constraining f and g functions in this way explores only a portion the design space. However, it will be shown that this is sufficient to develop very diverse switch blocks.

A specific switch block is constructed by selecting a set of values for a_i and b_i. These will be chosen to give the switch block maximum diversity for all two-turn paths. Diversity is measured by the number of pairs of paths that reach different destination tracks in the

same channel region.

The set of values or *solution set* for a particular switch block design instance can be expressed in vector form as:

$$\mathbf{x}_W = \left[\begin{array}{cccccccccccccc} a_{e,1} & a_{e,2} & a_{e,3} & a_{e,4} & a_{m,1} & a_{m,2} & a_{m,3} & b_{e,1} & b_{e,2} & b_{e,3} & b_{e,4} & b_{m,1} & b_{m,2} & b_{m,3} \end{array} \right]^T.$$

Since the mapping functions are all $\bmod W$, a solution set $\mathbf{x}_W$ is only valid for a specific value of W.

Enumerating the Path-Pairs

Before counting diversity, all possible paths containing two turns and all pairs of these paths, or *path-pairs*, leading to the same channel region must be enumerated. The enumeration of these path-pairs is described below.

There are two basic types of two-turn paths, an S-turn and a U-turn. Figure 7.8 shows the four possible types of S-turns, and four types of U-turns. Two of the S-turns are commutatively equivalent to the other two, so they can be ignored. This leaves six basic types of two-turn paths: ENE, ESE, ENW, WNE, NES and SEN, where N, S, E, or W refer to compass directions.

In Figure 7.8, the paths are shown in pairs separated by a single logic block. This exemplifies a path-pair using a different global route to reach the same destination. Additional path-pairs can be formed by changing the global route to increase the number of logic blocks between them.

In general, all possible two-turn path-pairs of an infinite routing fabric can be enumerated using the 8×8 grid or *supertile* in Figure 7.9. The size of this supertile arises from the length-four wire and the checkering of two layout tiles; without checkering, a 4×4 grid is sufficient. Within the supertile, each square is labelled with the pertinent mapping functions for one track group. Other track groups would have a similar, but shifted, labelling. Paths which fall within the supertile have their mapping function indicated by the label on that square. Paths which traverse longer horizontal or vertical distances can be ignored: they will reach another supertile and turn at a switch block equivalent to one already in this supertile.

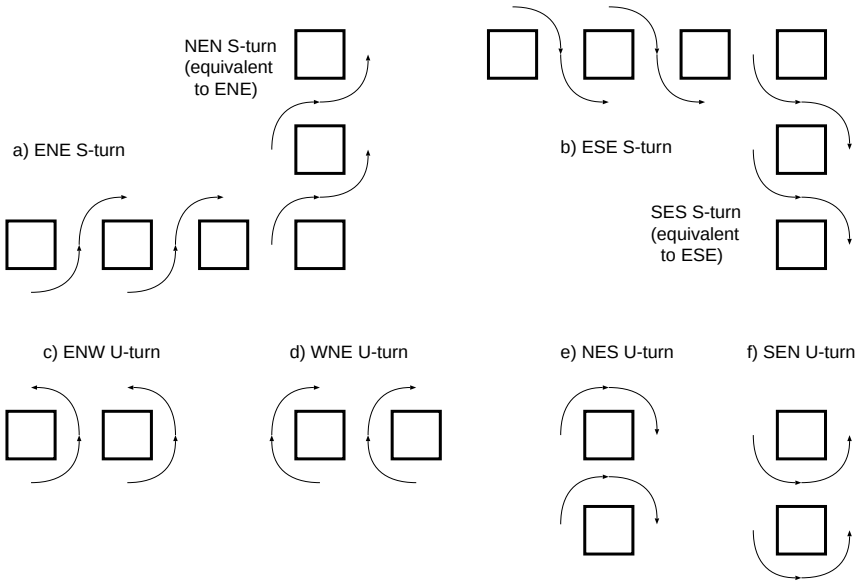


Figure 7.8: The six different two-turn path types.

To better understand the supertile and equivalent two-turn paths, consider the global route for an ENE connection. An input is shown entering the bottom row of Figure 7.9 at the lower left edge. The connection travels horizontally, turns North in one of eight columns A through H, travels vertically some distance, and then turns East into one seven possible rows labelled *out1* through *out7*. Hence, there are 8 different global routes or paths (one for each column) to reach each of the seven output channels.

A number of other isomorphic paths can be eliminated using the supertile and commutative property. For example, only a single input row (or column) needs to be considered for the path origin. Other input rows (or columns) merely rotate the contents of the supertile vertically (or horizontally), shifting the starting point to another square. As mentioned earlier, NEN and SES paths are commutatively equivalent to ENE and ESE paths, respectively, so they can be ignored. Also, all paths in the reverse directions to those shown in Figure 7.8 are commutatively equivalent (using the inverse mappings) and can be ignored. For example, a WSW path is commutatively equivalent to an ENE one. Finally, it has been assumed that connections straight across endpoint switch blocks stay on the same track, *i.e.*, $f_5(t) = f_6(t) = t$. If this is not true, then path-pairs which span more than one supertile can still be considered equivalent if commutative switch blocks are used.

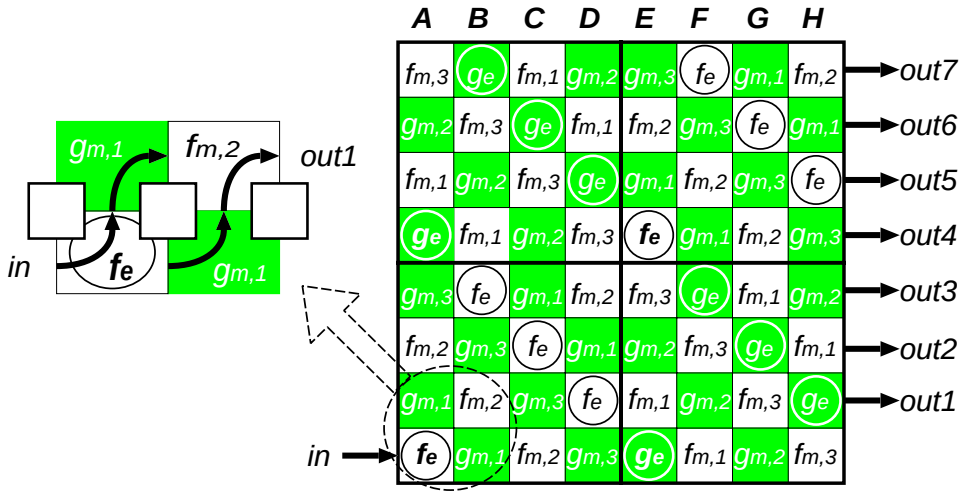


Figure 7.9: An 8×8 grid or supertile used for enumerating all two-turn paths.

For maximum diversity, each pair of paths that reach the same output row must reach different tracks. With 8 possible routes (columns A–H), there are $\binom{8}{2} = 28$ pairs of paths to be compared. Hence, for all turn types and all output rows (or columns), there are $6 \times 7 \times 28 = 1176$ *path-pairs* to be compared.

Counting Diversity

To detect diversity between a pair of paths, first compute the difference between the two permutation mappings $y = f_{pathA} - f_{pathB}$. The path-pair is diverse if y is non-zero. Since the expression for y is a simple linear combination of constant values from $\mathbf{x}_W$, the equations can be rewritten in matrix form,

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x}_W \mod W$$

where each row in $\mathbf{A}$ is predetermined based on the path-pair being considered. Each entry in $\mathbf{y}$ represents the diversity test result of the corresponding path-pair.

The size of $\mathbf{A}$, and subsequently the search space, has been considerably reduced by the large number of equivalent paths eliminated due to the commutative property. Other path types can be considered as well, not just two-turn paths. This requires additional rows in $\mathbf{A}$ and $\mathbf{y}$.

Diversity of a given switch block $\mathbf{x}_W$ is measured by counting the number of non-zero

solutions in $\mathbf{y}$. For the architecture used here, the maximum diversity is 1176, the number of path-pairs.

Searching Design Space

Rather than solve a large number of equations to maximise the number of non-zero entries in $\mathbf{y}$, the best result is taken from three types of random and brute-force searches of $\mathbf{x}_W$ for each W ranging from 2 to 18. The search techniques are described below.

The first search is an exhaustive search of $\mathbf{x}_W$. The second search assigns random values to $\mathbf{x}_W$. The third search is a randomized iterative sequential search. This last search starts with a random $\mathbf{x}_W$ and varies one element at a time from 0 to $W-1$. Before proceeding to the next element, the current one is fixed at the value that maximises diversity. Additional passes are done until the solution cannot be improved. At this point, a new random starting point is chosen.

To find switch blocks at a given W between the range from 2 to 18, each of these searches is allowed to run for roughly one CPU day.[1] The result with the greatest diversity is kept. For $W < 5$, the exhaustive search runs to completion and finds a best solution; all other solutions are the best-found upon termination of the time limit. For intermediate values of W, the exhaustive search nearly always produces a better result, even though it doesn't run to completion. The iterative search is not any more effective than the random guesses except that it is faster at quickly generating good results.

7.4 Results

This section begins by characterising the *diverse* and *diverse-clique* switch blocks created using the design methodology from the previous section. These two new switch blocks, plus the *shifty* switch block, are then evaluated in two ways. First, the diversity of the new switch blocks is counted to show that the new designs are significantly more diverse than the *disjoint* switch block. Second, routing experiments are performed using the switch blocks and the area results and delay results are compared.

[1] Performed on a 1 GHz Intel Pentium III computer.

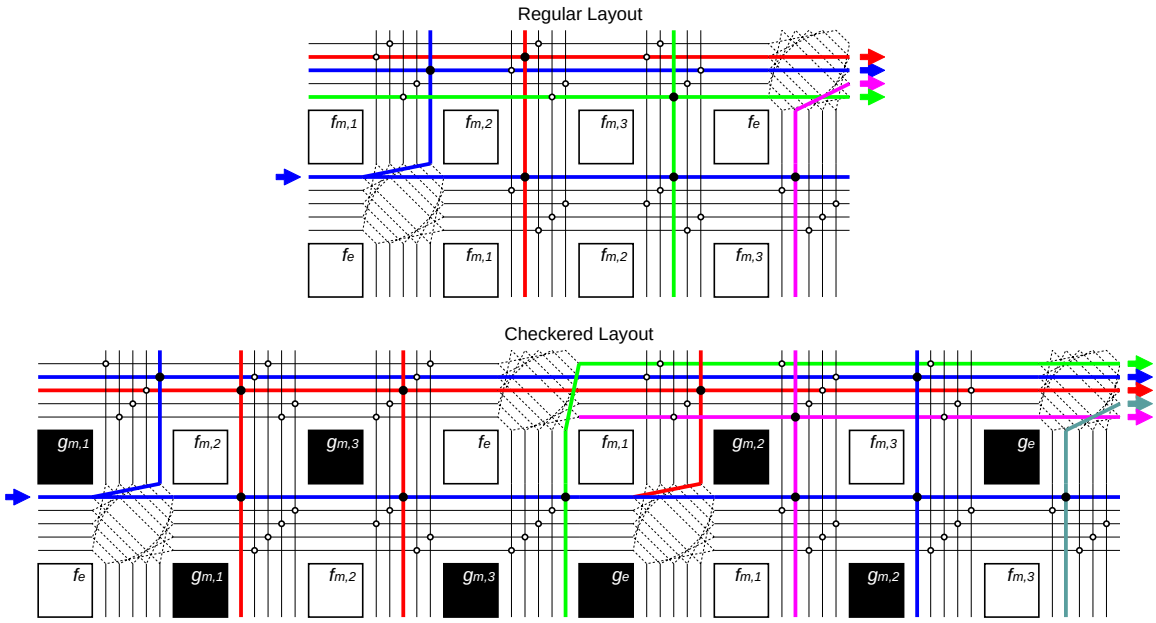


Figure 7.10: Regular and checkered layout of a $W = 5$ *diverse-clique* switch block.

7.4.1 Switch Block Design Results

Using the above procedure, switch blocks named *diverse* are designed for a variety of track group widths, $W \leq 18$. For each W, a solution set $\mathbf{x}_W$ is found. A similar procedure is followed to design *diverse-clique* switch blocks, except they contain an added constraint to preserve the 4-wire clique structures within the endpoint subblocks. A layout strategy for these cliques is given in [150].

