Architecture and Synthesis for Dynamically Reconfigurable Asynchronous FPGAs

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Benjamin Thomas Hill

February 2016

ARCHITECTURE AND SYNTHESIS FOR DYNAMICALLY RECONFIGURABLE ASYNCHRONOUS FPGAS

Benjamin Thomas Hill, Ph.D.

Cornell University 2016

The current slowdown in CMOS technology scaling presents opportunities for architectural innovation, in particular augmentation of general purpose processors with specialized units. Self-timed field-programmable gate arrays (FPGAs) are attractive in this space because of their high throughput, robustness, and modularity.

In my thesis, I present an architecture for a dynamically reconfigurable asynchronous field-programmable gate array, describe efforts to limit the overheads of asynchronous communication in the context of 3D integration, and develop an asynchronous-aware toolflow for mapping designs to the FPGA.

BIOGRAPHICAL SKETCH

Benjamin Hill was born in New York City to David and Lisa Hill. Growing up as a Coast Guard brat, he attended 9 different schools all across the country before his 18^{th} birthday. In lieu of a fixed mailing address, Ben received an enviable range of childhood experiences and a great education both at home and in the classroom. Ben exhibited the engineering gene early on, as evidenced by a trail of dissected household appliances (sometimes even returned to working condition). In 2003, he entered Franklin W. Olin College of Engineering as a member of its second ever graduating class. He left Olin four years later with a B.S. in Electrical and Computer Engineering and a kindled desire to help in their mission to transform engineering education. Ben went on to graduate study at Cornell University's Computer Systems Lab, where he joined the Asynchronous VLSI and Architecture group led by Dr. Rajit Manohar. Along the way, he also completed a minor in Education. One of the many wonderful people Ben met at Cornell was a biomedical engineer named Adelaide de Guillebon; the two married in 2015. Ithaca became the longest Ben had ever stayed in one place, and its waterfalls and wild places will always have a special place in his heart. But change is the only constant, and in 2012, Ben returned to his roots to become a part of the nascent Cornell Tech in New York City. Ben is currently a faculty member at Olin College.

ACKNOWLEDGEMENTS

I am deeply grateful to:

My thesis special committee: Rajit Manohar, Barbara Crawford, Christopher Batten, and David Albonesi. Thank you for always giving me encouragement and excellent guidance, any time I was smart enough to ask.

My colleagues and dear friends at CSL and Cornell Tech, for making my years there more than just a job. Thanks especially to the mighty λ-team: Jon Tse, Carlos Tadeo Ortega Otero, and Rob Karmazin, for sticking together through countless tapeout late nights.

Nathan Karst and Derek Lockhart, with whom I was truly blessed to share a home during my time in Ithaca. Thanks especially to Nathan for dragging me, kicking and screaming, across the dissertation-writing finish line.

My family and my wife Adelaide, for your unwavering love and support.

And to you: thanks for reading!

This research was supported by the Air Force Research Laboratories and by the Intelligence Advanced Research Projects Activity. I am grateful to them for making this work possible; any errors and shortcomings found within are mine alone.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Acknowledgements . . . iv
Table of Contents . . . v
List of Figures . . . viii
List of Abbreviations . . . x

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Asynchronous Logic . . . 3
1.3 FPGA Basics . . . 7
1.4 Organization of Thesis and Major Contributions . . . 10
1.5 Collaboration, Previous Publications, and Funding . . . 11

2 Asynchronous FPGA Architecture **14**
2.1 Introduction . . . 14
2.1.1 Related Work . . . 16
2.1.2 Contributions . . . 19
2.2 Asynchronous Dataflow FPGAs . . . 19
2.3 Asynchronous FPGA Circuits . . . 21
2.3.1 Computation . . . 22
2.3.2 Conditional Dataflow . . . 28
2.3.3 Programmable Routing . . . 31
2.4 Clustered Logic Blocks . . . 38
2.4.1 Clustering for Area . . . 39
2.4.2 Clustering for Performance . . . 40
2.5 Measured Results . . . 41
2.5.1 Throughput . . . 42
2.5.2 Area . . . 43

3 Dynamic Partial Reconfiguration **45**
3.1 Introduction . . . 45
3.1.1 Related Work . . . 47
3.1.2 Contributions . . . 49
3.2 Requirements for Dynamic Partial Reconfiguration . . . 50
3.2.1 Empty Pipeline Detection . . . 50
3.2.2 Preserving Internal State . . . 55
3.2.3 I/O Boundary Protection . . . 56
3.2.4 Region Control . . . 58
3.3 Power Gating . . . 60

4 3D FPGA Architecture **62**
4.1 Introduction . . . 62
4.1.1 Related Work . . . 63
4.1.2 Contributions . . . 65
4.2 Extending to 3D . . . 66
4.3 Need for Efficient Signaling . . . 69
4.4 Single-Bit Signaling Protocols . . . 71
4.4.1 WCHB . . . 73
4.4.2 RQDI . . . 74
4.4.3 ATLS . . . 74
4.4.4 STFB . . . 75
4.4.5 STATS . . . 75
4.5 Methodology . . . 79
4.5.1 Link Simulation . . . 79
4.5.2 Optimization Framework . . . 81
4.6 Evaluation and Discussion . . . 84
4.6.1 Planar Links . . . 86
4.6.2 TSV Links . . . 92
4.6.3 Link Failures and Reliability . . . 97
4.7 Conclusions . . . 100

5 Software Toolflow **102**
5.1 Introduction . . . 102
5.1.1 Related Work . . . 103
5.1.2 Contributions . . . 105
5.2 Phases of Synchronous Toolflow . . . 105
5.2.1 Synthesis . . . 106
5.2.2 Technology Mapping . . . 107
5.2.3 Clustering . . . 108
5.2.4 Placement . . . 109
5.2.5 Routing . . . 110
5.2.6 Bitstream Generation . . . 112
5.3 Asynchronous Performance . . . 112
5.3.1 Modeling Pipeline Performance . . . 114
5.3.2 Loops . . . 115
5.3.3 Reconvergent Paths . . . 119
5.4 Asynchronous FPGA Mapping Toolflow . . . 121
5.4.1 Synthesis . . . 122
5.4.2 Synchronous to Asynchronous Translation . . . 122
5.4.3 Bitstream . . . 123
5.4.4 Routing . . . 124
5.5 Partition-Based Clustering and Placement . . . 124
5.5.1 Graph Theory Background . . . 125
5.5.2 Recursive Bipartitioning . . . 127

5.5.3 Results 131
5.5.4 Discussion 133

A Summary of CHP Language **135**

Bibliography **137**

LIST OF FIGURES

1.1 Comparison of synchronous vs asynchronous circuit organization . 3
1.2 Asynchronous handshaking channel connecting two processes . . . 4
1.3 Timing behavior of synchronous and asynchronous systems 6
1.4 Island-style FPGA architecture 8

2.1 Synchronous logic implemented on an FPGA 20
2.2 Basic dataflow operators as implemented in the FPGA 21
2.3 4-way merge built from basic dataflow operators. 22
2.4 LUT process showing PCEVHB reshuffling 23
2.5 2x 1of2 to 1of4 converter 25
2.6 LUT pulldown computation stack 25
2.7 PCEHB two-bit adder process 26
2.8 Two-bit adder pulldown computation stacks 27
2.9 SPLIT, MERGE, and combined conditional unit 29
2.10 Conditional unit pulldown computation stacks 29
2.11 Conditional unit handshaking circuits 30
2.12 5-input LUT created from 2 LUTs and a MUX. 31
2.13 Disjoint switchbox with four switchpoints 32
2.14 Switchpoint implementation 33
2.15 Clustered logic block with multiple LUTs 38
2.16 Clustering examples 40
2.17 Chip layout for a single FPGA tile 42
2.18 FPGA area by category 44

3.1 Partial reconfiguration operations 46
3.2 Defining reconfigurable regions 59

4.1 3D extension of 2D AFPGA design 67
4.2 3D switchpoint buffer 68
4.3 3D tile floorplan showing relative TSV sizes 69
4.4 Asynchronous signaling protocols 72
4.5 WCHB Buffer 73
4.6 RQDI Buffer 74
4.7 STFB Buffer 76
4.8 Ternary Voltage Decoder 77
4.9 STATS transmit stage 78
4.10 Ternary return-to-null schemes 79
4.11 Link DUT, planar and TSV contexts 80
4.12 SPICE simulation harness for DUT 81
4.13 WCHB level shifters 82
4.14 Heuristic optimization framework for evaluating link circuit designs 83
4.15 Energy vs Throughput in 90 nm for a 1000 μm planar link 87
4.16 Area vs Throughput in 90 nm for a 1000 μm planar link 88

4.17 Energy-throughput Pareto-dominant points across planar technologies . . . 90
4.18 Area-throughput Pareto-dominant points across planar technologies 91
4.19 Energy vs Throughput in 90 nm for a 25 μm pitch TSV model . . 93
4.20 Area vs Throughput in 90 nm for a 25 μm pitch TSV model 94
4.21 Energy-throughput Pareto-dominant points for 25 μm pitch TSV . 96
4.22 Area-throughput Pareto-dominant points for 25 μm pitch TSV . . 97
4.23 Trace of STFB and STATS TSV links in 90nm 99

5.1 Intermediate stages of a design being FPGA mapped 103
5.2 Implementing abstract 5-input LUT using two 4-LUTs and MUX. . 107
5.3 Dataflow loop 116
5.4 Loop throughput 117
5.5 Loop throughput versus static slack for loops with varying numbers of tokens 117
5.6 Loop throughput versus static loop slack per token 118
5.7 Reconvergent path 119
5.8 Reconvergent path throughput for varying combinations of path slack 120
5.9 Reconvergent path throughput for varying path slack ratios 121
5.10 Recursive bipartitioning procedure 129
5.11 Sequence of bipartitioning operations assigns grid position 130
5.12 Throughput for partition-based placement with and without loop weighting 132
5.13 Throughput for simulated annealing versus partition-based placement. 133

LIST OF ABBREVIATIONS

AFPGA	Asynchronous FPGA
ASIC	Application Specific Integrated Circuit
ATLS	Asynchronous Ternary Logic Signaling
BLE	Basic Logic Element
CHP	Communicating Hardware Processes
CSP	Communicating Sequential Processes
DI	Delay-Insensitive
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous, Locally Synchronous
HDL	Hardware Description Language
LUT	Look-Up Table
MUX	Multiplexer
NMOS	N-type Metal-Oxide-Semiconductor
PCHB	Precharge Half Buffer
PMOS	P-type Metal-Oxide-Semiconductor
PVT	Process-Voltage-Temperature
QDI	Quasi Delay-Insensitive
RQDI	Relaxed Quasi Delay-Insensitive
RTN	Return-to-Null
SoC	System on a Chip
SRAM	Static Random Access Memory
STATS	Single-Track Asynchronous Ternary Signaling
STFB	Single-Track Full Buffer
TSV	Through Silicon Via
VPR	Versatile Place and Route
VLSI	Very Large Scale Integration
WCHB	Weak-Conditioned Half Buffer

CHAPTER 1

INTRODUCTION

1.1 Motivation

The history of computer architecture is marked by the pursuit of ever greater performance through the marriage of design innovations and steady increases in manufacturing technology. Over fifty years of this virtuous cycle have brought exponential gains in performance, integration density, and cost reduction, and completely reshaped the landscape of our modern world.

Unfortunately, manufacturing gains today are becoming increasingly hard-won. CMOS fabrication technology has historically benefited from Moore's law scaling, the empirical observation that the number of transistors that can economically integrated on a die grows exponentially over time. This trend continues today (albeit at a reduced pace), but crucially without the associated Dennard scaling that allowed that bounty of transistors to be used with approximately constant power requirements. This has caused some to (hyperbolically) predict a "coming dark silicon apocalypse" [137], in which increasing device counts with a constrained power budget force large swaths of future chips to remain unused.

A standard answer to the performance/energy wall ("dark silicon") is specialization [147]. Many researchers have taken up this challenge, and designs specialized to a given task have been shown to exhibit 10-100x performance gains compared to general purpose processors [38]. But specialization limits flexibility, and it is infeasible to design a specialized chip for every possible application.

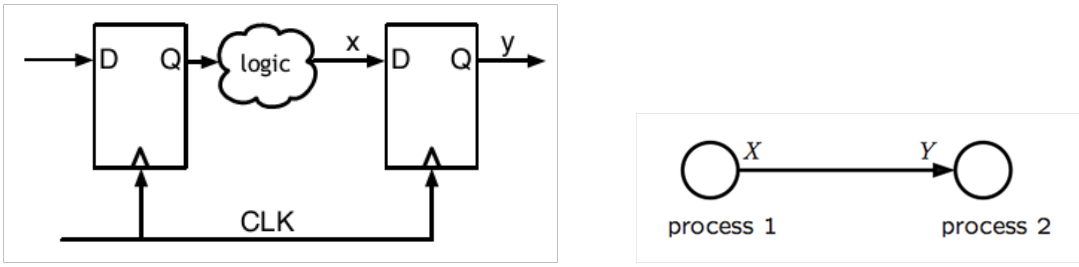
Reconfigurable computing systems combine the easy flexibility of general pur-

pose processing with the efficiency of specialized and reconfigurable accelerators. Often the accelerator in reconfigurable computing designs is implemented as a field-programmable gate array (FPGA), described further in Section 1.3. Many such systems have been proposed in the past [31, 36, 37, 58, 61, 123, 144, 147, 148]. Today, commercial reconfigurable computing systems including FPGAs and general purpose processors (such as Xilinx's Zynq [156]) have become more widely available.

So what is the ideal platform for reconfigurable computing? Standard architecture evaluation metrics apply: the system should have high performance and low energy. Previous work has shown that self-timed (Section 1.2) FPGAs can achieve high throughput compared to their synchronous counterparts [43].

Reconfigurable computing systems should also be capable of adapting on demand to changing workload requirements, without interrupting other ongoing operations. As systems grow to contain ever-larger amounts of varied accelerator resources, shutting down the entire ensemble to implement a new task becomes an unreasonable cost to bear. This suggests that the ability to dynamically reconfigure accelerator platforms will be important to future reconfigurable computing architectures.

The long-range aspiration that guides my work in this thesis is a reconfigurable computing architecture that includes a virtual hardware substrate for mapping accelerators on-demand. A major inspiration is the SCORE [21] computation model, which uses stream computation to implement applications on variable sized hardware platforms. Virtual memory systems provide the illusion of boundless memory space to an application while leaving the management of the physical details to the operating system. Virtual hardware would operate in a similar fashion by con-



(a) Synchronous logic (b) Asynchronous processes

Figure 1.1: Comparison of synchronous vs asynchronous circuit organization

trolling time-shared usage of a general purpose reconfigurable substrate, and could enable on-line synthesis of application-specific accelerators. In the short term, I seek to design a high-performance reconfigurable hardware substrate that could enable these applications, along with flexible software that allows experimentation on reconfigurable asynchronous platforms.

1.2 Asynchronous Logic

Asynchronous circuits are so-named based upon what they lack: a global clock signal that synchronizes actions and data transmission throughout the system. Instead, asynchronous systems are composed of a collection of self-timed processes that operate at their own maximum local rate and exchange data with other processes via communication channels.

Figure 1.1a shows a prototypical synchronous system. A 'cloud' of combinational logic computes a value (at X) which is stored in the state-holding flip-flop and propagated to Y when commanded by the clock signal. The clock period must be long enough to allow sufficient time for the combinational logic computation between the state-holding elements to complete. This is true for every path between state-holding elements, so the clock sets the timing globally for the entire

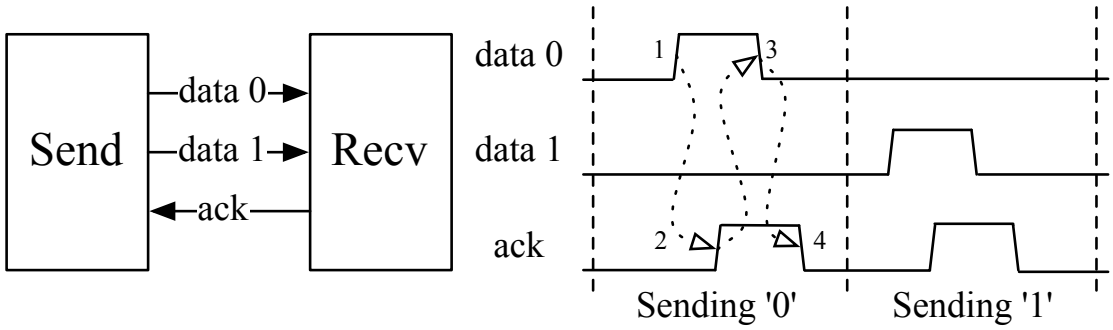


Figure 1.2: Asynchronous handshaking channel connecting two processes

system (or clock domain region if there are multiple clocks in the system).

By contrast, in Figure 1.1b the processes independently perform computations without making assumptions about the delays of wires and gates. When process 1 sends data to process 2, it places a data 'token' on port X of the communication channel between them. Process 2 can then receive that token on port Y of the same channel. These communication channels are generally point-to-point in an asynchronous system. The synchronization required to send a token across a channel is sequenced via a delay-insensitive local 'handshake' rather than a timed global or regional clock signal.

Figure 1.2 shows one specific implementation of a single-bit channel handshake. I discuss several other signaling protocols in Chapter 4. This simple example captures many of the features, strengths, and weaknesses of asynchronous communication in general. It shows a 'four-phase' handshake, with the following sequence of operations:

1. The sending process asserts one of two wires to encode the transmitted data in a delay-insensitive manner. This is known as a 1-of-2 encoding, and can be generalized to other delay insensitive encodings for larger numbers of bits (e.g. 1-of-4 for two bits, m-of-n more generally).

2. The receiving process confirms receipt of the data by raising an acknowledge wire traveling in the opposite direction as the data. At this point, the data has been transmitted and the processes are synchronized.

3. After seeing the acknowledge, the sending process resets its data values to their neutral state (indicating no data present).

4. Finally, after the data wires are neutral the receiver deasserts the acknowledge wire and the channel is reset to its initial state.

   The final two steps are known as the 'reset phase' of the handshake. They exist to simplify the circuits required to implement the handshake, and may be omitted (resulting in a 'two-phase' handshake).

In practice, synchronous vs asynchronous design style is not a binary choice, but rather two ends of a spectrum. For example, self-timed bundled data systems incorporate timing assumptions to remove the area overhead associated with delay-insensitive data encoding, and latch-based synchronous design can allow some timing flexibility throughout the system via clock borrowing.

There are a large variety of asynchronous design styles [10, 59, 112]. Unless otherwise noted, when I refer to asynchronous or self-timed systems in this thesis, I mean quasi delay-insensitive (QDI) circuits [105]. This class of circuits makes no assumptions about the delays on wires or gates, except that certain wire forks must be isochronic (have equal delay on both branches) [11]. Since fully delay-insensitive (DI) have been shown to be limited in what they can implement [102], QDI is considered to have the fewest possible timing assumptions among 'useful' design styles. Additionally I will consider only slack elastic systems [98].

Figure 1.3 shows the timing behavior of the same pipelined computation in

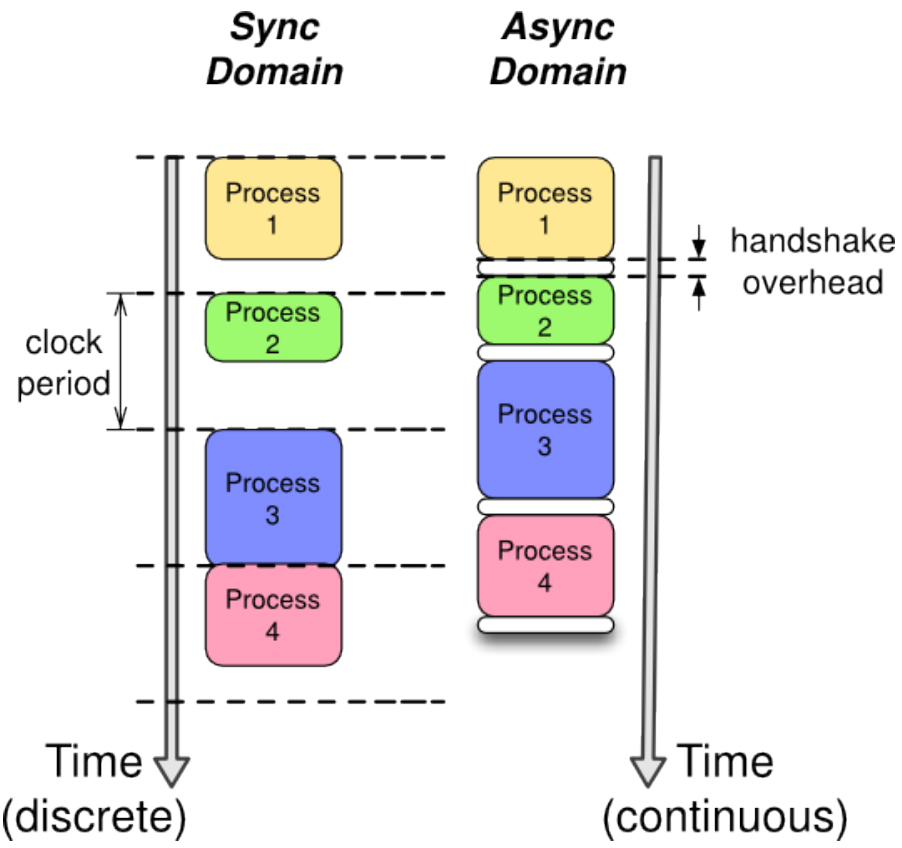


Figure 1.3: Timing behavior of synchronous and asynchronous systems

both synchronous and asynchronous systems. Each individual stage of the asynchronous computation may take longer, due to the overhead introduced by the round-trip local handshake (Figure 1.2) between each process. On the other hand, the synchronous implementation is constrained by the fact that the the global clock must accommodate the operation with the longest delay, while in the self-timed system each operation proceeds at its maximum local rate. The ability to exploit this average-case vs. worst-case algorithmic performance gap is a key benefit of asynchronous systems, and it allows for simple 'make the common case fast' local optimizations to have a global impact.

Other key asynchronous benefits stem from the fact that circuits operate correctly regardless of delay. This makes them tolerant of variations in manufacturing

process, voltage, and temperature. Furthermore, performance can be decoupled from correctness in self-timed systems. In slack elastic systems like those I will consider in the following chapters, physical pipelining is distinct from logical pipelining, which allows us to remap designs without retiming. Finally, the channel-based communication protocol allows simple composition of modules and interfacing with other independently designed systems, as is the case in a System on Chip (SoC) assembled from multiple reusable intellectual property blocks. It is also possible to achieve some of these benefits in synchronous systems by using well-defined protocols, such as in a globally asynchronous locally synchronous architecture.

There are however important drawbacks to asynchronous design. One major cost of delay-insensitive communication is that all transitions must be acknowledged as shown in Figure 1.2. Put simply, if we do not assume bounds on wire delay (other than that the delay is finite), we cannot know if a signal has been received unless the receiver replies. As a result, all delay-insensitive communications must be round trip. This adds both delay and additional wiring compared to schemes with timing assumptions. Additionally, both delay-insensitive encoding of values and additional acknowledge signals contribute to increased wire counts in asynchronous systems. I describe my efforts to reduce this disparity in Chapter 4.

1.3 FPGA Basics

Field-programmable gate arrays (FPGAs) are logic devices whose function may be changed after fabrication (field-programmable). Unlike other computing architectures they are traditionally organized to be flexible at an extremely fine granularity (in a gate array), allowing manipulation down to the level of single-bit computa-

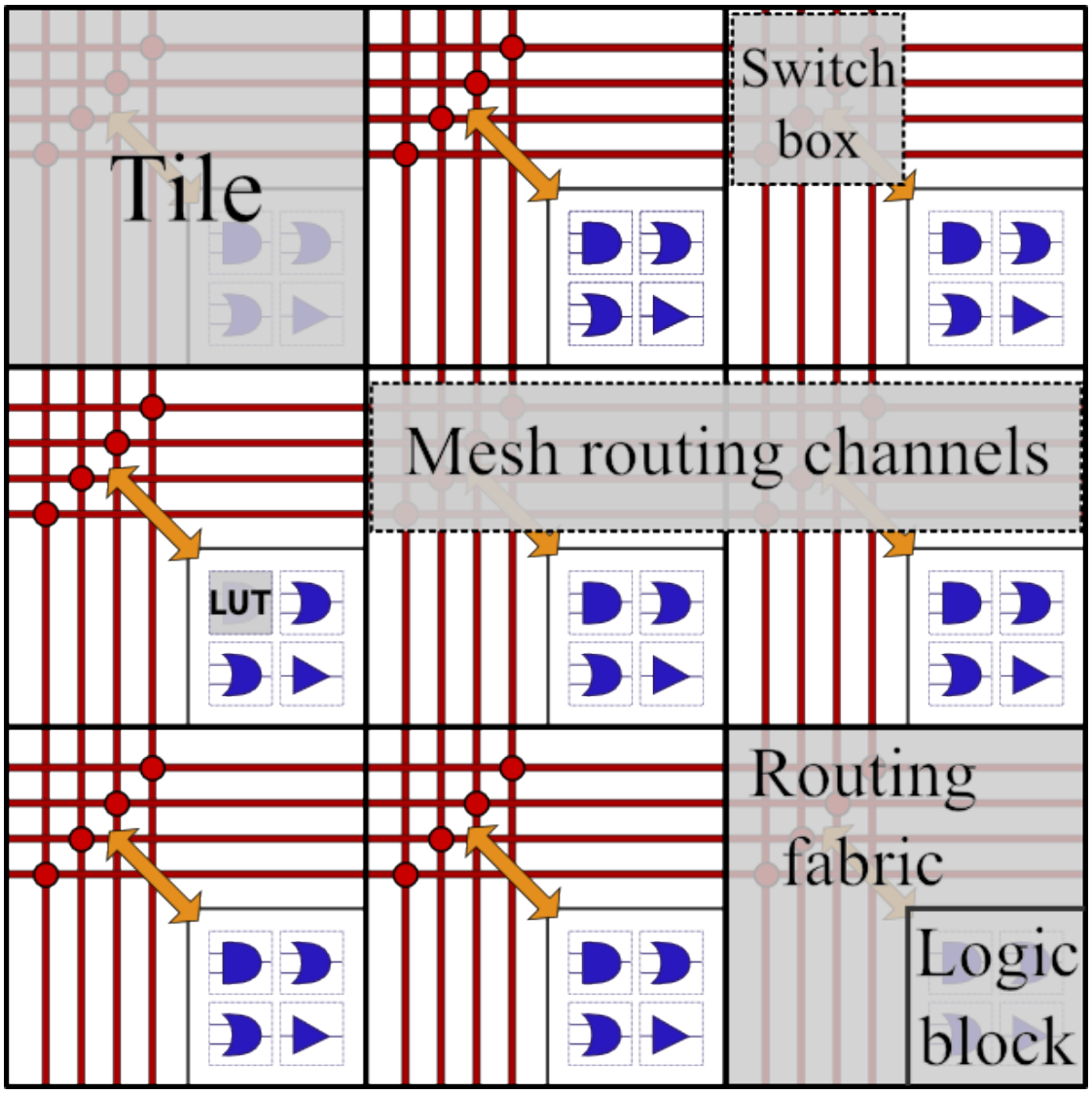


Figure 1.4: Island-style FPGA architecture

tions. Large collections of single bit computations can then be composed to build any larger digital system. FPGAs are a key enabling technology for specialization in reconfigurable computing systems.

Figure 1.4 shows the internals of an FPGA at the architectural level. FPGAs organized in this manner are called "island-style", because they have "islands" of logic surrounded by a "sea" of programmable interconnect. Other organizations of logic and routing are also possible, such as row-based and hierarchical [53].

The **tile** is the basic organizational unit of an island-style FPGA. Tiles are arranged in a two-dimensional grid, connected to their neighbor tiles in all compass directions by **mesh routing channels**. A routing channel is composed of multiple **routing tracks**, each of which carries a single logical signal. Connections between routing track segments are made at programmable **switch points**, and all the switch points within a tile are grouped into a **switch box**. Collectively the switch box and routing channels are known as **routing fabric** or global interconnect. In addition to making connections between tiles, the routing fabric also connects to the **logic block**. This computational core of the tile is made up of one or more programmable lookup tables or **LUT**s. Not shown but distributed throughput the FPGA is the **configuration memory** or programming bits that allow the functionality of the device to be changed.

Chapter 2 describes FPGA architecture in more detail, in the context of the design of an asynchronous FPGA platform.

The flexibility offered by FPGAs also means that they are necessarily less efficient than a custom implementation. To understand why this must be true, first map any design to an FPGA, then remove all unused logic blocks, reconfigurable interconnect, and programming bits to decrease circuit area, energy, and critical path. Application-specific integrated circuits (ASICs) have been shown to have a 3–4x improvement in critical path and a 40x reduction in area over FPGAs [87]. Custom designed chips can have a further 3-8x performance improvement over ASICs implemented in the same fabrication technology [28, 29]. On the other hand, implementing custom chips for specialized circuits comes with significant non-recurring engineering and fabrication costs. The generality of FPGAs shares these costs by allowing one chip to be manufactured and used for many different

purposes, especially for rapid prototyping and low-volume manufacturing.

A major reason aside from economic necessity to continue to use FPGAs despite their inefficiency is their inherent reconfigurability. Reconfigurable hardware can do things that fixed-function specialized accelerators cannot, such as adapting to changing data or system needs. In Chapter 3, I introduce my implementation of dynamically reconfigurable FPGAs to support this use case.

1.4 Organization of Thesis and Major Contributions

The remainder of this thesis describes my efforts working toward the vision of on-demand accelerators in reconfigurable computing systems.

Chapter 2 presents the basic architecture of my asynchronous FPGA. It builds upon previous work [139] and extends it with circuit enhancements and clustered logic blocks that increase performance, save area, and improve interconnect efficiency for dataflow designs. I fabricated multiple versions of this AFPGA, and demonstrated that it achieves higher peak performance than synchronous FPGAs in the same process technology.

In Chapter 3, I present a new implementation of dynamic partial reconfiguration for asynchronous architectures. This design is flexible enough enables a variety of system designs enabling on-the-fly replacement of accelerators, and requires very low hardware overhead.

One compelling approach for adding specialization hardware is to leverage recently-maturing 3D integration technology [8, 17] to physically stack accelerators on top of a computing system. In Chapter 4, I describe my 3D AFPGA

architecture that could enable such systems, discuss the major cost sources in 3D communication, and present a new signaling protocol that increases both energy and area efficiency.

Finally, in Chapter 5 I present my full electronic design automation toolflow for mapping high-level designs to an asynchronous FPGA platform, which includes a new asynchronous-aware approach to clustering and placing logic within the FPGA array.

1.5 Collaboration, Previous Publications, and Funding

The work described in this thesis could not have been completed without the advice, assistance, and support of a great many collaborators.

The AFPGA architecture described in Chapter 2 was first fabricated in collaboration with the Air Force Research Laboratories in Rome, NY. John Rooks, Tom Renz, Rich Linderman, and Qing Wu at AFRL, as well as James Stine at Oklahoma State University were instrumental in bringing the work to life in silicon.

Rob Karmazin assisted with the chip design, particularly the configuration memory interface. Filipp Akopyan designed the interface to synchronous system components outside the FPGA. Rob, along with Carlos Tadeo Ortega Otero and Jon Tse contributed mightily to the custom mask design for the physical layout implementation.

We also fabricated an obfuscated version of the AFPGA architecture using a split-foundry manufacturing method intended to foil IP theft and malicious insertion of hardware trojans [63]. The FPGA design adds an extra layer of security,

as its functionality is not set until it is loaded with the configuration bitstream. Carlos, Jon, and Rob developed an automated tool, split-cellTK [115], that made the design of that chip possible without extremely time consuming manual layout. This work was supported by IARPA award N66001-12-C-2009.

The 3D AFPGA described in Chapter 4 was fabricated in collaboration with Robert Geer and Harika Manem, researchers at the Colleges of Nanoscale Science and Engineering (CNSE) at SUNY Albany. They developed the TSV process that enables my die-stacked architecture.

Christopher LaFrieda invented the RQDI signaling protocol described in Chapter 4. In [88], we investigated the energy savings possible by using a voltage-scaled version of that design to replace the interconnect within my AFPGA architecture. That research was supported in part by the AFRL under contract FA8750-07-2-0191, and in part by the FCRP/DARPA Center on Circuits and Systems Solutions (C2S2).

I partnered with Jon Tse to conduct the single-bit signaling analysis described in Chapter 4 and in [145]. Jon had the inspiration for the heuristic optimization framework and the novel STATS signaling protocol, both of which we developed together. That work was supported in part by AFRL award FA8750-12-2-0035, and NSF awards CCF-1065307 and the TRUST STC center CCF-0424422.

Undergraduate researchers made helpful contributions to the software toolflow described in Chapter 5. Eashwar Rangarajan developed an early version of the synthesis step, which translated from Verilog to BLIF using Synopsys Design Compiler and custom software. Anay Joshi and Advait Muktibodh, visiting students from IIT Bombay, developed a prototype compiler for mapping designs from C to my

AFPGA architecture. Discussions with them were helpful in envisioning the possible applications of the architecture as an on-demand reconfigurable accelerator platform.

CHAPTER 2

ASYNCHRONOUS FPGA ARCHITECTURE

2.1 Introduction

In Section 1.1, I described several desirable characteristics for a high performance accelerator substrate. Asynchronous FPGA (AFPGA) architectures feature several key differences from their synchronous counterparts that make them an attractive choice for reconfigurable computing systems.

An accelerator that can keep pace with an associated general purpose processor is more useful than one that cannot, so high performance is a priority. AFPGAs are capable of sustaining high throughput thanks to their pipelined interconnect, described in Section 2.3.3. This fine-grained pipelining is more difficult to accomplish in synchronous FPGAs, and Section 2.5 shows that the fabricated devices offer higher peak throughput than commercial synchronous FPGAs in the same manufacturing technology. Delay-insensitive communication allows the constraints on mapping designs to an FPGA platform to be greatly simplified by breaking global timing dependencies.

Despite these advantages, asynchronous communication comes at a cost. Interconnect makes up a dominant fraction of FPGA area, and delay-insensitive interconnect is more expensive than synchronous in terms of area and wire count. I tackle this disparity directly in Chapter 4 by designing a new signaling protocol in which each wire is used more efficiently. I also help mitigate the area penalty in the architecture design by grouping related logic into local clusters, as described in Section 2.4. This saves area by reducing the communication burden on the global

routing fabric. Clustering also helps achieve high throughput in mapped designs by alignment with the optimization requirements of asynchronous systems, which differ from those of synchronous circuits as described in Section 5.3.

If we turn to accelerators for salvation from 'dark silicon', they must have low static power consumption. My AFPGA architecture is dataflow-driven (Section 2.2), which means that it is only active when computing new results. This is the equivalent of perfect clock gating in a synchronous system. In the current fabrication technology regime where leakage power has become significant, I further enhance the AFPGA architecture by adding low-cost, fine-grained power gating to help control leakage current in inactive regions (Section 3.3).

Finally, accelerators should have a clean modular design that allows us to easily compose and reuse them in reconfigurable computing systems. Thanks to their well-defined communication protocols and the robustness offered by delay-insensitivity, asynchronous circuits excel in this capacity. Asynchronous systems exhibit average rather than worst-case performance: slow modules only impact timing when they are actually used. One does not need to consider the global system (e.g. retiming) when designing modules. This unique ability to focus on "make the common case fast" enables an interesting potential use model in which accelerators are dynamically compiled on demand, and can be later refined and replaced if they are used more frequently.

The next subsection relates the evolution of asynchronous FPGA design that led to these desirable features. The remainder of the chapter describes how I improve the state of art for an AFPGA geared toward use as a high-performance reconfigurable accelerator.