The searches produced a number of switch block designs with a similar amount of diversity for each value of W. The precise solution sets used in this chapter and hard-coded into the VPRx router, an extended version of VPR [116, 19], are given in Appendix B.

Although no clique constraints are placed during the design of the *diverse* switch blocks, they are found in many of the solution sets. In particular, there are cliques in f_e for $W = \{2,3,4,5,7,9,10,11,12,16\}$ and in g_e for $W = \{3,4,6,7,8,10,11,12,13,14\}$. This means that many *diverse* switch blocks can also be constructed using simple layout structures.

The details of a $W = 5$ *diverse-clique* switch block are shown in Figure 7.10. This figure illustrates the diversity obtained for a number of two-turn paths by colouring the

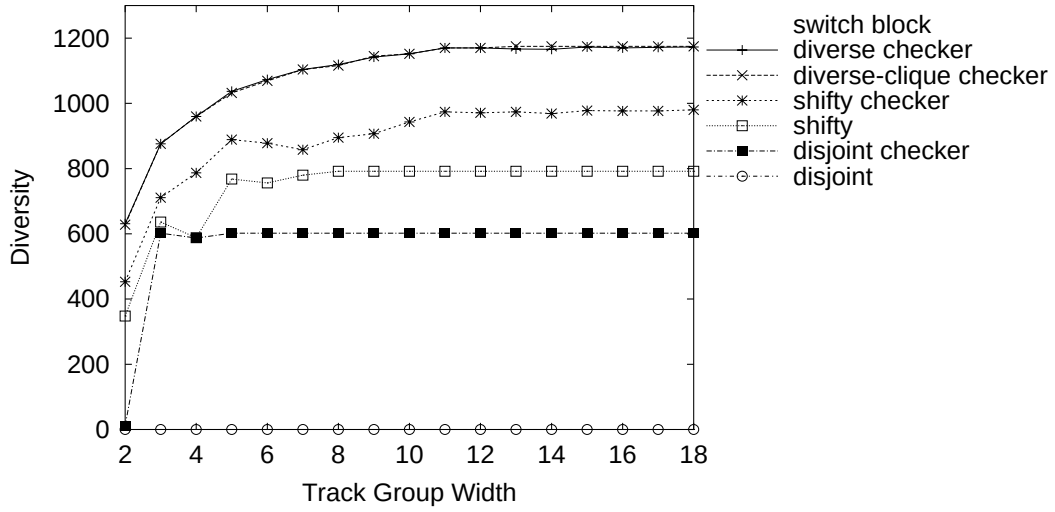


Figure 7.11: Diversity of various commutative switch blocks.

different tracks reached.

7.4.2 Diversity Results

The diversity of the new switch blocks are compared with the *disjoint* switch block in Figure 7.11. In this graph, diversity is measured as described in Section 7.3.2; the maximum diversity possible is 1176.

The *disjoint* switch block has no diversity but its checkered version has considerably more. The *shifty* switch block and its checkered version provide even more diversity. However, the *diverse* and *diverse-clique* checkered switch blocks reach the highest levels of diversity. For $W > 10$, these are within 99% of the maximum possible. Note, however, that it is impossible to attain maximum diversity when $W < 8$ because some of the 8 global routes (*e.g.*, columns A through H) must necessarily map to the same track. Considering this, the *diverse* and *diverse-clique* switch blocks perform very well at being diverse.

7.4.3 Routing Results

The new switch block designs are included in the VPRx router. The *diverse* and *diverse-clique* switch block designs described in Tables B.1 and B.2 have been hard-coded

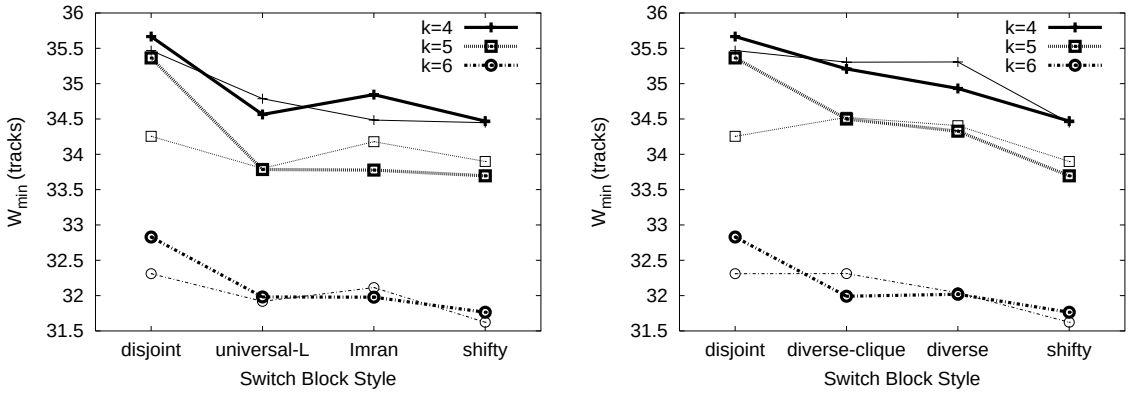


Figure 7.12: Minimum channel width results using the new switch blocks.

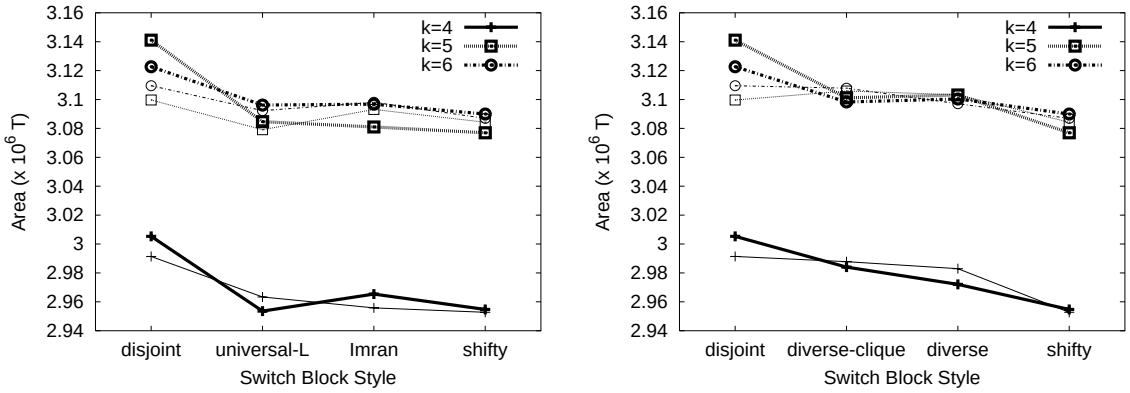


Figure 7.13: Area results using the new switch blocks.

for quick regeneration. In the situations where track groups have $W > 18$, the $W = 18$ switch block design is used instead.[2]

The experimental environment is similar to the one used throughout this dissertation. Routing experiments are performed with three different LUT sizes of $k = 4$, 5, and 6. The number of LUTs per cluster is fixed at $N = 6$, and only length four wires are used in the interconnect. Half of all wiring tracks use pass transistor switches of size 16 times minimum width, and the other half use buffers of size 6. Although not shown, similar results are obtained if all of the wiring tracks use buffers. Experiments are conducted with either one or two layout tiles.

The routability performance of the new switch blocks is presented in Figures 7.12

[2]This rarely occurs with the benchmarks used here.

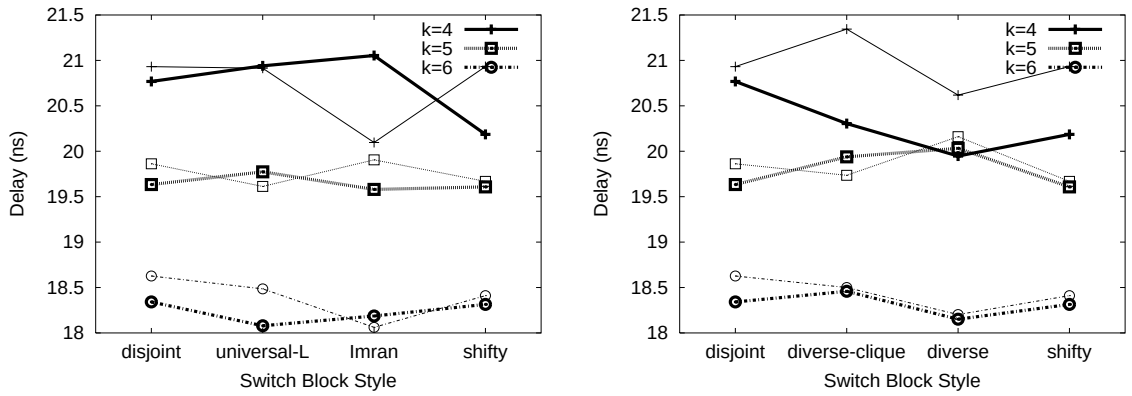


Figure 7.14: Delay results using the new switch blocks.

and 7.13. The former plots the minimum number of tracks required to route, W_{min}, while the latter plots the transistor area of the PLD architecture at the low-stress point of $1.2 \cdot W_{min}$ tracks. The graphs on the left compare *shifty* to the older switch blocks, and the graphs on the right compare *disjoint* to the newer switch blocks. A number of different curves are drawn in the graphs, corresponding to different LUT sizes and whether a plain or checkered layout is used. The **bold** curves show the results with one layout tile, while the thin curves show the results with two checkered layout tiles.

Critical-path delay results for circuits mapped into PLDs using the different switch blocks are presented in Figure 7.14. Unlike the routability results, there does not appear to be any strong connection between delay and switch block style.

7.4.4 Analysis

Overall, the results exhibit only small variations across the different designs, so conclusions might be sensitive to noise and must be carefully drawn. Since each data point is an arithmetic average across the twenty largest MCNC benchmark circuits, large variations should not be expected unless many circuits are affected. To mitigate the influence of noise, it is important to identify trends present in all of the routing results, *e.g.*, across the different LUT sizes, before drawing any conclusions.

One clear trend in the routing results is that the plain *disjoint* switch block performs worse than any perturbation of it (including its own checkered version). Beyond this, the

ranking of specific switch blocks is difficult. It appears that *shifty* is the best, followed closely by *universal-L* and *Imran*, then *disjoint*. The diversity-optimized switch blocks perform better than *disjoint*, but not as well as the *shifty* ad hoc design.

In addition to *shifty*, a variety of other ad hoc switch blocks (both commutative and non-commutative) were explored. The *shifty* design gives better results, but the differences are small. These experiments did not clearly suggest that one particular design is significantly better. The effectiveness of *shifty* demonstrates that the twist or non-commutative features of the *universal-L* and *Imran* blocks is not likely the key factor to their good performance. However, it leaves the following questions open: Why is track shifting effective? Is it because of increased diversity?

To investigate the latter question, the diversity-optimised switch blocks must be examined. While the diverse switch blocks do require fewer routing tracks than the *disjoint* baseline, *shifty* always outperforms them. This lack of improvement does not come from one or two circuits having significantly higher-than-average channel width requirements. Rather, it is a general trend distributed among many of circuits. This suggests that it is not only diversity that makes the *shifty*, *Imran* and *Wilton* switch blocks effective.

Another unexpected result arises when routing $F_s = 2$ variations of the *diverse* and *diverse-clique* switch blocks. The results are not shown here, but are given in Tables A.5, A.6, and A.7 of Appendix A. The anomaly is that several of the *low-stress* routings failed, making it impossible to compute geometric averages. (These are indicated by a dash in the table.) No other $F_s = 2$ switch blocks failed to route their low-stress cases.

Why do the diversity-optimised switch blocks not perform as well as anticipated? One conjecture is that negotiated-congestion type CAD tools, like the VPRx router, might have difficulty with too much diversity. This seems plausible because a local re-routing near the source of a net with a sharing violation would force downstream connections to use a new track. In an architecture with less diversity, it may be easier to avoid creating new routing conflicts by making it easier for the downstream portions of the net to resume using the routing tracks from the previous pass. This difficulty might increase the number of router iterations, but no such increase is seen in these experiments.

7.5 Conclusions

An analytical framework for the design of switch blocks has been presented. It is the first known framework which considers the switch block design problem as part of a larger switching fabric. It also provides a simple way to design using long wire segments. The framework provides an easy way to directly measure and control the amount of diversity in the routing network.

The most fundamental component of this framework is the new switch block model, called the track group model. By separating wiring tracks into independent groups, each one can be considered separately. Using permutation mapping functions to model switch block turns adds a mathematical representation to the framework. With commutative mapping functions and switch blocks, the order in which a net executes turns becomes unimportant and the network is easier to analyse. This framework can design diverse switch blocks, but it is not clear the router is utilising this diversity.

Not all of the components of the framework need to be adopted. For example, the new track group model can be used alone to design the *shifty* and *universal-L* switch blocks. Both of these switch blocks perform slightly better than the basic *disjoint* block.

The entire framework can also be applied to switch block design. An example is given to produce two switch blocks, *diverse* and *diverse-clique*, which maximise routing diversity. These switch blocks require fewer routing tracks than the *disjoint* block, but more tracks than *shifty*. This raises the interesting point that the diverse property may not be significant for creating good routing networks, or it may be difficult to efficiently utilise with existing CAD tools.

The switch block represents a very large proportion of the transistor area in mesh-based PLDs, so it is certainly desirable to make this block as small as possible. The work here has explored many options, but the small area reduction discovered does not provide encouragement that large gains will be found easily. It is likely that a better understanding of the problem and perhaps a more formal approach to switch block design are required to make significant improvements. The work presented in this chapter is only a beginning of this exploration, and some other avenues for future work are suggested in the next section.

7.6 Future Work

The work in this chapter is a first attempt at designing interconnect while examining the global impact of switch locations. A few questions have been raised while examining the results, and these need to be addressed in future work.

There are a number of minor architecture adjustments that can be made to better understand switch block performance. This chapter has only considered mapping functions of the form $f_i(t) = t + a_i$, but more general permutation maps can be considered as well. Track groups may be divided into smaller subgroup sizes, limiting diversity to stay within a few tracks per subgroup. It would be interesting to try increasing the number of switches in the midpoint subblocks. This would give a router a choice of which track assignment to choose when changing directions. For example, one could try the superposition of a *diverse* pattern and a *disjoint* pattern at the midpoint switch block. The area increase would be small, particularly since Chapter 6 has proposed a new routing switch circuit that is more efficient under higher fanin than previous designs.

The proposed track group model forces subsets of tracks to be disjoint. This step is necessary to begin to understand the interaction of the switch placements in the global fabric, but it may not be necessary in a real PLD routing architecture. This impact of the track group model should be better assessed.

In addition to the architecture adjustments, the interaction between the routing algorithm and diverse switch blocks should be examined. Presently, the routing algorithm is oblivious to diversity and this may be degrading performance. Ways of tuning the routing algorithm to better utilise diversity should be explored.

Other methods for designing switch blocks can be developed. This chapter highlights some of the important issues for such a design methodology: that long wire segments are important, that the global fabric must be considered, that the number of switches and their placement must be carefully considered, and that the routing algorithm must be able to efficiently utilise the network.

Chapter 8

Conclusions

Programmable logic devices provide a simple abstraction that permits digital system design to be completed at the RTL level. This abstraction conveniently hides from the designer an increasing number of physical design problems which are addressed at the PLD design stage instead. As a result, it is a promising way to cope with ever-increasing design costs.

The crux of successfully using the PLD abstraction for many future designs relies upon three things. First, the PLD itself must be area efficient and provide overall cost advantages, including design cost plus unit cost. Second, the PLD must meet the timing requirements of the circuits. Third, the physical design of the PLD itself must be feasible. This dissertation has focused on the first of these problems, although it has touched upon some of the preliminary physical design issues as well.

PLD interconnect is the largest contributor to area and unit cost of a PLD. By analysing the design of low-level interconnect components, switch blocks and sparse crossbars, they can be made more efficient. This efficiency is addressed in two ways in this dissertation: by making their implementations smaller, and by improving their ability to create a network which can reach higher levels of utilisation by CAD tools. These components have been studied so that they can be applied in a wide assortment of high-level interconnect styles. For example, although the mesh interconnect model is used in this dissertation, the sparse crossbar design technique can be applied equally to hierarchical networks or to CPLD design. A proper understanding of these low level components is necessary to explore these higher level designs.

The contributions made in this dissertation are summarised below.

First, sparse crossbars are presented as a highly routable alternative to full crossbars which use a fraction of the area. A method for designing routable sparse crossbars is given. This is the first known general method which generates a routable switch placement with an arbitrary number of inputs, outputs, and switches. The routability of a switch placement is measured statistically using a standard Monte Carlo technique and a network flow algorithm. These tools demonstrate that routability is very sensitive to the number of switches in the crossbar within a narrow region; having too few or too many switches is of little use. This sensitivity can be increased and fewer switches are needed if the outputs are intentionally under-utilised. A significant future contribution would involve characterising sparse crossbars so that a general formula expresses the number of switches and spare outputs needed to reach a certain level of routability.

Second, the use of sparse crossbars within clustered logic blocks is shown to be an effective way to save 10–18% in area. To achieve this savings and restore routability loss, a few additional spare cluster inputs are required. Since more area is saved in sparse clusters containing larger lookup table sizes, experiments here show that an architecture with 6-input lookup tables uses less area than one with traditional 4-input lookup tables. This results in a significant 22% area·delay advantage overall. Further investigation with more modern benchmarks and other cluster sizes is necessary to confirm that 6-input lookup tables are indeed better. However, there is sufficient evidence to confirm that the use of sparse crossbars within clusters offers an area advantage.

Third, new buffered routing switch circuits are shown to virtually eliminate delay caused by fanout. This is done by replacing a fanout structure with a fanin structure. With the previous switch design, fanout causes net delay to increase by 100% and critical-path delay by 16%. With one of the new designs, net delay increases by only 9% and critical-path delay increases by only 5%. In general, the new switch designs are also smaller than previously used circuits, leading to a 2% area savings and 7% delay savings overall. Future effort may consider other types of tristate circuits, using only directional buffers, and low-energy issues. Since delay is dominated by the interconnect, this must be considered in switch design as well.

Fourth, replacing some buffered routing switches with pass transistors is shown to reduce area requirements by 8–13% with a delay increase of only 1–2%. This is significant because many modern PLDs now use purely buffered interconnect. The savings is achieved by replacing buffered switches with pass transistors at every other wire endpoint and at every switch along the wire midpoints. An 11–14% improvement to area·delay and a 5% delay improvement is obtained by replacing only the buffered midpoint switches with pass transistors. Future work may improve delay by retaining some fully buffered routing tracks for critical high-fanout nets. As well, the routing tool can be modified to encourage fanout to occur at buffered switches rather than at pass transistor switches.

Fifth, a new framework is given for designing switch blocks with long wire segments. The framework focuses upon a track group model which separates the traditional switch block into sub-blocks. These sub-blocks create the disjoint track groups which subsequently simplify the routing network analysis. Using the framework, it is possible to optimise diversity of the routing network. This is the first known framework which designs switch blocks while simultaneously considering long wire segments and global network interactions. In routing experiments, diversity-optimised switch blocks perform nearly as well as the best-known switch blocks. They also offer a new type of flexibility, the ability to reach different tracks by permuting the global route. Although it is not clear that existing CAD tools can fully utilise this new flexibility, this is an avenue for future exploration. Future work can also explore other uses of a diverse network, such as providing additional paths to avoid faulty wires or switches.

The continuing trend toward designing larger chips requires simpler design methodologies and abstractions. PLD technology presents one simple methodology which allows designers to stop at the RTL level, but it suffers from significant area and delay overheads. This work has studied ways to design and implement these interconnect components so they are more efficient. Most of this design process can be automated and used in tools which generate custom PLDs for future markets such as SoC and embedded applications. Designing custom PLDs is a promising new area which requires significant new research to reach the full potential of providing a simple design methodology.

Appendix A

Switch Blocks with Reduced Flexibility

A.1 Introduction

All switch block styles explored in Chapter 7 have only considered a flexibility of $F_s = 3$ because early work by Rose and Brown [42] determined that $F_s = 2$ requires too many additional wiring tracks and is sometimes unroutable. However, the routing tools used in that study were limited by a separation of the global and detailed routing phases. In contrast, modern routing tools such as VPR are more effective. Considering the large amount of area contained within a switch block, it is important to consider whether switch blocks with fewer switches are a practical way to reduce the overall area cost.

This appendix investigates two $F_s = 2$ styles, *biased* and *asymmetric*, which can be combined with any of the $F_s = 3$ styles presented in this dissertation. Figure A.1 shows several examples of these switch blocks. A style with $F_s = 1$ is also investigated. Since turns at endpoints are not possible with $F_s = 1$, they must be made at midpoints instead.

It should be noted that when styles such as *disjoint*, *universal*, *shifty*, or *diverse* are used with the $F_s = 2$ patterns described here, they lose their defining properties. For example, the $F_s = 2$ *universal* switch block cannot meet all bandwidth constraints if there are only two switches per wire. As a result, the various $F_s = 2$ patterns should be considered as perturbations of $F_s = 3$ styles without any particular defining properties or performance guarantees. It may be possible to develop some $F_s = 2$ styles which preserve defining properties of the $F_s = 3$ switch blocks, but no particular effort was made to find them.

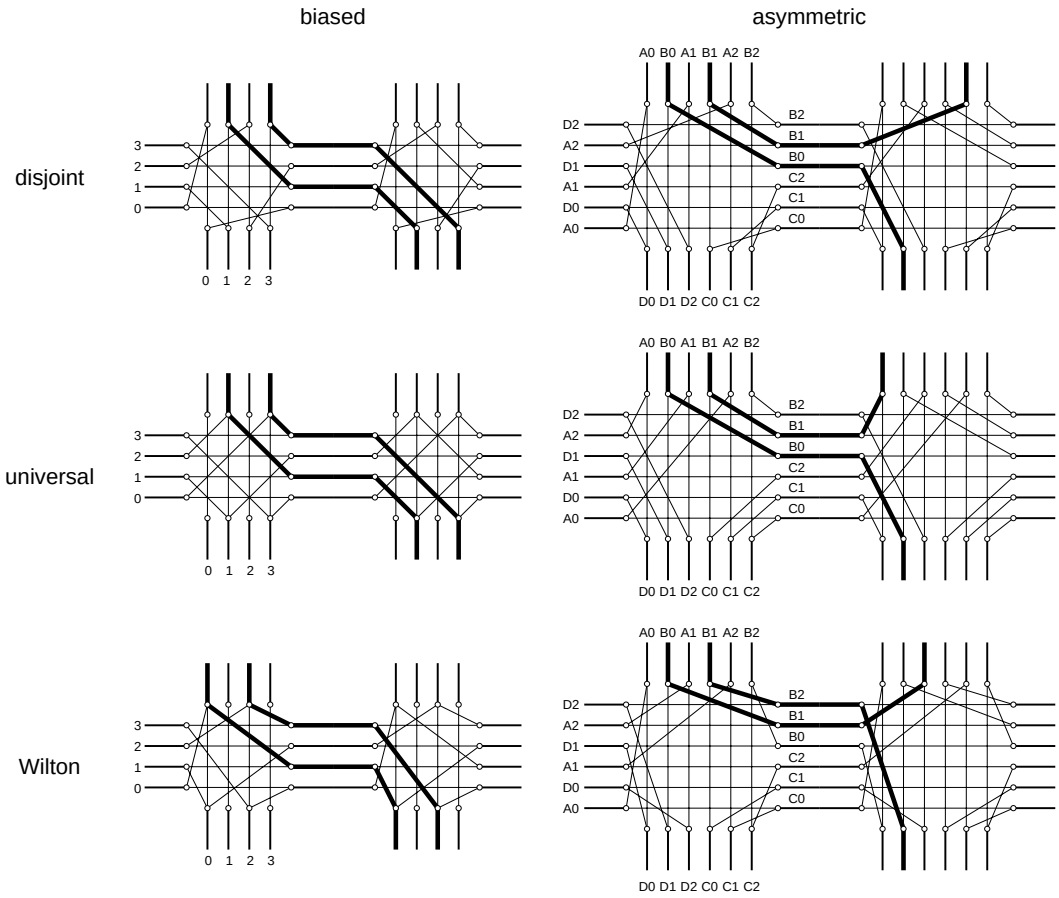


Figure A.1: Biased and asymmetric versions of different switch blocks.