2.1.1 Related Work

The design of the earliest asynchronous FPGAs was inspired by their synchronous counterparts. They consist of programmable gates and interconnect, but with modifications to account for the ways in which asynchronous circuits differ from synchronous. Without a global clock for synchronization, asynchronous circuits must usually be hazard-free. Tightly controlled delays are critical for some implementation styles. Asynchronous gates often have combinational loops or alternate state holding structures. Communication between modules occurs via channels, rather than the bare wires in a synchronous FPGA. Payne [117] presents an excellent (pre-)history of AFPGA architectures.

Montage [60] is generally recognized as first asynchronous FPGA. It was based on the earlier Triptych [18] FPGA design. Triptych featured a "sea-of-gates" organization, with logic gates connected directly to neighbors. Routing in this type of FPGA is implemented using logic gates rather than a separate interconnect [41]. Triptych's function block circuit design was already hazard-free, but Montage included additional accommodations for mapping asynchronous designs. Charge sharing and the timing of isochronic forks were addressed via careful layout. Montage also featured fast feedback paths for asynchronous state-holding circuits, as well as arbitration hardware.

Several other projects also took up the task of designing an FPGA specialized for implementing asynchronous logic of various styles. STACC [118] used four-phase bundled data signaling, which allows circuits to be very similar to their synchronous equivalents with some handshaking added. PGA-STC [96] used a two-phase bundled data protocol modeled after micropipelines [135]. Both of these architectures included reconfigurable delay lines to support the required bundled

data timing. PCA-1, the 'Plastic Cell Architecture' [86], was decomposed into two separate planes. The 'Plastic Part' implemented logic functions, while the 'Built-in Part' formed a communication network. Notably, PCA-1 supported dynamic reconfiguration.

Despite the mismatch in circuit requirements, there have been efforts to implement asynchronous logic using clocked FPGAs. The major advantage of this approach is that commercial synchronous FPGAs are widely available. Keller [77] was able to implement QDI circuits in the Null Convention Logic style on a Xilinx FPGA using the JBits tool. Ho and colleagues at TIMA [64] developed a design methodology for implementing hazard-free gates such as C-elements on commercial FPGAs.

Globally asynchronous, locally synchronous (GALS) FPGAs represent a "compromise" design point. Designs such as the one proposed by Royal and Cheung [122] subdivide the FPGA into synchronous island regions, which communicate using an asynchronous global interconnect. Such a system can be implemented atop a commercial synchronous FPGA, or as custom hardware. Another GALS FPGA design, GAPLA [70, 71], uses a hierarchical routing design with bundled data channels. GAPLA is implemented using separate timing and logic layers, and supports dynamic reconfiguration. The benefit of GALS design is that standard synchronous logic designs and mapping tools can be used for the local regions, while still retaining some of the benefits of asynchronous communications in long routes between regions. On the other hand, there are no self-timed benefits within regions. The selection of region size is also often fixed, and it is important to choose wisely to gain the asynchronous benefits without an excessive conversion penalty. For example, the FPGA presented in [94, 127] wraps asynchronous handshaking

circuitry including programmable delay lines around every LUT/BLE, leading to potentially excessive area cost.

Other FPGA designs explicitly offer support for multiple design styles (as did Montage). Huot et al. describe an FPGA that can support both QDI and micropipeline circuits [68]. Most recently, Manoranjan and Stevens [99] designed an FPGA based upon the Xilinx 7 series that can implement both synchronous and bundled data designs.

Another approach to implementing asynchronous circuits on a reconfigurable platform is to raise the level of abstraction. Rather than porting the primitive elements of existing FPGA designs to the asynchronous context, such a system uses dataflow building blocks based on based on high performance asynchronous circuit templates. This idea was first introduced with the Programmable Asynchronous Pipeline Array (PAPA) [140], which was intended to allow prototyping of high performance asynchronous circuits without requiring labor intensive custom layout. The concept was later developed into a complete asynchronous FPGA architecture [97, 141, 142].

Contemporaneously, Wong et al. proposed a similar process-level QDI FPGA implementation [154]. Each cell in their AFPGA can perform computation with configurable communication patterns, similar to a fusion of the computation and conditional units (Section 2.3). These cells are more flexible than the dataflow units proposed by Teifel, but also larger in area. They also propose clustering multiple cells together with local interconnect, unpipelined copying, and selectable slack buffering [155].

Komatsu et al. [84, 85] also use four-phase, dual rail processes as the core com-

putation elements in their AFPGA design, but they convert to a two-phase protocol for the interconnect to reduce switching. A similar hybrid signaling architecture was proposed in [88].

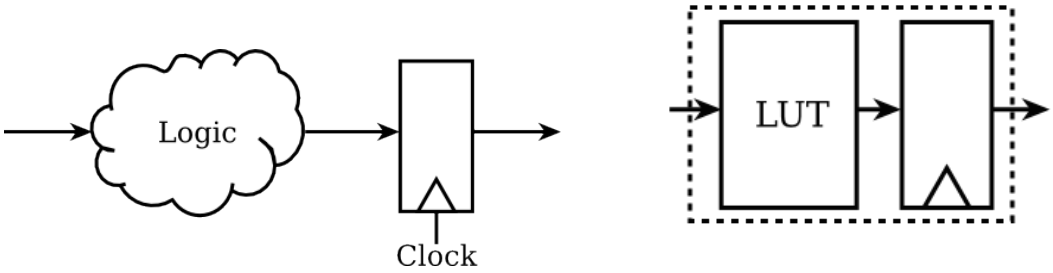
2.1.2 Contributions

My core architecture is based on the work of Teifel [139], which proposed an AFPGA design approach in which the primitive elements were dataflow building blocks. My work uses this idea, and extends it with the following enhancements:

- Improved circuit implementations (Section 2.3)
- Clustered logic blocks with fast local routing within a cluster (Section 2.4)
- Support for partial reconfiguration (Chapter 3)
- Fabricated AFPGA in multiple technologies and measured key architecture properties to validate architectural simulation tools (Section 2.5)

2.2 Asynchronous Dataflow FPGAs

At the most basic level, synchronous circuits are comprised of a collection of combinational logic interspersed with state-holding elements. Synchronous designs are mapped to an FPGA by implementing the combinational logic functions as a lookup table (LUT), with flip-flops serving to hold state and sequence operations. The grouping shown in Figure 2.1b is sometimes known as a basic logic element (BLE) [15]. In an FPGA, these logic elements are connected by reconfigurable



(a) Synchronous logic (b) Logic mapped to FPGA BLE

Figure 2.1: Synchronous logic implemented on an FPGA

interconnect elements, such as pass transistors and multiplexers, to implement the original design.

By contrast, we often think of an asynchronous circuit as a collection of concurrent processes executing in parallel and interacting with each other via channels as described in Section 1.2. This network of processes can be represented as a dataflow graph, showing the interconnections and interactions between processes. Tokens flowing through this graph indicate the movement of data in the system.

In an asynchronous FPGA, the components of a typical FPGA are replaced with their dataflow equivalents [141]. This allows us to implement asynchronous designs on the AFPGA by mapping their dataflow graph onto the AFPGA fabric. Just as a synchronous FPGA architecture uses generic lookup tables rather than implementing all forms of combinational logic, so too the asynchronous FPGA implements a basic set of dataflow operators as seen in Figure 2.2. More complex operations can be constructed from this basic set, as seen in Figure 2.3.

In a synchronous FPGA, the logic operators are connected with programmable interconnect. I take a largely similar approach in the asynchronous context, but with one key enhancement: the interconnect in an AFPGA can be pipelined at a fine granularity, allowing the system to support high throughput even in the

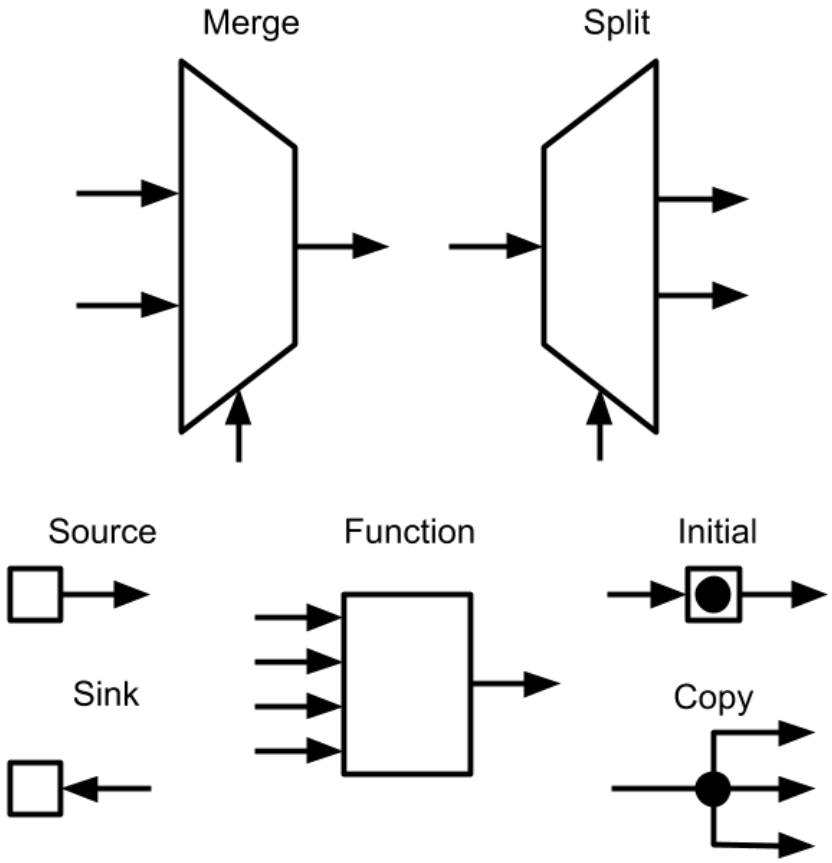


Figure 2.2: Basic dataflow operators as implemented in the FPGA

presence of long routes. This is discussed further in Section 2.3.3.

2.3 Asynchronous FPGA Circuits

My AFPGA is composed of three classes of circuits: computation modules, conditional dataflow modules, and configurable routing to tie the two former classes together. In this section, I describe the implementation of each class of circuits, focusing specifically on improvements made to the state of the art AFPGA design presented by Teifel in [139].

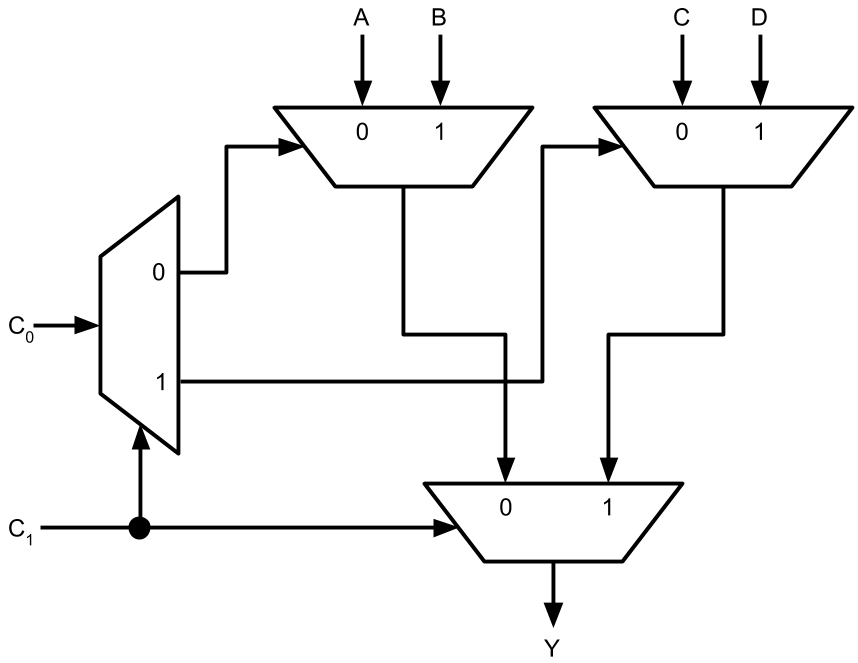


Figure 2.3: 4-way merge built from basic dataflow operators.

2.3.1 Computation

Lookup Table

The basic computational unit in the AFPGA is the dataflow function or LUT. We describe asynchronous circuits using the communicating hardware processes (CHP) language, which is explained further in Appendix A. A four-input LUT is represented by the following CHP:

$$LUT4 \equiv *[A?a, B?b, C?c, D?d; R!func(a, b, c, d)]$$

where A, B, C, D and R are one-bit channels, and $func$ is the function programmed into the lookup table configuration memory. To implement any possible function of four inputs, $2^4 = 16$ programming bits p are required.

If we were to consider the production rules needed to directly implement $func$,

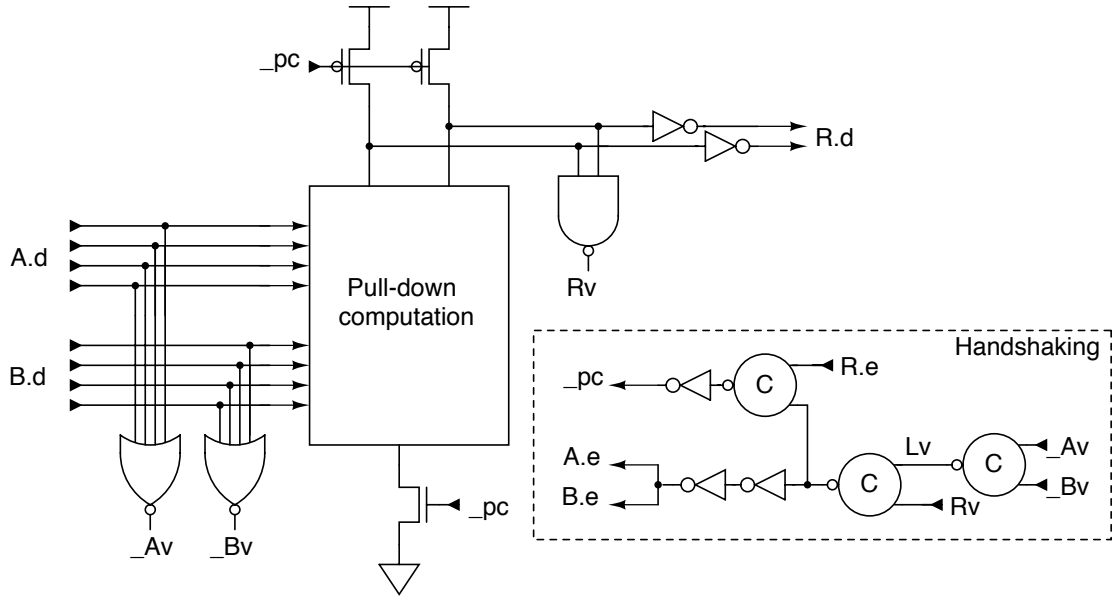


Figure 2.4: LUT process showing PCEVHB reshuffling

they would look something like:

$$(a.f \wedge b.f \wedge c.f \wedge d.f \wedge p[0]) \vee ... \vee (a.t \wedge b.t \wedge c.t \wedge d.t \wedge p[15]) \rightarrow r.t\uparrow$$

This implies the the use of five transistors in series, not including additional transistors that would be required for handshaking. As a rule of thumb, for high performance designs in modern CMOS technologies we limit transistor stacks to no more than 4 NMOS or 3 PMOS in series (hopefully fewer).

To implement the LUT process, I chose a precharged reshuffling, due to the complexity of the computation stack. If for example it was implemented as in the WCHB style, complementary logic would be required in the pullup stack, which would be far to expensive [92]. As an additional optimization, the LUT process is implemented using a PCEVHB reshuffling [44]. This design style, shown in Figure 2.4, allows us to use just one transistor instead of two in the pull down stack, and only one in the pullup, by placing $R.e$ and $L.e$ into a separate precharge computation.

Note: Figure 2.4 shows two inverters on the left acknowledges, since we never drive external channel signals using a dynamic staticized node like the output of a non-inverting C-element.

The second reduction in computation stack depth comes from converting the four 1of2 input channels into two 1of4 channels. A channel with 1of4 encoding uses the same number of wires to encode the data as two 1of2 channels (fewer if you consider the acknowledge signal), but only one of its wires switches per transition, saving energy. Despite this advantage, I use 1of4 data encoding only within the logic block, so that we can still support single-bit granularity in the FPGA routing. Rather than inserting a process to perform the translation, I simply modify the data on the wires and pass through the acknowledge signal using the circuit shown in Figure 2.5.

These two enhancements combine to give us manageable sized transistor stacks, with the final implementation shown in Figure 2.6. Note that the internal nodes in the pulldown tree (drain of the B transistors) are internally precharged high to avoid charge sharing issues (precharge transistors not shown).

The grey squares attached to the gates of transistors in the pulldown stack represent configuration memory, which can be set to zero or one to control the functionality of the LUT. Its implementation is discussed further in Section 2.3.3.

Hardware Arithmetic

LUTs can be used to implement any function, but this flexibility comes at a large cost in area and configuration memory (2^I bits are required for an I-input LUT).

Frequently, there are operations performed so often that it makes sense to in-

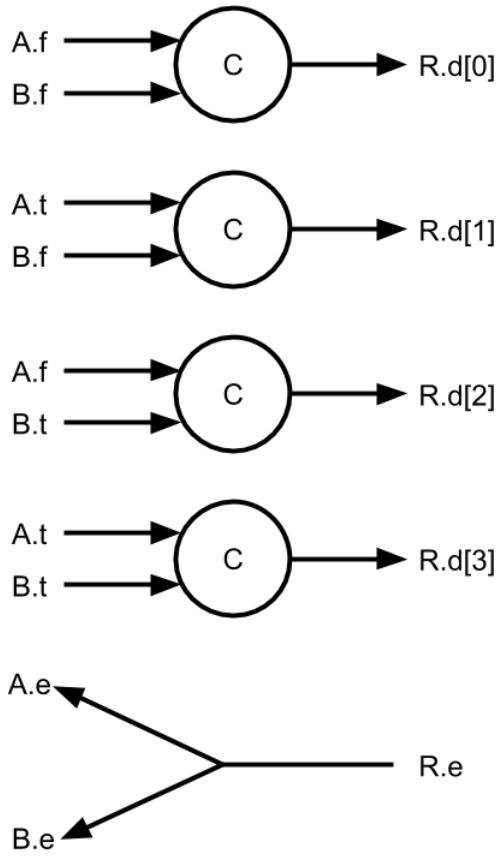


Figure 2.5: 2x 1of2 to 1of4 converter

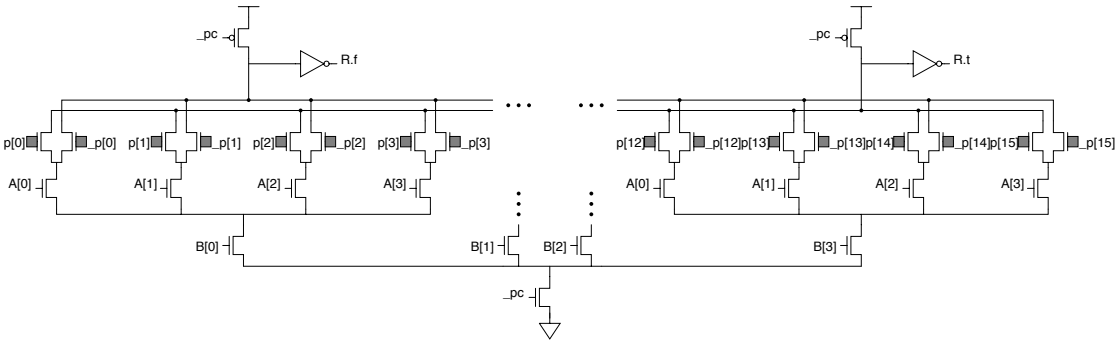


Figure 2.6: LUT pulldown computation stack

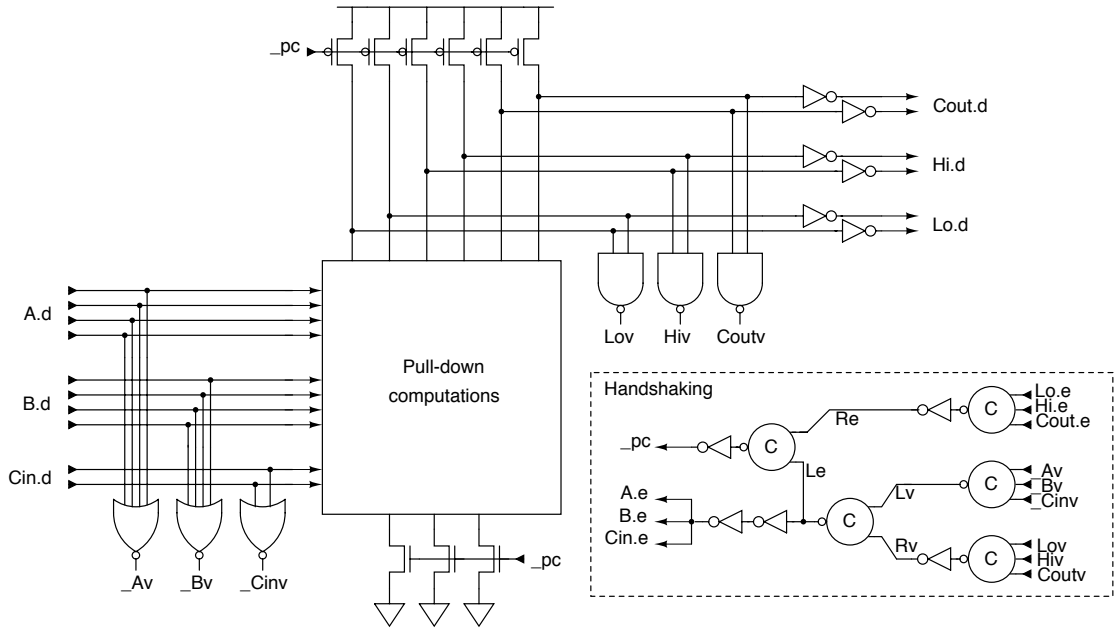
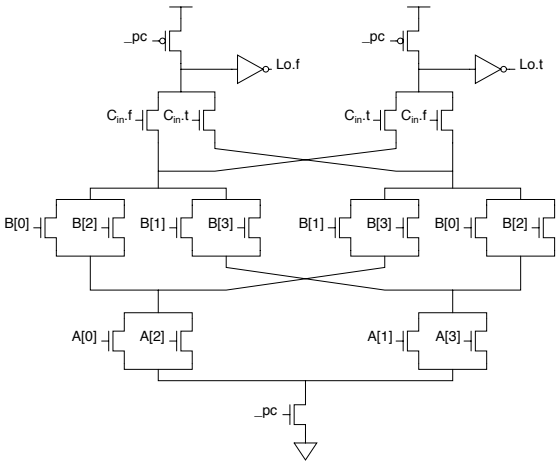


Figure 2.7: PCEHB two-bit adder process

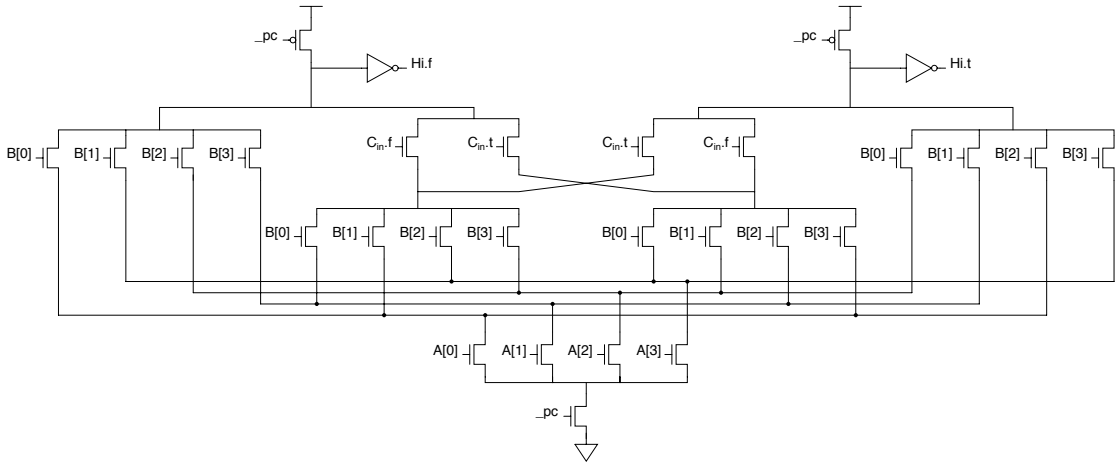
clude dedicated hardware to compute them. A classic example is addition/subtraction as seen in Figure 2.7 and Figure 2.8.

By implementing a hardware two-bit adder process instead of using two LUTs in series (plus dedicated carry logic), we use less area and perform the addition with lower latency than the LUT equivalent. We can also further optimize by adding quick direct routing paths for the carry chain, since this is the critical path for arithmetic operations.

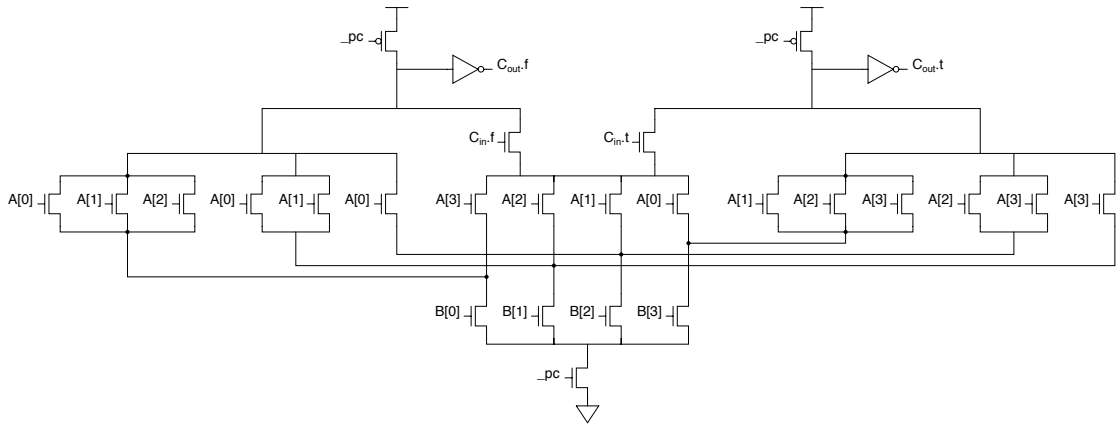
While it also possible to implement larger arithmetic functions (e.g. multipliers, DSP blocks), these typically do not fit into the tile context, and so should be considered separate hard macros. Due to the inherent modularity of asynchronous systems, it is easier to interface with other hardware in different regions such as these hard macros.



(a) Low bit



(b) High bit



(c) Carry Out bit

Figure 2.8: Two-bit adder pulldown computation stacks

2.3.2 Conditional Dataflow

The computation modules described in the previous section almost always have the same handshake behavior[1]. A function module waits until all of its inputs arrive, and then computes its outputs.

By contrast, there are two basic conditional dataflow operators that can change token flow based upon a control input. A dataflow *SPLIT* sends an input token to one of two outputs, based on a control input. A dataflow *MERGE* chooses one of two input tokens to pass to its output, based on a control input.

$$SPLIT \equiv *[C?c; A?x; [\neg c \rightarrow Y!x [] c \rightarrow Z!x]]$$

$$MERGE \equiv *[C?c; [\neg c \rightarrow A?x [] c \rightarrow B?x]; Y!x]$$

Teifel [142] introduced the idea of the conditional unit — a single process that can act as either a dataflow split or dataflow merge depending on a set of configuration bits. This sharing saves area by amortizing the complex circuitry required to implement the conditional handshakes.

I enhance this process by adding a third mode of operation, which I call *MUX* by analogy to a synchronous multiplexer. The *MUX* consumes both input tokens, then passes one to the output and discards the other based on control input value.

$$MUX \equiv *[C?c; [\neg c \rightarrow A?x, B?bucket [] c \rightarrow A?bucket, B?x]; Y!x]$$

One can select between the three possible behaviors using three configuration memory bits: S for *SPLIT*, Mg for *MERGE*, and Mx for *MUX*.

[1]It is possible to create e.g. an adder with early sum generation, that does not wait for the carry-in signal.

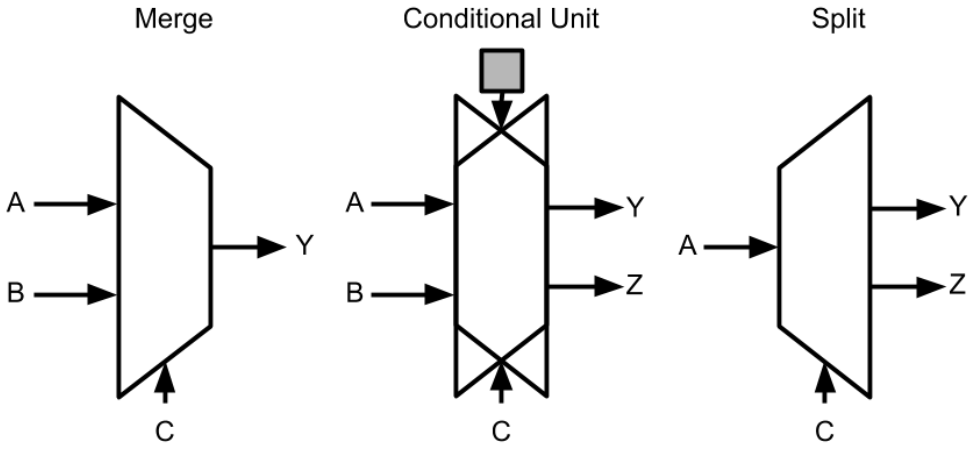
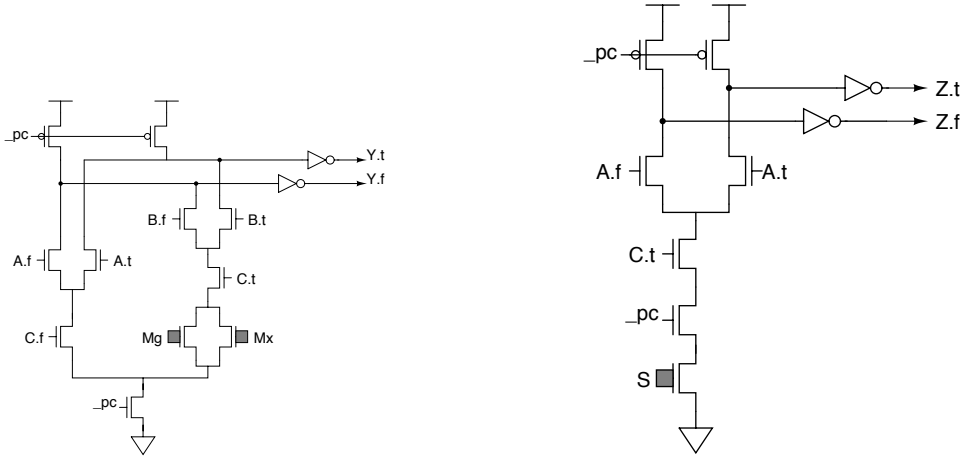


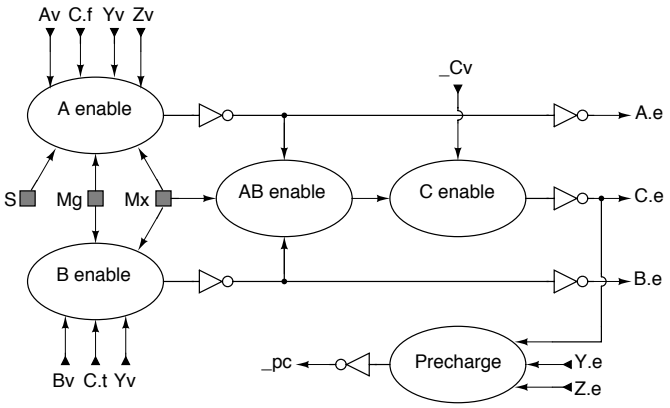
Figure 2.9: SPLIT, MERGE, and combined conditional unit



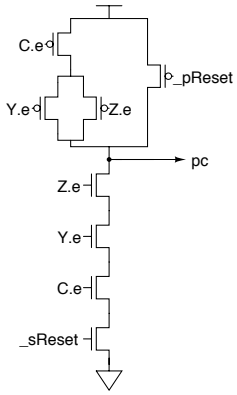
(a) Y output (b) Z output (only used in *SPLIT* mode)

Figure 2.10: Conditional unit pulldown computation stacks

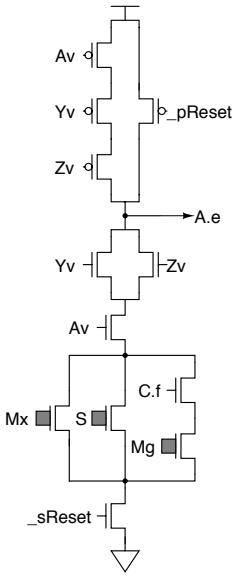
Unlike the *SPLIT* and *MERGE*, the *MUX* does not have conditional handshaking and its behavior could be implemented by a LUT. The conditional unit serves essentially the same role as the multiplexer within a Xilinx 'slice'. By using two LUTs whose outputs feed into a conditional unit configured as a MUX, one can effectively create a 5-input LUT as seen in Figure 2.12. This functionality is particularly attractive in the context of clustered logic blocks, as described in Section 2.4.



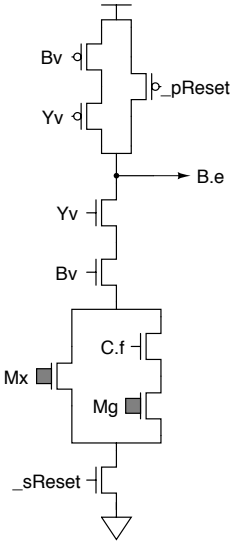
(a) Conditional unit handshaking structure



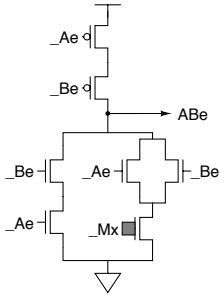
(b) Precharge for Y and Z



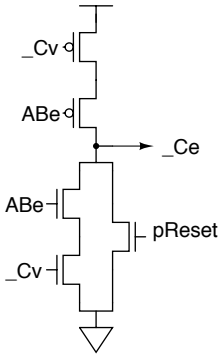
(c) Input A enable



(d) Input B enable



(e) Combine A and B enables



(f) Control input C enable

Figure 2.11: Conditional unit handshaking circuits

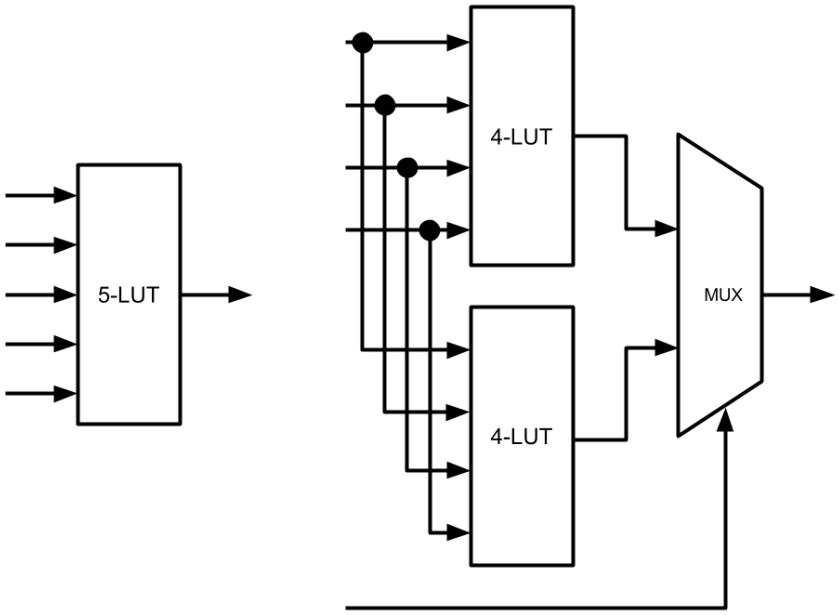


Figure 2.12: 5-input LUT created from 2 LUTs and a MUX.

2.3.3 Programmable Routing

The basic computation and conditional elements described above represent all the building blocks needed to implement any dataflow computation. All that remains is to connect their channels together to match the connectivity of the dataflow graph for the design. These connections are made within the routing fabric, a multitude of configurable switches organized to form a programmable interconnect. Unlike the conditional routing elements described below, the dataflow through these interconnect switches is usually statically set at configuration time and does not change while the FPGA is in use. The following subsections describe the most important pieces of the programmable interconnect, as well as how they differ from their synchronous counterparts.

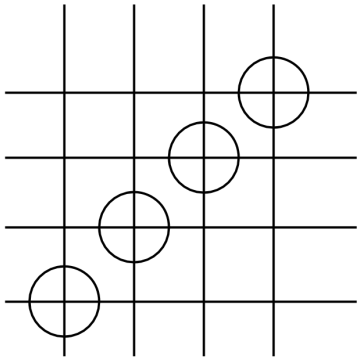


Figure 2.13: Disjoint switchbox with four switchpoints

Switch Box

Connections between mesh routing channels are made within the switch box. The example connection pattern shown in Figure 2.13 is known as a disjoint switch box, in which connections are always made between channels with the same track number. Each circle in the switch box is a switchpoint, which statically makes or breaks connections between routing channels based on how it is programmed.

Synchronous FPGAs often use another routing structure known as a connection box that taps routing channels midway to bring signals into and out of the logic block. This is possible because copying signals in synchronous logic can be implemented by simply fanning out the signal to multiple locations. By contrast, asynchronous channels are generally point-to-point links, and copying must be implemented using explicit token copy modules. As a result, my AFPGA architecture uses a unified switch box design that combines the functionality of the switch box and connection box in a synchronous FPGA, as shown in the next subsection.

For the same reason, longer routing tracks such as quads, hexes, and globals that are often used in synchronous FPGAs to widely distribute signals with

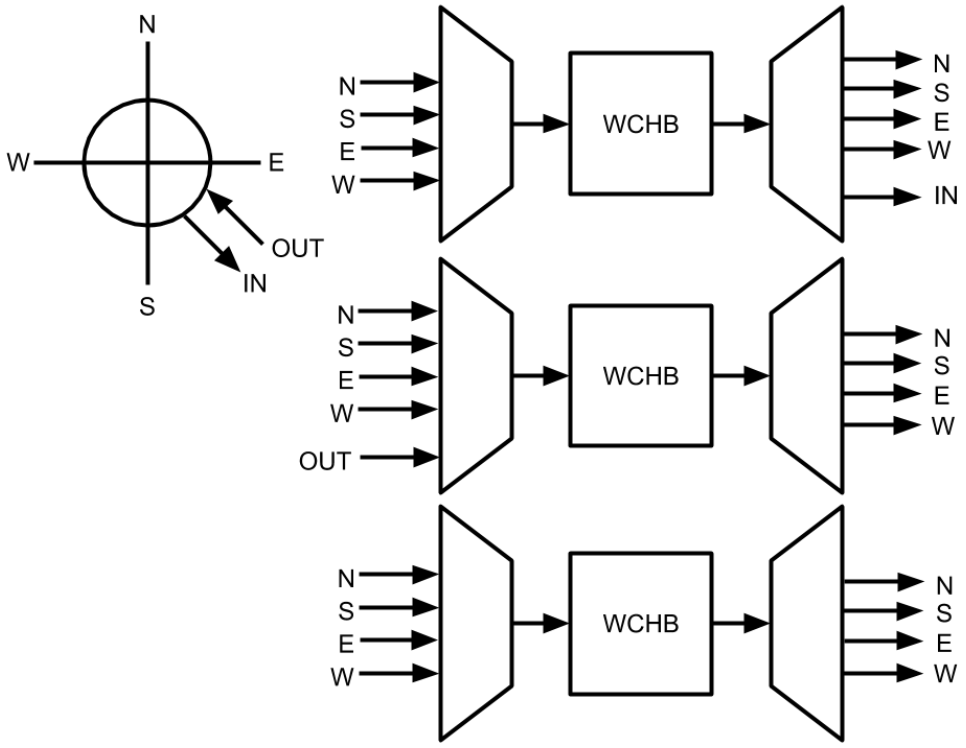


Figure 2.14: Switchpoint implementation

multiple fanouts are less useful in an AFPGA, since channels cannot be tapped midway without inserting an explicit copy. Longer routing hops may still be useful to limit the number of pipelined buffers on a long route, as further explored in Section 5.3.1.

Switchpoint Buffer

An example switchpoint is shown in Figure 2.14. In the abstract (shown at left), it makes connections between the routing channels in all four compass directions, as well as into and out of the logic cluster in the case of my unified switch box. In Section 4.2 I describe how to extend the routing fabric to three dimensions in an AFPGA system with die stacking.

Connections between the routing channels are made via statically configured passgate MUXes, as shown in the right half of Figure 2.14. A single route through the switchpoint consumes one of the MUX-buffer-MUX structures, so we need three to simultaneously implement all possible routes through a degree 6 switchpoint like the one shown. In the physical implementation, I remove one of the three buffers, sacrificing some routing flexibility to save $\frac{1}{3}$ of the area. Programmable interconnect is the dominant fraction of AFPGA area, as shown in Section 2.5. This is true in general for FPGAs, but especially critical for those with asynchronous communications because each single bit routing channel is composed of three wires rather than one. I present my efforts to reduce this overhead in Chapter 4.

Because the AFPGA is fabricated in a standard CMOS process without access to special ultra-low threshold voltage transistors or a separate gate-boosting voltage, I use fully-complementary passgates in my static routing. This prevents the leakage current that would otherwise result due to the threshold voltage drop through an NMOS-only passgate [27].

The major difference between asynchronous and synchronous FPGAs is that the AFPGA routing fabric can include pipelined buffer processes. In a synchronous FPGA the critical path is set by the longest route (which may make several hops through the unpipelined interconnect), but for the AFPGA each route segment constitutes a local handshake cycle. This fine-grained pipelining allows us to maintain high system throughput after designs are mapped to the AFPGA fabric, only limited by algorithmic dependencies as described in Section 5.3.1.

This optimization is possible because in slack elastic asynchronous systems, physical and logical pipelining are distinct. Adding or removing buffer stages may affect the performance of an asynchronous dataflow graph, but the number

of tokens in the graph does not change, so it will not change the correctness of the operation. By contrast, adding flip-flops on a route in a synchronous system implicitly adds data tokens, which results in signals arriving at their destinations misaligned in time, changing the meaning of the computation. It is possible to pipeline the interconnect in a synchronous FPGA, but this requires expensive banks of retiming registers and significantly complicates the mapping software [128, 129].

In the AFPGA switchpoint buffers, I use a simple WCHB process. This offers not only high speed, but also low area, which is important given the large fraction of FPGA die area already dedicated to programmable routing. My WCHB design uses a van Berkel-style C-element modified with a reset cutoff in the H-bridge. This allows us to keep only two transistors in the computation stacks (for high performance [42]), while also offering a neutral channel protocol at reset (which simplifies partial reconfiguration, described in Chapter 3).

Copying

As described above, fanout is not free in asynchronous dataflow graphs and must be implemented by explicit dataflow copy modules. I mitigate the impact of routing multiple copies of the same logical signal through the interconnect by using clustering, described in Section 2.4. Each clustered logic block has input copies that allow them to share logical inputs, as well as output copies, which allow them to route their outputs to multiple destinations. It is also possible to include copying elsewhere in the routing fabric (such as within a switchpoint buffer), at the cost of extra area and handshake cycle time.

Initial tokens

There must be at least one token in every loop in the dataflow graph. This is implemented in my AFPGA using an initial token buffer (show in Figure 2.2). This process starts execution holding a single token with programmable value, then acts as a simple buffer for the rest of the execution. Since each programmable initial token buffer can start with zero or one initial tokens, we can alter the number of tokens in a dataflow loop to match the algorithmic requirements. Section 5.3.1 has more details about choosing optimal token occupancy for performance within dataflow loops.

Configuration Memory

The configuration memory for my FPGA is implemented with static random access memory (SRAM). SRAM is also used for many synchronous FPGAs, and the selection criteria are exactly the same as for the AFPGA case. The key benefits of an SRAM configuration memory include the ability to reprogram at will, and ease of implementation in standard process technology. This quick reprogrammability is also crucial for partial reconfiguration, as described in Chapter 3. A drawback for SRAM memory is its volatility. Designs must be reloaded into configuration memory every time the device is powered down. Alternative memory structures that do not have this limitation, such as flash and antifuse, are also supported by my architecture.

Configuration memory is logically arranged in the same fashion as any normal memory array, with a collection of words accessed by intersecting wordlines and bitlines. This allows for random access reads and writes at the word level to the

configuration memory by providing an address and data. For bulk configuration of the FPGA in practice, one would likely wrap this base array with address generation logic to allow for DMA-style streaming configuration writes at some granularity, so that it is not necessary to provide an address for each word.

I implement full-swing bitline reads, rather than using sense amplifiers, since configuration reads are not the common case[2]. Once the array gets too big to effectively read/write, that region should be wrapped in a layer of hierarchy.

In its physical implementation, configuration memory cannot be arrayed as densely as it would be in e.g. cache memory. The simple reason for this is that every bit cell of the array must have an output that connects to some FPGA circuit, so none can be completely surrounded by other bit cells. I trade off density for routability in the physical implementation of the FPGA by interspersing a word (or two back to back) of SRAM with the FPGA logic they control.

Another consequence of having the configuration bits in close proximity to active circuits is that you must protect the internal state from being affected by capacitive coupling from fast switching wires nearby. I ensure this protection by 1) using an output inverter to protect the state node from coupling into the (potentially long) wire connecting the configuration bit to the FPGA logic it controls, and 2) disallowing routing channels that may be active from passing over the SRAM cells on low-level metal (below M4).

[2]FPGA configuration memory is perhaps the one context where a write-only memory is not an April Fools joke [130]

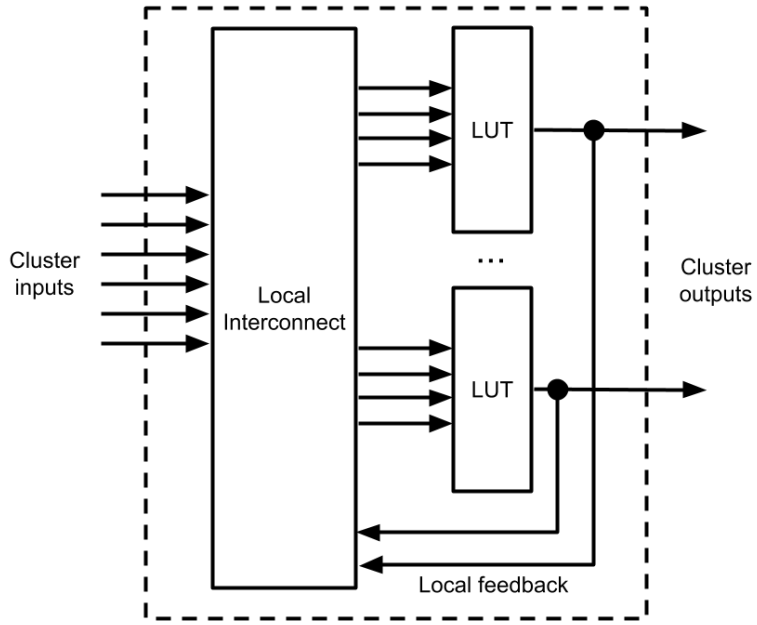


Figure 2.15: Clustered logic block with multiple LUTs

2.4 Clustered Logic Blocks

Clustering is a strategy for grouping related logic functions together in order to save resources or improve performance. Figure 2.15 shows an example cluster of LUTs. The cluster has some number of inputs, which are connected through local interconnect to the set of LUTs. LUT outputs become cluster outputs, and they are also fed back through the local interconnect to be optionally used as LUT inputs. Clusters may also contain more than just LUTs.

Most modern synchronous FPGAs use some form of clustering. Altera Stratix IV FPGAs group two LUTs together in an adaptive logic module (ALM), then cluster eight ALMs in a logic array block. Xilinx Virtex 5 FPGAs group four LUTs together in a slice, then cluster two slices in a configurable logic block [5].

2.4.1 Clustering for Area

A majority of FPGA area is interconnect, and clustering allows us to conserve global interconnect resources by grouping related logic. This is the equally valid for both synchronous and asynchronous FPGAs.

There are two main opportunities for clustering related logic. First, LUTs may share inputs. This allows the input to be routed from its source to the cluster and then distributed locally, instead of routing to two separate tile destinations. Second, the output of a LUT is generally connected to one or more LUTs. If we pack these destination LUTs together in a cluster with the source, the output net can become local feedback, which need not leave the cluster. Both of these use cases are demonstrated in Figure 2.16.

Clustering related logic can reduce routing pressure on the global interconnect. The challenge is finding a balance between the amount of local interconnect connectivity needed to implement sharing and the resulting global interconnect savings. There have been numerous studies into the optimal amount of clustering and input sharing for area efficiency [12, 13]. In general, both LUT size (number of inputs) and cluster size (number of LUTs) have increased over time as shrinking technology nodes permit more logic to be efficiently packed together.

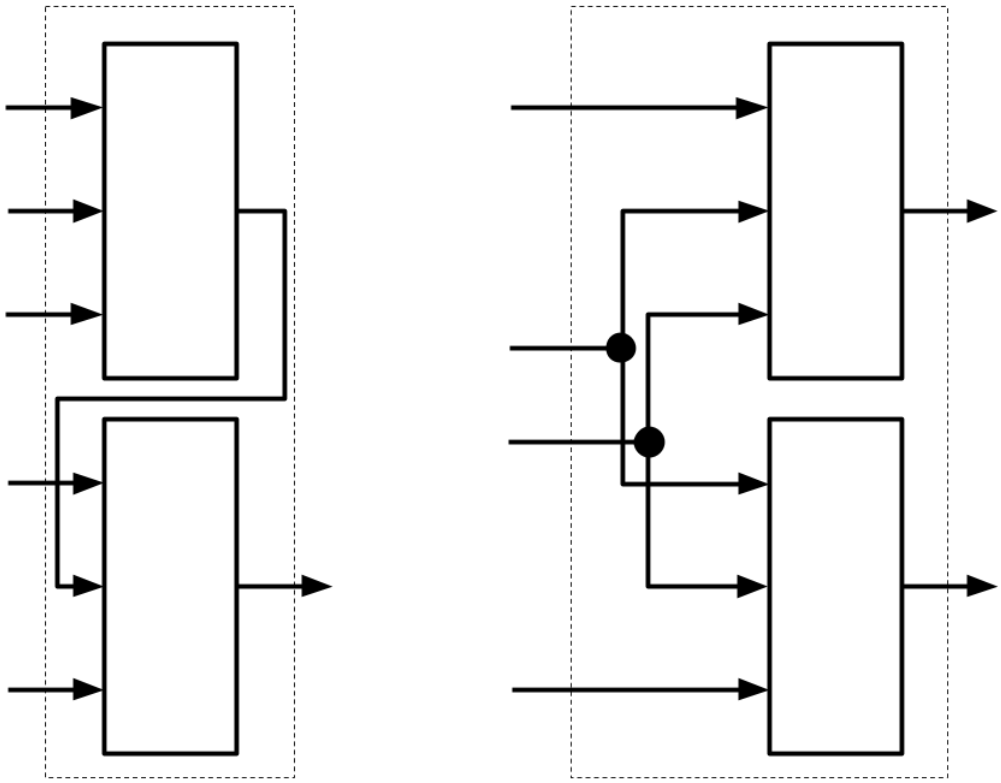


Figure 2.16: Examples of clustering to share inputs and clustering to share outputs.

2.4.2 Clustering for Performance

Synchronous

As described in Section 1.2, operating frequency in a synchronous FPGA is set by the worst case delay of the critical path: the longest path that exists between any two flip-flops in the circuit.

Because the interconnect is unpipelined, longer paths through global interconnect increase this delay. Synchronous FPGAs attempt to optimize critical path delay by providing routing resources of various lengths (e.g. singles, doubles, hexes) so that destination of nets various distances away can be reached efficiently.

In order to reduce the critical path delay, it helps to pack related logic into a single cluster. This way, the flop-flop path can be shorter, since it does not need to leave the cluster.

Asynchronous

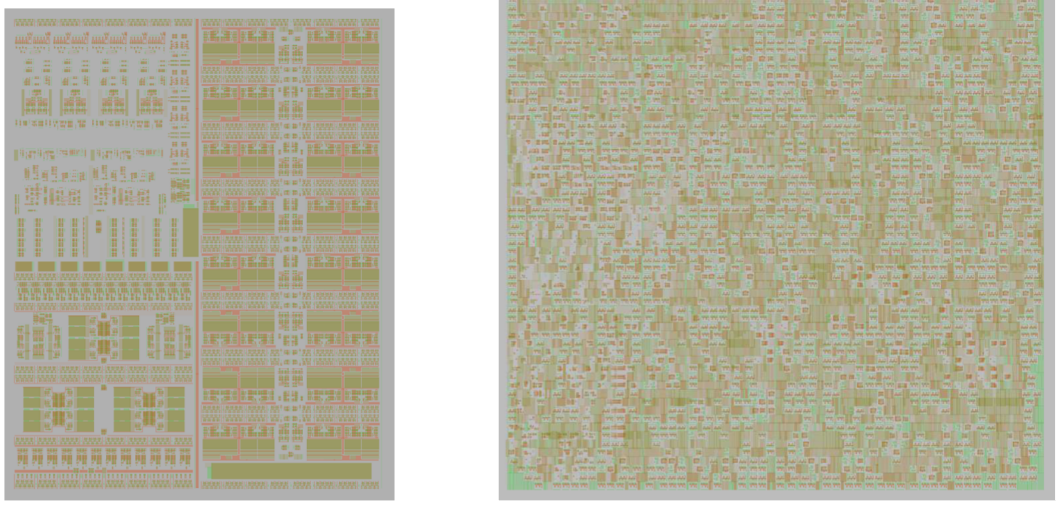
By contrast, in my asynchronous FPGA, the interconnect is pipelined and elastic. As a result, the maximum throughput is not constrained by a critical path between logic blocks, and even long routes through the global interconnect can operate at maximum throughput.

Instead, benchmark performance for designs mapped to an asynchronous FPGA is driven by data dependencies in the dataflow graph. Specifically, these manifest in the AFPGA as token loops and reconvergent split-join paths. We can affect the performance of the design by altering the amount of slack, or buffer stages, on these paths within the dataflow graph. A major benefit of clustering in the AFPGA is that it allows there to be lower slack on dataflow token loops. Asynchronous performance modeling and strategies for clustering are discussed more fully in Chapter 5.

2.5 Measured Results

I have fabricated the AFPGA architecture described above in several technology nodes. Figure 2.17 shows the layout for two versions of a single AFPGA tile.

Figure 2.17a is a full custom implementation in 65 nm technology. The structure of the tile is clearly visible even at this scale, with the repetitive switch box



(a) Full custom, 125x160 μm^2 in 65nm (b) Obfuscated, 485x485 μm^2 in 130nm

Figure 2.17: Chip layout for a single FPGA tile

on the right side and the less dense logic core on the right.

Figure 2.17b is a test of an automated obfuscated layout system [63], implemented in 130nm technology.

2.5.1 Throughput

The 130nm FPGA operates at a peak throughput of 340 MHz. The 65nm full custom FPGA operates at approximately 800 MHz peak.

For context, the commercial Virtex 5 FPGA boasts a maximum operating frequency of 550 MHz in an FPGA-optimized 65nm technology, while the competitor Altera Stratix III claims a peak of 530 MHz.

The throughput of the asynchronous FPGA is largely limited by the capacitance of the programmable interconnect. For instance, in the 65nm FPGA the

Logical unit	Area (μm^2)	Fraction of total area
Switchbox	5,544	28%
Logic core	4,815	24%
Config memory	4,391	22%
Crossbar	2,165	11%
Other buffers	1,070	5%
Output copy	846	4%
Power gating	751	4%
Partial reconfiguration	370	2%

Table 2.1: Area breakdown for a single FPGA tile

LUT can actually perform at over 1.3 GHz, and a bare switchpoint buffer has a peak throughput above 2.7 GHz. The diminished results echo the sentiment in Section 1.3 about the performance differences between FPGA, ASIC, and custom implentations.

2.5.2 Area

The area of each part of the tile is broken out in Table 2.1 and summarized in Figure 2.18. As is typical for FPGAs, interconnect dominates the total area. The actual logic cluster accounts for about a quarter of the tile area, and the configuration memory is approximately another quarter. Implementing support for partial reconfiguration (described in Chapter 3) consumes less than 2% of the total tile area.

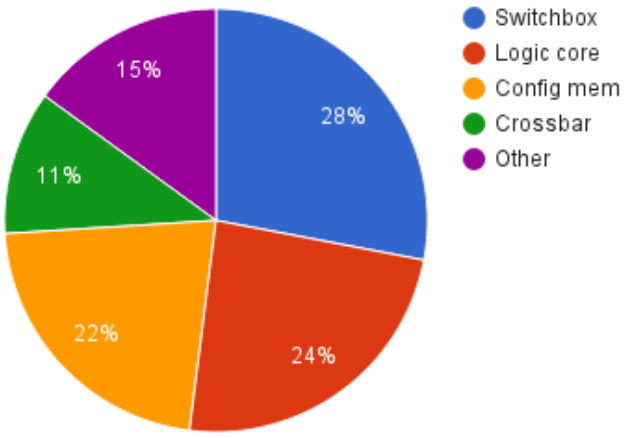


Figure 2.18: FPGA area by category

CHAPTER 3

DYNAMIC PARTIAL RECONFIGURATION

3.1 Introduction

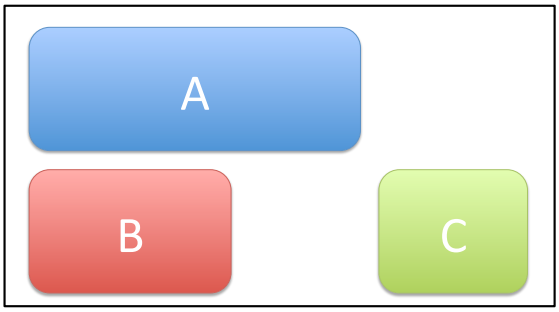
Our long-range aspiration informing this work is a model of virtual infinite hardware for on-demand accelerators. An important piece of this goal is developing a high-performance accelerator substrate that can be reconfigured on the fly to serve a new purpose.

Reconfiguration simply means that an FPGA's programming can be changed more than once. This is a property of all FPGAs with reprogrammable configuration memory (i.e. SRAM, flash). It is possible for an FPGA to be programmable but not reconfigurable (e.g. anti-fuse).

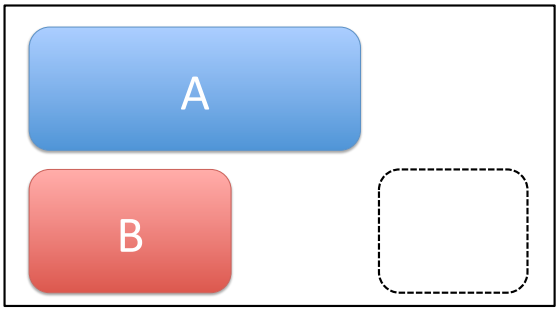
Partial reconfiguration is the ability to change the functionality of one portion of the FPGA without reprogramming the entire device. The system may be halted to do so, but the salient feature is only the regions of the FPGA being replaced are modified.

Dynamic partial reconfiguration (DPR) adds the ability to perform this reprogramming while other parts of the device remain active. Note that when we use the term dynamic reconfiguration, partial reconfiguration is also implied. If the entire FPGA is being reprogrammed, there are no other parts to remain active. Dynamic reconfiguration is also sometimes known as run-time reconfiguration.

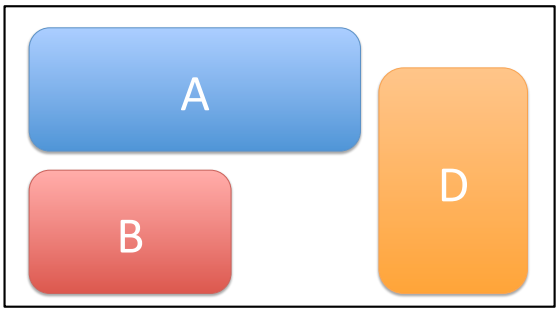
The benefit of partial reconfiguration is its ability to let you "virtualize" hardware by reusing it in time, which allows for more functionality in a fixed size FPGA.



(a) FPGA configuration with three distinct modules in operation



(b) Module C completes its operation and is no longer needed



(c) Module D replaces module C. If we only reconfigure the resources needed by D, it is partial reconfiguration. If A and B continue to operate during the reprogramming, it is dynamic reconfiguration.

Figure 3.1: Partial reconfiguration operations

It also allows for shorter configuration times and smaller configuration bitstreams.

Partial reconfiguration has been implemented before in synchronous FPGAs, and is supported in some modern FPGAs [82]. In the next subsection, we will discuss some past efforts in this area. Despite the existence of some hardware support however, partial reconfiguration remains far from common.

Why isn't partial reconfiguration more common? Part of the answer is that de-

signing modular interoperable synchronous systems is difficult. Adding or changing modules requires either retiming (local changes can have global impacts), starting with worst-case performance assumptions, or inserting clock domain crossings. There is also the issue of software support. The system needs to compute and store the set of all partial configurations, which may be tied to a particular physical location in the FPGA.

Our asynchronous FPGA may offer a way forward for partially reconfigurable systems. Self-timed designs feature inherent modularity and resilience due to their of lack of timing assumptions, which allows us to make local changes without affecting the correctness of global operations.

In this chapter, we discuss the challenges associated with adding support for dynamic partial reconfiguration to an asynchronous FPGA. We present a low cost hardware implementation, atop which multiple reconfiguration schemes (as well as other desirable architectural features) can be built.

3.1.1 Related Work

Research in this area has largely focused on implementing partial reconfiguration on top of commercial FPGAs. Koch [82] published an excellent survey of the history and state of the art of FPGA partial reconfiguration. Xilinx, Altera, Lattice, and Atmel have all commercial released products supporting some degree of dynamic partial reconfiguration [40].

As an example, support for dynamic reconfiguration on Xilinx FPGAs historically went through at least four phases [6]:

1. Early devices such as the XC6200 added the hardware capability for partial reconfiguration, but lack of tool support and a less-compelling FPGA architecture limited adoption
2. Researchers made efforts to implement partial reconfiguration on more "mainstream" architectures, resulting in tools such as JBits
3. FPGA vendors begin to add support for partial reconfiguration within their existing software design flow, but with large designer effort to insert constraints such as bus macros
4. Partial reconfiguration begins to be more seamlessly integrated into the design flow. Commercial products are released integrating microprocessors on die with FPGAs, resulting in lower barriers for implementing reconfigurable computing systems.

Researchers were able to use these platforms to implement a wide variety of applications using partial reconfiguration, including Software Defined Radio [107], reprogrammable automotive cabin functions [9], computer vision systems [16], encryption systems, network switches and packet processors [156].

At each stage of technology maturation, researchers attempting to implement DPR pushed the boundaries to beyond what was supposed to be possible. For example, [62,125] implemented 2D modular partial reconfiguration on earlier Xilinx Virtex-II FPGAs that did not support it by using a read-modify-writeback strategy. Systems such as Wires on Demand [6] and ReCoBus [83] implement reconfigurable modules on top of commercial FPGAs by using structured communication channels with plug-in sockets for logic. These systems raise the level of FPGA abstraction to free designers from manually dealing with issues of floorplanning and bus macros, at the cost of some flexibility. Our architecture provides a clean foundation that

makes it easy to support multiple types of DPR systems, without jumping through these types of implementation hoops.

A closely related work is APL, a dynamically reconfigurable asynchronous FPGA designed by Jia et al. [70]. This system uses globally asynchronous, locally synchronous architecture, with multiple traditional synchronous logic regions communicating via four-phase handshakes. APL is able to disable the timing cells that control communications for a reconfigurable region during programming using an array addressing scheme, so that a timing region undergoing reconfiguration cannot affect other regions. APL is an interesting design point due to the potential for using existing synchronous FPGA mapping tools to program the regions. Our architecture uses asynchronous logic throughout, and does not require predefined fixed sized regions.

3.1.2 Contributions

In this chapter, we:

- Explain the requirements for implementing dynamic partial reconfiguration (Section 3.2)
- Design and fabricate mechanisms supporting flexible, fine-grained DPR in AFPGAs, with low area cost and no performance degradation (Section 3.2)
- Build additional FPGA architectural enhancements such as power gating (Section 3.3) using the same framework

3.2 Requirements for Dynamic Partial Reconfiguration

There are four main questions one must answer when considering the dynamic and partial reconfiguration process:

- When is it safe to reconfigure?
- How do we maintain the state of a computation when we do?
- How do we protect other regions when a section is replaced?
- On what granularity is it possible to reconfigure the FPGA, and how can we manage the process?

We will discuss each of these points in order below, with special attention paid to origins of the underlying issue, the solution we have deployed as related to other possible approaches, and the equivalent concepts in the synchronous domain.

3.2.1 Empty Pipeline Detection

In traditional reconfigurable hardware, we accept that all data currently residing on the FPGA will be lost during reconfiguration. Since the entirety of the device is reconfigured simultaneously, this assumption presents no real problem. In the case of dynamic and partial reconfiguration, however, we need to take more care to assure that a module is not reconfigured while still in use.

When is it safe to reconfigure? When performing partial reconfiguration of a system, we need to ensure that we do not remove a section of the system that is still in use. "Partial" is the problem here — not "dynamic."

It's important to note that this problem also exists even in regular FPGA systems. In general, we rarely consider whether a particular portion of the FPGA fabric is still in use, because the FPGA computation is typically intentionally infinite and/or has predictable latency. The additional complications here center on the fact that we would like to perform reconfiguration with finer granularity both in space (multiple active regions) and in time (swapped in and out frequently).

The reconfiguration process is an 'out of band' operation (i.e., it uses channels and mechanisms entirely separate from and overlaid on top of the normal FPGA data communication channels) that alters the structure of the computation without any regard for the internal state. If we want to use it safely, we must externally guarantee that it is safe to reconfigure the system. By 'safe,' we mean that the computation is complete (data is no longer in flight), and all results of the computation have propagated somewhere outside of the region that will be replaced.

To do this, we need to know when a computation is complete. In the general case, this question is impossible to answer (halting problem). Fortunately, in most reasonable systems, we can assume and leverage some regularity in the computation – bounded latency and/or a known relationship between number of inputs and outputs.

If we assume a simple model of the asynchronous FPGA computation where inputs are processed and generate outputs, then the problem of determining whether a computation is complete reduces to empty pipeline detection. There are several possible mechanisms for ensuring that the region to be reconfigured is empty.

Timing

The first and simplest involves a timing assumption: we assume that there exists predictable and bounded latency for each operation taking place in the reconfigured region. In this way, any computation takes a known amount of time to complete, and so one need only wait for some set period of time after introducing the last set of inputs before initiating reconfiguration. In theory, this assumption is a poor one due to the halting problem. In practice, we can leverage the (presumably) known function of the reconfigured block to make reasonable estimates about its execution time.

This is likely the mechanism of choice for synchronous systems: if a given operation takes ten clock cycles to complete, it is safe to reconfigure after that much time has elapsed since the last set of inputs were introduced. Note that a timing-based approach is a valid option even for asynchronous circuits. The latency of an operation in an asynchronous system may be data-dependent, or it may change based on process, voltage, and temperature variation, but it is seldom random or unpredictable. This assurance is the basis for an entire class of self-timed circuit design (bundled data).

Environment tracks dependencies

If we are unwilling or unable to use timing information to determine when a reconfigurable module is quiescent, the next simplest approach is for the environment (i.e. the other parts of the system connected to and communicating with the reconfigurable module) to detect when the module is inactive. There is often dependency tracking elsewhere in the system, and this can be leveraged to monitor

the reconfigurable module.

In order for the environment to be able to make this determination, there must be some predictable communication pattern. For example, an encryption block has a simple deterministic relationship between input and outputs: one message in yields exactly one encrypted message out. The deterministic dataflow architecture of our asynchronous FPGA is ideally suited to this sort of token counting approach.

Not every module will have a convenient external heartbeat that can be derived simply from its communication patterns. In these cases, it is often possible to augment the module design to generate a `DONE` signal specifically for use in the context of a dynamically reconfigured system.

Hardware dependency tracking

It is also possible to add dedicated hardware to perform empty pipeline detection. By instrumenting channels entering and exiting the reconfigurable region, we can implement a credit-based token counting scheme to track when a module is in use [45, 114].

This solution in essence performs the same task as the environment-based tracking above, but less flexibly, since one must choose where the tracking hardware is placed. In the FPGA context, this extra hardware might be added on-demand in the fabric.

Hardware probe

In an even more hardware-intensive solution to empty pipeline detection, one could simply inspect the internal state of the region about to undergo reconfiguration. Whereas in the counter-based approach we need only inspect a limited number of locations (e.g., input/output channels), here inspection takes on a much more global flavor. Several equally invasive methods are well suited to this approach, including direct probes of the region and current detection.

We can tell when a given process is active by inspecting the state of its communication channels. We could insert a small amount of logic to watch the process and output a signal when it quiescent. This is analogous to a hardware version of the probe operator [104] — including the fact that as a negated probe it is unstable. If we have a method of detecting when a single process is not in use, we could detect that our computation is complete using the brute force approach of inspecting every process.

Obviously the hardware overhead of such an approach makes it completely infeasible for as system with a huge number of fine-grained processes like the AFPGA. Similar in spirit, however, is the idea of current-sensing completion detection [55]. This scheme measures the current consumed as a process communicates, and when it drops below a certain threshold the process can be assumed to be idle.

Preferred solution

We can perform all these forms of dependency tracking in the environment in an FPGA context except for the hardware instrumentation. Extra hardware is only needed and useful when computations take an unpredictable amount of time *and*

do not reliably produce a result (i.e., there is an unpredictable relationship between inputs and outputs). This is certainly possible (e.g. deep packet inspection where packets may be dropped based on a complex ruleset), but not the common case. Further, in these systems one can generally add a heartbeat to the design as described above. So, our preferred implementation is to omit hardware instrumentation and leave dependency tracking to the environment.

3.2.2 Preserving Internal State

How do we preserve internal state after reconfiguring? In asynchronous systems state is everywhere. It exists in the tokens within the dataflow graph, which in general are not guaranteed to be at any given place at a set time.

By contrast, in a synchronous system all state information is guaranteed to always be in registers at the end of a clock cycle. In this way, the synchronous case represents a more limited set of resources one needs to preserve when reconfiguring.

We have the ability in our architecture to replace internal state using initial token buffers (ITBs). This process is even inexpensive — you can modify an existing configuration to change its initial state by simply flipping a small number of bits that control the ITB state. The difficult part is actually measuring what the state is at any given point so that it can be restored.

If you want to preserve state while replacing a piece of the system, you can choose not to replace the piece that contains the tokens representing the state. The problem only arises when you want to remove and replace an entire system, then put it back later with the same internal state.

If the internal state replacement problem cannot be avoided in system design, previous work in asynchronous testability offers a solution. The design objective here is nearly the same — adding the ability to inspect the internal state of a dataflow graph at a given point in time. This can be accomplished by inserting multiplexers [138], a partial scan chain [78], or by adding a synchronous scannable mode to the circuits [146]. All of these solutions come at the cost of additional hardware, so it makes sense to deploy them sparingly, but they do include the additional benefits of increasing testability.