A.2 Biased $F_s = 2$ Style

There are many ways of removing switches from an $F_s = 3$ switch block to create one with $F_s = 2$. One such method involves removal of the switches:

- between any track on top and an *odd* track on the left,
- between any track on top and an *even* track on the right,
- between any track on bottom and an *even* track on the left, and
- between any track on bottom and an *odd* track on the right.

This removal policy creates a *biased* $F_s = 2$ style. The bias is evident from the bold paths in Figure A.1. When combined with styles other than *disjoint*, individual vertical tracks

may connect to one, two, or three wires, but the average flexibility is still two. A *universal* biased $F_s = 2$ switch block exists with two switches on every track, but it is not produced by this method.

The *disjoint* $F_s = 2$ biased topology contains a strong directional bias for each track. For example, odd tracks will have a diagonal bias traveling down and to the right. Non-*disjoint* $F_s = 2$ biased blocks do not have the same degree of bias because some vertical tracks can turn in both directions, while others cannot turn at all. However, each particular horizontal track is biased to turn only up (or only down).

A.3 Asymmetric $F_s = 2$ Style

A general method for generating an asymmetric $F_s = 2$ style is presented below. The asymmetric style solves two problems with the biased style: it guarantees that $F_s = 2$ on all tracks (on all sides), and it does not contain the directional bias.

Rather than starting with an $F_s = 3$ style and removing switches, a direct construction procedure is followed. First divide the tracks into smaller groups, A through D, according to steps 1 through 4 of the algorithm given in Figure A.2. In steps 6 through 8, connections within partitions are formed to establish the permissible turns. These steps treat each of the A through D groups individually and connect them according to the rules for some $F_s = 3$ style. Note that the cardinalities of each group are always even, with exactly half the tracks on one side. Hence, a one-to-one mapping within each group is always possible. Lastly, steps 9 and 10 form the straight connections. This asymmetric algorithm is a generalisation of the asymmetric switch block given by Rose and Brown in [42]. Applying this algorithm with a *disjoint* switch block produces the same topology used in [42].

The advantage of this asymmetric style is its balance. An even track going right may prefer to turn up onto an even track, but it will be able to turn either left or right again depending on whether it is in the lower or upper half of the channel. With the biased style, it would only be able to turn right.

This general method of creating an asymmetric switch block works along with any $F_s = 3$ style, but it destroys the structural properties of the $F_s = 3$ style. For example, it

1. Place the even tracks on the left into A, and the odd tracks into D.
2. Place the even tracks on the top into A, and the odd tracks into B.
3. Place the right-side tracks into C if they are in track $i < \lceil W/2 \rceil$, otherwise they are placed into B.
4. Place the bottom-side tracks into D if they are in track $i < \lfloor W/2 \rfloor$, otherwise they are placed into C.
5. For each side, consecutively renumber the tracks in the same group starting at 0.
6. For group A, reset W to the number of tracks in group A on either side.
7. In group A, connect track t on the left to track $f_{e,1}(t)$ above using the new track numbers.
8. Repeat steps 6 and 7 for groups B, C, and D, using $f_{e,2}(t)$, $f_{e,3}(t)$, and $f_{e,4}(t)$, respectively.
9. Connect track t on the left side to track t on the right side.
10. Connect track t on the top side to track t on the bottom side.

Figure A.2: Asymmetric $F_s = 2$ algorithm.

breaks apart the simple layout elements used in the $F_s = 3$ *disjoint* and *universal* topologies. Hence, these asymmetric switch blocks do not have a simple layout structure. Since these asymmetric switch blocks did not significantly reduce routing area or delay, it did not seem worthwhile to search for other asymmetric styles that would preserve the regular layout structure.

A.4 Results

The results of place and route experiments with $F_s = 1$ are presented in Table A.1. These results are normalised to the baseline *disjoint* $F_s = 3$ architecture. Compared to the baseline, the data shows that $F_s = 1$ reduces total area by 2–3%. This makes it better than all of the other $F_s = 3$ switch block styles, which only save 1–2% in area. However, the $F_s = 1$ switch block has larger increases to delay (3–9% versus -1 to 6% at $W_{min} + 10\%$) and to minimum channel width (10–12% versus -1 to -4%) than the other $F_s = 3$ styles.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at $W_{min}+$			Area·Delay (T·s) at $W_{min}+$		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 4$												
unnormalized				***35.7***		***3.01***	***20.43***			***0.0644***		
disjoint	3	plain	37.35	***1.000***	3.97	***1.000***	***1.000***	1.017	0.975	***1.000***	1.063	1.072
diverse	3	plain	36.60	0.979	3.93	0.989	1.001	0.976	0.975	0.992	1.010	1.059
diverse-clique	3	plain	36.85	0.987	3.95	0.993	1.057	0.994	0.987	1.048	1.032	1.075
imran	3	plain	36.40	0.977	3.92	0.987	1.033	1.031	0.979	1.019	1.063	1.061
shifty	3	plain	36.05	0.966	3.90	0.983	0.986	0.988	0.977	0.969	1.014	1.054
universal	3	plain	36.10	0.969	3.91	0.983	1.036	1.025	0.980	1.018	1.053	1.058
disjoint	1	plain	42.35	1.123	3.88	0.971	1.030	1.040	1.009	0.997	1.055	1.070
LUT size $k = 5$												
unnormalized				***35.4***		***3.14***	***19.62***			***0.0642***		
disjoint	3	plain	36.95	***1.000***	4.15	***1.000***	***1.000***	1.001	0.982	***1.000***	1.041	1.063
diverse	3	plain	35.95	0.971	4.11	0.988	1.008	1.021	0.971	0.995	1.045	1.036
diverse-clique	3	plain	36.05	0.976	4.10	0.987	1.019	1.016	0.998	1.006	1.043	1.067
imran	3	plain	35.40	0.955	4.08	0.981	1.013	0.998	0.989	0.992	1.016	1.046
shifty	3	plain	35.30	0.953	4.07	0.980	1.006	0.999	0.996	0.983	1.016	1.052
universal	3	plain	35.40	0.955	4.09	0.982	0.998	1.008	0.977	0.977	1.024	1.035
disjoint	1	plain	40.70	1.095	4.01	0.967	1.091	1.035	1.022	1.054	1.035	1.063
LUT size $k = 6$												
unnormalized				***32.8***		***3.12***	***18.40***			***0.0594***		
disjoint	3	plain	34.60	***1.000***	4.26	***1.000***	***1.000***	0.997	0.988	***1.000***	1.027	1.049
diverse	3	plain	33.80	0.975	4.22	0.993	1.016	0.986	0.990	1.007	1.007	1.043
diverse-clique	3	plain	33.85	0.974	4.23	0.992	1.008	1.003	0.991	1.000	1.025	1.043
imran	3	plain	33.65	0.974	4.21	0.992	1.005	0.988	0.987	0.996	1.008	1.039
shifty	3	plain	33.40	0.968	4.21	0.990	0.995	0.995	0.976	0.983	1.012	1.025
universal	3	plain	33.70	0.974	4.22	0.992	1.014	0.982	0.984	1.003	1.002	1.036
disjoint	1	plain	38.80	1.114	4.20	0.984	1.054	1.036	1.013	1.035	1.046	1.055

Table A.1: Performance of $F_s = 1$ switch blocks.

Although $F_s = 1$ is the most effective technique for reducing area, the large increase in channel width may make it impractical.

The results of $F_s = 2$ experiments are presented in Tables A.2, A.3, and A.4. These data are normalised to the same $F_s = 3$ baseline architecture. These same $F_s = 2$ results are repeated in Tables A.5, A.6, and A.7 where they are normalised to the *disjoint* biased $F_s = 2$ switch block. This latter set of tables show that the *disjoint* biased $F_s = 2$ switch block has the highest W_{min} and area of all $F_s = 2$ styles (with one exception, noted below). Hence, it is useful as a high water mark in comparing with the $F_s = 3$ and $F_s = 1$ styles.

There is one anomaly in the $F_s = 2$ data. The asymmetric versions of the *diverse* and

diverse-clique switch blocks often had difficulty completing the *low-stress* route for one or more circuits. When this occurs, delay and area·delay averages cannot be computed. These cases are shown with a long dash ('—') in the tables. These cases are also the only $F_s = 2$ styles which have larger area numbers than the high water mark.

Nearly all of the architectures with $F_s < 3$ require up to 6% more routing tracks than the $F_s = 3$ baseline. In comparison, all of the $F_s = 3$ variations require 1–5% fewer routing tracks than the baseline. Hence, $F_s < 3$ is not an effective way to reduce track count. Instead, it is always better to keep $F_s = 3$ and use a non-*disjoint* switch block such as *shifty*.

Although more tracks were used, the reduced F_s architectures (except the anomaly cases) always require less area than the best $F_s = 3$ result. In general, the $F_s = 3$ topologies save 0.7–2.0% compared to the baseline, but the $F_s = 2$ topologies save 2.1–3.9% and $F_s = 1$ saves 1.6–2.9%. Hence, $F_s = 2$ is the best way to save area.

The delay results are normalised against the $W_{min} + 10\%$ low-stress channel width so that the effect of increasing the channel width for each architecture can be examined as well as the topological influences. At $W_{min} + 10\%$, nearly all topologies perform more slowly than the baseline. The typical increase is in the 1–3% range. The largest increase is 11.5% in the $k = 4$, *Imran*, biased $F_s = 2$ architecture. This case seems to be an anomaly because an increase of that magnitude is not present at $k = 5$ or $k = 6$. The cause may be tool-related: three small circuits had large delay increases (up to 70%), but a simple re-route with a more gradual sharing-penalty cost fully mitigates the problem for all three cases. The delay increases at $F_s = 1$ are caused by similar problems, but re-routing is not as effective — it is certainly a slower architecture.

When the channel width is increased to $W_{min} + 20\%$ there is a small improvement to delay for $k = 5$ and $k = 6$, but not for $k = 4$. At $W_{min} + 30\%$ all architectures tend to achieve their lowest delays and, except for $F_s = 1$, are faster than the baseline.

There is one other interesting delay trend evident from the channel width sweep data. The best low-stress delay requires more tracks with the 4-LUT (closer to $W_{min} + 30\%$) than the 5-LUT and 6-LUT architectures (closer to $W_{min} + 20\%$). Since different architectures are usually evaluated at the same low-stress point (often $W_{min} + 20\%$), the 4-LUT is placed

at a slight disadvantage for delay. In contrast, delay in $k = 5$ and $k = 6$ architectures is lower and less sensitive to excess channel capacity. Both of these are desirable characteristics.

Other than the anomaly found with the asymmetric *diverse* and *diverse-clique* area results, a comparison between the biased and asymmetric styles does not clearly indicate a superior style. This is unexpected because the biased style seems like it should have a natural handicap.

In contrast, the $F_s = 1$ delay results do not improve much as the channel width is increased. In fact, they always remain slower than the baseline while other topologies typically become faster than the baseline. The poor delay result is expected because turns can only be made at midpoints. This shortens the effective wire length for all connections that require a turn. Although $F_s = 1$ switch blocks require less area, there is a significant delay penalty from removing all endpoint turns.

Comparing the area·delay product results, it can be seen that the high area penalty but low delay improvement from increasing the channel width favours architecture design with only $W_{min} + 10\%$ routing tracks. As well, due to the small variations in area and delay, the choice of the "best" area·delay architecture nearly always changes depending on evaluation point (10%, 20%, or 30% more tracks). In general, however, $F_s = 2$ architectures are favoured since the area savings is larger than the delay increase.

Overall, reduced F_s architectures are feasible alternatives to $F_s = 3$ architectures. This is a new result, since earlier work had shown $F_s < 3$ to be completely impractical. Unfortunately, these architectures do not result in significant area savings, and generally they tend to cause an increase in delay. Unless the area of a routing switch rises dramatically in the future, it is unlikely these architectures will be considered useful.

A.5 Summary

The work in Chapter 7 focuses on different $F_s = 3$ topologies to determine whether switch placement is a more effective way to save area than removing switches.