3.2.3 I/O Boundary Protection

In a system with dynamic partial reconfiguration, the region being replaced communicates with the other fixed active regions before and/or after the reconfiguration process. If the boundaries of the fixed region are not protected, errors may be introduced during the reconfiguration process. For instance, data out of the fixed region may be lost through a "dangling" channel or spurious data may enter the fixed region through a floating input channel. We must ensure that the region being replaced does not impact the regions that are not (this is specifically a problem for 'dynamic').

Early versions of the Xilinx synthesis flow used manually-placed 'Bus Macros' which enforced the separation between static and reconfigurable regions. The latest versions automate this process by placing 1-LUTs as proxy logic, but the user must still place enabled registers on partition pins between reconfiguration boundaries, to prevent corruption of adjacent regions during reconfiguration [156].

In systems with delay-insensitive communication channels, there is a specific

handshake protocol that must be followed (Section 1.2). If there are glitches or unintended transitions on any of the channel wires, tokens may be inadvertently created or destroyed.

Asynchronous handshakes do provide a particular advantage for boundary protection, in that they can simply rely on the delay insensitivity of the channels. It is possible to "pause" a handshake on a channel entering the reconfigurable region by not simply not acknowledging its communication, and you can take as long as you need in this state to reconfigure the region.

We take advantage of this feature of asynchronous channels to implement a low-cost boundary protection system for partial reconfiguration. Once the region to be replaced is empty (Section 3.2.1), it is held in reset. At this point, it is safe to remove and replace the reconfigurable region. While they remain in reset, all channels entering and exiting the reconfigured region are held in their neutral state. Outgoing channels will not inject data into the static region, and incoming channels will not acknowledge any data they may receive, pausing all incoming data tokens at the border of the reconfigurable region. Once the reconfigurable region has been reprogrammed, it is released from reset and communications with the static region pick up where they left off.

This system allows for reconfigurable boundaries to be placed anywhere, without requiring any special interface circuitry. All that is needed is the ability to hold the reconfigurable region in reset independent of the static region, a topic we discuss in the next subsection. Note that if there are channels that pass between regions both before and after reconfiguration, it is still necessary for those channels to be aligned physically. The advantage is that these ports can be placed anywhere at configuration time, rather than in a discrete set of locations. Further,

if the ports of two modules do not align (e.g. they are pre-compiled IP blocks), it is easy to insert a simple shim route due to the delay insensitive nature of the asynchronous interconnect.

3.2.4 Region Control

On what granularity is it possible to reconfigure regions?

On one extreme, the granularity is the entire FPGA. This is the normal case, with no partial reconfiguration.

On the opposite extreme, it is possible to modify even a single bit of the configuration memory. One bit is probably not that useful (except possibly in a LUT function), since you usually want to modify a related cluster based on functionality.

The first limiting factor is the granularity with which you can modify the configuration memory. Our configuration memory architecture permits random access writes at a word level. Furthermore, the SRAM memory in our design features non-destructive write operation: if a bit is overwritten with the same value it currently holds, there will not be a glitch on its output. This allows for sub-word configuration memory writes, making it possible to modify the configuration even a single bit at a time.

In our AFPGA design, we can change the configuration of any region subset. The limiting factor is that as described in the precious subsection, we need to hold the replaced region (and only that region) in reset when we do.

Since state is everywhere, asynchronous circuits require a reset signal when they begin operation to put them in a known state. The same issue exists in

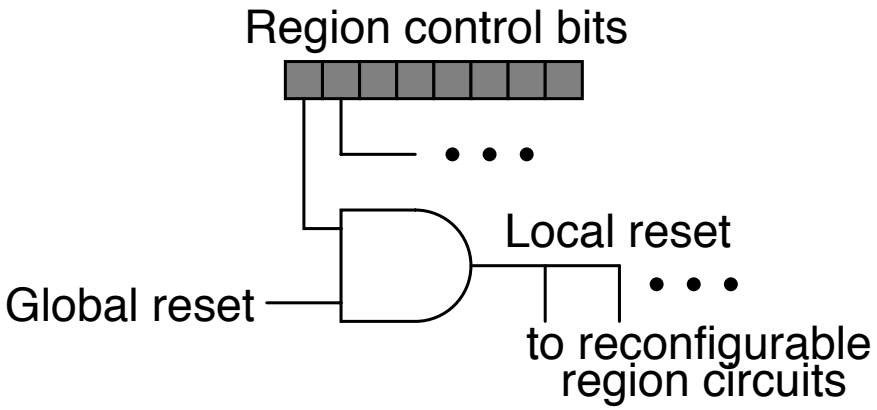


Figure 3.2: Reset behavior for reconfigurable regions is controlled using configuration memory

synchronous circuits, but again the state is limited in where it can appear, so resetting e.g. all the flip flops is sufficient. Aside: through reset discipline you can avoid putting reset to *every* process, but you need it at least in some places (and definitely at borders of reconfigurable regions as discussed in the previous section).

Partial reconfiguration regions in our design are managed using additional configuration memory bits — one per region. These extra bits are used to control reset behavior in their region, as shown in Figure 3.2. This system lets us flexibly implement any granularity we desire, by simply spending more memory bits to achieve a higher granularity.

For example, the architecture presented in the previous chapter uses extremely fine-grained reconfiguration regions. The logic cluster (and associated local interconnect) form one region, while routes in the switchbox are in independent regions. This sub-tile granularity allows for routes through a single switchbox to belong to separate reconfigurable regions, offering maximum flexibility when placing and

routing these regions. By contrast, the smallest possible possible reconfigurable region in a Xilinx Virtex 7 FPGA is the configuration frame, which captures an entire clock region and contains 50 CLBs [156]. Despite the fine granularity, the hardware cost of implementing reconfiguration regions was less than 2% of the total tile area.

There are other possible implementations of asynchronous reconfigurable regions. For example, we could predefine reconfiguration regions as in commercial FPGAs and use a separate global reset signal for each. Unfortunately, such a system unnecessarily constrains the boundaries of each region, while increasing wire density, pin count, and top-level control complexity. The advantage of our implementation is that it reuses the existing configuration memory mechanism to allow regions of nearly any description to be constructed with minimal hardware cost. Its only drawback versus a static region system is that it requires an extra configuration memory write to the region control bits before a partial reconfiguration event.

3.3 Power Gating

Asynchronous circuits are data-driven by construction: they are quiescent whenever they are not processing tokens. Because there is no switching activity - the equivalent of perfect clock gating in the synchronous world - their dynamic power is zero.

Unfortunately, in modern technologies leakage power has become a growing concern. We can reduce leakage power by power gating inactive circuits. Our AFPGA architecture has selectable power gating regions, so that each tile can be

in one of three states:

- Fully active - logic in tile in use
- Passthrough - logic disabled, but routes pass through the global interconnect within tile
- Shutdown - everything including configuration memory gated

The same hardware mechanism (extra configuration bits) that controls reconfiguration regions can also be used to control power gating. This consumes a very small amount of area, as shown in Section 2.5. In fact in our custom layout implementation, these structures fit into existing white space in the tile (though their area was still counted in the reported totals).

An additional benefit of power gating in FPGAs is preventing contention from illegal configurations before the device is completely programmed. This contention is caused by multiple drivers attempting to drive a net to different voltages, which leads to short circuits and possibly even device damage. Our FPGA architecture with power gating solves the driver contention issue by placing all circuits in a gated state at power up or device reset.

Even the configuration memory can be powered down in our architecture. This is accomplished by designating one 'privileged' word (the same as was used for partial reconfiguration region control). This privileged word contains bits that control power gating to the different regions of the tile. The privilege word is always powered, and resets to a known state.

CHAPTER 4

3D FPGA ARCHITECTURE

4.1 Introduction

Integrated circuit manufacturing has recently expanded into the third dimension. It is now possible to stack multiple CMOS die atop one another, using Through Silicon Vias (TSVs) to connections between the die layers [8, 17].

Three dimensional integration offers several attractive benefits to an FPGA architecture. The extra dimension reduces the effective diameter of the interconnection network, providing a shorter average distance and lower propagation delay between circuit regions. Additionally, the fact that logic has more immediate neighbors (up and down, in addition to the usual planar compass directions) increases the routing flexibility possible [4]. Finally, 3D stacking allows for the same amount of logic to be fabricated using multiple die, each of which are smaller in area. This helps to increase manufacturing yield, which is especially important given that FPGAs are some of the largest chips currently produced today.

Vertical stacking is not an unqualified success, however. Current TSVs are a scarce resource compared to normal vias due to their much larger size, so we must use the 3D interconnect wisely. Stacking multiple die fabricated separately can introduce additional variability due to process, voltage, and temperature, which complicates timing closure in synchronous systems [3]. Adding more layers of active logic within a fixed package can also lead to heat dissipation issues [52].

Asynchronous systems are excellent candidates for 3D integration, because they do not need to synchronize clock domains across multiple die tiers. Delay-

insensitivity allows them to gracefully tolerate the different electrical environment between planar and TSV routes.

As we saw in Chapter 2, a significant fraction of FPGA area is devoted to interconnect. In asynchronous FPGAs the problem is multiplied due to the increased wiring needed to implement handshaking channels (Section 1.2). Ideally we would find a way to reduce this interconnect penalty, which is especially critical for 3D communication.

This chapter describes how our asynchronous FPGA (AFPGA) architecture can be extended to 3D (Section 4.2), explains the need for efficient signaling protocols (Section 4.3), describe available single-bit signaling protocols including our own novel design (Section 4.4), and evaluate their suitability in both 3D and planar contexts.

4.1.1 Related Work

The benefits of 3D integration to FPGAs were first articulated in the mid-1990s. Depreitere et al. [39] designed an "optoelectronic" 3D system. It was built from a collection of 2D FPGAs (each similar in design to Triptych), connected in 3D at a printed circuit board level using free-space optical links. Another early 3D FPGA proposal used multiple 2D FPGA tiers as multi-chip modules, directly connected with solder bumps [4]. These stacking technologies allow for only coarse integration with limited connectivity between tiers, with a wire pitch of 200 μm or greater.

The achievable bandwidth between 3D tiers was greatly increased after the development of Through Silicon Via fabrication technology. Rothko [111] is the first 3D FPGA architecture to use TSVs. Its architecture was also based on the

Triptych sea-of-gates architecture, extended with vertical connections in each cell. A later version of this design was extended to support dynamic partial reconfiguration, saving state between configuration events in special registers [30] similar to the SCORE architecture concept.

Cevrero et al. [22] designed another interesting FPGA architecture using TSVs, creating a multicontext system by stacking DRAM on top of a planar FPGA. Multicontext FPGAs have multiple parallel sets of configuration memory, and are able to switch between these "personalities" rapidly to permit time sharing of the FPGA fabric. In general stacking configuration memory on a separate 3D tier makes little sense, since current TSVs are as large or larger than an individual SRAM cell, but the multicontext application takes good advantage of the greater density of DRAM vs SRAM to effectively multiplex the TSVs in time.

Although smaller than solder bumps and other packaging technologies, TSVs are still much larger than normal interconnect vias. Monolithically stacked FPGA architectures [91] avoid the TSV penalty altogether by assuming new active devices can be fabricated vertically interspersed with interconnect metal layers. Devices fabricated within these interconnect layers may be degraded (e.g. N-type transistors only), but it is possible to work around these limitations by keeping active logic on the bottom (normal CMOS) tier and only placing programmable routing and memory on upper tiers. Such an architecture is likely limited to only one logic tier, and so is perhaps more properly considered a more efficient form of 2D FPGA.

As part of their latest high-end series, Xilinx currently offers "3D FPGAs". These are implemented as separate die stacked on a silicon interposer, and are more accurately termed 2.5D. They share the benefits of smaller die size and thus

better yield, but not the shorter average interconnect diameter that comes with a true 3D FPGA.

The closest related work to the architecture described below is the three-tier asynchronous FPGA created by Fang et al. [46]. It is constructed from dataflow elements like the 2D AFPGA in [141], but expands the switchpoint to include links up and down to neighboring tiers. This 3D AFPGA was fabricated in MIT Lincoln Labs experimental 3D SOI technology.

There has also been work in extending FPGA mapping toolflows (Chapter 5) to three dimensions. Ababei et al. published TPR, a three dimensional extension of the VPR package [1, 2]. Their technique is to first partition the design to assign logic to tiers, and then perform constrained placement and routing for each tier serially. A similar approach could also be used with our 3D AFPGA architecture.

Related to our interconnect study, a wide variety of different asynchronous signaling protocols have been previously proposed. Each of these are discussed in detail below in Section 4.4. In general, these works make a one-to-one comparison to an existing system perceived to be their closest rival. Our comparison study is the first I know of to quantitatively evaluate all of the principal asynchronous single-bit signaling options within the same framework.

4.1.2 Contributions

In this chapter, we:

- Propose and fabricate a 3D AFPGA architecture that flexibly extends the 2D implementation (Section 4.2)

- Invent a new asynchronous signaling protocol and buffer design suitable for 3D communication using only a single wire (Section 4.4.5)
- Construct a heuristic optimization framework for evaluating circuit designs on relevant performance metrics (Section 4.5)
- Use the framework to compare all major families of single-bit asynchronous signaling, in both planar and 3D applications (Section 4.6)

4.2 Extending to 3D

One possible FPGA architectural innovation suggested by 3D stacking would be to house configuration memory elements on one level and relegate logic to another. In this way, memory elements could be packed in a tight array, and would be safe from inadvertent bit flips due to the fast switching of logic units.

Unfortunately, in modern technologies TSVs are substantially larger than planar vias (by a factor of 10-100 or more). In fact, each TSV is actually larger than a configuration memory cell, completely negating any possible area savings. Hopefully TSV size and spacing will continue to shrink as manufacturing technology improves, but for now we must find ways to use this scarce resource more efficiently.

A more viable architecture houses one or more layers of FPGA fabric on top of more traditional processing units. In this way, the FPGA can be used as an on-demand accelerator. When paired with the dynamical partial reconfiguration techniques discussed in Chapter 3, this approach presents a compelling step towards more generic hardware primitives — the same die can be used in a very wide array of ways atop different controllers.

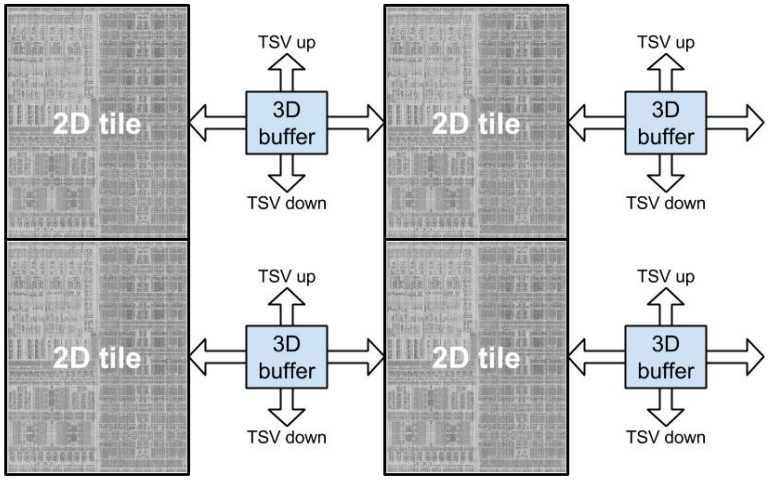


Figure 4.1: 3D extension of 2D AFPGA design

We can extend our existing 2D AFPGA architecture with a few simple modifications to the routing fabric. We could expand every switchpoint (Figure 2.14) in the switchbox to include extra connections up and down to adjacent die tiers. Unfortunately, the brute force approach of increasing the switchpoint degree increases the multiplexer size and adds capacitance, energy, and delay to all routes, regardless of whether 3D connectivity is used or not.

Instead, we choose to augment the 2D routing fabric by inserting '3D buffers' on the existing mesh channels connecting 2D FPGA tile as seen in Figure 4.1. These buffers can be configured to simply pass through a 2D route, so that existing 2D designs can be reused (down to the same bitstream). Their additional feature is allowing a route to connect up or down to a TSV channel, which connects to the adjacent die tiers.

The 3D buffer design shown in Figure 4.2 is very similar to the 2D switchpoint,

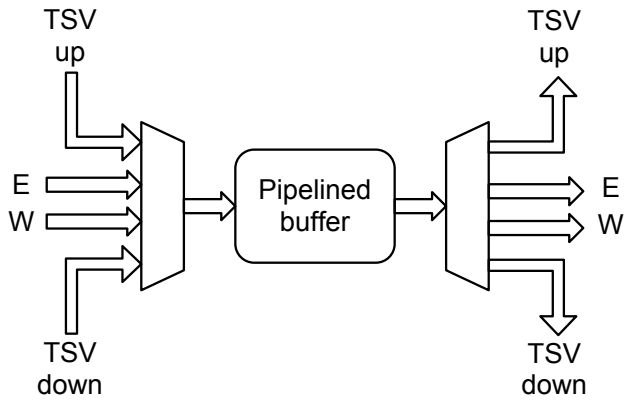


Figure 4.2: 3D switchpoint buffer connected to existing 2D mesh interconnect

and has similarly small multiplexer sizes. One potential enhancement to the design shown is to connect multiple 2D routing channels to a single TSV channel through their multiplexers. This 'party line' configuration allows several 2D channels to share a single (large and scarce) TSV channel statically. It even lets the TSVs serve as a 2D 'dogleg' — if a given TSV channel is not being used for 3D communication, it can instead be used to swap between multiple 2D tracks in the same tier. This adds additional routing diversity, which may makes it easier to map designs to the FPGA.

Figure 4.3 shows an approximate floorplan of the 3D-capable tile. TSVs are indicated as yellow circles, and are grouped with their associated 3D buffers in channels using three TSVs per channel (for *true*, $false$, and *acknowledge* signals). Particularly of note is the scale of the TSVs compared to the entire 2D tile. A mere three 3D TSV channels are able to fit in the same linear space as a tile, which encompasses more than 32 planar channels and their associated switchbox.

The large disparity between planar and 3D TSV communication raises a clear opportunity: is it possible to reduce the cost of 3D signaling? Ideally, we would like a solution that can match the synchronous equivalent: a single wire, plus

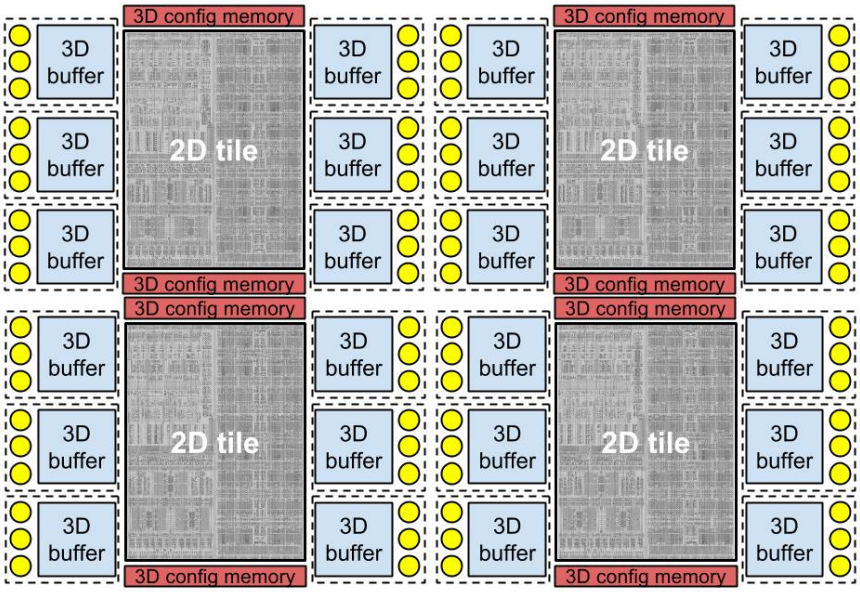


Figure 4.3: 3D tile floorplan showing relative TSV sizes

shared/amortized clock. The following sections describe our work toward that goal, both in the 3D domain and for asynchronous signaling in general.

4.3 Need for Efficient Signaling

Increasing system complexity has begun to put serious pressure on planar wiring resources [65]. At first glance, new process nodes and better back-end-of-line (BEOL) manufacturing have kept the problem mostly at bay. Unfortunately, while designers might have enough wires to meet connectivity requirements in all but the most wire-starved designs, the RC characteristics of the wires have not scaled with transistors. In order to keep shrinking BEOL features without dramatically increasing wire resistance, chip foundries have increased the cross-sectional height of wires. The resistance of long wires can no longer be ignored—the lumped capacitor model

is no longer valid in deep-submicron technologies [54]. Furthermore, taller, more closely spaced wires have resulted in large coupling capacitance values, increased crosstalk, and decreased performance. Some designers of high-frequency systems have resorted to increasing planar wire spacing to decrease wiring capacitance and crosstalk, thereby preserving performance. Over-reliance on this technique can artificially increase pressure on wiring resources, especially for wide buses.

Regardless of bus width, wire spacing, or signaling protocol, the energy of intra-chip communication represents a non-trivial portion of total chip energy consumption [93]. Some projections show wide, cross-chip links consuming a hundredfold more energy in wire transitions alone than in computation [76]. One way to alleviate this problem is to move to 3D integration, for both energy [17] and performance [8], as transmitting data inter-die through a through-silicon-via (TSV) is lower in energy and delay than transmitting data through planar wires across a die. 3D integration has its own problems, such as variability [3] and thermal management [56]. However, for the purposes of this study we focus on the fact that TSV resources are quite limited in comparison with planar wire resources. Achievable TSV pitch ranges from several micron [20] to more than 25 μm [149], well over tenfold the pitch of modern planar wiring.

Self-timed single-bit links like those used throughout our AFPGA architecture are uniquely situated in this complex design space. While they are robust to delay variations, the encodings used incur additional overheads in transition counts and wiring resources—especially important in the TSV case. In comparison, synchronous links make efficient use of wiring resources but suffer from clock distribution and recovery problems. As such, the benefits they provide in comparison with self-timed links are largely dependent on usage case [132].

In light of the pressures on planar and TSV wiring resources by today's asynchronous designers, we present an analysis of self-timed single-bit links. We evaluate representatives from the various classes of self-timed links on the metrics of throughput, energy per bit (token) transmitted, and circuit area. We also present our Single-Track Asynchronous Ternary Signaling (STATS) single-bit link design, which is a single-wire link intended for use in wire/TSV limited environments. However, evaluating each link type at a single point in the throughput/energy/area space is unfair, as factors such as transistor sizing, V_{DD}, and circuit topology can easily change that point. As part of this work we developed an optimization framework to obtain throughput/energy/area Pareto efficiency fronts.

4.4 Single-Bit Signaling Protocols

Table 4.1 shows the self-timed single-bit signaling protocols we chose to study, representative of the various classes of competing schemes. Figure 4.4 shows the wire transitions required for each to send the same token pattern. We provide a brief description of each protocol and justification for our choices below.

Other self-timed techniques, such as bundled data [133] and GaSP [134], leverage traditional clock-based datapath elements like flip-flops and latches for pipelining. They generate "clock signals" for each pipeline stage locally, and amortize the cost of this control circuitry over the many bits of a wide datapath. These are more appropriate for multi-bit communication channels, so they are not considered here. We also omit link protocols which do not include any handshaking flow control, such as [136].

[1]For brevity, we use the same initialism to refer to both the signaling protocol and the buffer that implements it.

Table 4.1: Self-Timed Single-Bit Signaling Protocols

Name[1]	Handshake	Timing	Voltage	Wires
ATLS	4-Phase	QDI	Ternary	2
RQDI	2-Phase NRTN	RQDI	Full-Swing	3
STATS	2-Phase RTN	Single-Track	Ternary	1
STFB	2-Phase RTN	Single-Track	Full-Swing	2
WCHB	4-Phase	QDI	Full-Swing	3

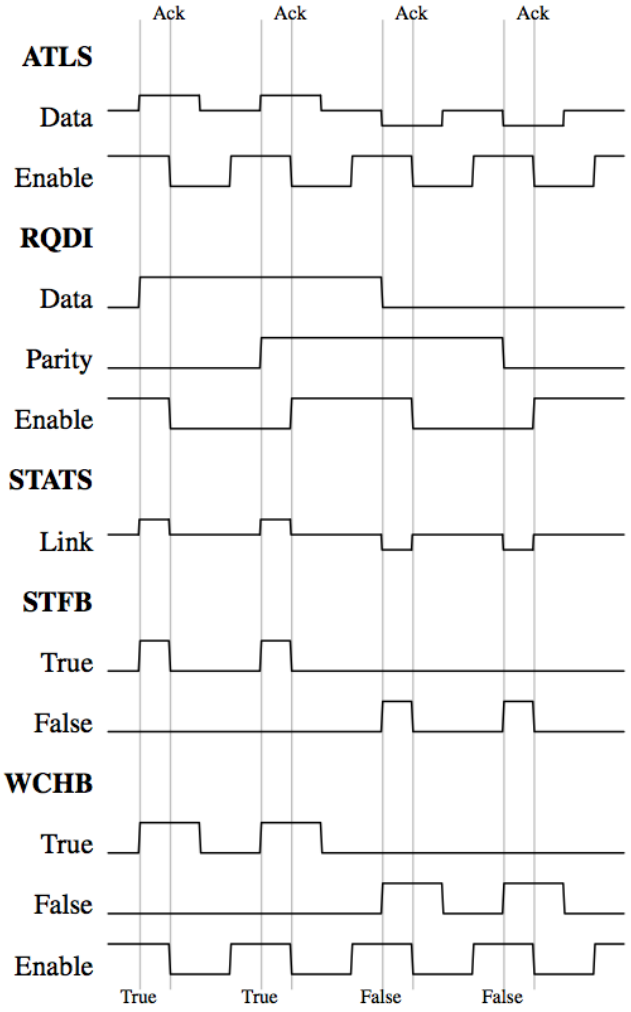


Figure 4.4: Signaling Protocols. Transitions are aligned in time for readability; in general the different buffer types will *not* have the same latencies.

4.4.1 WCHB

The Weak-Conditioned Half Buffer (WCHB) [92] is a handshake reshuffling of the 4-phase dual-rail Quasi Delay-Insensitive [102] protocol, which we refer to as *e1of2* (*e* for "*enable*", an inverted-sense *acknowledge* signal). While there are other possible reshufflings such as the PCHB and PCFB [92] used for logic, we chose the WCHB variant because the buffer implementation is small, simple, and fast. WCHB buffers are used throughout the interconnect in the baseline AFPGA architecture described in Chapter 2. Of all the schemes we study in this paper, the e1of2 protocol is the most conservative. The other link types relax timing assumptions or use more aggressive signaling techniques (e.g. low swing, single track). Evaluating the WCHB allows us to compare the effects of those decisions on throughput, energy, or area.

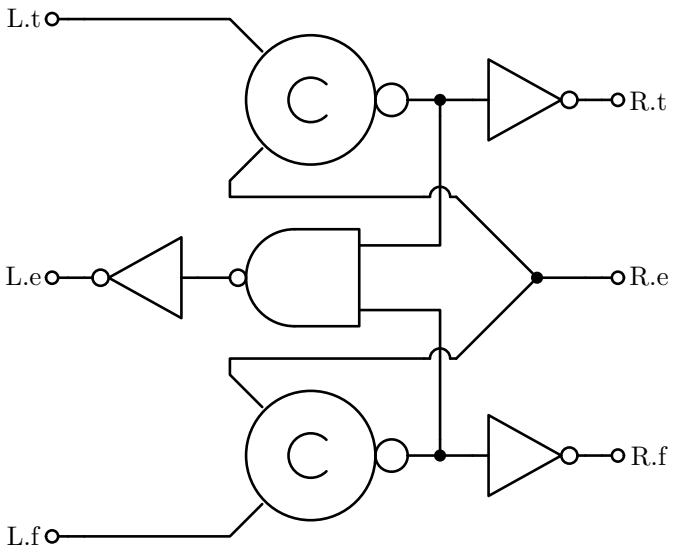


Figure 4.5: WCHB Buffer

4.4.2 RQDI

The Relaxed Quasi Delay-Insensitive (RQDI) buffer design [89] implements a 2-phase, non-return-to-null (NRTN) protocol. It leverages a timing assumption already present in QDI circuits to reduce circuit complexity. We have implemented the LEDR [34] 2-phase encoding, although RQDI supports other 2-phase encodings. We use RQDI to represent the state of the art in 2-phase, single-bit QDI links.

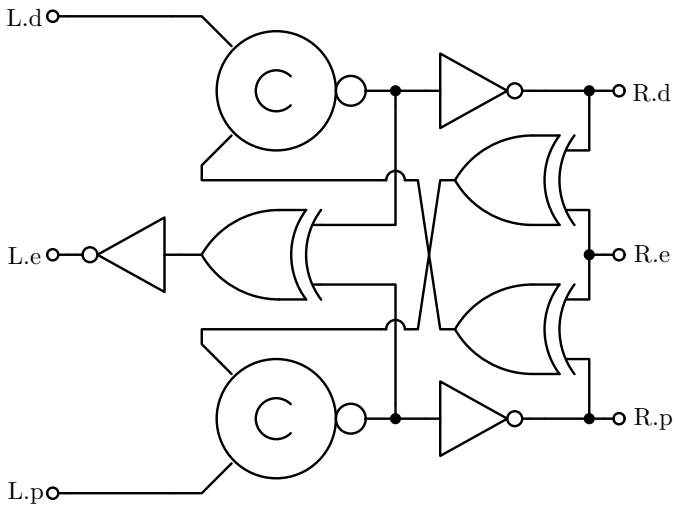


Figure 4.6: RQDI Buffer

4.4.3 ATLS

Asynchronous Ternary Logic Signaling (ATLS) [47,120] is a 4-phase, QDI signaling protocol with a ternary delay-insensitive data encoding. This encoding compacts the dual-rail data wires into a single wire. V_{DD} encodes a *true* token, *GND* a *false* token, and $\frac{1}{2}V_{DD}$ represents the *null* state of the dual-rail encoding. The half-swing encoding reduces the energy cost of data rail transitions, which is attractive as a power saving measure but lowers static noise margins. The enable rail is still

full-swing. ATLS simultaneously attacks the problem of limited wiring resources and power consumption, hence its inclusion in our study.

Our implementation of ATLS differs from the proposed circuits in [47] and [120] as the proposed ternary decoding structures have not scaled well into deep submicron technologies. As in the original proposed circuits, we assume a $\frac{1}{2}V_{DD}$ power supply is available and account for it in our power measurements. We use the circuits described in Section 4.4.5 to encode/decode the ternary data rail. Since ATLS as proposed does not include any pipelining, we use an additional WCHB buffer when necessary as a pipelining element.

4.4.4 STFB

The Single-Track Full Buffer (STFB) [48] is designed for throughput. It uses a 2-phase, return-to-null (RTN) protocol with no control wires. It is, however, dual-rail, using a total of two wires to transmit a single bit. An upgoing transition on the *true* (*false*) rail encodes a *true* (*false*) token, and a downgoing transition on the rail signals an RTN. The sending process is responsible for raising a rail and the receiving process is responsible for lowering it. The single-track timing assumption requires that the sender and receiver are not simultaneously driving the rail, to avoid shorting the chip power supplies across a link.

4.4.5 STATS

Single-Track Asynchronous Ternary Signaling (STATS) is a single-track buffer template of our own design. The design goal was to reduce the total wiring resource

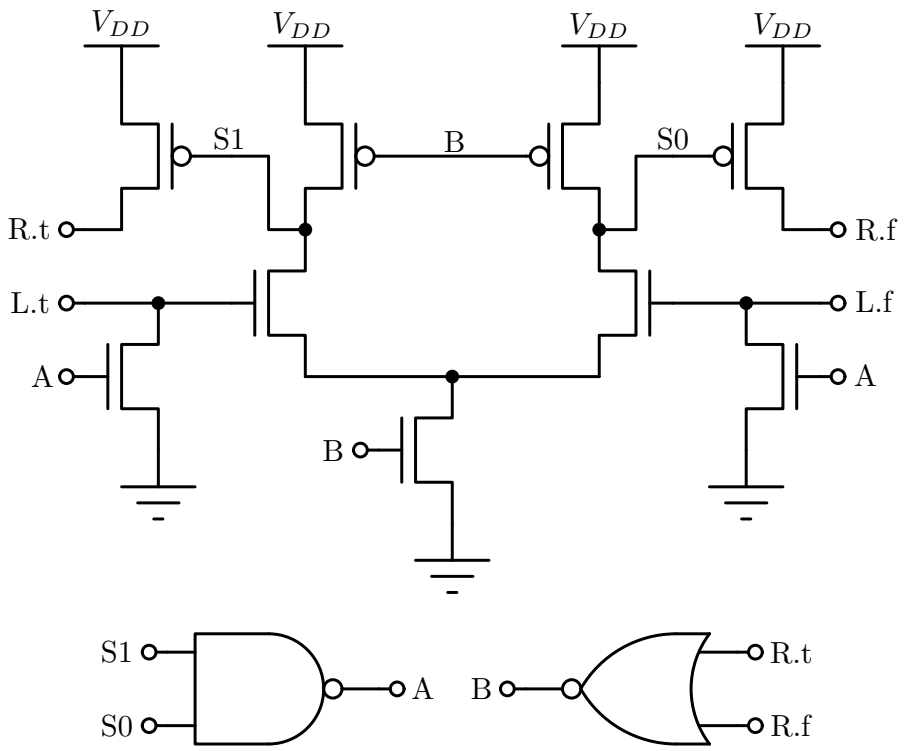


Figure 4.7: STFB Buffer

requirements to a single wire. It combines the ternary encoding of ATLS with the 2-phase RTN, single-track handshake of STFB. To send a *true* (*false*) token, the sending process sets the wire to V_{DD} (GND). The receiving process returns the state to *null* by driving the wire to $\frac{1}{2}V_{DD}$. As with STFB, the single-track timing assumption requires that the sending and receiving process do not simultaneously drive the link.

To decode the ternary link, we use the pair of level shifter structures shown in Figure 4.8. The cross-coupling ensures full rail-to-rail swing, minimizing static power dissipation. While the level shifters are fragile to pathological imbalances in pullup/pulldown network sizings, weakening the pullup/pulldown cross-coupled stacks with respect to their pulldown/pullup counterparts to a ratio of 1:2 is sufficient. Further increasing the drive strength disparity by changing transistor thresholds is recommended. The inverters and NAND gate should be sized to equalize

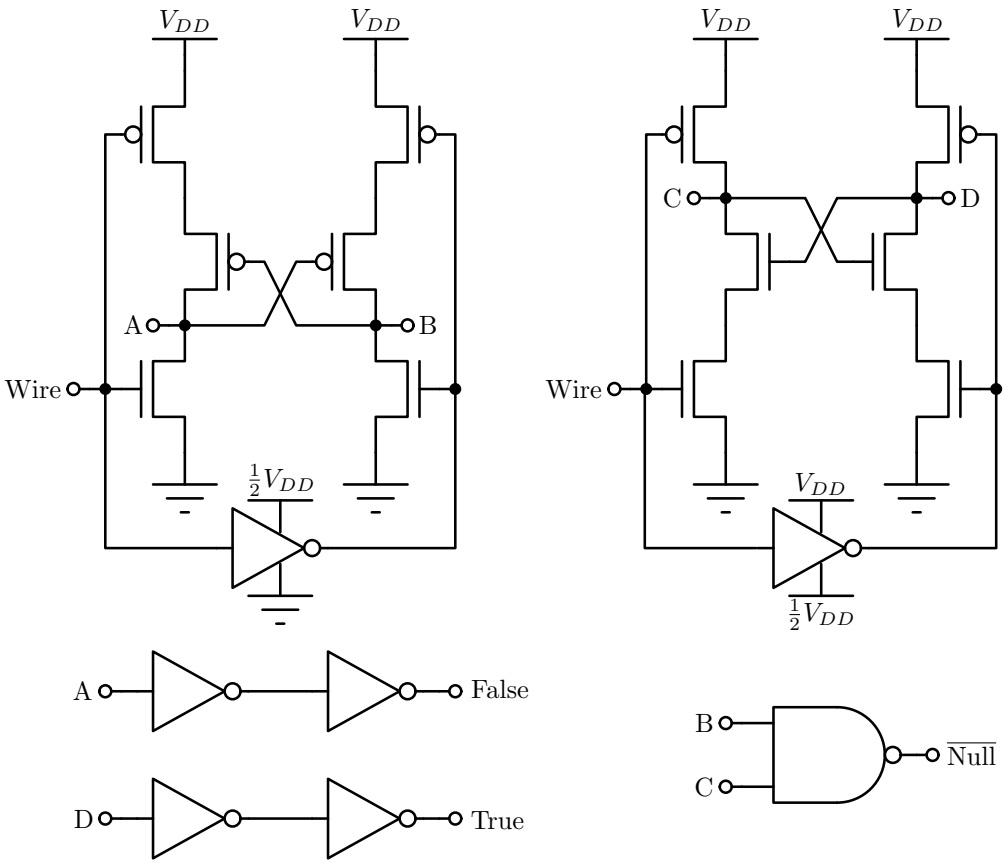


Figure 4.8: Ternary Voltage Decoder

load capacitances on nodes A, B, C, and D.

The link is driven to V_{DD} or *GND* by a single appropriately-sized PMOS or NMOS transistor, respectively, as shown in Figure 4.9. A parallel combination of one or more of the circuits in Figure 4.10 returns the link to the *null* state at $\frac{1}{2}V_{DD}$. We allow our analysis framework, described in Section 4.5.2, to permute the combination and sizing of the RTN circuits to fully explore the tradeoff space.

- **Passgate** (Figure 4.10a): This circuit drives the link to $\frac{1}{2}V_{DD}$ using the least energy, by connecting to the $\frac{1}{2}V_{DD}$ supply. It is the most conservative of the three, but also the slowest.
- **Self-Invalidating Driver** (Figure 4.10b): The self-invalidating driver is

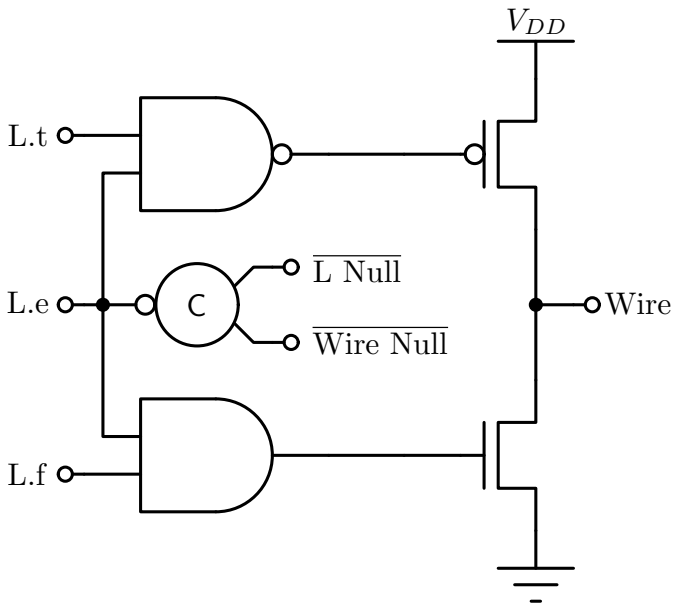


Figure 4.9: STATS Transmit Stage: The null calculation for the Wire and L are obtained from the NAND from Figure 4.8 and the traditional NOR dualrail calculation (not pictured), respectively.

the most aggressive of the three designs, as it is essentially a full rail-to-rail transition interrupted halfway. While it offers the best slew-rate (a single RC time constant is more than a $\frac{1}{2}V_{DD}$ swing), it relies on the level-shifter structures in Figure 4.8 to resolve the state of the wire quickly and switch the True/False signals depicted in Figure 4.8 and Figure 4.10b. A slow transition on either of those two signals will result in an overshoot of $\frac{1}{2}V_{DD}$ and potentially a spurious token on the link.

- **Shorted Inverter** (Figure 4.10c): The shorted inverter makes use of the CMOS inverter voltage transfer curve behavior to drive the wire very quickly to $\frac{1}{2}V_{DD}$. It is faster than the Passgate technique, but very energy inefficient as it essentially shorts V_{DD} to GND while enabled.

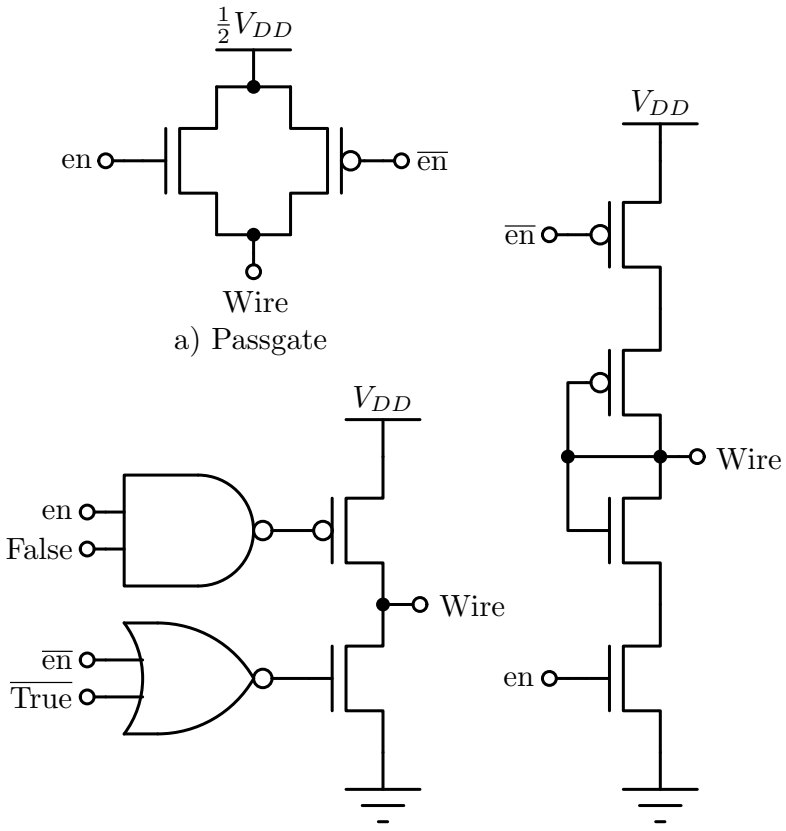


Figure 4.10: Ternary Return to Null Schemes. en high starts the RTN process, and True and False are signals from the decoder shown in Figure 4.8.

4.5 Methodology

We constructed a framework to evaluate the various link types described in Section 4.4 across a wide range of operating points. Links were studied in two contexts: on-chip planar communication, and 3D signaling through TSVs.

4.5.1 Link Simulation

We used SPICE simulation to determine the throughput and energy for each link type. Figure 4.11 shows the basic Device Under Test (DUT) for these simulations.

The link DUT is a FIFO pipeline, implemented at the transistor level. It is driven by an environment that generates pseudorandom tokens as fast as the link can accept them.

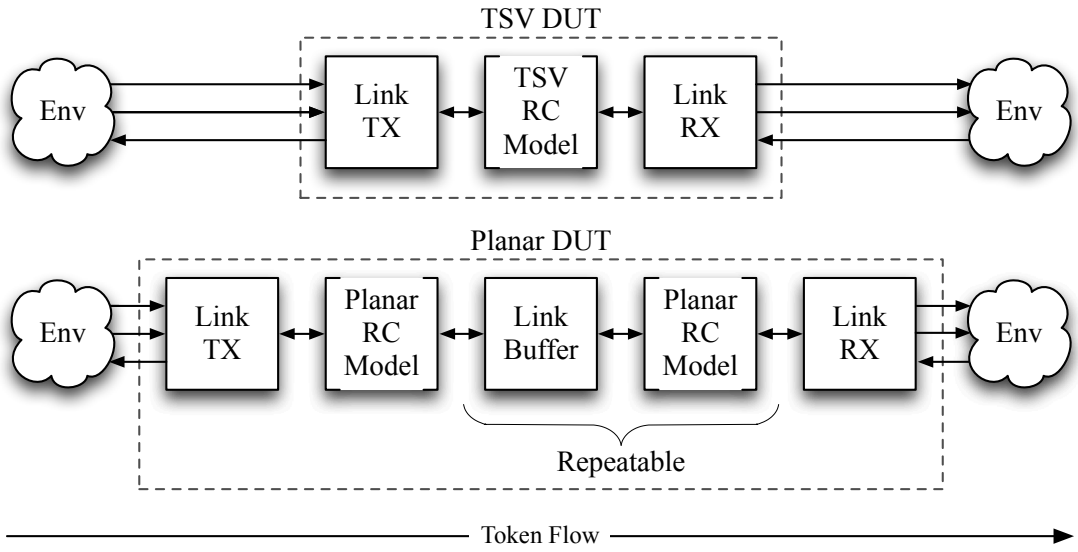


Figure 4.11: Link DUT, for both planar and TSV contexts. Double-headed arrows represent link channels (e.g. STFB); 3-wire channels from the environment are e1of2. "Link TX" and "Link RX" convert to and from the link protocol, respectively, while "Link Buffer" is a native buffer for the protocol.

The link DUT also includes a distributed RC interconnect model (planar wire or TSV). Planar wires of a given length may be broken up into several shorter wires by adding extra buffers as pictured. TSV links cannot be so divided, as there is no way to insert a buffer in the middle of a TSV. We discuss interconnect models in more detail in Section 4.6.

In our simulations, the environment communicates using e1of2, and the cost of conversion to the protocol used by the DUT is included as part of the energy and area costs of the link. This simulates a fully-asynchronous system where computation is done with islands of 4-phase QDI logic and the links are used to shuttle data across planar links or TSVs [103]. This assumption penalizes 2-phase protocols, but it is generally accepted that 2-phase computation is unwieldy in

comparison to 4-phase [108] (with the possible exception of STFB [48]).

Figure 4.12 shows the complete simulation harness. Since the environment source and sink are implemented in Verilog, we use two WCHBs to decouple the DUT from any digital boundary effects. The harness is designed to operate faster than the DUT, so that link throughput is governed primarily by the DUT itself and the RC characteristics of the interconnect.

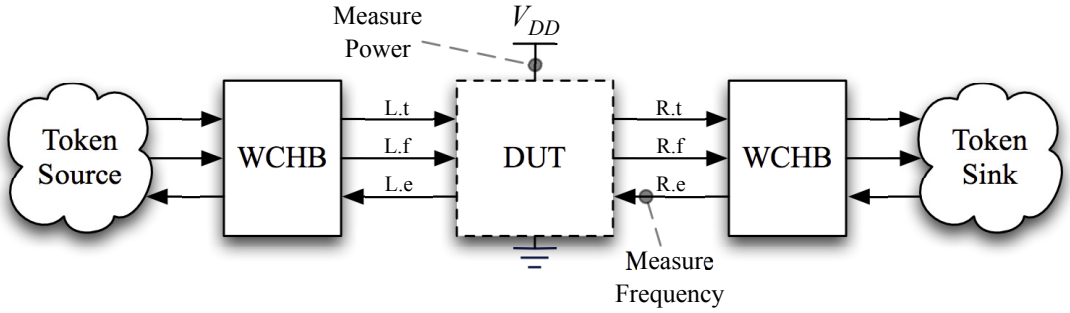


Figure 4.12: SPICE simulation harness for the DUT. Average frequency is measured using the right-side enable signal, and power dissipation is measured for the DUT alone using a dedicated power supply.

ATLS and STATS buffers require an additional $\frac{1}{2}V_{DD}$ supply. In order to be fair, we allow RQDI, STFB, and WCHB links to run at a voltage lower than the harness V_{DD}. To support this, we implemented pipelined level shifters based on the WCHB template, shown in Figure 4.13. These are considered part of the environment, so they are not counted against the link energy and area. The usual protocol converters are still required in addition to these level shifters for non-e1of2 links.

4.5.2 Optimization Framework

The goals for our links are to maximize throughput, minimize energy dissipation, and minimize buffer silicon area. Because these measures are not independent,

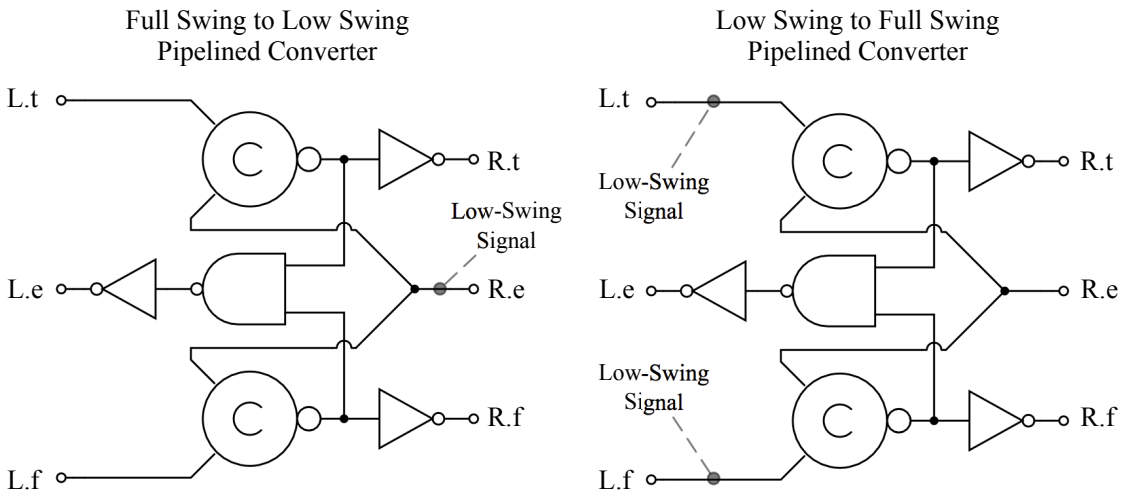


Figure 4.13: WCHB Level Shifters. The C-elements have one low-swing input and one full-swing input. In the C-elements, the NMOS transistors connected to the low-swing input are LVT and sized double-width, and the PMOS transistors are HVT. All other transistors are standard VT.

the solution to this multi-objective problem is a Pareto front of different buffer configurations situated in a three-dimensional tradeoff space between throughput, energy, and area.

To explore this space, we can apply multi-objective heuristic optimization algorithms. While heuristic optimization algorithms are not guaranteed to find the true Pareto front of a given space, i.e. the globally optimal front, in practice a reasonable approximation can be obtained.

We chose the DEAP [50] toolkit and its implementations of the $(\mu + \lambda)$ genetic algorithm[2] (GA) [7] and the widely-used NSGA-II [35] population selection algorithm. NSGA-II-based genetic algorithms are designed to provide a well-distributed family of points along a Pareto front, allowing us to capture the engineering tradeoffs in the design space. Some commercial tools [25] converge to a near-globally-optimal Pareto front faster than NSGA-II-based algorithms, but

[2]Two-Point Crossover ($c_p = 0.7$), Gaussian Mutation ($m_p = 0.2$), $\mu = 20$, $\lambda = 60$, $n_{gen} = 60$

the same result can be obtained by NSGA-II given enough design space samples—typically, a few thousand is sufficient, and we sample at least 2850 points in the space for each link.

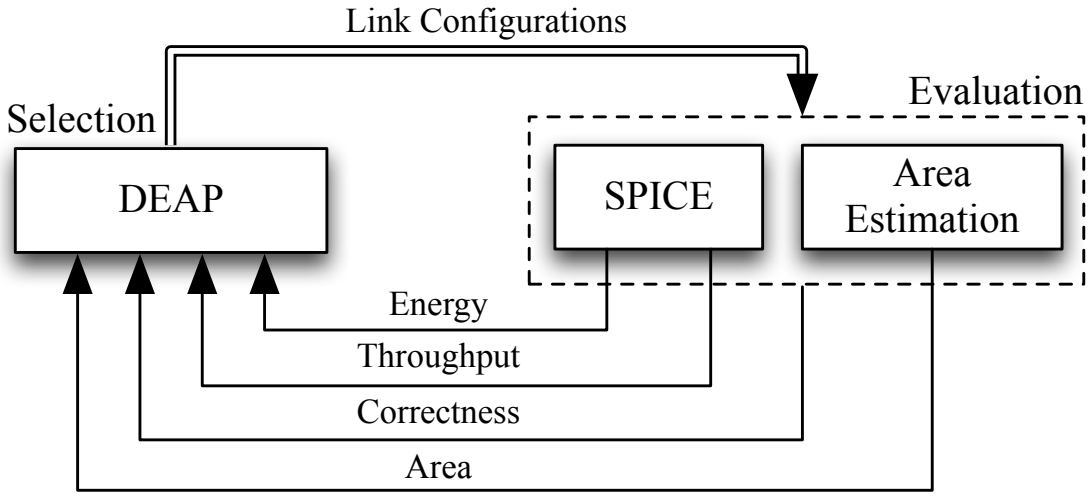


Figure 4.14: Heuristic optimization framework for evaluating link circuit designs

To find the Pareto front for each link design, we built the optimization tool shown in Figure 4.14. Candidate link configurations are selected by DEAP, simulated, and evaluated based on relevant fitness criteria. Throughput and energy are measured using the SPICE harness described in Section 4.5.1. Area is estimated as the total transistor area, i.e. the sum of $W \times L$ for all MOSFETs in the DUT. While this does not account for routing, etc., it provides a lower bound to make reasonable direct comparisons. Some link configurations may deadlock, send spurious tokens, violate dual-rail encodings, or present other failure modes. The environment checks for this and removes any failing configurations. All these evaluation results are fed back to DEAP and used to direct the selection of further candidate configurations.

It is worth noting that this optimization approach is not limited to on-chip links. The same general framework can be applied to any system with a parameterized

configuration (genome) and a set of performance metrics (fitness).

Table 4.2: Link Configurations Explored By Framework

Link Type	Transistor Sizing	Circuit Topology	Link Voltage
ATLS	✓	✓	
RQDI	✓		✓
STATS	✓	✓	
STFB	✓		✓
WCHB	✓		✓

The specific configuration parameters selected by our tool depend on the link type being optimized, and are summarized in Table 4.2. For all link types, the framework selects transistor sizes for the circuits. Transistor sizing is usually handled by convex optimization algorithms, but since we want to explore the multi-objective space we allow DEAP to choose sizing, from minimum transistor width up to 100 times minimum.

For the ternary link types (ATLS, STATS), the framework can alter the circuit topology by choosing which combination of RTN schemes to use (Figure 4.10). For the other types, it can voltage-scale the link as described in Section 4.5.1. Finally, in the planar context our tool can vary the number of buffer stages used (Figure 4.11). We account for the area and energy consumed by multiple planar buffers, but not the additional pipeline slack they provide (which may or may not be desirable depending on the specific system).

4.6 Evaluation and Discussion

We evaluate each link on the metrics of throughput, energy per token, and planar buffer area. In order to definitively conclude that one link protocol is "better" than another on these metrics, the Pareto front of the better link must completely

dominate the other front—the three dimensional surfaces of the fronts must not intersect. Intersecting surfaces imply that the links being compared are situationally better than one another.

The rest of this section is devoted to examining the throughput/energy/area tradeoff space. To present the data in the most readable format, we have chosen to show two-dimensional projections of the three-dimensional Pareto front, omitting the dominated points on each plot. The bottom right quadrant of each plot represents the most attractive link configurations, as we are trying to maximize throughput and minimize energy/area.

In this study we look at three different technology nodes: a low-power 90 nm bulk process, a low-power 65 nm bulk process, and a high-performance 45 nm Silicon-on-Insulator (SOI) process. While we did not fabricate test structures in all three technologies, we were able to build WCHB and STATS planar links in our 90 nm process. We did not obtain isolated power measurements, but the SPICE-predicted frequency numbers for our test structures were within 7% of the actual silicon measurements. Since we have performed the same technology characterization steps for all three technologies when building our SPICE environments, we are reasonably confident in the simulation results presented in this section.

Our methodology does not directly account for robustness to noise or process, voltage, and temperature (PVT) variation. We provide a qualitative analysis of these factors here, but a complete characterization is pending in our future work.

4.6.1 Planar Links

In our planar link simulations, we model wires using a 100-segment π-model with RC parameters obtained from extracted layout. It is vital to use distributed wire models when studying long links, in order to avoid unrealistically optimistic results [54]. As a concrete example: STATS transceivers sense the state of the wire locally to determine when to stop driving. A lumped wire model would yield misleading results, since sender and receiver observe the same voltage. In reality, charge relaxation across the wire means that the voltages may differ and the sender may stop driving too soon—this places a restriction on the maximum slew rate possible for a given wire length.

Figure 4.15 shows the energy/throughput Pareto front for each buffer type in a 90 nm process. Each point in the front represents a different buffer configuration (transistor sizing, V_{DD}, number of buffer stages, etc.). The relative merit of each link type is similar across process technology generations, so we omit the 65 nm and 45 nm plots for brevity. Our evaluation framework allows us to examine the configuration of each individual point on a Pareto front and uncover Pareto-front-wide trends.

From Figure 4.15, we can see that RQDI and STFB are more energy efficient than WCHB with only a few exceptions. This is unsurprising, as 2-phase protocols like RQDI and STFB expend less energy by halving the number of transitions on the RC link. STFB goes one step further by removing the acknowledge wire and the associated drive circuitry, offering additional energy savings. STATS and ATLS are almost completely dominated by the full-swing protocols (RQDI, STFB, and WCHB). The obvious conclusion for the designer is that ternary signaling is a poor choice for planar wiring, which essentially behaves like an RC lowpass network and

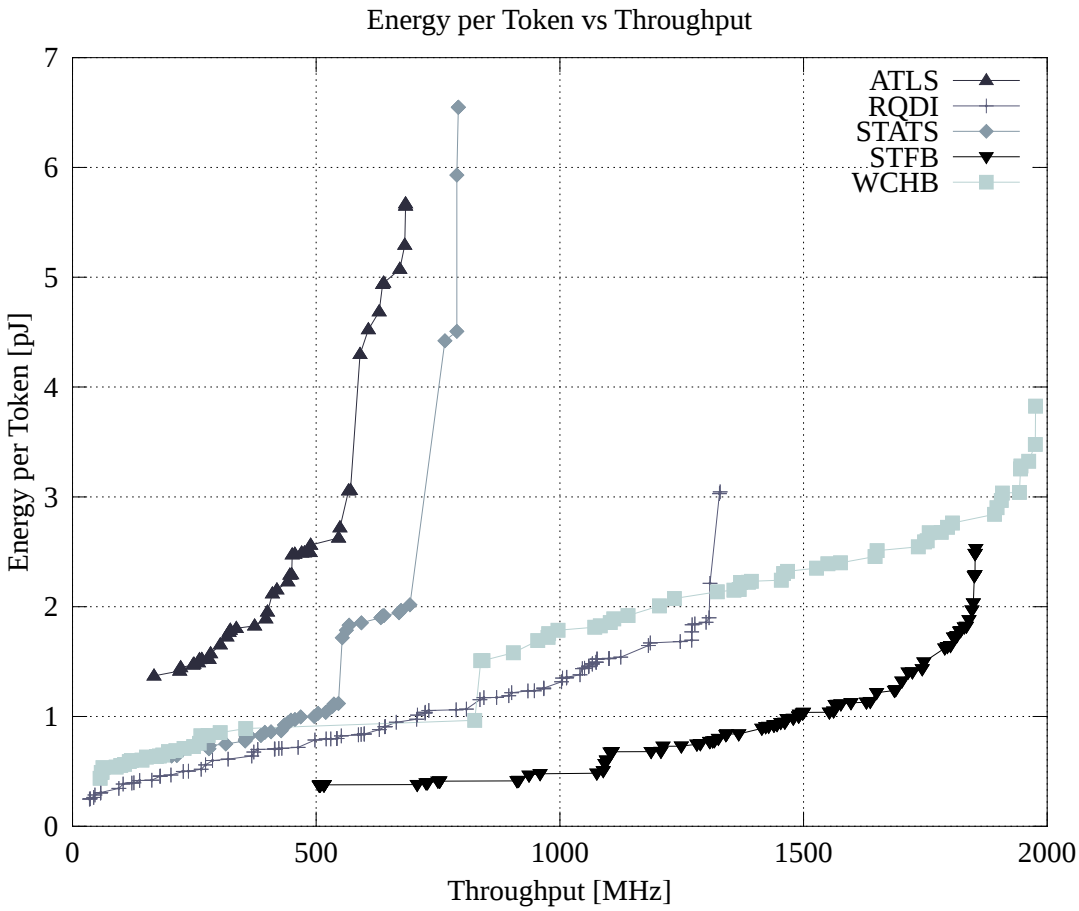


Figure 4.15: Energy vs Throughput in 90 nm for a 1000 μm planar link

limits the throughput of low-swing signals. This is borne out by the fact that the high-throughput Pareto-optimal points for RQDI, STFB, and WCHB all run at full V_{DD} for all technologies, in spite of the capability to aggressively reduce V_{DD}.

An added downside to ternary signaling is that the energy cost of voltage level conversion in ATLS and STATS is quite high, especially when replicated many times in a multi-hop link. The sharp increases in energy per token in Figure 4.15 represent the addition of more buffers on a planar link. Examining the trends across links, STFB and WCHB gradually increase the number of buffers on the link as throughput increases—more buffers driving shorter links allows for higher frequencies. Conversely, STATS and ATLS increase the number of buffers only if

aggressive transistor sizing is unable to achieve additional throughput. Figure 4.15 demonstrates the much greater energy cost of adding buffers to a STATS or ATLS link compared to a similar addition for STFB or WCHB. RQDI also mainly uses transistor sizing to achieve higher throughput. The vertical jump in energy and area at the very highest throughputs represents the addition of more buffers when aggressive sizing is not enough.

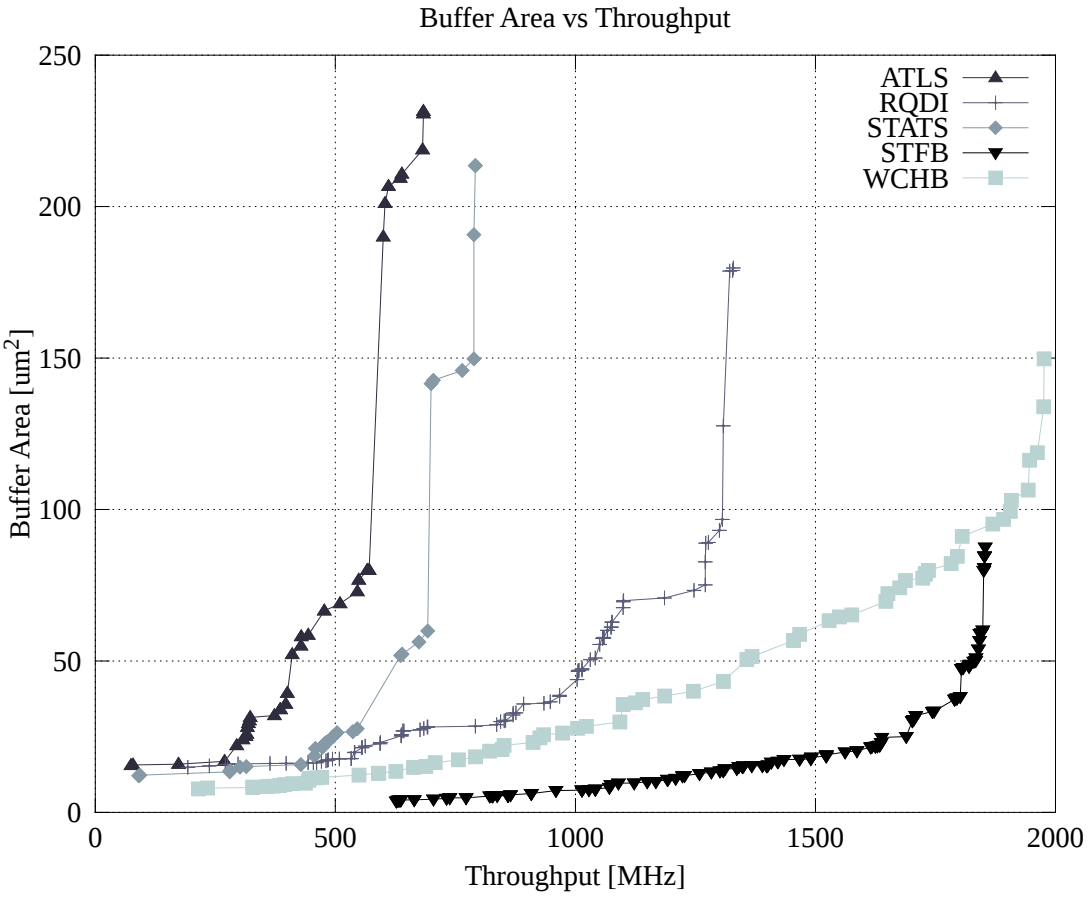


Figure 4.16: Area vs Throughput in 90 nm for a 1000 μm planar link

Figure 4.16 shows the area/throughput Pareto front for each buffer type in our 90 nm process. The aggressive sizing of STATS and ATLS buffers can be seen here—the almost 100 μm^2 increase in area around 400 MHz and 550 MHz represents the addition of a single ATLS or STATS buffer stage, respectively. STFB is best in area, as it has the lowest transistor count per buffer of any link.

As discussed in Section 4.4.5, the Passgate RTN scheme is used for low-throughput, energy-efficient STATS and ATLS configurations, while the faster, more aggressive Self-Invalidating Driver is used in high-throughput links. For average throughput, a mix of these two schemes is used. The Shorted Inverter RTN scheme is only used for the highest throughput link configurations, where energy costs are already high.

As a general observation (that also holds for TSV links as seen in Section 4.6.2), ATLS is never optimal and almost always dominated by every other buffer. The link, as proposed by [47], is more of a data encoding than an actual link design. While we improve some of the circuit designs as described in Section 4.4.3, we still use WCHBs as a pipelining element. Including WCHBs in series adds extra transitions/cycle and power, adversely affecting throughput and energy. In an attempt to maximize the frequency, DEAP selected large transistor sizes, which makes ATLS look unattractive in area as well. A redesign of ATLS that combines the pipelining element with the encode/decode structures could improve its Pareto efficiency performance.

Figure 4.17 and Figure 4.18 show composite Pareto fronts across all technology nodes, for energy/throughput and area/throughput respectively. In other words, the curves on these plots represent the best buffers in that technology at each given operating point. In order to compare results across technology nodes, we scale link length by the technology feature size. Results presented below are for a link length of 20,000λ, equivalent to 1000 μm in a 90 nm technology.

The results are consistent across technologies: STFB buffers are the most energy- and area-efficient for planar signaling across most of the range. At the very highest throughputs WCHB (and 45 nm RQDI) buffers continue to operate after STFB fails, but at a greatly increased cost in energy per token. This high

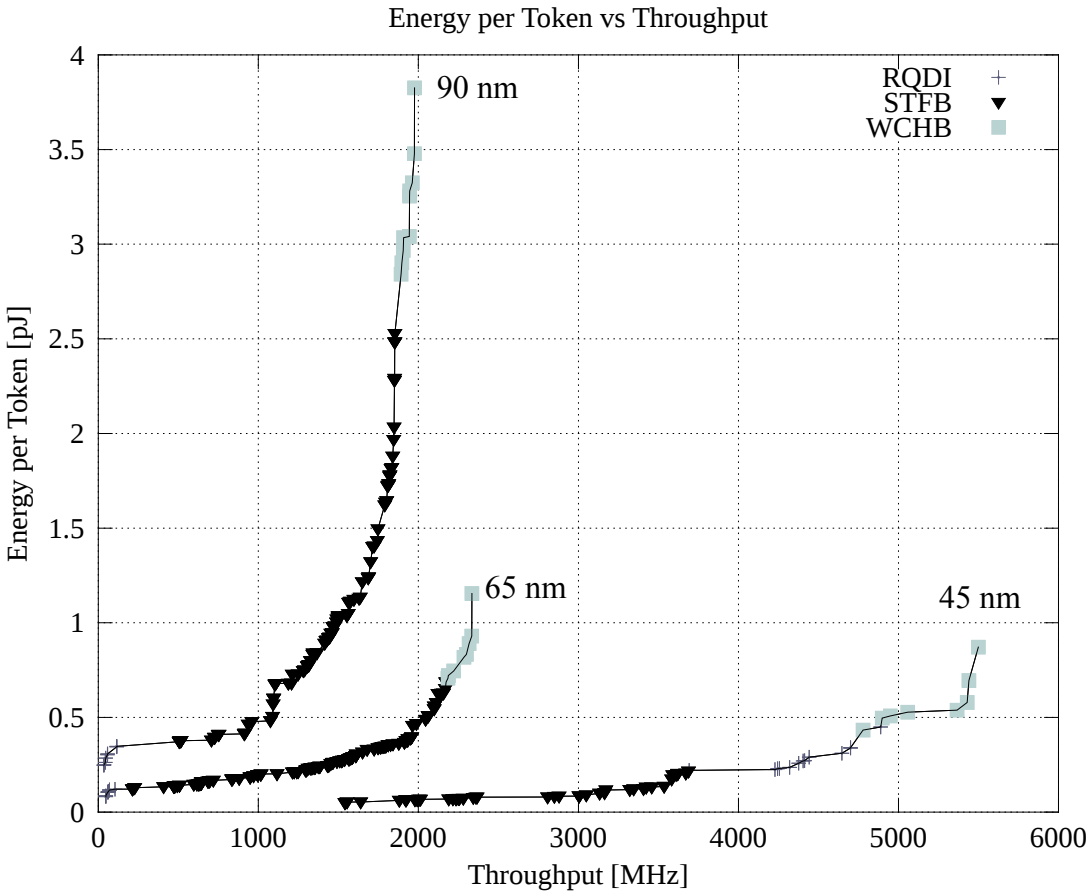


Figure 4.17: Energy-throughput Pareto-dominant points across planar technologies

energy is due to aggressive transistor sizings, reflected in extravagant area usage as shown in Figure 4.18. From these results alone, STFB is the clear winner in the planar context for all but the most aggressive throughput targets. However, the single-track timing assumption makes STFB less robust than QDI buffers, as we discuss in Section 4.6.3. This presents a tradeoff to the designer between energy/area usage and ease of design. The additional cost of "robustness" is not prohibitive, as can be seen by comparing STFB against the QDI buffers (WCHB for high throughput, RQDI for lower) in Figure 4.15 and Figure 4.16.