The work here the first known re-examination of $F_s < 3$ styles using modern routing tools. Unlike before, these new results show that $F_s = 2$ and even $F_s = 1$ topologies are

routable. Furthermore, although they require a up to 4% more tracks, both $F_s = 2$ styles tend to require 3% less transistor area and increase delay by less than 4% compared with the $F_s = 3$ *disjoint* baseline architecture. Although the $F_s = 1$ architecture also reduces transistor area by 3%, it is inferior to $F_s = 2$ because it produces larger increases in minimum channel width and delay.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at $W_{min}+$			Area·Delay (T·s) at $W_{min}+$		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 4$												
		unnormalized		***35.7***		***3.01***	***20.43***			***0.0644***		
disjoint	3	plain	37.35	***1.000***	3.97	***1.000***	***1.000***	1.017	0.975	***1.000***	1.063	1.072
diverse	3	plain	36.60	0.979	3.93	0.989	1.001	0.976	0.975	0.992	1.010	1.059
diverse-clique	3	plain	36.85	0.987	3.95	0.993	1.057	0.994	0.987	1.048	1.032	1.075
imran	3	plain	36.40	0.977	3.92	0.987	1.033	1.031	0.979	1.019	1.063	1.061
shifty	3	plain	36.05	0.966	3.90	0.983	0.986	0.988	0.977	0.969	1.014	1.054
universal	3	plain	36.10	0.969	3.91	0.983	1.036	1.025	0.980	1.018	1.053	1.058
disjoint	2	biased	38.95	1.041	3.89	0.979	1.009	1.036	0.982	0.986	1.058	1.054
diverse	2	biased	37.55	1.005	3.81	0.962	1.025	1.004	0.977	0.983	1.007	1.025
diverse-clique	2	biased	37.85	1.014	3.84	0.966	1.044	1.007	0.976	1.008	1.014	1.029
imran	2	biased	37.80	1.011	3.84	0.964	1.115	1.044	0.975	1.072	1.050	1.028
shifty	2	biased	37.85	1.015	3.82	0.964	1.038	1.022	0.974	1.000	1.032	1.029
universal	2	biased	38.20	1.021	3.84	0.970	1.020	1.022	0.972	0.988	1.035	1.032
disjoint	2	biased, checker	38.00	1.016	3.83	0.967	1.012	1.008	0.967	0.977	1.017	1.022
diverse	2	biased, checker	37.80	1.010	3.83	0.964	1.015	0.999	0.977	0.975	1.004	1.029
diverse-clique	2	biased, checker	37.90	1.014	3.83	0.966	1.025	1.046	0.985	0.988	1.054	1.040
imran	2	biased, checker	38.00	1.016	3.85	0.967	1.035	1.018	0.989	0.999	1.027	1.044
shifty	2	biased, checker	37.85	1.013	3.83	0.965	1.056	1.005	0.971	1.015	1.012	1.025
universal	2	biased, checker	37.75	1.006	3.83	0.963	1.023	1.013	0.982	0.982	1.016	1.033
disjoint	2	asym.	37.90	1.012	3.85	0.966	1.018	1.033	0.972	0.982	1.040	1.028
diverse	2	asym.	38.25	1.038	4.00	1.013	—	—	—	—	—	—
diverse-clique	2	asym.	38.25	1.038	4.00	1.013	—	—	—	—	—	—
imran	2	asym.	37.90	1.013	3.83	0.967	1.026	1.036	0.986	0.989	1.045	1.042
shifty	2	asym.	38.00	1.016	3.84	0.967	1.042	1.046	0.975	1.004	1.054	1.029
universal	2	asym.	38.05	1.018	3.84	0.967	1.034	0.998	0.975	0.998	1.008	1.033
disjoint	2	asym., checker	38.25	1.021	3.86	0.970	1.021	1.012	0.980	0.988	1.023	1.039
diverse	2	asym., checker	38.10	1.033	4.00	1.014	—	1.004	—	—	1.069	—
diverse-clique	2	asym., checker	38.10	1.034	3.99	1.009	—	0.994	—	—	1.054	—
imran	2	asym., checker	38.10	1.017	3.84	0.967	1.005	1.027	0.982	0.971	1.036	1.039
shifty	2	asym., checker	37.55	1.005	3.81	0.961	1.024	1.008	0.975	0.982	1.012	1.023
universal	2	asym., checker	37.90	1.014	3.84	0.966	1.055	1.046	0.992	1.016	1.052	1.047
disjoint	1	plain	42.35	1.123	3.88	0.971	1.030	1.040	1.009	0.997	1.055	1.070

Table A.2: Performance of $F_s = 2$ switch blocks for $k = 4$, normalised to $F_s = 3$.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at W_{min}+			Area·Delay (T·s) at W_{min}+		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 5$												
		unnormalized		***35.4***		***3.14***	***19.62***			***0.0642***		
disjoint	3	plain	36.95	***1.000***	4.15	***1.000***	***1.000***	1.001	0.982	***1.000***	1.041	1.063
diverse	3	plain	35.95	0.971	4.11	0.988	1.008	1.021	0.971	0.995	1.045	1.036
diverse-clique	3	plain	36.05	0.976	4.10	0.987	1.019	1.016	0.998	1.006	1.043	1.067
imran	3	plain	35.40	0.955	4.08	0.981	1.013	0.998	0.989	0.992	1.016	1.046
shifty	3	plain	35.30	0.953	4.07	0.980	1.006	0.999	0.996	0.983	1.016	1.052
universal	3	plain	35.40	0.955	4.09	0.982	0.998	1.008	0.977	0.977	1.024	1.035
disjoint	2	biased	37.65	1.013	4.05	0.972	1.004	1.010	0.967	0.974	1.018	1.015
diverse	2	biased	37.20	1.003	4.02	0.968	1.021	1.005	0.972	0.986	1.009	1.014
diverse-clique	2	biased	36.50	0.985	3.99	0.961	1.024	1.008	0.978	0.983	1.002	1.012
imran	2	biased	37.20	1.004	4.02	0.969	1.038	1.001	0.981	1.004	1.005	1.023
shifty	2	biased	37.15	1.002	4.02	0.967	1.011	1.009	0.987	0.976	1.013	1.029
universal	2	biased	37.45	1.010	4.04	0.970	1.006	1.016	0.982	0.974	1.023	1.027
disjoint	2	biased, checker	37.05	1.000	4.02	0.967	1.015	0.995	0.979	0.980	0.997	1.022
diverse	2	biased, checker	37.25	1.003	4.02	0.969	1.022	1.011	0.984	0.987	1.013	1.027
diverse-clique	2	biased, checker	36.95	0.996	4.01	0.964	0.998	0.998	0.975	0.962	0.997	1.015
imran	2	biased, checker	37.35	1.007	4.03	0.969	1.018	1.005	0.973	0.984	1.012	1.018
shifty	2	biased, checker	37.15	1.001	4.02	0.967	1.009	1.004	0.977	0.974	1.005	1.020
universal	2	biased, checker	37.15	1.004	4.01	0.969	1.007	1.008	0.980	0.975	1.010	1.025
disjoint	2	asym.	37.20	1.003	4.02	0.969	1.034	1.003	0.978	0.999	1.005	1.023
diverse	2	asym.	37.75	1.036	4.16	1.009	—	0.996	0.998	—	1.048	1.094
diverse-clique	2	asym.	37.75	1.036	4.16	1.009	—	0.996	0.998	—	1.048	1.094
imran	2	asym.	37.40	1.009	4.04	0.972	1.015	1.019	0.983	0.984	1.025	1.030
shifty	2	asym.	36.95	0.997	4.01	0.967	1.010	1.009	0.977	0.973	1.010	1.016
universal	2	asym.	37.20	1.002	4.04	0.969	1.028	1.002	0.975	0.994	1.005	1.018
disjoint	2	asym., checker	36.85	0.993	4.01	0.963	1.030	1.009	0.980	0.991	1.008	1.018
diverse	2	asym., checker	37.70	1.035	4.17	1.011	1.044	—	1.007	1.056	—	1.108
diverse-clique	2	asym., checker	37.60	1.028	4.16	1.005	1.024	—	0.984	1.032	—	1.073
imran	2	asym., checker	37.05	1.000	4.01	0.967	1.033	1.001	0.977	0.996	1.003	1.019
shifty	2	asym., checker	37.00	0.999	4.01	0.966	1.011	0.989	0.978	0.976	0.991	1.020
universal	2	asym., checker	37.10	1.001	4.02	0.968	1.037	1.021	0.980	1.000	1.023	1.024
disjoint	1	plain	40.70	1.095	4.01	0.967	1.091	1.035	1.022	1.054	1.035	1.063

Table A.3: Performance of $F_s = 2$ switch blocks for $k = 5$, normalised to $F_s = 3$.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at $W_{min}+$			Area·Delay (T·s) at $W_{min}+$		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 6$												
		unnormalized		***32.8***		***3.12***	***18.40***			***0.0594***		
disjoint	3	plain	34.60	***1.000***	4.26	***1.000***	***1.000***	0.997	0.988	***1.000***	1.027	1.049
diverse	3	plain	33.80	0.975	4.22	0.993	1.016	0.986	0.990	1.007	1.007	1.043
diverse-clique	3	plain	33.85	0.974	4.23	0.992	1.008	1.003	0.991	1.000	1.025	1.043
imran	3	plain	33.65	0.974	4.21	0.992	1.005	0.988	0.987	0.996	1.008	1.039
shifty	3	plain	33.40	0.968	4.21	0.990	0.995	0.995	0.976	0.983	1.012	1.025
universal	3	plain	33.70	0.974	4.22	0.992	1.014	0.982	0.984	1.003	1.002	1.036
disjoint	2	biased	35.45	1.023	4.19	0.985	1.022	1.001	0.978	1.003	1.011	1.020
diverse	2	biased	35.00	1.011	4.16	0.980	1.024	1.006	0.969	1.001	1.012	1.006
diverse-clique	2	biased	34.95	1.009	4.17	0.980	1.047	1.014	0.981	1.022	1.019	1.017
imran	2	biased	35.15	1.013	4.17	0.982	1.029	1.017	0.976	1.007	1.024	1.014
shifty	2	biased	35.15	1.018	4.17	0.982	1.029	1.012	0.969	1.008	1.020	1.009
universal	2	biased	35.40	1.021	4.18	0.983	1.024	1.019	0.973	1.004	1.029	1.014
disjoint	2	biased, checker	35.10	1.013	4.17	0.982	1.047	1.003	0.976	1.025	1.009	1.015
diverse	2	biased, checker	35.25	1.017	4.17	0.983	0.999	0.998	0.973	0.979	1.006	1.012
diverse-clique	2	biased, checker	35.00	1.010	4.16	0.980	1.018	1.004	0.974	0.995	1.009	1.011
imran	2	biased, checker	35.15	1.012	4.17	0.981	0.992	1.018	0.979	0.971	1.024	1.017
shifty	2	biased, checker	35.15	1.015	4.16	0.982	1.003	1.005	0.970	0.982	1.013	1.008
universal	2	biased, checker	34.95	1.009	4.16	0.980	0.998	1.009	0.985	0.976	1.014	1.022
disjoint	2	asym.	35.30	1.016	4.19	0.982	1.018	1.000	0.979	0.998	1.008	1.019
diverse	2	asym.	35.95	1.055	4.29	1.013	1.020	0.999	0.995	1.035	1.044	1.075
diverse-clique	2	asym.	35.95	1.055	4.29	1.013	1.020	0.999	0.995	1.035	1.044	1.075
imran	2	asym.	35.25	1.014	4.18	0.982	1.009	1.002	0.971	0.988	1.010	1.009
shifty	2	asym.	34.95	1.010	4.17	0.981	1.051	1.017	0.983	1.028	1.023	1.018
universal	2	asym.	35.10	1.011	4.17	0.981	1.007	1.006	0.974	0.985	1.013	1.011
disjoint	2	asym., checker	35.10	1.011	4.18	0.981	1.002	1.003	0.977	0.981	1.009	1.015
diverse	2	asym., checker	36.00	1.057	4.29	1.016	1.014	0.997	—	1.031	1.044	—
diverse-clique	2	asym., checker	35.95	1.054	4.30	1.012	1.008	0.995	—	1.021	1.040	—
imran	2	asym., checker	35.15	1.016	4.17	0.982	1.014	1.010	0.979	0.994	1.019	1.017
shifty	2	asym., checker	35.30	1.020	4.18	0.983	1.020	1.008	0.977	1.000	1.016	1.018
universal	2	asym., checker	35.25	1.015	4.18	0.982	1.007	1.010	0.972	0.986	1.018	1.011
disjoint	1	plain	38.80	1.114	4.20	0.984	1.054	1.036	1.013	1.035	1.046	1.055