In the planar context, designers also have control over interconnect wire spacing, which has a direct effect on coupling capacitance. We found that a change

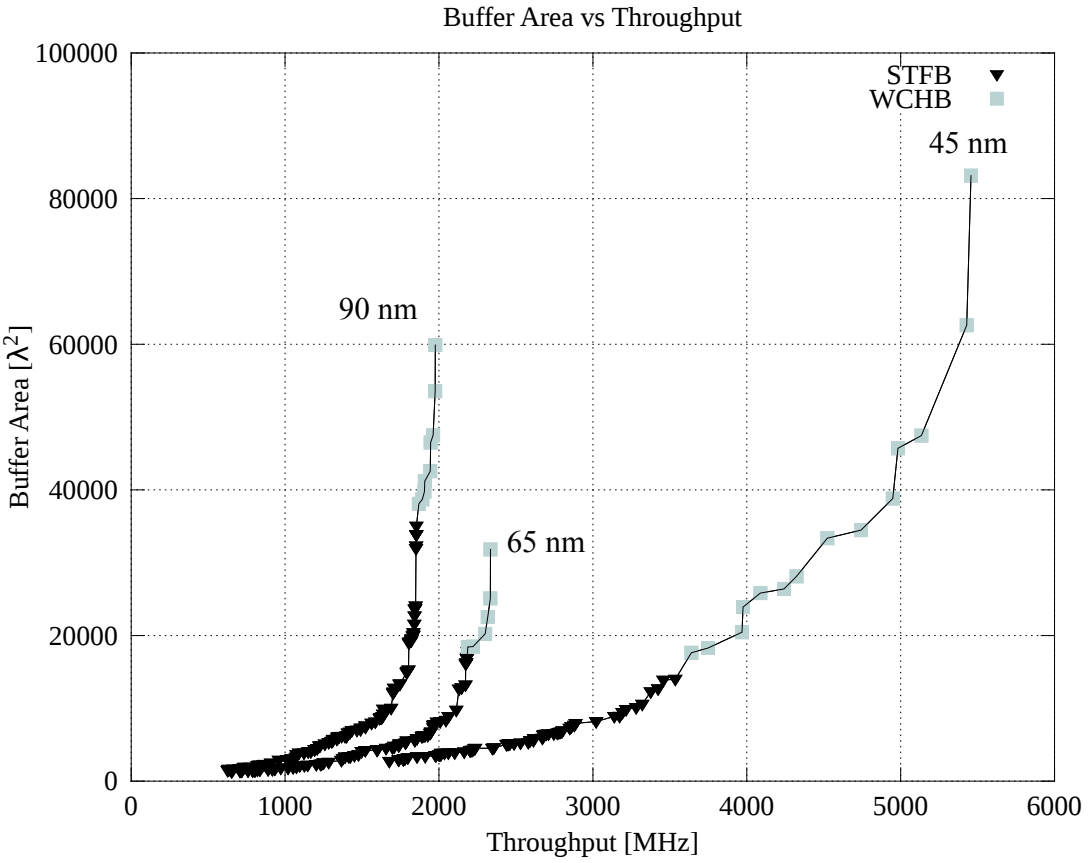


Figure 4.18: Area-throughput Pareto-dominant points across planar technologies

from minimum to sparse spacing (twice minimum) chiefly impacted energy per token. For brevity, we report the average energy improvements at a given frequency for each link in Table 4.3, as opposed to including additional Pareto fronts. Note that this table does not capture the additional benefits of reduced crosstalk due to the increased spacing.

Table 4.3: Percentage Improvement in Sparse Wiring Energy

Link	90 nm	65 nm	45 nm
ATLS	47.36	16.93	-24.67
RQDI	33.71	7.22	13.98
STATS	27.42	-92.28	-112.87
STFB	39.04	18.11	12.26
WCHB	49.66	28.43	20.99

For most link/technology parings, the results shown in Table 4.3 are as ex-

pected. To first order, wire resistance remains constant with increased wire spacing while coupling capacitance decreases. This decrease in capacitance leads to lower $\frac{1}{2}CV^2$ energy dissipation.

Strangely, the sparse wiring energy *increases* for STATS and ATLS. The root cause of this energy increase is that for high-throughput link configurations, DEAP selects more highly pipelined links. As an example, in 45 nm, the fastest running ATLS and STATS configurations divided the planar wiring into 10 sections for the sparse wire spacing case, and only 5-6 for the minimum spacing case.

In order to implement single-track timing (sender and receiver must not drive a wire simultaneously), STATS inspects the local voltage to determine when to cut off the driving transistors. If the interconnect resistance is high relative to its capacitance (as in sparse wiring), the buffer may see the local voltage change and turn off before moving enough charge to resolve the state transition at the remote end of the wire. This tends to favor shorter wires with more buffers, leading to greater energy consumption. High-throughput ATLS configurations use the fast Self-Invalidating Driver, which has the same property.

4.6.2 TSV Links

To simulate 3D links between stacked dies, we use the TSV model from [149], modified to have distributed rather than lumped RLC components. It represents a 20 μm diameter copper TSV with 25 μm pitch in a digital process. We also model coupling capacitance to Manhattan neighbor TSVs. TSV fabrication is usually a separate step from the rest of the CMOS process and scales at a different rate, so we use the same TSV model for all process technologies in this study.

Because TSV pitch is much larger than the standard via pitch, we assume that TSVs are a scarce resource. As a result, we report throughput per-TSV below (by scaling using wire counts from Table 4.1). This penalizes buffers that require more wires to send a single bit of data.

Figure 4.19 and Figure 4.20 show the energy/throughput and area/throughput Pareto fronts in a 90 nm process for each buffer type communicating vertically through a TSV link. Since buffers in each technology must drive the same TSV structure, reported buffer area is not scaled as it was for the planar results.

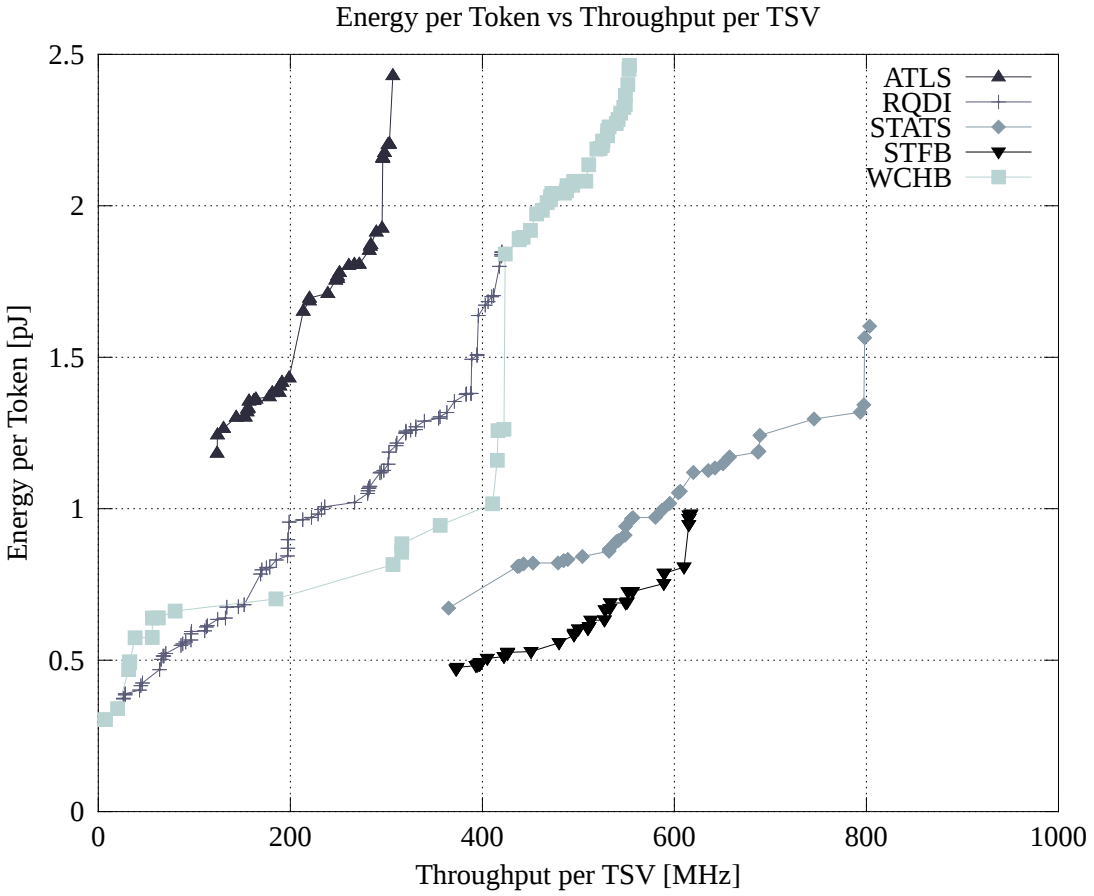


Figure 4.19: Energy vs Throughput in 90 nm for a 25 μm pitch TSV model

In the TSV context, STATS is a strong contender due to its efficient use of TSV resources. Furthermore, TSVs have high capacitance but low resistance compared

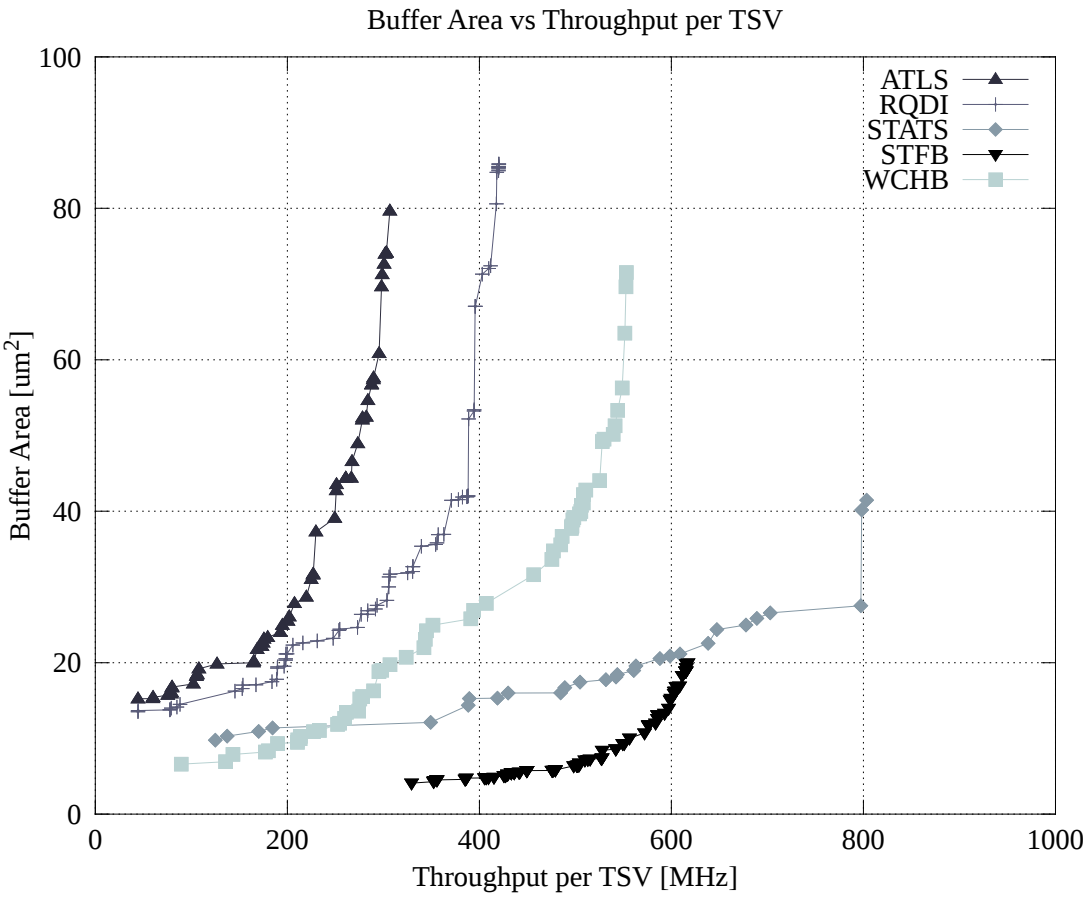


Figure 4.20: Area vs Throughput in 90 nm for a 25 μm pitch TSV model

to planar wires, due to the sheer amount of conductive material. This environment approaches the ideal lumped capacitance case where a low-swing link such as STATS excels—theoretically, a half-swing protocol would expect to see 4x savings in $\frac{1}{2}CV^2$ switching energy. In practice, the ternary conversion energy cost cuts into this savings, but STATS is more attractive in energy/throughput-per-TSV than all other links save STFB.

An interesting phenomenon is the sharp energy increase for WCHB buffers in Figure 4.19 around 400 MHz. Examining the link configurations that straddle this increase revealed essentially identical configurations save for one gate: the inverter driving the returning L.e acknowledge signal (shown in Figure 4.5) for the Link RX unit (shown in the TSV DUT section of Figure 4.11). The higher-energy

configuration was a *maximal* sizing of this inverter, whereas the lower-energy point was a *minimal* sizing of the same inverter. This phenomenon is present in all three technologies. It occurs in the planar case as well: the slight discontinuity in WCHB energy per token seen in Figure 4.15 around 800 MHz displays the same jump in driver sizing. There are other effects at work in the planar case (e.g. number of planar buffers), but the trend is still noticeable.

While further investigation is warranted, this suggests two Pareto efficient operating regimes for the WCHB link. In the first mode, throughput is unaffected by a slow transition on the enable signal—the link is not token-hole-limited. At some throughput threshold, however, the system becomes token-hole-limited and a fast acknowledge transition (with associated energy cost) is required to see further improvement. Examination of the dominated points in the DEAP runset revealed that DEAP had tried many similar configurations, more or less holding all other parameters constant and varying the sizing of the L.e inverter across the range of allowable sizings—in other words, this phenomenon is quite unlikely to be an artifact of the heuristic optimization algorithm. Intuitively, a small increase in the inverter sizing would offer negligible throughput gains with an energy penalty. Conversely, a downsizing of a maximal inverter would penalize throughput without much benefit to energy. Furthermore, it is likely that an algorithm that sizes transistors based on their electrical environment alone would not have discovered these two operating regimes. Such an algorithm would have sized the L.e inverter to drive the large TSV capacitance and missed out on the low-energy WCHB configurations.

A cross-technology examination of TSV links, plotted in Figure 4.21 and Figure 4.22, is slightly more complicated than the planar scenario. We use the same

TSV structure across all technologies, so the electrical characteristics of the physical link remain constant while the transistors shrink. This leads to STFB failure (described in Section 4.6.3) and its disappearance from the Pareto front after 90 nm.

Measured on a throughput/TSV basis, STATS (which uses only one TSV) dominates. The QDI buffers (RQDI, WCHB) are penalized due their 3-wire interface, but also appear on the Pareto fronts at low throughput/TSV.

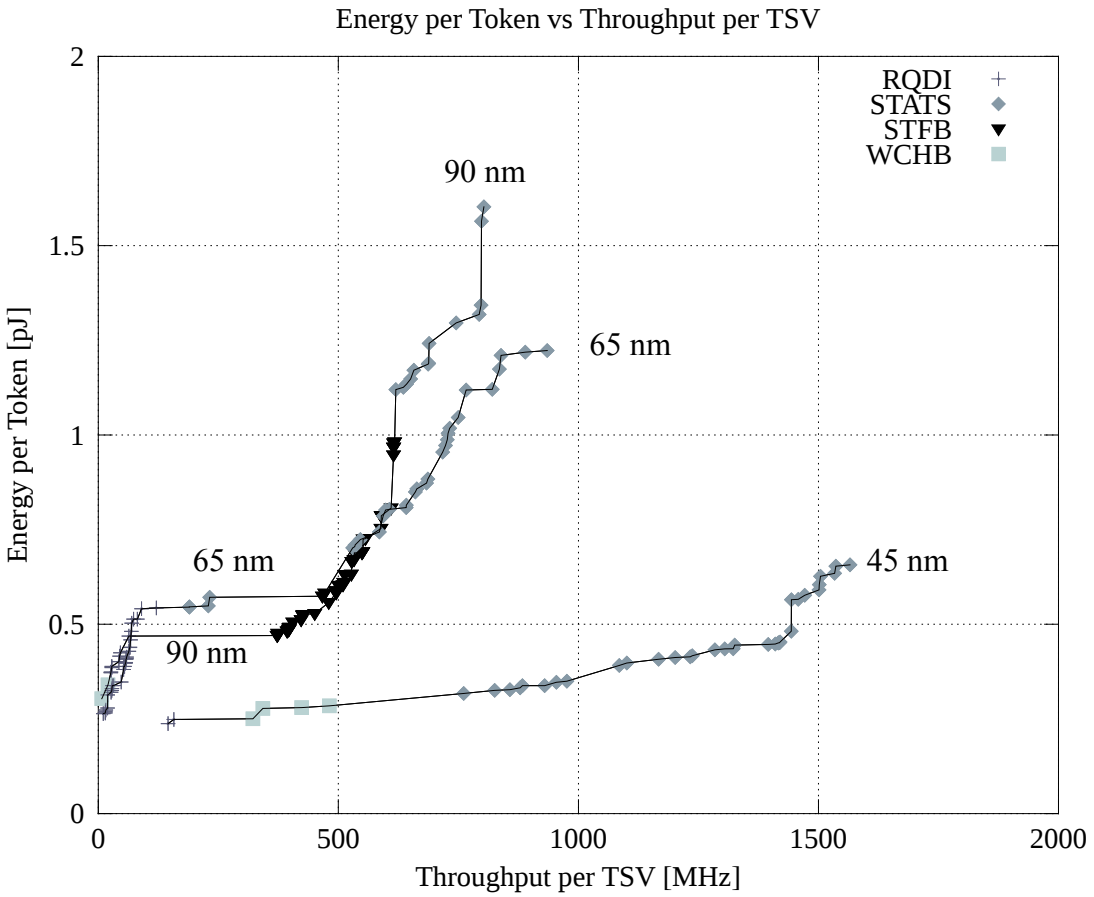


Figure 4.21: Energy-throughput Pareto-dominant points for 25 μm pitch TSV

Examining silicon area (Figure 4.22) instead of energy per token, WCHB (not RQDI) is the smallest for low per-TSV throughput, for similar reasons to the planar case. The other rankings are the same as for energy.

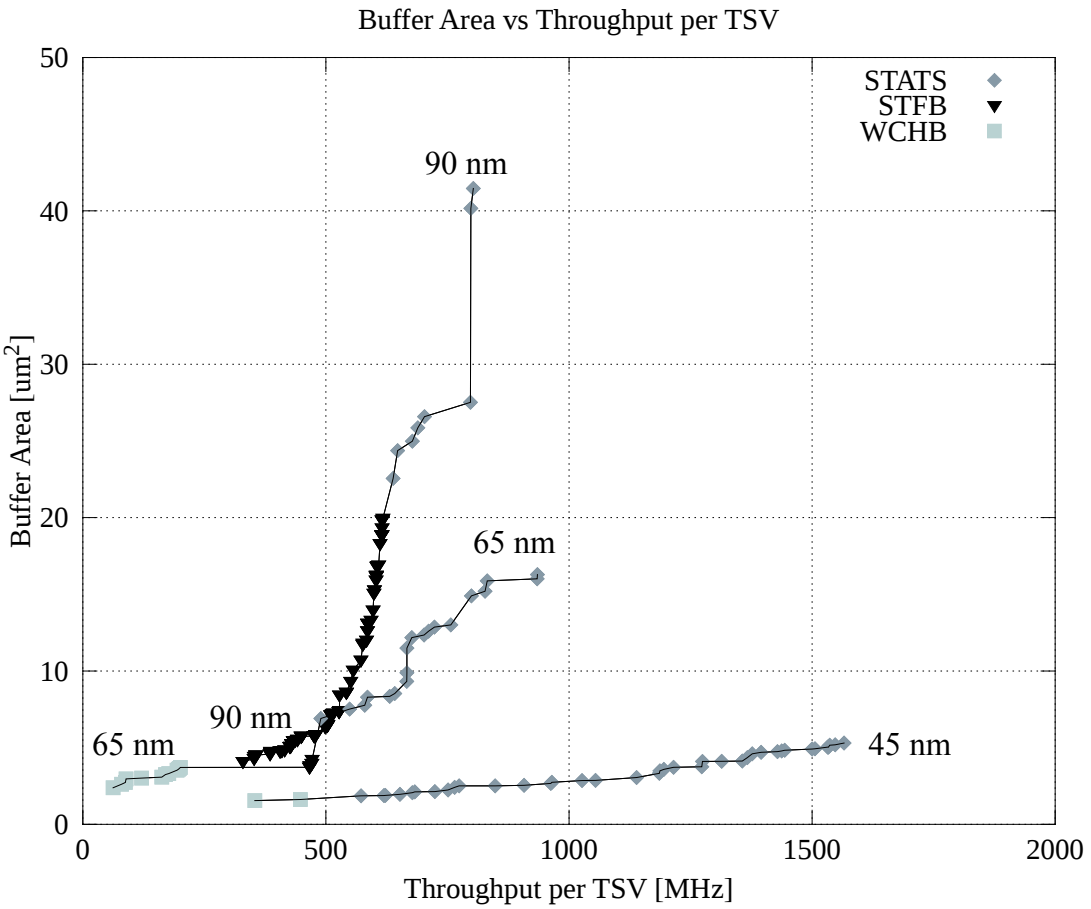


Figure 4.22: Area-throughput Pareto-dominant points for 25 μm pitch TSV

4.6.3 Link Failures and Reliability

In addition to throughput, energy, and area, we record the failure rate of individual configurations selected by DEAP. These statistics are reported in Table 4.4 for planar and TSV links across our three process technology nodes. As discussed in Section 4.5, we verify that links send tokens correctly and do not deadlock. Failures are typically due to poorly-sized transistors driving large RC loads, since DEAP can choose sizes at random. Roughly speaking, these failure rates provide information about how easy a link is to design and how robust it is to sizing variation. While a significant amount of additional work is required to quantify link robustness, we believe the failure rate is of use in building an intuitive understanding of a link's

timing assumptions and relative design difficulty.

Table 4.4: Link Failure Rates

Link	% Planar Failure			% TSV Failure		
	90 nm	65 nm	45 nm	90 nm	65 nm	45 nm
ATLS	23.94	16.34	19.23	17.72	20.83	15.54
RQDI	25.60	23.93	17.80	19.72	21.52	24.68
STATS	42.40	36.26	45.45	33.26	33.96	33.31
STFB	28.18	21.99	33.63	29.19	99.33	100.00
WCHB	10.67	8.49	12.43	12.79	12.80	25.32

Note: $2856 \leq n \leq 11158$

STATS has the highest failure rates of all buffer types in planar and the second highest in the TSV context. This is not surprising, as STATS combines both ternary encoding and the single-track timing assumption to achieve its single-wire goal, and each of these techniques reduce reliability compared to a delay-insensitive link.

Ternary decoders (Figure 4.8) are particularly sensitive to sizing variations, and their failure prevents the buffer from sensing the link state correctly. Even an accurate but slow decoder may cause link failure, for example by causing a Self-Invalidating RTN Driver (Figure 4.10b) to overshoot $\frac{1}{2}V_{DD}$. This impacts the failure rates of both STATS and ATLS.

As discussed in Section 4.6.1, single-track timing can cause STATS to fail if the interconnect resistance is too high (planar wiring). This is less of an issue in the high-capacitance, low-resistance TSV context, so we see correspondingly lower failure rates in Table 4.4. ATLS, RQDI, and WCHB do not suffer from this problem, due to the QDI timing of their handshake. Reasonably slow transitions are acceptable, as they will be not be acknowledged until the receiving end can resolve the wire state.

STFB uses an even more aggressive single-track timing assumption. The STATS level shifter structures offer a better inspection of the wire state due to hysteresis, whereas the link wire directly drives the STFB handshaking logic as seen in Figure 4.7. As a result, STFB has simpler circuits and better throughput, but at the expense of robustness. The traces shown in Figure 4.23 were selected from the fastest five STFB and STATS TSV link configurations in 90 nm. The STFB *true* and *false* rails do not complete full-swing transitions. Examining the figure, it takes at least two tokens traversing a link (and driving the link pulldown network) to return the wire state to *GND*. In contrast, STATS transitions cleanly between V_{DD}, $\frac{1}{2}V_{DD}$, and *GND* because it inspects the voltage before cutting off the transistors driving the wire.

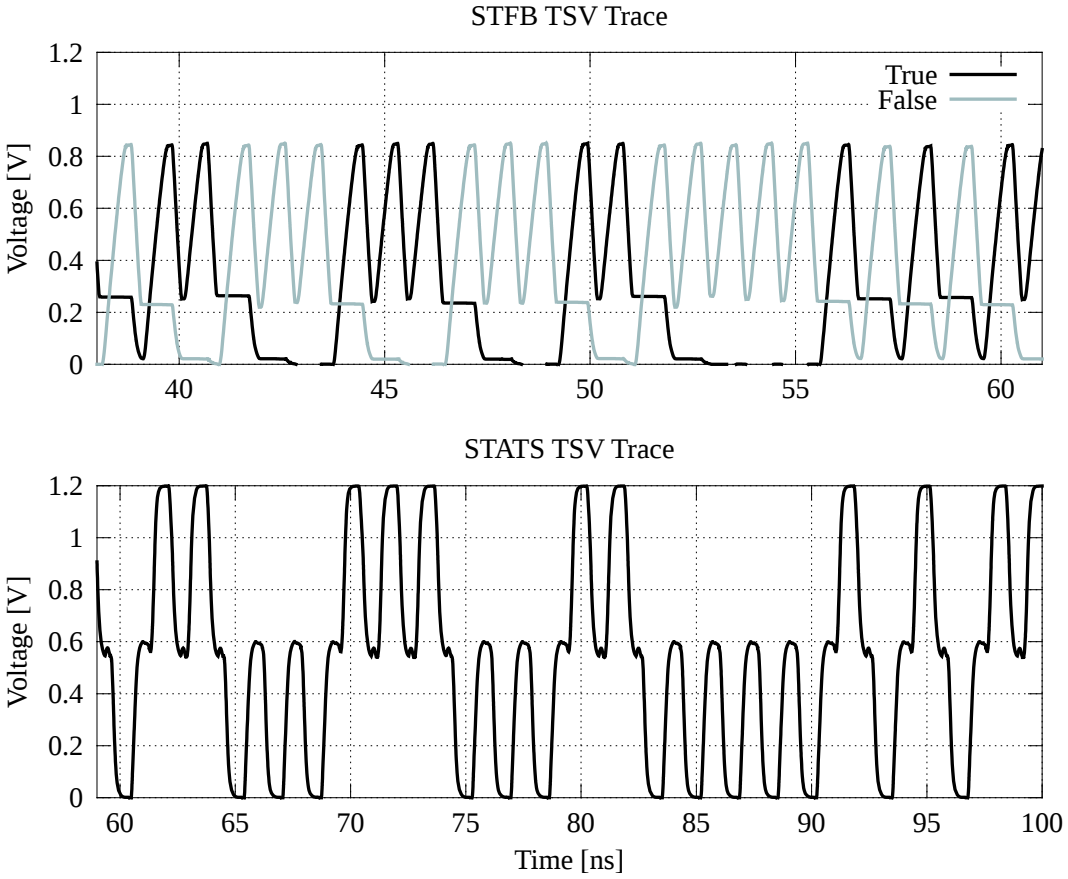


Figure 4.23: Trace of STFB and STATS TSV links in 90 nm. V_{DD} is 1.2 V. Times shown are matched by sent tokens.

In short, STFB is releasing the pull-up network and pull-down networks too early. Large capacitances exacerbate this problem. Because the TSV RC characteristics in our model do not scale with technology—we assume they are separately fabricated—but transistor switching speed does, the timing margins become progressively worse for STFB with each smaller technology. This is reflected in the increased number of failing configurations, shown in Table 4.4, and the complete failure of STFB to drive the TSV link in 45 nm.

One side effect of this timing failure is that STFB becomes a de facto low-swing signaling protocol, which artificially improves its energy efficiency. While this is certainly not without merit, it comes at the cost of noise margins and robustness. While we do not model noise sources, the STFB traces shown in Figure 4.23 are more susceptible to noise than a full-swing signal would be. Note that ternary encodings (ATLS, STATS) also have reduced noise margins compared to a full-swing signal.

4.7 Conclusions

We studied five self-timed single-bit signaling protocols with widely varied properties (timing assumption, wire count, voltage swing), including our proposed STATS single-wire link design. We developed a multi-objective optimization tool to evaluate the performance (throughput, area, and energy per bit) of these protocols for both traditional planar wiring and 3D inter-die communication using TSVs.

Pareto front analysis is a powerful framework for evaluating competing objectives. From this study, we draw several conclusions useful for circuit designers. For planar links, STFB offers the best performance across the range of process

technologies studied, though its tight timing requirements do not cope well with non-ideal wires. WCHB also performs well, trading some energy and area for increased robustness, and remains our buffer of choice in 2D AFPGA designs.

When extending the AFPGA in 3D, however, the results change. STATS buffers perform poorly with planar wiring but are a good match for TSV electrical characteristics. Their efficient use of scarce TSV resources makes the extra expense worth contemplating for 3D signaling, bringing our wire efficiency up to the same level as synchronous designs.

CHAPTER 5

SOFTWARE TOOLFLOW

5.1 Introduction

FPGA mapping is the process by which a design expressed in a hardware description language (HDL) is converted into an FPGA configuration bitstream. Mapping software tools span this gulf between very high and very low level descriptions via a sequence of smaller steps: synthesis, technology mapping, clustering, placement, routing, and finally bitstream generation.

Many commercial FPGA mapping toolflows in effect combine all of these operations into a single black box. Device manufacturers like Altera and Xilinx provide software that exclusively supports the chips that they sell. They have no commercial incentive to expose the steps of the mapping toolflow in an open or reusable way, since this work would also benefit sales of their competitors' hardware. This approach also has some technical benefits, in that it allows manufacturers to deeply optimize for their specific hardware. On the other hand, opaque solutions such as this prevent efficient and expedient exploration in educational and research environments.

Fortunately, open source academic tool flows like VPR [14] have opened the door to a wealth of experimentation. We hope to extend this universe to include self-timed FPGAs, while taking into account their unique challenges and opportunities.

In this chapter, we discuss the stages of the traditional FPGA mapping toolflow (Section 5.2), analyze the different optimization criteria posed by asynchronous

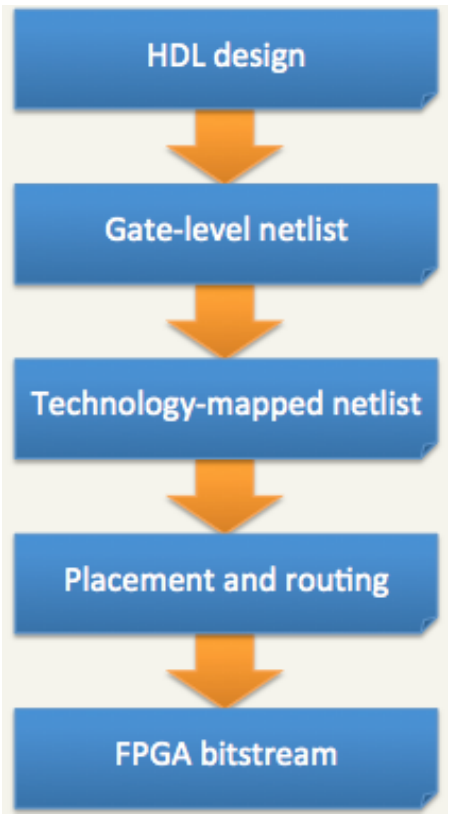


Figure 5.1: Intermediate stages of a design being FPGA mapped

logic (Section 5.3), and describe modifications we made to the mapping toolflow to allow it to target asynchronous logic (Section 5.4), including a new partition-based clustering and placement strategy (Section 5.5).

5.1.1 Related Work

As discussed above, the majority of commercial FPGA mapping toolflows are closed source, and by design only target a given company's series of devices. While this makes sense commercially, it limits the possibilities for research in this area.

The main open source FPGA tool flow project is called Versatile-Place-and-Route (VPR) [14], published by the University of Toronto. This collection of tools

maps designs to an abstract FPGA architecture, rather than a particular commercial FPGA. The parameters of this architecture may be varied by researchers to explore the system impacts of different FPGA implementation choices. With VPR5 [95], the tool was expanded to support heterogeneous FPGA fabrics, as well as single-driver routing within the interconnect. VPR5 also improves electrical modeling, allowing for more accurate (synchronous) timing simulations of mapped designs. The latest embodiment of this project is Verilog-to-Routing (VTR) [121]. The starting point for the VPR flow previously was a structural netlist, but VTR adds Verilog elaboration using Odin II [69] so that it can accept HDL input like commercial toolflows.

VPR/VTR targets architectural research, so it stops short of generating programming bitstreams for actual hardware. Other researchers have separately added this functionality, extending VPR to map to Xilinx Virtex 6 devices [67]. This extension uses RapidSmith [90], an API to the Xilinx Design Language, to create bitstreams and program commercial FPGAs.

Other groups have also developed similar efforts. Zhou et al. created VDK [158], a full toolflow designed to support formal verification throughout. VDK uses Icarus for Verilog parsing, Synplify for clustering, simulated annealing for placement, and a Pathfinder-based router. The TORC project [131] laid out a roadmap for a toolflow similar to VPR that supports reconfigurable computing applications. Most recently, Project IceStorm [152] developed a complete working open source flow targeting Lattice FPGAs. It uses Yosys [151] along with ABC [19] for design elaboration and synthesis, arachne-pnr [126] for placement and routing, and generates bitstreams for the Lattice iCE40 series based on reverse engineering the (relatively simple) architecture for that FPGA.

In addition to the full toolflows above, there are also a wealth of independent tools that comprise the various steps of the flow. Many of these can be reused or adapted to target asynchronous FPGA architectures. These are tools are discussed in the relevant subsections of Section 5.2.

Our goal is to build a complete toolflow targeting asynchronous FPGAs like the one designed in Chapter 2, from HDL through to a mapped design. Like VPR, this flow should have sufficient flexibility to allow it to be used for future architectural research. In the remainder of the chapter, we describe how we constructed just such a toolflow.

5.1.2 Contributions

In this chapter, we:

- Create a complete, flexible and extensible asynchronous FPGA mapping toolflow (Section 5.4)
- Reframe asynchronous performance analysis to be more directly useful for designers of AFPGA architectures and mapping tools (Section 5.3)
- Design and implement an async-aware placement and clustering tool, based on dataflow graph partitioning (Section 5.5)

5.2 Phases of Synchronous Toolflow

In this section, we outline the sequence of steps used by mapping toolflows to perform the series of design transformations shown in Figure 5.1. Chen et al. [26]

published an excellent survey of the complete FPGA mapping process. The sequence of operations given mainly follows those performed by VPR. Depending on the specific toolflow implementation, some of these logical stages may be combined, or fractured into sub-steps.

The FPGA mapping process is largely similar to an ASIC circuit design flow, but with more constraints due to the fixed/discrete nature of the FPGA platform. It is worth noting that for each of these stages, finding an optimal solution is generally computationally intractable. As a result, each relies on heuristic algorithms to find a reasonable balance between mapping speed and mapped design performance.

5.2.1 Synthesis

Synthesis is the process of converting a design expressed behaviorally in a high-level design language (e.g. Verilog, VHDL, or more recently C) to a structural netlist. This netlist is built from a discrete set of (possibly abstract) gates, and it includes all connectivity information between those gates. The output of the synthesis stage does not target any particular FPGA architecture.

The synthesis process is not unique to the FPGA context; it is the first step in many digital design flows. As a result, there are a range of commercial and research tools that can be used, in conjunction with or independently from a commercial FPGA mapping toolflow.

Open source tools that can perform synthesis and netlist elaboration include Yosys [151] and ODIN II [69].

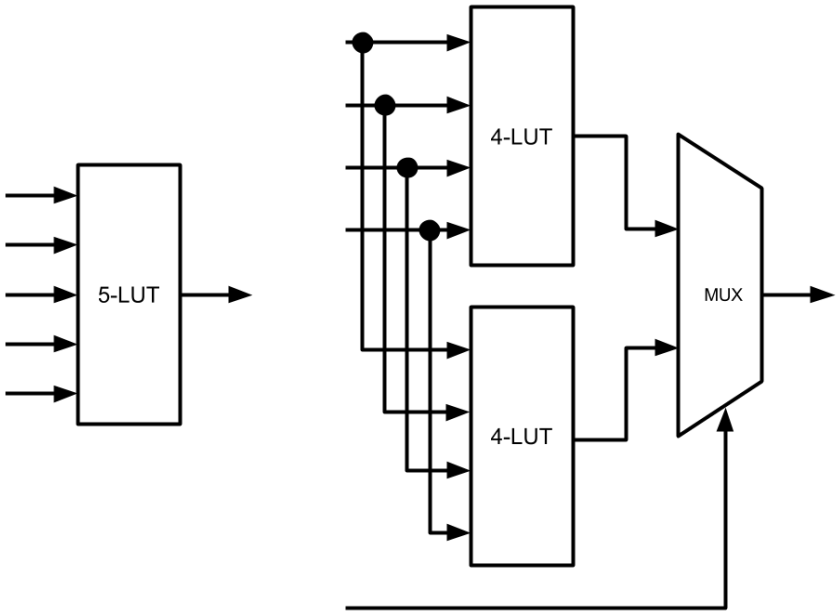


Figure 5.2: Implementing abstract 5-input LUT using two 4-LUTs and MUX.