Table A.4: Performance of $F_s = 2$ switch blocks for $k = 6$, normalised to $F_s = 3$.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at W_{min}+			Area·Delay (T·s) at W_{min}+		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 4$												
		unnormalized		***37.1***		***2.94***	***20.61***			***0.0635***		
disjoint	2	biased	38.95	***1.000***	3.89	***1.000***	***1.000***	1.026	0.973	***1.000***	1.073	1.069
diverse	2	biased	37.55	0.966	3.81	0.983	1.016	0.995	0.969	0.998	1.022	1.040
diverse-clique	2	biased	37.85	0.974	3.84	0.987	1.035	0.998	0.967	1.022	1.029	1.044
imran	2	biased	37.80	0.972	3.84	0.985	1.105	1.035	0.966	1.088	1.065	1.043
shifty	2	biased	37.85	0.976	3.82	0.985	1.029	1.013	0.965	1.014	1.047	1.044
universal	2	biased	38.20	0.982	3.84	0.991	1.011	1.012	0.963	1.002	1.050	1.047
disjoint	2	biased, checker	38.00	0.976	3.83	0.987	1.003	0.999	0.959	0.991	1.031	1.037
diverse	2	biased, checker	37.80	0.971	3.83	0.985	1.005	0.990	0.968	0.989	1.019	1.044
diverse-clique	2	biased, checker	37.90	0.975	3.83	0.987	1.016	1.036	0.976	1.002	1.069	1.055
imran	2	biased, checker	38.00	0.977	3.85	0.988	1.026	1.009	0.980	1.013	1.041	1.059
shifty	2	biased, checker	37.85	0.973	3.83	0.985	1.046	0.996	0.962	1.030	1.027	1.040
universal	2	biased, checker	37.75	0.967	3.83	0.984	1.013	1.004	0.974	0.996	1.031	1.048
disjoint	2	asym.	37.90	0.972	3.85	0.987	1.008	1.024	0.963	0.996	1.055	1.043
diverse	2	asym.	38.25	0.998	4.00	1.035	—	—	—	—	—	—
diverse-clique	2	asym.	38.25	0.998	4.00	1.035	—	—	—	—	—	—
imran	2	asym.	37.90	0.973	3.83	0.988	1.016	1.027	0.977	1.003	1.060	1.057
shifty	2	asym.	38.00	0.977	3.84	0.987	1.033	1.037	0.966	1.019	1.069	1.044
universal	2	asym.	38.05	0.979	3.84	0.988	1.025	0.989	0.966	1.013	1.023	1.048
disjoint	2	asym., checker	38.25	0.981	3.86	0.991	1.012	1.003	0.971	1.003	1.038	1.054
diverse	2	asym., checker	38.10	0.993	4.00	1.036	—	0.995	—	—	1.084	—
diverse-clique	2	asym., checker	38.10	0.994	3.99	1.031	—	0.985	—	—	1.070	—
imran	2	asym., checker	38.10	0.977	3.84	0.988	0.996	1.018	0.974	0.985	1.051	1.054
shifty	2	asym., checker	37.55	0.966	3.81	0.982	1.015	0.999	0.966	0.996	1.026	1.038
universal	2	asym., checker	37.90	0.975	3.84	0.987	1.045	1.036	0.983	1.031	1.068	1.062

Table A.5: Performance of $F_s = 2$ switch blocks for $k = 4$.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at $W_{min}+$			Area·Delay (T·s) at $W_{min}+$		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 5$												
		unnormalized		***35.8***		***3.05***	***19.70***			***0.0625***		
disjoint	2	biased	37.65	***1.000***	4.05	***1.000***	***1.000***	1.006	0.963	***1.000***	1.045	1.043
diverse	2	biased	37.20	0.991	4.02	0.996	1.017	1.001	0.969	1.012	1.036	1.042
diverse-clique	2	biased	36.50	0.973	3.99	0.989	1.021	1.005	0.975	1.009	1.029	1.040
imran	2	biased	37.20	0.991	4.02	0.997	1.034	0.997	0.977	1.031	1.033	1.051
shifty	2	biased	37.15	0.989	4.02	0.995	1.007	1.005	0.983	1.003	1.040	1.057
universal	2	biased	37.45	0.997	4.04	0.998	1.002	1.012	0.978	1.001	1.051	1.055
disjoint	2	biased, checker	37.05	0.988	4.02	0.995	1.012	0.991	0.975	1.007	1.024	1.050
diverse	2	biased, checker	37.25	0.991	4.02	0.997	1.018	1.007	0.980	1.014	1.041	1.055
diverse-clique	2	biased, checker	36.95	0.984	4.01	0.992	0.994	0.994	0.971	0.989	1.024	1.042
imran	2	biased, checker	37.35	0.994	4.03	0.997	1.014	1.001	0.970	1.011	1.039	1.046
shifty	2	biased, checker	37.15	0.989	4.02	0.995	1.005	1.000	0.974	1.001	1.033	1.048
universal	2	biased, checker	37.15	0.991	4.01	0.996	1.003	1.004	0.977	1.001	1.037	1.053
disjoint	2	asym.	37.20	0.990	4.02	0.997	1.030	0.999	0.974	1.026	1.033	1.051
diverse	2	asym.	37.75	1.023	4.16	1.038	—	0.992	0.994	—	1.077	1.124
diverse-clique	2	asym.	37.75	1.023	4.16	1.038	—	0.992	0.994	—	1.077	1.124
imran	2	asym.	37.40	0.996	4.04	0.999	1.012	1.015	0.979	1.011	1.053	1.058
shifty	2	asym.	36.95	0.984	4.01	0.995	1.006	1.005	0.973	0.999	1.037	1.044
universal	2	asym.	37.20	0.989	4.04	0.997	1.024	0.999	0.971	1.021	1.032	1.045
disjoint	2	asym., checker	36.85	0.981	4.01	0.991	1.026	1.005	0.976	1.018	1.035	1.045
diverse	2	asym., checker	37.70	1.022	4.17	1.040	1.040	—	1.003	1.085	—	1.138
diverse-clique	2	asym., checker	37.60	1.016	4.16	1.034	1.020	—	0.980	1.060	—	1.102
imran	2	asym., checker	37.05	0.987	4.01	0.995	1.029	0.997	0.973	1.023	1.030	1.047
shifty	2	asym., checker	37.00	0.987	4.01	0.994	1.007	0.985	0.974	1.003	1.018	1.047
universal	2	asym., checker	37.10	0.989	4.02	0.996	1.033	1.017	0.977	1.028	1.051	1.052

Table A.6: Performance of $F_s = 2$ switch blocks for $k = 5$.

Type	F_s	Style	W_{min} (tracks)		Area ($\times 10^6$ T)		Delay (ns) at $W_{min}+$			Area·Delay (T·s) at $W_{min}+$		
			Arith.	Geom.	Arith.	Geom.	10%	20%	30%	10%	20%	30%
LUT size $k = 6$												
		unnormalized		***33.6***		***3.08***	***18.81***			***0.0595***		
disjoint	2	biased	35.45	***1.000***	4.19	***1.000***	***1.000***	0.980	0.957	***1.000***	1.008	1.017
diverse	2	biased	35.00	0.989	4.16	0.996	1.002	0.985	0.949	0.998	1.009	1.003
diverse-clique	2	biased	34.95	0.986	4.17	0.995	1.024	0.992	0.960	1.019	1.015	1.014
imran	2	biased	35.15	0.990	4.17	0.997	1.007	0.996	0.955	1.004	1.021	1.011
shifty	2	biased	35.15	0.995	4.17	0.997	1.007	0.991	0.949	1.005	1.017	1.005
universal	2	biased	35.40	0.998	4.18	0.998	1.002	0.997	0.952	1.001	1.026	1.011
disjoint	2	biased, checker	35.10	0.990	4.17	0.997	1.025	0.981	0.955	1.022	1.006	1.012
diverse	2	biased, checker	35.25	0.994	4.17	0.998	0.978	0.977	0.952	0.976	1.003	1.008
diverse-clique	2	biased, checker	35.00	0.988	4.16	0.995	0.996	0.983	0.954	0.992	1.006	1.008
imran	2	biased, checker	35.15	0.990	4.17	0.996	0.970	0.996	0.958	0.968	1.021	1.014
shifty	2	biased, checker	35.15	0.992	4.16	0.997	0.981	0.984	0.949	0.979	1.010	1.005
universal	2	biased, checker	34.95	0.987	4.16	0.995	0.977	0.987	0.964	0.973	1.011	1.019
disjoint	2	asym.	35.30	0.993	4.19	0.997	0.996	0.979	0.958	0.995	1.005	1.015
diverse	2	asym.	35.95	1.032	4.29	1.029	0.998	0.978	0.974	1.032	1.041	1.071
diverse-clique	2	asym.	35.95	1.032	4.29	1.029	0.998	0.978	0.974	1.032	1.041	1.071
imran	2	asym.	35.25	0.991	4.18	0.997	0.988	0.981	0.950	0.985	1.007	1.006
shifty	2	asym.	34.95	0.987	4.17	0.996	1.029	0.995	0.962	1.025	1.020	1.015
universal	2	asym.	35.10	0.989	4.17	0.996	0.986	0.984	0.953	0.982	1.010	1.008
disjoint	2	asym., checker	35.10	0.988	4.18	0.996	0.980	0.982	0.956	0.978	1.006	1.012
diverse	2	asym., checker	36.00	1.034	4.29	1.032	0.993	0.976	—	1.028	1.041	—
diverse-clique	2	asym., checker	35.95	1.030	4.30	1.028	0.986	0.974	—	1.018	1.037	—
imran	2	asym., checker	35.15	0.994	4.17	0.997	0.992	0.989	0.958	0.991	1.016	1.014
shifty	2	asym., checker	35.30	0.997	4.18	0.998	0.998	0.986	0.956	0.997	1.012	1.015
universal	2	asym., checker	35.25	0.992	4.18	0.997	0.986	0.989	0.951	0.983	1.015	1.008

Table A.7: Performance of $F_s = 2$ switch blocks for $k = 6$.

Appendix B

Diverse Switch Block Design Instances

This appendix lists the precise switch block designs computed for length-4 wires. Table B.1 and B.2 show the solutions for *diverse* and *diverse-clique* switch block styles, respectively.

The tables can be read as follows. The solution set $\mathbf{x}_W$ found for a particular value of W is given in each column. Each row represents one turn type. The constant given at a particular row and column is used in that turn map for that value of W. For example, $a_{m,1}$ is the constant for a turn at the first midpoint block. At $W = 14$, the turn map is given by

$$f_{m,1}(t) = t + a_{m,1} = t + 3.$$

A clique structure can be verified in the *diverse-clique* results by checking that $a_{e,1} = a_{e,4}$, $a_{e,2} = a_{e,3}$, and $a_{e,1} + a_{e,2} = W$. Although no clique constraints are placed during the design of the *diverse* switch blocks, they are found in many of the solution sets. In particular, there are cliques in f_e for $W = \{2,3,4,5,7,9,10,11,12,16\}$ and in g_e for $W = \{3,4,6,7,8,10,11,12,13,14\}$. This means that many *diverse* switch blocks can also be constructed using simple layout structures.

Solution	**Track Group Channel Width,** $W =$																
$\mathbf{x}_W$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$a_{e,1}$	1	0	2	0	0	0	7	0	0	0	0	12	6	9	5	10	13
$a_{e,2}$	1	0	2	0	3	0	0	0	0	0	0	6	11	6	11	7	6
$a_{e,3}$	1	0	2	0	3	0	0	0	0	0	0	1	8	6	11	0	5
$a_{e,4}$	1	0	2	0	0	0	0	0	0	0	0	12	13	11	5	14	5
$a_{m,1}$	0	1	1	0	5	5	0	0	7	5	7	10	3	4	9	11	13
$a_{m,2}$	0	0	1	4	0	2	2	5	8	7	9	2	3	0	11	15	12
$a_{m,3}$	0	1	2	4	2	5	6	8	2	10	2	11	9	6	4	14	8
$b_{e,1}$	1	0	2	2	0	0	0	2	0	0	0	6	0	2	8	0	16
$b_{e,2}$	0	0	2	0	0	0	0	0	0	0	0	7	0	13	8	2	2
$b_{e,3}$	0	0	2	0	0	0	0	7	0	0	0	7	0	13	8	0	1
$b_{e,4}$	0	0	2	2	0	0	0	0	0	0	0	6	0	1	2	15	17
$b_{m,1}$	1	0	0	3	3	1	3	4	6	8	4	0	13	0	15	15	11
$b_{m,2}$	0	2	3	0	1	1	1	2	1	1	1	0	1	1	15	6	13
$b_{m,3}$	0	2	1	3	1	0	0	1	0	0	0	2	4	8	13	8	1

Table B.1: *Diverse* switch block solution sets.