5.2.2 Technology Mapping

FPGAs consist of a fixed set of hardware, as described in Section 1.3. In the technology mapping phase, the structural netlist produced by synthesis is transformed to use only gates or hardware units that exist in the particular FPGA targeted by the technology mapping tool.

For instance, if a given FPGA architecture only include four-input LUTs, larger LUTs can be implemented by technology mapping as shown in Figure 5.2.

The logic manipulation tool ABC [19] is the most popular open source tool for technology mapping. ABC evolved from the earlier SIS, and manipulates the netlist as a large AND-Inverter graph (AIG), with the ability to optimize for logic depth, gate size, and other criteria.

5.2.3 Clustering

For FPGAs that include Clustered Logic Blocks (CLBs), the technology mapped netlist must next be clustered to match. The optimization goals for the clustering step the same as those described in Section 2.4.

One clustering tool, RASP [32], begins by assigning a weight to every pair of logic elements. A higher weight indicates that it is more desirable to place the two logic blocks near one another. A pair of nodes that cannot be placed in the same cluster (i.e., could not be part of a legal placement as discussed below) are given weight zero. The algorithm then finds a collection of pairs such that (i) total weight is maximal; (ii) each logic element appears in at most one pair. This process can be iterated to construct clusters that have 2^k logic elements for any positive integer k simply by using clusters of size 2^{k-1} in the pair-forming procedure.

While RASP builds all clusters simultaneously, other tools tools builds clusters sequentially, typically through the greedy application of a simple attraction rule. This is the approach taken by VPack [12] — at each iteration of the algorithm, a logic element that has not yet been clustered is selected, and additional logic elements are selected greedily to fill the cluster according to a desirability metric. The well known variant T-VPack [100] improves this desirability metric by more faithfully capturing timing information. It improves critical path delay by preferentially grouping LUTs along the critical path into clusters, to take advantage of the faster intracluster connections compared to global routing [5].

5.2.4 Placement

After a design is technology mapped and clustered, we can assign a physical location within the FPGA fabric to each unit or cluster of units. This step is known as placement, and there are several different methods commonly used. A placement is considered 'legal' once every unit has been assigned to a valid position within the FPGA, respecting the capacity of each tile.

Tools like VPR [12] and TimberWolf [124] use simulated annealing [81] to produce high quality legal placements. An initial (poor quality) placement is produced. At each step of the algorithm, a random perturbations is introduced, and some measurement of cost is computed via a fixed cost function. If overall cost decreases, the perturbation is automatically accepted. If overall cost increases, then the perturbation is accepted with probability $e^{-\Delta/T}$, where Δ is the change in cost induced by the perturbation, and T is the current "temperature" of the algorithm. Temperature starts high and is gradually decreased according to a so-called cooling schedule. In effect, this means that poor perturbations are more likely to be accepted earlier in the simulated annealing process, and become exponentially less likely to be accepted as the cooling schedule progresses. Not surprisingly, this cooling schedule plays an integral role in shaping the overall nature and quality of simulated annealing results, and VPR's success in large part can be attributed to the flexibility with which it sets this schedule [61].

Simulated annealing becomes quite expensive when considering very large arrays, a growing concern since FPGA size doubles every two to three year [61]. In order to have some confidence that the algorithm will produce quality results, the total number of perturbations executed over the course of the cooling schedule must scale with the total number of possible configurations of the system. The

number of placements of N logic elements is approximately $N!$, and so this approach becomes untenable as N grows large. In these cases, we might prefer a more structured approach.

Partition-based placement attempts to assign clusters to physical regions of the FPGA fabric such that connections between regions are minimized. A typical approach is bipartitioning, in which regions are recursively divided in two, with the clusters in the parent region being assigned to one of the child regions. By iteratively taking into account local connectivity information, partition-based routing can reduce total wire length [61]. This approach has been successful in ASIC design for minimizing wire length in cell placement. Note that this strategy can be applied to both hierarchical and island-style FPGAs. We will discuss our contributions to recursive bipartitioning in more detail below.

5.2.5 Routing

With all the logic placed in a physical location within the FPGA, the remaining step to complete the design mapping is to connect the net terminals in the netlist.

Routing is often split into two phases. The first is a global routing phase, in which nets are assigned to routing channels within the global interconnect. Here, congestion can be negotiated and routing resources assigned in a way that allows everything to fit. Global routing is generally performed using a more abstract representation of the interconnect, omitting accurate detail in favor of quicker solutions. This is followed by a second detail routing phase, in which the final connections are made from the global routing channels, through the local interconnect, and to the terminals of the logic.

Routing in the FPGA context is inherently difficult, as it is prohibitively expensive to fully connect components of the FPGA fabric (as discussed in Section 2.3.3. To address this issue, some tools iterate between placement and routing phases in order to perform joint optimization. Improvement to global metrics like average wire length comes at the cost of algorithmic complexity and run time.

The PathFinder routing algorithm [109] operates on a high-level description of FPGA connectivity. This approach is highly advantageous, as it is inherently architecture-agnostic, allowing for rapid and low cost exploration of new architectures [61]. At this level of description, one can leverage well known shortest-path algorithms (e.g. Djikstra) or heuristics (e.g. A* [57]) in order to reduce total routing cost. In all approaches, connections between FPGA elements are assigned a weight based on the routing resources necessary to implement the channel, and a solution representing minimal total routing cost in terms of these weights is sought. Obviously, the choice of weights is critical the final quality of a routing solution, and much work has gone into finding weighting functions that faithfully represent network properties like congestion [113], fanout [14], and heterogeneous path lengths [51].

At the end of the routing step, the design has been fully mapped to the FPGA — every unit and net has an assigned position and dedicated resources within the FPGA. Research-oriented toolflows such as VPR can stop at this level of description; all information needed for detailed simulation is in hand. In order to write this specification onto a physical FPGA, we need one additional step: bitstream generation.

5.2.6 Bitstream Generation

Bitstream generation is the process of creating the configuration memory image for a mapped FPGA design. It is by its nature the most device-specific step in the toolflow, and it requires knowledge of what function each bit of the configuration memory controls. This process is comprehensive, in that it includes the function programmed into every LUT, the position of each MUX, passgate, etc. After bitstream generation, the design is finally ready to be implemented on the physical FPGA by loading or burning the bitstream image into the configuration memory.

In commercial FPGAs, the details of the bitstream are generally somewhat secret, since detailed knowledge of the FPGA architecture that they represent could allow for reverse engineering of the hardware. No major FPGA manufacturer has publicly disclosed the bitstream format for their devices since the Xilinx XC6200 series in 1998 [110]. As a result bitstream generation has in general been closed source, but there do exist a few open source options for limited families of hardware that have allowed for research exploration. JBits [116] manipulated configuration bitstreams for the Xilinx XC4000 and Virtex device families, abits [110] targeted the Atmel FPLSIC series, and the more recent IceStorm [152] supports Lattice iCE40 FPGAs.

5.3 Asynchronous Performance

In this section we describe the foundations of asynchronous performance modeling, indicate how asynchronous optimization criteria differ from synchronous, and provide guidelines useful for designing both AFPGA architectures and the associated

mapping tools.

Our goal is maximize system throughput, defined as the number of operations per unit time. Throughput has units of Hertz, and is directly analogous to synchronous frequency.

By definition, all operations in a synchronous system are paced by a global clock. The period of this clock is set by the longest path between two stateholding elements (flip-flops) for all such paths in the system (see figure in Section 1.2). Synchronous FPGA mapping tools attempt to reduce this critical path to increase maximum system throughput.

By contrast, our AFPGA architecture is physically pipelined throughout at a very fine granularity. The "longest path between two stateholding elements" is represented by a single handshaking channel, and its timing is relatively uniform throughout the AFPGA.

The limits to performance in an asynchronous system are the true data dependencies in the dataflow graph. These algorithmic data dependencies also limit synchronous throughput, whether or not they are the determining factor in setting the clock period for a given implementation. Data dependencies are manifested within an asynchronous dataflow graph as token loops and reconvergent paths.

We don't just want to compute pipeline performance, we want to optimize it as part of our mapping process. Our main tool in this effort is the ability to vary the number of buffer stages on a given path within the dataflow graph by altering the mapping. Assuming the system is slack elastic [98] like our AFPGA architecture, altering the path slack can change the performance without impacting correctness. Logically this makes sense: since the channel communications are delay-insensitive,

downstream processes cannot detect that extra slack has been added.

Unfortunately, Kim [79,80] demonstrated that slack optimization is NP-complete (by reduction to 3SAT). Regardless, we can use pipeline performance to guide our async-aware mapping heuristics.

5.3.1 Modeling Pipeline Performance

The classic formulation [92, 150] for modeling asynchronous pipeline performance is to consider a linear pipeline of buffer stages. Each buffer stage is characterized by its forward latency, reverse latency, and maximum stage throughput T, which is limited by its local cycle time.

A pipeline has static slack s (related to the number of buffer stages), which indicates the maximum number of tokens that can occupy the pipeline without stalling.

It also has a dynamic slack d, which is the number of tokens ($< s$) for which pipeline's throughput peaks. If throughput is limited by internal cycle time, this is a range d_{min} to d_{max}.

Given these pipeline and buffer characteristics, we can derive an expression for the throughput γ of the pipeline as a function of the number of tokens x it contains:

$$\gamma(x) = \begin{cases} T\frac{x}{d_{min}} & \text{if } x < d_{min} \\ T & \text{if } d_{min} \leq x < d_{max} \\ T\frac{s-x}{s-d_{min}} & \text{if } d_{min} \leq x < d_{max} \end{cases} \tag{5.1}$$

Figure 5.4 shows the classic 'umbrella' plot that results, representing how the throughput of a pipeline changes with the number of tokens resident. There are three main phases of this plot. The first (left-most) phase is token-limited — tokens can race through the pipeline without interference, so adding more increases throughput (positive slope). The final (right-most) phase is known as bubble-limited or hole-limited — tokens in the pipeline begin to collide and their transit of the pipeline is limited by the free space ahead of them, so adding more decreases throughput (negative slope). The point where these two slopes intersect is the maximum throughput d. If the throughput is limited by each buffer's internal cycle time, there is an additional central control-limited phase, and the performance triangle is truncated into a trapezoid.

5.3.2 Loops

An example dataflow loop structure is shown in Figure 5.3. Here, an output of a copy node eventually feeds back to its input. The source and sink in the example attempt to send/consume tokens at the highest possible throughput, so the performance of the structure is limited only by the loop. Note that every loop must have at least one token (otherwise nothing can ever happen), and the number of tokens in a loop is assumed to be constant.

Buffer loops are the first and easiest place to operationalize the performance

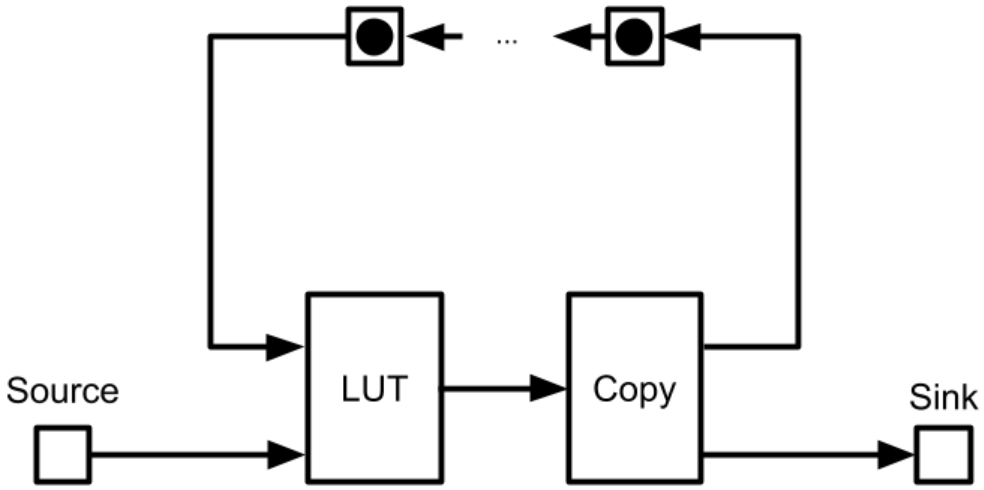


Figure 5.3: Dataflow loop

model discussed above. A loop is simply a linear pipeline with the output connected to the input, so the model directly applies and we can see the classic trapezoid in Figure 5.4 as the number of tokens is varied within a loop with fixed static slack.

This performance data can help us optimize both the AFPGA architecture and the mapping flow to achieve maximum throughput. Unfortunately, it is not in the most useful form. In general, the number of tokens in a dataflow graph loop is determined by the algorithm it implements. On the other hand, thanks to slack elasticity we have the ability to adjust the static slack of the loop by adding or removing buffer stages.

Figure 5.5 shows the throughput achieved for various amounts of static slack, given a fixed number of tokens within the loop. We can use this information as a lookup table to determine the throughput for any given scenario.

Figure 5.6 shows the final and most useful form of the same simulation data. It shows how the loop throughput varies as a function of the static slack *per token*

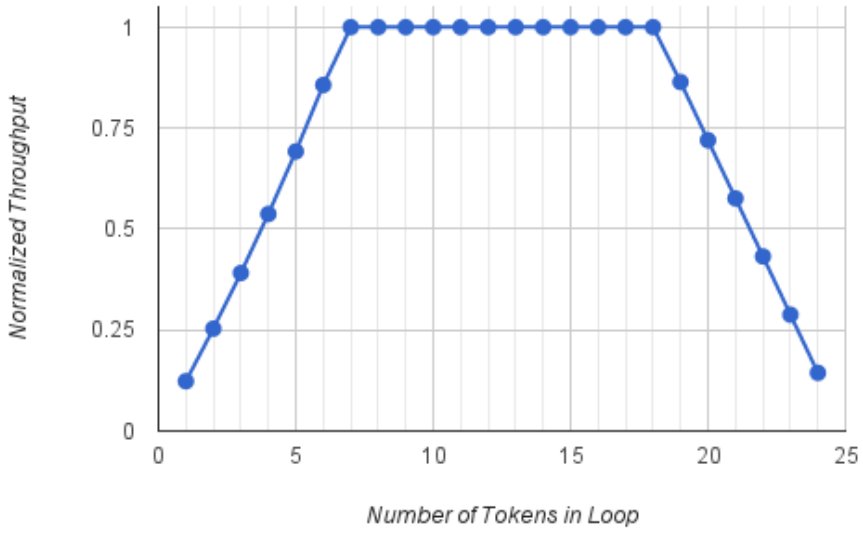


Figure 5.4: Throughput in a loop with static slack $s = 25$ as the number of tokens x is varied

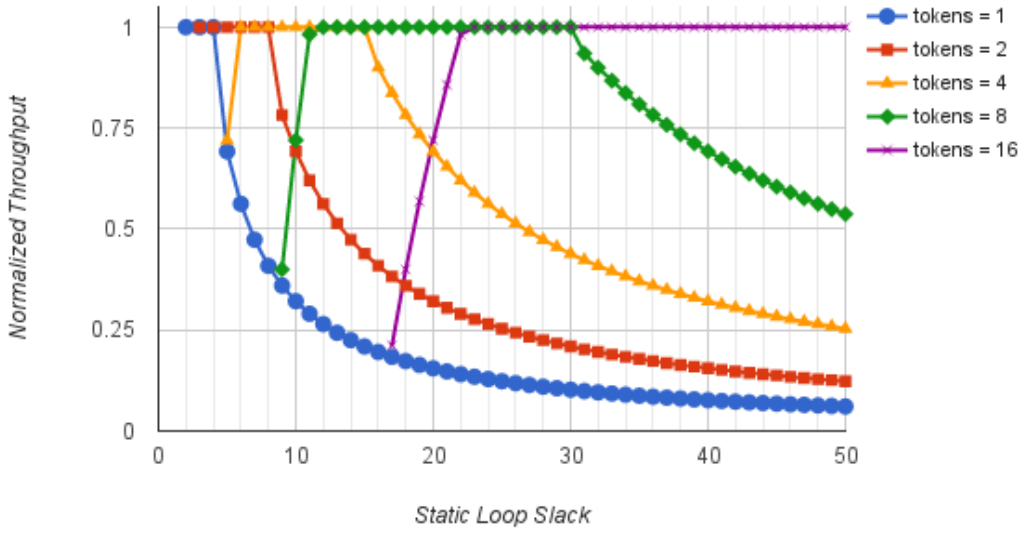


Figure 5.5: Loop throughput versus static slack for loops with varying numbers of tokens

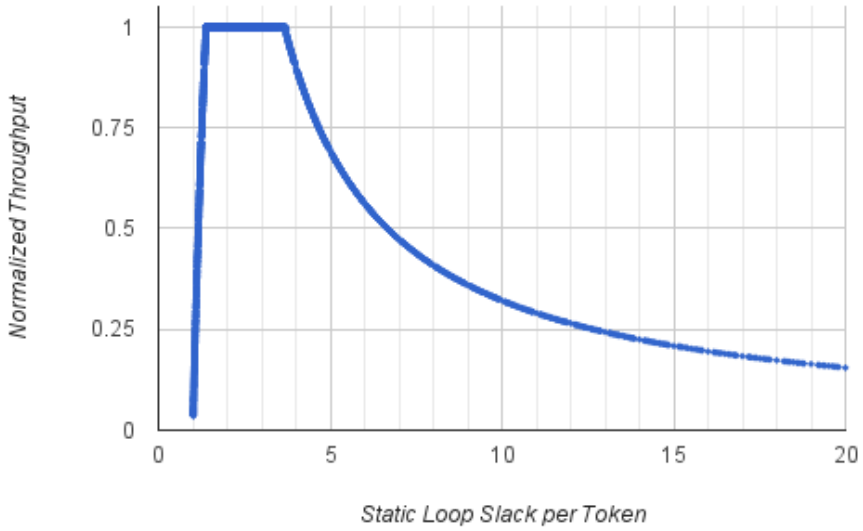


Figure 5.6: Loop throughput versus static loop slack per token

in the loop. Tokens traveling in a pipeline spread out across multiple stages, and at optimal throughput the slack surrounding each token is known as its dynamic wavelength [150] ($= \frac{1}{d}$). This makes logical sense, as e.g. a loop with one token and $s = 5$ will have the same throughput as a loop with two tokens and $s = 10$.

From Figure 5.6, we can directly determine how many slack buffers should be included in any loop in the dataflow graph, as well as the negative performance implications if the actual slack differs from the ideal. Given the particular buffer characteristics simulated here, we can see that as long the loop has slack per token between 1.5 and 4, it can reach peak throughput.

When mapping dataflow graphs to the AFPGA, it is extremely unlikely that loops will be token-limited due to the extra routing buffers in the AFPGA interconnect. As a result, our main optimization task is to keep all loops short enough to achieve high throughput.

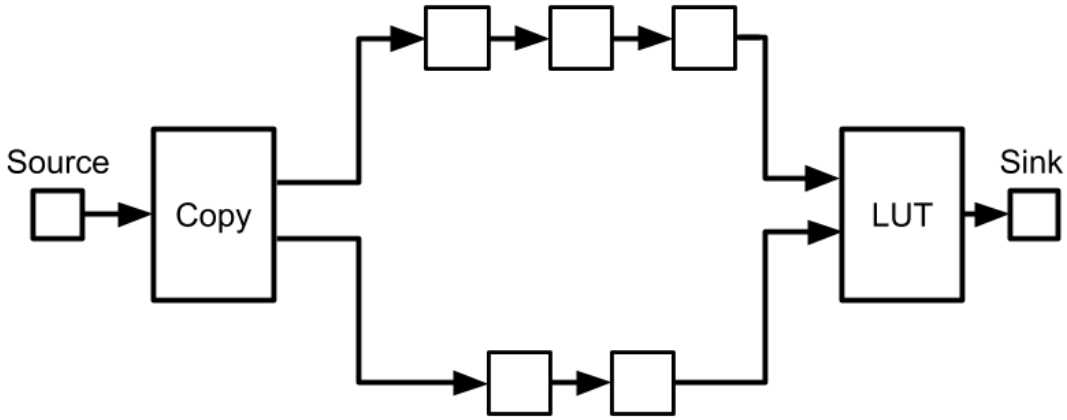


Figure 5.7: Reconvergent path

5.3.3 Reconvergent Paths

The other structure in a dataflow graph that can limit throughput is the reconvergent path, shown in Figure 5.7. In this scenario, two linear pipelines (with potentially different static slack) are fed by the same source and terminate at the same sink. As a result, the number of tokens in each branch of the path must be identical, which can force one of the branches into the token- or hole-limited regime if the paths are not balanced.

Just as we did for the loops we can distill the performance impact of reconvergent paths into Figure 5.8, which lets us tabulate throughput for any reconvergent path with path slacks (A, B).

If we try to create a unified metric similar to static loop slack per token, the story is not as quite as clean in the case of reconvergent paths. Figure 5.9 shows the throughput of reconvergent paths based upon the ratio of their path slacks $\frac{A}{B}$, where ratio of 1 indicates that the paths are completely slack matched. Unfortunately, sets of reconvergent paths with the same ratio can have differing thoughputs: path

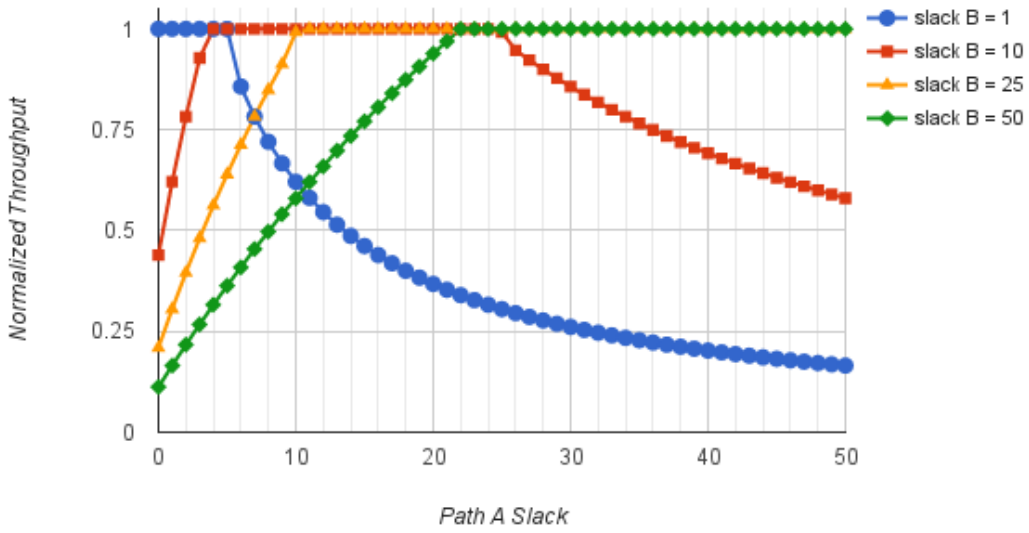


Figure 5.8: Reconvergent path throughput for varying combinations of path slack

$(5, 1)$ can operate at full throughput while $(50, 10)$ has throughput cut nearly in half, even though both have a ratio of 0.2.

Fortunately, we can still derive useful design guidelines from the path ratio data in Figure 5.9. For instance, given the particular buffer characteristics simulated here and regardless of absolute slack, one path can more than twice as long as the other before we start to see any throughput degradation. Additionally, if the path slacks diverge beyond that, we can see a clear lower bound for the resulting throughput.

The main designer intuition available here is that there is a lot more leeway available in rebalancing reconvergent paths than there is in setting loop slack. Thus, our first priority in mapping for asynchronous performance is to optimally map short loops, after which we can rebalance reconvergent paths if necessary by adding slack to the shorter path.

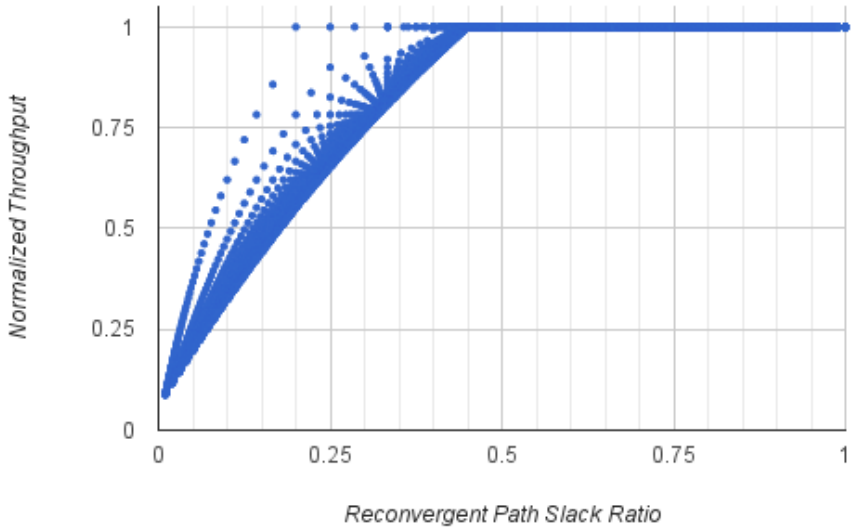


Figure 5.9: Reconvergent path throughput for varying path slack ratios

5.4 Asynchronous FPGA Mapping Toolflow

We have constructed a mapping toolflow for asynchronous FPGAs, which is able to take a high-level design through to placement and routing, while also emitting intermediate information to help with architectural exploration. We adopt many of the approaches outlined in the synchronous toolflow description above with several important changes which we will outline below. The largest change is an async-aware partition-based clustering and placement step, which is discussed separately in Section 5.5.

5.4.1 Synthesis

The output of the synchronous synthesis step is a structural netlist, but the true starting point of the AFPGA mapping flow is a dataflow graph.

Peng et al. developed a synthesis procedure for generating such a graph [119, 143]. This process starts with CHP, which is then converted to Static Token form (a variant of Static Single Assignment), and finally decomposed via projection into the dataflow blocks used in our AFPGA architecture (Section 2.2).

Wong et al. also created a similar synthesis tool using data-driven decomposition [153]. It begins with sequential CHP, converts this to Dynamic Single Assignment form, and projects the into concurrent PCHB processes. These processes are more general than our dataflow primitives, but each may be mapped to a collection of simpler dataflow blocks.

Dataflow graphs for the AFPGA could also be generated via synchronous to asynchronous translation, described in the next section.

5.4.2 Synchronous to Asynchronous Translation

If we want the highest possible performance from an AFPGA, we should begin with an asynchronous dataflow graph optimized for its architecture. Despite this, it is sometimes useful to have access to the broader world of synchronous benchmark designs. Our AFPGA can also run synchronous designs, after they have been mapped to a dataflow graph.

In order to map a synchronous design to a dataflow graph, two main steps are

required:

- All fanout within the synchronous netlist must be replaced with explicit copy processes
- All flip-flops in the synchronous netlist must be replaced with initial token buffers, with the same value token as the flip-flop held at reset

Our toolflow includes a tool to perform this translation. The synchronous design must only have a single clock domain. The results will be cycle accurate/token accurate, but beyond that timing information will not necessarily be preserved.

5.4.3 Bitstream

Our bitstream generator uses an abstract representation of the AFPGA, which has several useful properties.

First, since the bitstream is always generated from the abstract representation, it can never have illegal configurations such as shorts between drivers that could damage the physical FPGA.

It is also fully relocatable, which means that a design can be trivially remapped to anywhere within the AFPGA array. This allows us to pre-compile a design for use in a partial reconfiguration flow, and only insert the physical location at the last minute through a process similar to binary linking in the software context.

The bitstream format supports word-level configuration modifications if necessary, which enables difference-based partial reconfiguration. This granularity

requires you to specify an address for each write which is inefficient when programming large regions of the AFPGA, so it also supports a compressed run-length encoded format. This permits DMA-style block writes with auto-incrementing address generation, similar to the Xilinx configuration frame model.

5.4.4 Routing

In general, the Pathfinder router algorithm is still good for our problem specification. The final detail routing stage is simpler for AFPGAs, because they do not have multi-terminal nets.

We can also do a post-hoc slack matching optimization pass by opportunistically *lengthening* the shorter legs of unbalanced reconvergent paths, as described in Section 5.3.3. A reasonably simple architectural enhancement that would help with this step would be to use un-pipelineable buffers in the routing fabric. These buffers could be configured to have slack 0 and act solely as electrical drivers on the channel, permitting us to reduce slack if needed during the routing stage. This is similar to the concept of retiming registers in synchonous FPGAs [129], but (i) much simpler and cheaper to implement, and (ii) completely optional for correctness.

5.5 Partition-Based Clustering and Placement

In this section, we explain the graph theoretical background underpinning partition-based mapping, describe our implementation of the algorithm, and share results compared to other mapping strategies.

5.5.1 Graph Theory Background

A directed graph $G = (V, E)$ is composed of a collection of vertices V and a collection of edges $E \subseteq V \times V$. This abstraction is extremely general, with applications from modeling modular dependencies to capturing the flow of blood microvasculature. In our context, nodes can be thought of as dataflow elements, and edges as the channels that convey information between these elements. We say that two vertices u and v are adjacent if there is an edge connecting u and v, that is, if (u, v) is in E.

A classic problem in graph theory and theoretical computer science is to find the "minimum cut" of a directed graph, that is, the minimum number of edges that must be severed in order to reduce the graph to two disconnected components. This idea has an immediate extension in which a function $w : E \to \mathbb{R}$ assigns a weight to each edge, and the objective is to produce a cut with minimal total weight. Exact solutions to this problem have been obtainable for a half century, e.g., via Ford-Fulkerson.

Traditional min-cut algorithms make no guarantees as to the size of the disconnected components of the cut graph; it can be the case that one component has many more nodes than the other. Not surprisingly, partitioning becomes more difficult when one attempts to minimize this disparity. The situation is further complicated if we allow nodes to have weights (as we did with edges) and search for a cut that equitably distributes node weight, not just node count, across partitions. One common use case of this approach is the allotment of jobs (nodes) with certain dependencies (edges) and expected run times (weights) across a pool of processors (partitions). Minimizing the cut reduces the volume of between-processor communication, and attempting to allot equal weight to each processor evenly dis-

tributes total load. We will use the same mathematical approach to assign each dataflow element a physical location on the FPGA fabric.

Modern partitioning algorithms take a heuristic approach to deal with the large graphs commonly encountered in practice. The key problem here is that partitioning complexity generally scales with graph size, and so directly partitioning large graphs is not feasible. The METIS suite [73–75] that we will employ in our work here takes a coarsening-partitioning-refining approach to circumvent this issue. The original graph $G_0 = (V_0, E_0)$ is first put through several rounds of coarsening to produce a sequence $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_k = (V_k, E_k)$ of progressively smaller but more highly connected graphs. In step i, a maximal collection of adjacent vertices is found in V_{i-1}, and the vertices in each pair are merged, producing a new node set V_i. This induces a corresponding change in the connectivity information of the graph, resulting in a new set of edges E_i. Notice while the total number of nodes decreases by up to a factor of 2 in each iteration, the overall connectivity of the graph increases, because (intuitively) nodes become closer as a result of the merging procedure.

Once the number of nodes in G_k for some k passes below a predetermined threshold, the coarsening procedure is halted; the graph is now small enough to be quickly partitioned by simple methods. But the partitioning produced here is of the coarsened graph, which may or may not be a quality partitioning of the original graph G_0. METIS takes a refinement approach to deal with this issue. First, the algorithm begins to undo the coarsening procedure by reproducing the sequence of graphs $G_k, G_{k-1}, G_2, G_1, G_0$ via expansion of vertex pairs that were previously merged. At each step, the partitioning applied to G_i is first naively applied to G_{i-1} (i.e., if node v was in partition j in G_i, then the associated unmerged nodes

u', v' are initially placed in partition j in G_{i-1}) and then heuristically improved by trading vertices in G_{i-1} between partitions such that overall partition quality increases. At the end of the refining procedure, we have a quality partition of the original graph G_0 that has been obtained with far less computational effort than would have been necessary to partition G_0 directly.

Note that this method is quite flexible, in the sense that we can tune both edge weights and node weights to capture the various implementation constraints in the system.

5.5.2 Recursive Bipartitioning

We use the graph-based techniques described above to partition a dataflow graph into smaller pieces. The same mechanism is flexible enough to be used for both clustering and placement, depending on input and stop conditions.

The partitioning algorithm begins with a directed graph representation of the dataflow graph for a design. The general procedure is as follows:

1. Partition the input graph into two subgraphs, respecting provided constraints
2. If the new partitions meet the stop condition, return the generated set of partitioned subgraphs.

   Otherwise, start at step 1 with each new subgraph as an input

In order to use this strategy for clustering, begin with the dataflow graph containing all nodes, and stop when the partitions contain the correct number of nodes to form a legal cluster.

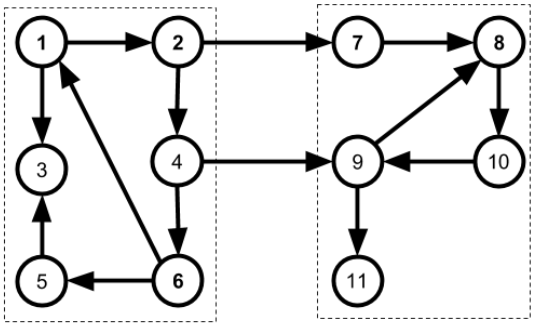
To use the algorithm for placement, begin with an already clustered dataflow graph. Stop when each partition contains the correct number of clusters to form a legal FPGA tile.

You can also perform unified clustering and placement, as shown in Figure 5.10. We begin with the complete dataflow graph, unclustered and unplaced. First, we partition the graph into two subgraphs (Figure 5.10a), physically assigning each half to a fraction of the FPGA fabric (Figure 5.10b). Since each partition is still larger than what can be placed into an FPGA tile, the process continues by partitioning each subgraph (Figure 5.10c). At this point in our example we assume that each of these new subgraphs is small enough to form a cluster, each cluster is assigned to an FPGA tile location (Figure 5.10d), and the combined clustering and placement is complete.

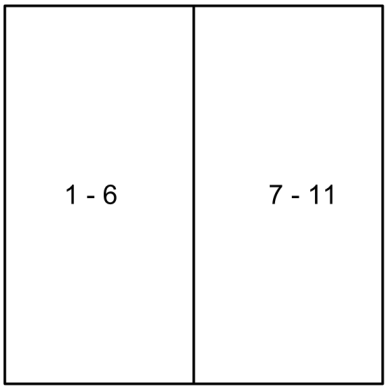
This recursive bipartitioning procedure leads to a physical position within the FPGA grid following the assignment rules shown in Figure 5.11. The numbered arrows show the cut order, and each partition is assigned a grid position from the history of cuts that generated it (using power-of-two indices starting from the center of the grid).

The main mechanism we have to influence the partitioning process are the weights assigned to nodes and edges of the graph during partitioning.

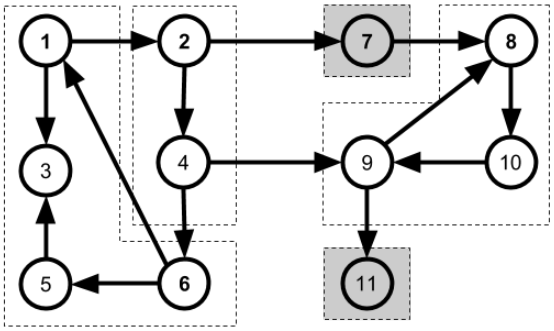
Node weights ensure that the correct units are placed in each partition. Usually we want an equal number of nodes on each side of a bipartition, but this can also be used as an unequal constraint. For instance, we can bias the partitioning by setting an extra "I/O weight" that will serve to force primary inputs and outputs of the dataflow graph toward partitions that will end up on the outside of the grid,



(a) Dataflow graph after step 1



(b) FPGA fabric after step 1



(c) Dataflow graph after step 2

(d) FPGA fabric after step 2

Figure 5.10: Two steps of a recursive bipartitioning procedure applied to a simple dataflow. Note that in some steps, dataflow elements that are not connected in the dataflow (e.g., nodes 7 and 11) may be placed in the the same region of the FPGA fabric (gray region).

while still respecting the total node weight for each partition.

Edge weights by default ensure that connected components of the dataflow graph are placed together. This is generally what we want in a quality placement, since it keeps communication local and decreases use of global routing resources.

In Section 5.3, we showed that the most critical consideration for optimizing performance of asynchronous circuits is to keep token loops short. We capture this condition by searching the graph for loops, and adding extra weight to each

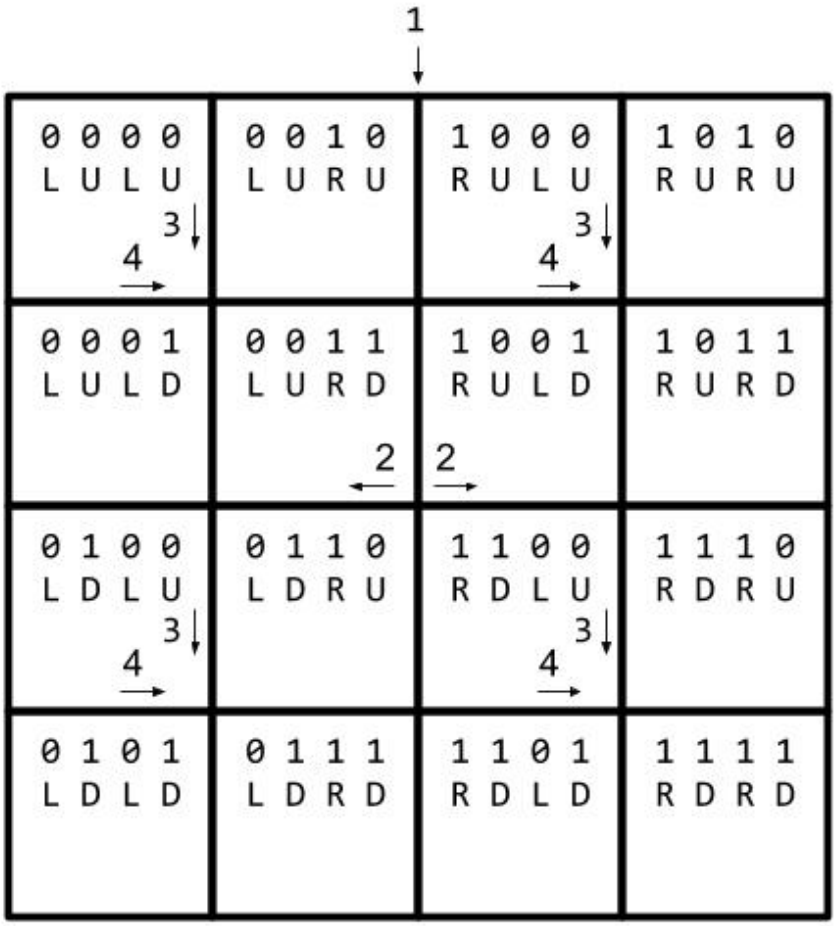


Figure 5.11: Sequence of bipartitioning operations assigns grid position

edge that is part of a loop. Since finding all circuits in a large directed graph is an expensive proposition [72, 106], we make this process tractable by limiting the depth of the search. Intuitively, all the loops found and weighted by this process are the ones we'll be able to improve, by placing them into clusters and/or adjacent tiles.

In the next section, we compare the performance of this recursive partition-based placement strategy (both with and without loop edge weighting) to a traditional simulated annealing placement.

5.5.3 Results

For "async-friendly" dataflow graphs (i.e. those without long loops or unbalanced reconvergent paths), our toolflow finds an optimal full-throughput mapping with ease. This is largely a function of the AFPGA architecture itself, with its fine-grained pipelined interconnect.

To analyse the toolflow further, we turn to less 'friendly' designs. The benchmarks used in the following results are the twenty largest designs in the MCNC benchmark suite, which are commonly used for benchmarking FPGA mapping toolflows [12, 53]. A full description of the benchmarks can be found at [157].

Each benchmark started as a synchronous structural netlist and was run through our asynchronous mapping toolflow. For the results presented, we used ABC for technology mapping and our own tool for synchronous-to-asynchronous translation. The resulting dataflow graphs were then run through three different mapping flows:

1. Partition-based placement
2. Partition-based placement with loop weighting
3. VPR placement using simulated annealing

Since each of these placement alternatives is stochastic, each data point is the best result from five trials with different random seeds. Averages across benchmarks are computed using the geometric mean [49].

Figure 5.12 shows the results of experiments (1) and (2), and highlights the benefit of adding edge weights to loops. Results are shown only for sequential

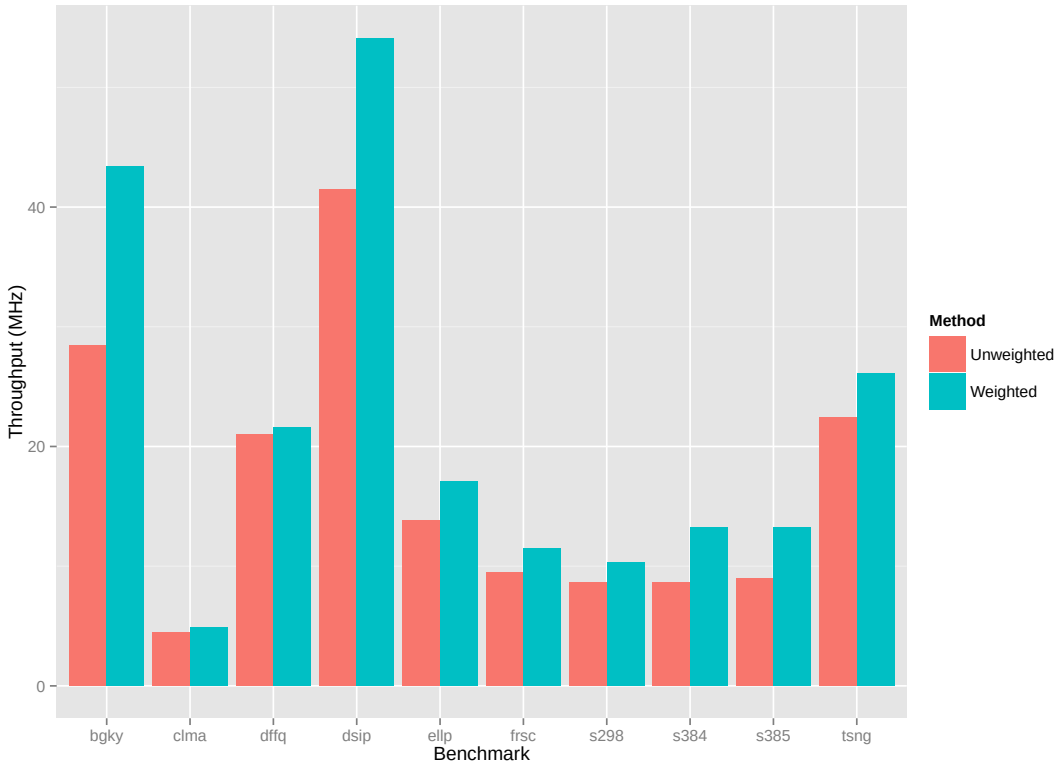


Figure 5.12: Throughput for partition-based placement, with and without loop weighting.
Geometric mean across all benchmarks is 43.11 and 48.49 for unweighted and weighted, respectively.

benchmarks; combinational designs have no loops and thus have identical results. Designs placed with extra loop weight were superior in every case, and showed an average 12.5% throughput improvement over the default equally-weighted graph partitioning.

Figure 5.13 compares the results of experiments (2) and (3), pitting the partition-based placement strategy against VPR's simulated annealing placer. Results here are mixed (partition-based placer yields higher throughput in four of the combinational benchmarks) but ultimately decisive in favor of simulated annealing, which showed an average 52.5% throughput improvement.

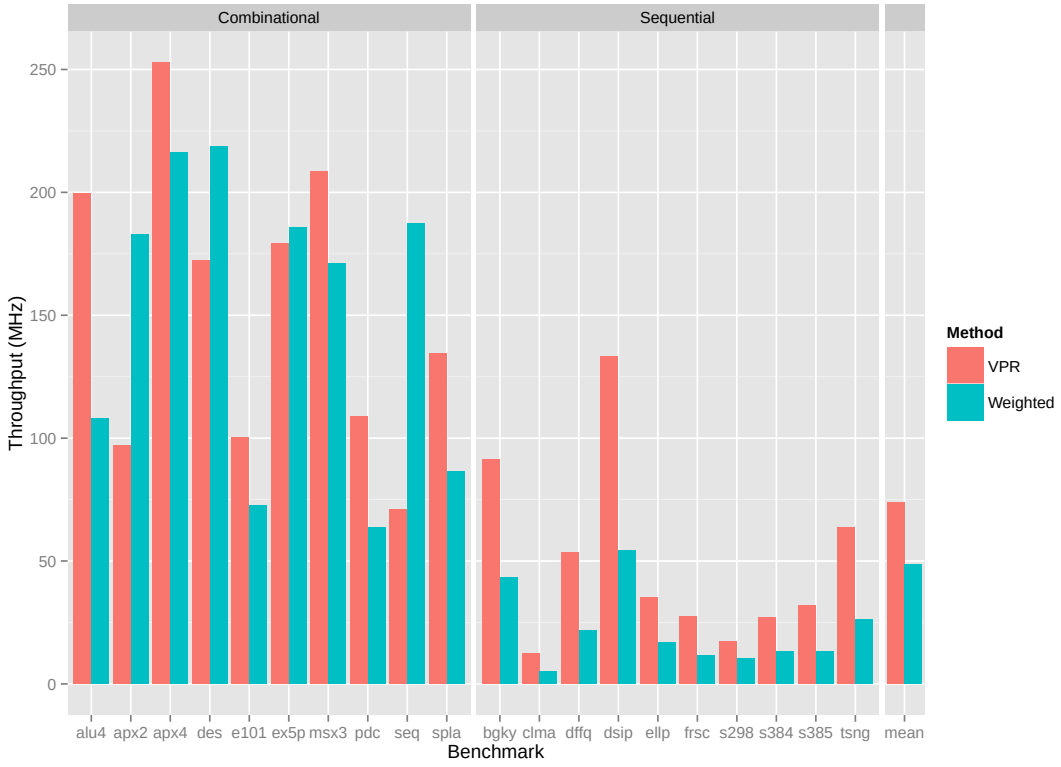


Figure 5.13: Throughput for simulated annealing versus partition-based placement.

5.5.4 Discussion

Partition-based placement is quick and flexible. We can use the same mechanism to perform clustering, placement, and partitioning simultaneously (e.g. ensuring that a fraction of a design that will be dynamically reconfigured is grouped together).

Unfortunately, when operating on 'async-unfriendly' benchmarks, the quality of the resulting mappings was in most cases inferior to that generated by simulated annealing. The partition-based algorithm was able to detect and improve some performance limiting structures in the dataflow graph, but not all of them. The ones that remained limited total achievable throughput, and were actually better suited to the global wirelength minimizing cost function of the simulated annealing placer.

A leading critique of partition-based placement is that the thread of the desired global optimizations, e.g., of total wire length, is lost as the algorithm makes a sequence of cuts based solely on *local* connectivity information [33]. A natural solution to this problem is to look at the connectivity information at a coarser level – instead of considering how each individual logic element is connected to other logic elements, we consider how entire regions of the dataflow are connected to other regions. When viewing the dataflow from this level, we have a much more global picture of overall connectivity (which has come at the price of fine-grain resolution of the dataflow's constituent parts). One additional advantage of coarsening procedures is that placement procedures that were intractable for the large collection of logic elements (e.g., simulated annealing) can easily be applied to this much reduced collection of regions [24]. The key issue here is that an optimal placement of the coarsened depiction of the dataflow may no longer be optimal (or even very good) when directly applied to the full resolved dataflow – a multilevel approach is needed [23]. Ultimately, the best solution may be a hybrid approach, leveraging the scalability and configurability of the graph-based approach with the global perspective of simulated annealing in multiple phases.

APPENDIX A

SUMMARY OF CHP LANGUAGE

The CHP notation we use is based on Hoare's CSP [66]. A full description of CHP and its semantics can be found in [101]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := \mathit{true}$, and $a\downarrow$ for $a := \mathit{false}$.
- Selection: $[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard. The notation $[G]$ is short-hand for $[G \rightarrow \mathit{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the *true* guards and executing the corresponding statement, repeating this until all guards evaluate to *false*. The notation $*[S]$ is short-hand for $*[\mathit{true} \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is *true* iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.

- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

BIBLIOGRAPHY

[1] C Ababei, H Mogal, and K Bazargan. Three-dimensional place and route for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1132–1140, May 2006.

[2] Cristinel Ababei, Pongstorn Maidee, and Kia Bazargan. Exploring Potential Benefits of 3D FPGA Integration. In *Field Programmable Logic 2004*, pages 874–880, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] F Akopyan, C Otero, D Fang, S Jackson, and R Manohar. Variability in 3-D integrated circuits. *IEEE CICC*, pages 659–662, September 2008.

[4] M J Alexander, J P Cohoon, J L Colflesh, J Karro, and G Robins. Three-dimensional field-programmable gate arrays. In *Eighth International Application Specific Integrated Circuits Conference*, pages 253–256. IEEE, September 1995.

[5] Mohab Anis and Hassan Hassan. *Low-Power Design of Nanometer FPGAs*. learning.acm.org, October 2009.

[6] P Athanas, J Bowen, T Dunham, C Patterson, J Rice, M Shelburne, J Suris, M Bucciero, and J Graf. Wires on Demand: Run-Time Communication Synthesis for Reconfigurable Computing. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 513–516. IEEE, 2007.

[7] T Back, F Hoffmeister, and H P Schwefel. A survey of evolution strategies. *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991.

[8] K Banerjee, S.J Souri, P Kapur, and K Saraswat. 3-D ICs: a novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration. In *Proc. IEEE*, 2001.

[9] J Becker, M Hubner, G Hettich, R Constapel, J Eisenmann, and J Luka. Dynamic and Partial FPGA Exploitation. *Proceedings of the IEEE*, 95(2):438–452, 2007.

[10] Peter A Beerel, Recep O Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, February 2010.

[11] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI Journal*, 13(2):103–128, June 1992.

[12] V Betz, J Rose, and A Marquardt. *Architecture and CAD for Deep-Submicron FPGAS*. 1999.

[13] Vaughn Betz and Jonathan Rose. Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size. In *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, pages 551–554, May 1997.

[14] Vaughn Betz and Jonathan Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. *International Workshop on Field Programmable Logic and Applications*, pages 1–10, 1997.

[15] Vaughn Betz and Jonathan Rose. How much logic should go in an FPGA logic block. *Design & Test of Computers, IEEE*, 15(1):10–15, 1998.

[16] M Birla and K Vikram. Partial run-time reconfiguration of FPGA for computer vision applications. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6, April 2008.

[17] S Borkar. 3D integration for energy efficient system design. In *IEEE DAC*, pages 214–219, 2011.

[18] G Borriello, C Ebeling, S A Hauck, and S Burns. The Triptych FPGA architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):491–501, 1995.

[19] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] James Burns. TSV-Based 3D Integration. In *link.springer.com.proxy.library.cornell.edu*, pages 13–32. Springer US, Boston, MA, November 2010.

[21] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score). In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, pages 605–614, London, UK, UK, 2000. Springer-Verlag.

[22] Alesandro Cevrero, Panagiotis Athanasopoulos, Hadi Parandeh-Afshar, Maurizio Skerlj, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. Using 3D integration technology to realize multi-context FPGAs. In *2009 International Conference on Field Programmable Logic and Applications (FPL)*, pages 507–510. IEEE, September 2009.

[23] Tony F Chan, Jason Cong, Tim Kong, Joseph R Shinnerl, and Kenton Sze. *An Enhanced Multilevel Algorithm for Circuit Placement*. IEEE Computer Society, November 2003.

[24] Chin-Chih Chang, J Cong, Zhigang Pan, and Xin Yuan. Multilevel global placement with congestion control. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(4):395–409, April 2003.

[25] N Chase, M Rademacher, E Goodman, R Averill, and R Sidhu. A Benchmark Study of Multi-Objective Optimization Methods. Technical Report BMK-3021, Red Cedar Technology.

[26] Deming Chen, Jason Cong, and Peichen Pan. FPGA Design Automation: A Survey. *Foundations and Trends® in Electronic Design Automation*, 1(3):195–334, 2006.

[27] C Chiasson and V Betz. Should FPGAS abandon the pass-gate? In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, 2013.

[28] D G Chinnery and K Keutzer. Closing the gap between ASIC and custom: an ASIC perspective. In *Design Automation Conference, 2000. Proceedings 2000*, pages 637–642, 2000.

[29] David Chinnery and Kurt Keutzer. *Closing the Gap Between ASIC & Custom*. Tools and Techniques for High-Performance ASIC Design. Springer Science & Business Media, May 2007.

[30] S Chiricescu, M Leeser, and M M Vai. Design and analysis of a dynamically reconfigurable three-dimensional FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):186–196, February 2001.

[31] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *Computing Surveys (CSUR*, 34(2), June 2002.

[32] J Cong, J Peck, and Yuzheng Ding. RASP: A General Logic Synthesis

System for SRAM-Based FPGAs. In *Fourth International ACM Symposium on Field-Programmable Gate Arrays*, pages 137–143. IEEE, 1996.

[33] Jason Cong, Joseph R Shinnerl, Min Xie, Tim Kong, and Xin Yuan. Large-scale circuit placement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(2):389–430, April 2005.

[34] Mark E Dean, Ted E Williams, and David L Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In *Advanced Research IN VLSI*, pages 55–70. IEEE ICCD, April 1991.

[35] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *Kan-GAL report*, 200001, 2002.

[36] Andre DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, MIT, September 1996.

[37] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30(6):334–354, September 2006.

[38] André DeHon and John Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, June 1999.

[39] J Depreitere, H Neefs, H Marck, J Campenhout, R Baets, B Dhoedt, H Thienpont, and I Veretennicoff. An optoelectronic 3-D Field Programmable Gate Array. In *Field-Programmable Logic Architectures, Synthesis and Applications*, pages 352–360, Berlin, Heidelberg, June 1994. Springer Berlin Heidelberg.

[40] S Donthi and R Haggard. A survey of dynamically reconfigurable FPGA devices. *System Theory, 2003. Proceedings of the 35th Southeastern Symposium on*, pages 422–426, 2003.

[41] C Ebeling, L McMurchie, S A Hauck, and S Burns. Placement and routing tools for the Triptych FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):473–482, December 1995.

[42] Oussama Elissati, Eslam Yahya, Sébastien Rieubon, and Laurent Fesquet.

Optimizing and Comparing CMOS Implementations of the C-Element in 65nm Technology: Self-Timed Ring Case. In *link.springer.com*, pages 137–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[43] D Fang, John Teifel, and Rajit Manohar. A high-performance asynchronous FPGA: test results. *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 271–272, April 2005.

[44] David Fang. Width-Adaptive and Non-Uniform Access Asynchronous Register Files. Master's thesis, Cornell University, Cornell University, 2004.

[45] David Fang, C Otero, F Akopyan, and Rajit Manohar. Data-Dependent Slack Matching. Technical Report CSL-TR-2005-1046, Cornell University, December 2005.

[46] David Fang, Song Peng, Christopher LaFrieda, and Rajit Manohar. A Three-Tier Asynchronous FPGA. In *Proceedings of the 23rd International VLSI/ULSI Multilevel Interconnection Conference (VMIC) 2006*, pages 1–8. Cornell University, September 2006.

[47] T Felicijan and S.B Furber. An asynchronous ternary logic signaling system. *IEEE VLSI*, 11(6):1114–1119, 2003.

[48] M Ferretti and P.A Beerel. Single-track asynchronous pipeline templates using 1-of-N encoding. In *IEEE DATE*, pages 1008–1015, 2002.

[49] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[50] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 2012.

[51] R Fung, Vaughn Betz, and W Chow. Simultaneous short-path and long-path timing optimization for FPGAs. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 838–845, November 2004.

[52] Aman Gayasen, Vijaykrishnan Narayanan, Mahmut Kandemir, and Arifur Rahman. Designing a 3-D FPGA: Switch Box Architecture and Thermal

Issues. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(7):882–893, June 2008.

[53] Varghese George and Jan M Rabaey. *Low-Energy FPGAs — Architecture and Design*. Architecture and Design. Springer Science & Business Media, June 2001.

[54] S.M Gilla, M Roncken, and I Sutherland. Long-Range GasP with Charge Relaxation. *IEEE ASYNC*, pages 185–195, 2010.

[55] E Grass and S Jones. Asynchronous circuits based on multiple localised current-sensing completion detection. In *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on*, pages 170–177, 1995.

[56] T R Harris, S Priyadarshi, S Melamed, C Ortega, R Manohar, S R Dooley, N M Kriplani, W R Davis, P D Franzon, and M B Steer. A Transient Electrothermal Analysis of Three-Dimensional Integrated Circuits. *IEEE CPMT*, 2(4), 2012.

[57] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[58] R Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01 Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, March 2001.

[59] S Hauck. Asynchronous design methodologies: an overview. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 83(1):69–93, 1995.

[60] Scott Hauck, Steven Burns, Gaetano Borriello, and Carl Ebeling. An FPGA for implementing asynchronous circuits. *Design & Test of Computers, IEEE*, 11(3):60, 1994.

[61] Scott Hauck and André DeHon. *Reconfigurable Computing*. learning.acm.org, November 2007.

[62] M Hiibner, C Schuck, M Kiihnle, and J Becker. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic

circuits. *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 00, 2006.

[63] Benjamin Hill, Robert Karmazin, Carlos Tadeo Ortega Otero, Jonathan Tse, and Rajit Manohar. A split-foundry asynchronous FPGA. In *2013 IEEE Custom Integrated Circuits Conference*, pages 1–4. Cornell University, IEEE, September 2013.

[64] Quoc Thai Ho, Jean-Baptiste Rigaud, Laurent Fesquet, Marc Renaudin, and Robin Rolland. Implementing Asynchronous Circuits on LUT Based FPGAs. In *link.springer.com*, pages 36–46. Springer Berlin Heidelberg, Berlin, Heidelberg, August 2002.

[65] Ron Ho, Kenneth W Mai, and Mark A Horowitz. The future of wires. In *Proc. IEEE*, pages 490–504, 2001.

[66] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 1978.

[67] E Hung, F Eslami, and S J E Wilton. Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices. *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 45–52, 2013.

[68] N Huot, H Dubreuil, L Fesquet, and M Renaudin. FPGA architecture for multi-style asynchronous logic [full-adder example]. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 32–33 Vol. 1. IEEE, 2005.

[69] P Jamieson, K Kent, F Gharibian, and L Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 149–156, 2010.

[70] Xin Jia, Jayanthi Rajagopalan, and Ranga Vemuri. A Dynamically Reconfigurable Asynchronous FPGA Architecture. In *Field Programmable Logic and Application, 14th International Conference*, pages 836–841, 2004.

[71] Xin Jia and R Vemuri. The GAPLA: a globally asynchronous locally synchronous FPGA architecture. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 291–292. IEEE, 2005.

[72] Donald B Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.*, 4(1):77–84, March 1975.

[73] G Karypis and V Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, page 28, 1998.

[74] George Karypis. METIS 5.0 Manual. Technical report, University of Minnesota, March 2013.

[75] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998.

[76] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[77] Eric Keller. Building Asynchronous Circuits with JBits. In *link.springer.com*, pages 628–632. Springer Berlin Heidelberg, Berlin, Heidelberg, August 2001.

[78] A Khoche and E Brunvand. Testing self-timed circuits using partial scan. In *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on*, pages 160–169, May 1995.

[79] Sangyun Kim. Pipeline Optimization for Asynchronous Circuits. pages 1–124, March 2003.

[80] Sangyun Kim and P A Beerel. Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithm. In *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, pages 296–302, 2000.

[81] S Kirkpatrick, C D Gelatt, and M P Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[82] Dirk Koch. *Partial Reconfiguration on FPGAs*, volume 153 of *Lecture Notes in Electrical Engineering*. Springer New York, New York, NY, 2013.

[83] Dirk Koch, Christian Beckhoff, and Jüergen Teich. A communication architecture for complex runtime reconfigurable systems and its implementation

on spartan-3 FPGAs. *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, February 2009.

[84] Y Komatsu, M Hariyama, and M Kameyama. An Asynchronous Field-Programmable VLSI Using LEDR/4-Phase-Dual-Rail Protocol Converters. *ERSA*, 2009.

[85] Yoshiya Komatsu, Shota Ishihara, Masanori Hariyama, and Michitaka Kameyama. An implementation of an asychronous FPGA based on LEDR/four-phase-dual-rail hybrid architecture. In *ASPDAC '11: Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 89–90. IEEE Press, January 2011.

[86] R et al Konishi. PCA-1: a fully asynchronous, self-reconfigurable LSI. In *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 54–61, February 2001.

[87] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, February 2006.

[88] Christopher LaFrieda, Benjamin Hill, and Rajit Manohar. An Asynchronous FPGA with Two-Phase Enable-Scaled Routing. In *IEEE ASYNC*, pages 141–150. IEEE, 2010.

[89] Christopher LaFrieda and Rajit Manohar. Reducing Power Consumption with Relaxed Quasi Delay-Insensitive Circuits. *IEEE ASYNC*, pages 217–226, May 2009.

[90] C Lavin, M Padilla, J Lamprecht, P Lundrigan, B Nelson, and B Hutchings. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 349–355. IEEE, 2011.

[91] Mingjie Lin, Abbas El Gamal, Yi-Chang Lu, and Simon Wong. Performance Benefits of Monolithically Stacked 3-D FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):216–229, February 2007.

[92] A.M Lines. Pipelined Asynchronous Circuits. Master's thesis, California Institute of Technology, June 1995.

[93] Dake Liu and C Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE SSC*, 29(6), June 1994.

[94] Hock Soon Low, Delong Shang, Fei Xia, and A Yakovlev. Variation Tolerant AFPGA Architecture. In *Asynchronous Circuits and Systems (ASYNC), 2011 17th IEEE International Symposium on*, pages 77–86, 2011.

[95] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Fang, and Jonathan Rose. VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, February 2009.

[96] Kapilan Maheswaran and Venkatesh Akella. PGA-STC: programmable gate array for implementing self-timed circuits. *International Journal of Electronics*, 84(3):255–267, March 1998.

[97] Rajit Manohar. Reconfigurable Asynchronous Logic. In *Conference 2006, IEEE Custom Integrated Circuits*, pages 13–20, September 2006.

[98] Rajit Manohar and Alain J Martin. Slack Elasticity in Concurrent Computing. In *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, pages 1–14, 1998.

[99] Jotham Vaddaboina Manoranjan and Kenneth S Stevens. An a-FPGA architecture for relative timing based asynchronous designs. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2014.

[100] Alexander Sandy Marquardt, Vaughn Betz, and Jonathan Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. *the 1999 ACM/SIGDA seventh international symposium*, pages 37–46, February 1999.

[101] A. J. Martin. Compiling Communicating Processes for Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1986.

[102] AJ Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *6th MIT Conference on Advanced Research in VLSI*. Proceedings of the 6th MIT Conference on Advanced Research in VLSI, 1990.

[103] A.J. Martin and M Nystrom. Asynchronous Techniques for System-on-Chip Design. *Proc. IEEE*, 94(6):1089–1120, June 2006.

[104] Alain J Martin. The Probe: An Addition to Communication Primitives. *Information Processing Letters*, 20(1), 1985.

[105] Alain J Martin. A synthesis method for self-timed VLSI circuits. In *ICCD '87: IEEE International Conference on Design: VLSI in Computers and Processors, Proceedings of*, pages 224–229. California Institute of Technology, October 1987.

[106] Prabhaker Mateti and Narsingh Deo. On Algorithms for Enumerating All Circuits of a Graph. *SIAM J. Comput.*, 5(1):90–99, March 1976.

[107] E McDonald. Runtime FPGA Partial Reconfiguration. In *Aerospace Conference, 2008 IEEE*, pages 1–7, March 2008.

[108] W F McLaughlin, A Mitra, and S M Nowick. Asynchronous Protocol Converters for Two-Phase Delay-Insensitive Global Communication. *IEEE VLSI*, 17(7):923–928, July 2009.

[109] L McMurchie and C Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Third International ACM Symposium on Field-Programmable Gate Arrays*, pages 111–117. IEEE, 1995.

[110] Megacz. A Library and Platform for FPGA Bitstream Manipulation. *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 45–54, 2007.

[111] W M Meleis, M Leeser, P Zavracky, and M M Vai. Architectural design of a three dimensional FPGA. In *Seventeenth Conference on Advanced Research in VLSI*, pages 256–268. IEEE Comput. Soc, September 1997.

[112] Chris J Myers. *Asynchronous Circuit Design*. John Wiley & Sons, April 2004.

[113] R Nair. A Simple Yet Effective Technique for Global Wiring. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(2):165–172, March 1987.

[114] C Ortega, J Tse, and Rajit Manohar. Static Power Reduction Techniques for

Asynchronous Circuits. In *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium on*, pages 52–61. IEEE Computer Society, 2010.

[115] Carlos Tadeo Ortega Otero, Jonathan Tse, Robert Karmazin, Benjamin Hill, and Rajit Manohar. Automatic obfuscated cell layout for trusted split-foundry design. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 56–61. IEEE, 2015.

[116] C Patterson and S Guccione. JBits Design Abstractions. *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 251–252, 2001.

[117] R Payne. Asynchronous FPGA architectures. *Computers and Digital Techniques, IEE Proceedings*, 143(5):282–286, September 1996.

[118] Rob Payne. Self-Timed FPGA Systems. *FPL '95: Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, 975:21–35, September 1995.

[119] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated synthesis for asynchronous FPGAs. *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, February 2005.

[120] Jean-Mark Philippe, Ekue Kinvi-Boh, Sebastien Pillement, and Oliver Sentieys. An energy-efficient ternary interconnection link for asynchronous systems. *IEEE ISCAS*, pages 4 pp.–1014, 2006.

[121] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeff Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 77–86. ACM, 2012.

[122] Andrew Royal and Peter Y K Cheung. Globally Asynchronous Locally Synchronous FPGA Architectures. In *link.springer.com*, pages 355–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[123] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh. A quick safari through the reconfiguration jungle. *Design Automation ...*, pages 172–177, 2001.

[124] C Sechen and A Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *Solid-State Circuits, IEEE Journal of*, 20(2):510–522, 1985.

[125] P Sedcole, B Blodget, J Anderson, P Lysaghi, and T Becker. Modular partial reconfiguration in Virtex FPGAs. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 211–216, August 2005.

[126] Cotton Seed. Arachne-pnr. `https://github.com/cseed/arachne-pnr`.

[127] Delong Shang, Fei Xia, and A Yakovlev. Asynchronous FPGA architecture with distributed control. *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1436–1439, 2010.

[128] Akshay Sharma, Katherine Compton, Carl Ebeling, and Scott Hauck. Exploration of pipelined FPGA interconnect structures. *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, February 2004.

[129] Akshay Sharma, Carl Ebeling, and Scott Hauck. PipeRoute: a pipelining-aware router for FPGAs. *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, February 2003.

[130] Signetics. *Fully Encoded, Random Access Write-Only Memory*, April 1972.

[131] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: towards an open-source tool flow. In *FPGA '11: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM Request Permissions, February 2011.

[132] K.S Stevens. Energy and performance models for clocked and asynchronous communication. In *IEEE ASYNC*, pages 56–66, 2003.

[133] I Sutherland. Micropipelines. *CACM*, 32(6), June 1989.

[134] I Sutherland and S Fairbanks. GasP: a minimal FIFO control. *IEEE ASYNC*, pages 46–53, 2001.

[135] I E Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[136] Christer H Svensson and Ji-Ren Yuan. A 3-level asynchronous protocol for a differential two-wire communication link. *IEEE JSSC*, 29(9), September 1994.

[137] M B Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136. IEEE, 2012.

[138] F te Beest and A Peeters. A multiplexer based test method for self-timed circuits. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pages 166–175, February 2005.

[139] John Teifel. *Fast Prototyping of Asynchronous Logic*. PhD thesis, Cornell University, Cornell University, June 2004.

[140] John Teifel and Rajit Manohar. Programmable Asynchronous Pipeline Arrays. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 1–10. Cornell University, May 2003.

[141] John Teifel and Rajit Manohar. An asynchronous dataflow FPGA architecture. *Computers, IEEE Transactions on*, 53(11):1376–1392, November 2004.

[142] John Teifel and Rajit Manohar. Highly pipelined asynchronous FPGAs. *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, February 2004.

[143] John Teifel and Rajit Manohar. Static tokens: using dataflow to automate concurrent pipeline synthesis. In *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pages 17–27, 2004.

[144] R Tessier, K Pocek, and A DeHon. Reconfigurable Computing Architectures. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 103(3):332–354, 2015.

[145] Jonathan Tse, Benjamin Hill, and Rajit Manohar. A Bit of Analysis on Self-Timed Single-Bit On-Chip Links. In *Asynchronous Circuits and Systems (ASYNC), 2013 19th IEEE International Symposium on*, pages 124–133, 2013.

[146] K van Berkel, A Peeters, and F te Beest. Adding synchronous and

LSSD modes to asynchronous circuits. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 161–170, March 2002.

[147] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Taylor. Conservation cores: reducing the energy of mature computations. *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, March 2010.

[148] M A Watkins and D H Albonesi. ReMAP: A Reconfigurable Architecture for Chip Multiprocessors. *Micro, IEEE*, 31(1):65–77, 2011.

[149] R Weerasekera, M Grange, D Pamunuwa, and H Design Automation Test in Europe Conference Exhibition DATE 2010 Tenhunen. On signalling over Through-Silicon Via (TSV) interconnects in 3-D Integrated Circuits. In *IEEE DATE*, pages 1325–1328, 2010.

[150] Ted Eugene Williams. *Self-timed Rings and their Application to Division*. PhD thesis, Stanford University, Stanford University, May 1991.

[151] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[152] Clifford Wolf and Mathias Lasser. Project IceStorm. Technical report.

[153] C Wong and Alain J Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Design Automation Conference, 2003. Proceedings*, pages 508–513, 2003.

[154] C Wong, Alain J Martin, and P Thomas. An architecture for asynchronous FPGAs. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 170–177, 2003.

[155] Catherine Grace Wong. *High-level synthesis and rapid prototyping of asynchronous VLSI systems*. PhD thesis, California Institute of Technology, California Institute of Technology, May 2004.

[156] Xilinx. Partial Reconfiguration User Guide. Technical report, April 2013.

[157] Saeyang Yang. Logic Synthesis and Optimization Benchmarks, User's Guide v.3. Technical report, Microelectronics Center of North Carolina, 1991.

[158] Huabing Zhou, Minghao Ni, S Chen, and Zhongli Liu. The Design and Verification of FPGA CAD Toolset. *Integrated Circuits, 2007. ISIC '07. International Symposium on*, pages 461–464, 2007.