Solution	**Track Group Channel Width,** $W =$																
$\mathbf{x}_W$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$a_{e,1}$	1	0	2	0	3	0	0	0	5	0	0	0	0	0	8	0	17
$a_{e,2}$	1	0	2	0	3	0	0	0	5	0	0	0	0	0	8	0	1
$a_{e,3}$	1	0	2	0	3	0	0	0	5	0	0	0	0	0	8	0	1
$a_{e,4}$	1	0	2	0	3	0	0	0	5	0	0	0	0	0	8	0	17
$a_{m,1}$	0	2	1	2	2	5	5	4	9	5	7	11	3	7	3	10	12
$a_{m,2}$	0	1	0	3	2	2	6	7	2	7	9	8	5	12	13	12	4
$a_{m,3}$	0	2	1	2	0	5	1	7	4	10	2	9	6	13	6	4	5
$b_{e,1}$	0	0	0	0	0	0	0	2	5	0	0	5	9	3	3	3	3
$b_{e,2}$	0	0	0	0	0	0	0	7	5	0	0	8	5	12	13	14	15
$b_{e,3}$	0	0	0	0	0	0	0	7	5	0	0	8	5	12	13	14	15
$b_{e,4}$	0	0	0	0	0	0	0	2	5	0	0	5	9	3	3	3	3
$b_{m,1}$	1	1	2	1	4	1	2	6	2	8	4	1	1	1	1	1	1
$b_{m,2}$	0	0	0	1	0	1	0	2	1	1	1	1	1	1	1	1	1
$b_{m,3}$	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0

Table B.2: *Diverse-clique* switch block solution sets.

Bibliography

[1] Intel Corporation, "Microprocessor quick reference guide," October 30, 2002. http://www.intel.com/pressroom/kits/quickreffam.htm.

[2] ATI Technologies Inc., "ATI launches RADEON 9700, establishing a new foundation for the future of graphics," *ATI Technologies Press Release*, July 18, 2002. http://mirror.ati.com/companyinfo/press/2002/4512.html.

[3] F. Abazovic, "ATI Radeon 9700 spec revealed," *the inquirer*, July 15, 2002. http://www.theinquirer.net/?article=4431.

[4] S. Thompson, P. Packan, and M. Bohr, "MOS scaling: Transistor challenges for the 21st century," *Intel Technology Journal*, pp. 1–19, 1996.

[5] S. Ohr and R. Wilson, "Trouble at 130-nm node causes finger pointing," *EE Times*, June 14, 2002.

[6] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Transactions on Computers*, vol. C-20, pp. 1469–1479, December 1971.

[7] J. Pistorius and M. Hutton, "Placement rent exponent calculation methods, temporal behaviour, and fpga architecture evaluation," in *ACM/IEEE 5th International Workshop on System Level Interconnect Prediction*, April 5–6 2003.

[8] V. Betz and J. Rose, "Using architectural families to increase FPGA speed and density," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 10–16, 1995.

[9] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[10] K. Compton, A. Sharma, S. Phillips, and S. Hauck, "Flexible routing architecture generation for domain-specific reconfigurable subsystems," in *International Conference on Field Programmable Logic and Applications*, pp. 59–68, September 2002.

[11] Actel Corporation, Sunnyvale, CA, *Online Databook*, 2002. http://www.actel.com/techdocs/ds/index.html.

[12] eASIC Corporation, San Jose, CA, *Online Databook*, 2002. http://www.easic.com/products/index.html.

[13] Leopard Logic Incorporated, Cupertino, CA, *Online Databook*, 2002. http://www.leopardlogic.com/home.html.

[14] A. Cataldo, "Hybrid architecture embeds Xilinx FPGA core into IBM ASICs," *EE Times*, June 24, 2002.

[15] K. Zhu, D. F. Wong, and Y.-W. Chang, "Switch module design with application to two-dimensional segmentation design," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 481–486, November 1993.

[16] P. Hall, "On representatives of subsets," *Journal of the London Mathematical Society*, vol. 10, pp. 26–30, 1935.

[17] G. A. Margulis, "Explicit construction of concentrators," *Problems of Information Transmission*, vol. 9, no. 4, pp. 325–332, 1973.

[18] O. Gabber and Z. Galil, "Explicit construction of linear sized superconcentrators," *Journal of Computer and System Sciences*, pp. 407–420, 1981.

[19] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Boston: Kluwer Academic Publishers, 1999.

[20] S. Brown, R. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, May 1992.

[21] C. Sung, R. Cliff, J. Huang, B. Wang, K. Nguyen, X. Wang, K. Veenstra, B. Pedersen, and J. Turner, "A silicon efficient FLEX6000 programmable logic architecture," in *IEEE Custom Integraged Circuits Conference*, pp. 273–276, 1998.

[22] K. Veenstra, B. Pedersen, J. Schleicher, and C. Sung, "Optimizations for a highly cost-efficient programmable logic architecture," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 20–24, 1998.

[23] V. Aggarwal, "Actel's ProASIC family: The non-volatile, reprogrammable gate array," in *Actel-Synopsis fpgaforum Quarterly Application Notes*, pp. 2–3, October 1999.

[24] A. El Gamal, J. Greene, and V. Roychowdhury, "Segmented channel routing is nearly as efficient as channel routing (and just as hard)," in *Advanced Research in VLSI Conference*, pp. 192–211, March 1991.

[25] J. Greene, V. Roychowdhury, S. Kaptanoglu, and A. El Gamal, "Segmented channel routing," in *Design Automation Conference*, pp. 567–572, 1990.

[26] J. W. Greene, V. P. Roychowdhury, and A. El Gamal, "Segmented channel routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 79–95, January 1993.

[27] Actel, Sunnyvale, CA, *FPGA Data Book and Design Guide*, 1994.

[28] K. Zhu and D. F. Wong, "On channel segmentation design for row-based FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 26–29, 1992.

[29] M. Pedram, B. S. Nobandegani, and B. T. Preas, "Architecture and routability analysis for row-based FPGAs.," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 230–235, 1993.

[30] Altera Corporation, San Jose, CA, *Online Databook*, 2002. `http://www.altera.com/literature/lit-index.html`.

[31] Xilinx, Inc., San Jose, CA, *Online Databook*, 2002. `http://www.xilinx.com/partinfo/databook.htm`.

[32] Vantis (AMD) Corporation, *On-line Mach Family Data Sheets*, March 1997. `http://www.vantis.com/products/products.html`.

[33] Lattice Semiconductor Corporation, *On-line ispLSI/pLSI Data Sheets*, March 1997. `http://www.latticesemi.com/cgi-bin/lattice_list_files`.

[34] A. K. Aggarwal, "Routing algorithms and architectures for hierarchical field programmable gate arrays," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, January 1994.

[35] A. K. Aggarwal and D. M. Lewis, "Routing architectures for hierarchical field programmable gate arrays," in *IEEE International Conference on Computer Design*, pp. 475–478, 1994.

[36] V. C. Chan and D. M. Lewis, "Area-speed tradeoffs for hierarchical field-programmable gate arrays," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 51–57, 1996.

[37] Y.-T. Lai and P.-T. Wang, "Hierarchical interconnection structures for field programmable gate arrays," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 186–196, June 1997.

[38] D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, R. Cliff, and J. Rose, "The Stratix routing and logic architecture," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 12–20, February 2003.

[39] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size," in *IEEE Custom Integraged Circuits Conference*, (Santa Clara, CA), pp. 551–554, 1997.

[40] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 3–12, 2000.

[41] E. Ahmed, "The effect of logic block granularity on deep- submicron FPGA performance and density," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2001.

[42] J. Rose and S. Brown, "Flexibility of interconnection structures in field-programmable gate arrays," *IEEE Journal of Solid State Circuits*, vol. 26, pp. 277–282, March 1991.

[43] S. D. Brown, G. 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.

[44] V. Betz and J. Rose, "Directional bias and non-uniformity in FPGA global routing architectures," in *IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, CA), pp. 652–659, 1996.

[45] V. Betz and J. Rose, "Effect of the prefabricated routing track distribution on FPGA area-efficiency," *IEEE Transactions on VLSI Systems*, pp. 445–456, September 1998.

[46] B. Tseng, J. Rose, and S. Brown, "Using architectural and CAD interactions to improve FPGA routing architectures," in *First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp. 3–8, February 1992.

[47] V. Betz and J. Rose, "Circuit design, transistor sizing, and wire layout of FPGA interconnect," in *IEEE Custom Integraged Circuits Conference*, pp. 171–174, May 1999.

[48] V. Betz and J. Rose, "FPGA routing architecture: Segmentation and buffering to optimize speed and density," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 59–68, 1999.

[49] A. Roopchansingh, "Nearest neighbour interconnect architecture in deep-submicron FPGAs," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2002.

[50] A. Roopchansingh and J. Rose, "Nearest neighbour interconnect architecture in deep submicron FPGAs," in *IEEE Custom Integraged Circuits Conference*, 2002.

[51] Y.-L. Wu and M. Marek-Sadowska, "Orthogonal greedy coupling — a new optimization approach to 2-D FPGA routing," in *ACM/IEEE Design Automation Conference*, June 1995.

[52] Y.-W. Chang, D. F. Wong, and C. K. Wong, "Universal switch-module design for symmetric-array-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 80–101, January 1996.

[53] G. G. Lemieux, S. D. Brown, and D. Vranesic, "On two-step routing for FPGAs," in *International Symposium on Physical Design*, (Napa, CA), April 1997.

[54] R. G. Tessier, *Fast Place and Route Approaches for FPGAs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[55] Y.-W. Chang, D. F. Wong, and C. K. Wong, "Universal switch-module design for symmetric-array-based FPGAs," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 80–86, 1996.

[56] M. Shyu, Y.-D. Chang, G.-M. Wu, and Y.-W. Chang, "Generic universal switch blocks," in *IEEE International Conference on Computer Design*, pp. 311–314, 1999.

[57] M. Shyu, Y.-D. Chang, G.-M. Wu, and Y.-W. Chang, "Generic universal switch blocks," *IEEE Transactions on Computers*, vol. 49, pp. 348–359, April 2000.

[58] H. Fan, Y.-L. Wu, and Y.-W. Chang, "Comment on "generic universal switch blocks"," *IEEE Transactions on Computers*, vol. 51, pp. 93–95, January 2002.

[59] H. Fan, J. Liu, Y.-L. Wu, and C.-C. Cheung, "On optimum designs of universal switch blocks," in *International Conference on Field Programmable Logic and Applications*, pp. 142–151, September 2002.

[60] H. Fan, J. Liu, and Y.-L. Wu, "General models for optimum arbitrary-dimension FPGA switch box designs," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 93–98, November 2000.

[61] H. Fan, J. Liu, and Y.-L. Wu, "Combinatorial routing analysis and design of universal switch blocks," in *Asia South Pacific Design Automation Conference*, pp. 641–644, January 2001.

[62] H. Fan, J. Liu, Y.-L. Wu, and C.-C. Cheung, "On optimum switch box designs for 2-D FPGAs," in *ACM/IEEE Design Automation Conference*, June 2001.

[63] S. J. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1997.

[64] M. I. Masud, "FPGA routing structures: A novel switch block and depopulated interconnect matrix architectures," Master's thesis, Department of Electrical and Computer Engineering, University of British Columbia, December 1999.

[65] M. I. Masud and S. Wilton, "A new switch block for segmented FPGAs," in *International Workshop on Field Programmable Logic and Applications*, August 1999.

[66] A. Yan, R. Cheng, and S. Wilton, "On the sensitivity of FPGA architectural conclusions to experimental assumptions, tools, and techniques," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 147–156, February 2002.

[67] Y.-W. Chang, D. F. Wong, and C. K. Wong, "Design and analysis of FPGA/FPIC switch modules," in *IEEE International Conference on Computer Design*, pp. 394–401, 1995.

[68] G.-M. Wu and Y.-W. Chang, "Switch-matrix architecture and routing for FPDs," in *International Symposium on Physical Design*, pp. 158–163, 1998.

[69] G.-M. Wu and Y.-W. Chang, "Maximally routable switch matrices for FPD design," in *IEEE International Symposium on Circuits and Systems*, vol. 6, pp. 131–134, 1998.

[70] G.-M. Wu and Y.-W. Chang, "Quasi-universal switch matrices for FPD design," *IEEE Transactions on Computers*, vol. 48, pp. 1107–1122, October 1999.

[71] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska, "Computational complexity of 2-D FPGA routing for arbitrary switch box topologies," in *ACM International Workshop on Field-Programmable Gate Arrays*, 1994.

[72] Y.-L. Wu and M. Marek-Sadowska, "An efficient router for 2-D field-programmable gate arrays," in *European Design Automation Conference*, pp. 412–416, 1994.

[73] Y.-L. Wu and D. Chang, "On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 362–366, 1994.

[74] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska, "Graph based analsysis of 2-D FPGA routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, January 1996.

[75] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman and Company, 1979.

[76] Y. Takashima, A. Takahashi, and Y. Kajitani, "Routability of fpgas with extremal switch-block structures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E81-A, pp. 850–856, May 1998.

[77] Y. Sun, T.-C. Wang, C. K. Wong, and C. L. Liu, "Routing for symmetric FPGAs and FPICs," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 486–490, 1993.

[78] Y. Sun, T.-C. Wang, C. K. Wong, and C. L. Liu, "Routing for symmetric FPGAs and FPICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 20–31, January 1997.

[79] P. Hallschmid and S. J. Wilton, "Detailed routing architectures for embedded programmable logic ip cores," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 69–74, February 2001.

[80] S. Nakamura and G. M. Masson, "Lower bounds on crosspoints in concentrators," *IEEE Transactions on Computers*, vol. C-31, pp. 1173–1179, December 1982.

[81] A. Y. Oruç and H. M. Huang, "New results on sparse crossbar concentrators," in *Proceedings of Information Sciences and Systems Conference*, (Princeton University), 1994.

[82] K. Fujiyoshi, Y. Kajitani, and H. Niitsu, "Design of optimum totally-perfect connection-blocks of FPGA," in *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 221–224, May 1994.

[83] K. Fujiyoshi, Y. Kajitani, and H. Niitsu, "Design of minimum and uniform bipartites for optimum connection blocks of FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 1377–1383, November 1997.

[84] W. Guo and A. Y. Oruç, "Regular sparse crossbar concentrators," *IEEE Transactions on Computers*, vol. 47, pp. 363–368, March 1998.

[85] G. M. Masson, "Binomial switching networks for concentration and distribution," *IEEE Transactions on Communications*, vol. COM-25, pp. 873–883, September 1977.

[86] A. Y. Oruç and H. M. Huang, "Crosspoint complexity of sparse crossbar concentrators," *IEEE Transactions on Information Theory*, vol. 42, pp. 1466–1179, September 1996.

[87] K. Azegami, "Integrated circuit device with programmable junctions and method of designing such integrated circuit device," *United States Patent Number 6,323,678*, November 2001.

[88] A. Y. Oruç, "Multiple tracks of research on interconnection networks," Technical Report INRL-95-04, Interconnection Network Research Laboratory, University of Maryland, 1995.

[89] V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic Press, 1965.

[90] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, pp. 406–424, March 1953.

[91] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *Bell System Technical Journal*, vol. 43, pp. 1641–1656, 1964.

[92] G. W. Richards and F. K. Hwang, "A two-stage rearrangeable broadcast switching network," *IEEE Transactions on Communications*, vol. COM-33, no. 10, pp. 1025–1035, 1985.

[93] M. Butts, J. Batcheller, and J. Vargese, "An efficient logic emulation system," in *Proceedings of IEEE International Conference on Computer Design*, pp. 138–141, 1992.

[94] J. Vargese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Transactions on VLSI Systems*, vol. 1, pp. 171–174, June 1993.

[95] D. M. Lewis, D. R. Galloway, M. van Ierssel, J. Rose, and P. Chow, "The transmogrifier-2: A 1 million gate rapid prototyping system," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 53–61, February 1997.

[96] D. M. Lewis, D. R. Galloway, M. van Ierssel, J. Rose, and P. Chow, "The transmogrifier-2: A 1 million gate rapid prototyping system," *IEEE Transactions on VLSI Systems*, vol. 6, pp. 188–198, June 1998.

[97] M. A. Khalid and J. Rose, "A hybrid complete-graph partial-crossbar routing architecture for multi-FPGA systems," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 45–54, 1998.

[98] M. A. Khalid and J. Rose, "A novel and efficient routing architecture for multi-FPGA systems," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 30–39, February 2000.

[99] P. K. Chan and M. D. Schlag, "Architectural trade-offs in field-programmable device based computing systems," in *IEEE Workshop on FPGA's for Custom Computing Machines*, pp. 138–141, April 1993.

[100] W.-K. Mak and D. F. Wong, "On optimal board-level routing for FPGA-based logic emulation," in *ACM/IEEE Design Automation Conference*, pp. 552–556, 1995.

[101] W.-K. Mak and D. F. Wong, "On optimal board-level routing for FPGA-based logic emulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 282–289, March 1997.

[102] X. Song, W. N. Hung, A. Mishchenko, M. Chrzanowska-Jeske, A. Coppola, and A. Kennings, "Board-level multiterminal net assignment," in *ACM 12th Great Lakes Symposium on VLSI*, April 2002.

[103] A. Ejnioui and N. Ranganathan, "Multi-terminal net routing for partial crossbar-based multi-FPGA systems," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 176–184, February 1999.

[104] S.-S. Lin, Y.-J. Lin, and T. Hwang, "Net assignment for the FPGA-based logic emulation system in the folded-Clos network structure," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 316–320, March 1997.

[105] F. R. Chung, "On concentrators, superconcentrators, generalizers, and nonblocking networks," *Bell System Technical Journal*, vol. 58, pp. 1765–1777, October 1978.

[106] W. Guo and A. Y. Oruç, "Semi-explicit cosntruction of linear size concentrators and superconcentrators," Technical Report INRL-95-01, Interconnection Network Research Laboratory, University of Maryland, 1995.

[107] M. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson, "Randomness conductors and constant-degree lossless expanders," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, May 19–21 2002.

[108] M. Sheng and J. Rose, "Mixing buffers and pass transistors in FPGA routing architectures," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 75–84, February 2001.

[109] W. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, pp. 55–63, January 1948.

[110] T. Okamoto and J. Cong, "Buffered steiner tree construction with wire sizing for interconnect layout optimization," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 739–752, 1992.

[111] A. Marquardt, V. Betz, , and J. Rose, "Speed and area tradeoffs in cluster-based FPGA architectures," *IEEE Transactions on VLSI Systems*, pp. 84–93, February 2000.

[112] Collaborative Benchmarking Laboratory, *LGSynth93 suite*. North Carolina State University. `http://www.cbl.ncsu.edu/`.

[113] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit analysis," Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[114] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–12, January 1994.

[115] A. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 37–46, 1999.

[116] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Seventh International Workshop on Field-Programmable Logic*, (London, UK), pp. 213–222, 1997.

[117] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 203–213, 2000.

[118] J. Swartz, "A high-speed timing-aware router for FPGAs," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1998.

[119] J. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 140–149, February 1998.

[120] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider, "Teramac – configurable custom computing," in *IEEE Workshop on FPGA's for Custom Computing Machines*, 1995.

[121] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider, "Plasma: An FPGA for million gate systems," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 10–16, 1996.

[122] Altera Corporation, San Jose, CA, *News & Views Newsletter*, August 1999.

[123] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1993.

[124] I. S. Honkala and P. R. J. Östergård, "Applications in code design," in *Local Search in Combinatorial Optimization* (E. Aarts and J. Lenstra, eds.), ch. 12, Wiley, 1997.

[125] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith, "A new table of constant weight codes," *IEEE Transactions on Information Theory*, vol. IT-36, pp. 1334–1380, November 1990.

[126] R. L. Graham and N. J. A. Sloane, "Lower bounds for constant weight codes," *IEEE Transactions on Information Theory*, vol. IT-26, pp. 37–43, January 1980.

[127] K. J. Nurmela, M. K. Kaikkonen, and P. R. J. Östergård, "New constant weight codes from linear permutation groups," *IEEE Transactions on Information Theory*, vol. IT-43, pp. 1623–11630, September 1997.

[128] A. El Gamal, L. A. Heinachandra, I. Shperling, and V. K. Wei, "Using simulated annealing to design good codes," *IEEE Transactions on Information Theory*, vol. 33, pp. 116–123, January 1987.

[129] P. Leventis, "Placement algorithms and routing architecture for long-line based FPGAs," Bachelor of Applied Science thesis, Division of Engineering Science, Faculty of Applied Science and Engineering, University of Toronto, 1999.

[130] Altera, San Jose, CA, *1996 Data Book*, 1996.

[131] A. DeHon, "Entropy, counting, and programmable interconnect," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 73–79, 1996.

[132] A. Takahara, T. Miyazaki, T. Murooka, M. Katayama, K. Hayashi, A. Tsutsui, T. Ichimori, and K. nosuke Fukami, "More wires and fewer LUTs: a design methodology for FPGAs," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 12–19, February 1998.

[133] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 69–78, 1999.

[134] E. S. Ochotta, P. J. Crotty, C. R. Erickson, C.-T. Huang, R. Jayaraman, R. C. Li, J. D. Linoff, L. Ngo, H. V. Nguyen, K. M. Pierce, D. P. Wieland, J. Zhuang, and S. S. Nance, "A novel predictable segmented FPGA routing architecture," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 3–11, February 1998.

[135] S. Trimberger, K. Duong, and B. Conn, "Architecture issues and solutions for a high-capacity FPGA," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 3–9, February 1997.

[136] M. Sheng, "Mixing buffers and pass transistors in FPGA routing architecture," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2001.

[137] I. Dobbelaere, M. Horowitz, and A. El Gamal, "Regenerative feedback repeaters for programmable interconnections," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, 1995.

[138] P. Chow, S. Ong Seo, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja, "The design of an SRAM-based field-programmable gate array — part II: Circuit design and layout," *IEEE Transactions on VLSI Systems*, vol. 7, pp. 321–330, September 1999.

[139] M. Khellah, S. Brown, and Z. Vranesic, "Modelling routing delays in SRAM-based FPGAs," in *Canadian Conference on VLSI*, pp. 6B.13–18, November 1993.

[140] V. George and J. Rabaey, *Low Energy FPGAs: Architecture and Design*. Boston: Kluwer Academic Publishers, 2001.

[141] E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," in *International Symposium on Low Power Electronics and Design*, pp. 155–160, 1998.

[142] V. George and J. Rabaey, "The design of a low energy FPGA," in *International Symposium on Low Power Electronics and Design*, pp. 188–193, 1999.

[143] H. Zhang, V. George, and J. Rabaey, "Low-swing on-chip signaling techniques: effectiveness and robustness," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 264–272, June 2000.

[144] G. Lemieux and D. Lewis, "Using sparse crossbars within LUT clusters," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 59–68, February 2001.

[145] J. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Upper Saddle River, NJ: Prentice Hall, 1996.

[146] K. Martin, *Digital Integrated Circuit Design*. New York, NY: Oxford University Press, 2000.

[147] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *ACM/SIGDA Int. Symp. on FPGAs*, pp. 135–143, 1999.

[148] K. Bernstein, M. Bhushan, and N. Rohrer, "On the selection of the optimal threshold voltages for deep submicron CMOS technologies," *IBM MicroNews*, vol. 7, no. 1, pp. 29–31, 2001.

[149] A. Cataldo, "Altera samples 0.13-micron high-density PLD," *EE Times*, January 7, 2002.

[150] H. Schmit and V. Chandra, "FPGA switch block layout and evaluation," in *ACM/SIGDA Int. Symp. on FPGAs*, (Monterey, CA), pp. 11–18, February 2002.