Copyright

by

Johnathan Andrew York

2011

The Dissertation Committee for Johnathan Andrew York
certifies that this is the approved version of the following dissertation:

Multiple Personality Integrated Circuits and the Cost of Programmability

Committee:

Derek Chiou, Supervisor

Brian Evans

Craig Chase

Thomas Gaussiran

David Pan

Keshav Pingali

Multiple Personality Integrated Circuits and the Cost of Programmability

by

Johnathan Andrew York, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2011

Acknowledgments

This dissertation required the support of many colleagues, family, friends, and well-wishers. I am grateful both for those listed below and those who go unsung.

Firstly, my wife Rosa provided encouragement, proofreading, and undeserved patience over many years. My adviser Derek served as an appreciated voice of experience and wisdom, ensuring that abstract notions never strayed far from the realm of the tangible (written word). The late Margarida Jacome served ably in the early years of this effort as both adviser and creative crucible. Her knowledge and willingness to argue energetically about the way the design process "should be" shaped my thinking about this work and will not be forgotten.

My parents, John and Betty, were my first "reference librarians" and provided a fertile environment in which a young engineer could prosper. My brothers, Richard and Robert, have served faithfully as co-conspirators and able instructors in humility. My daughter Abigail provided immeasurable motivation upon her arrival.

Finally, my many colleagues and friends at ARL:UT have provided encouragement, opportunity, and flexibility over the last decade. I surely would not have attended, nor persevered, in graduate school without their gracious support.

Johnathan Andrew York

The University of Texas at Austin

May 2011

Multiple Personality Integrated Circuits and the Cost of Programmability

Publication No. ____________________

Johnathan Andrew York, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Derek Chiou

This dissertation explores the cost of programmability in computing devices as measured relative to fixed-function devices implementing the same functionality using the same physical fabrication technology. The central claim elevates programmability to an explicit design parameter that (1) can be rigorously defined, (2) has measurable costs amenable to high-level modeling, (3) yields a design-space with distinct regions and properties, and (4) can be usefully manipulated using computer-aided design tools. The first portion of the the work is devoted to laying a rigorous logical foundation to support both this and future work on the subject. The second portion supports the thesis within this established logical foundation, using a specific engineering problem as a narrative vehicle. The engineering prob-

lem explored is that of mechanically adding a useful degree of programmability into preexisting fixed-function logic while minimizing the added overhead. Varying criteria for usefulness are proposed and the relative costs estimated both analytically and through case-study using standard-cell logic synthesis. In the case study, a methodology for the automatic generation of reconfigurable logic highly optimized for a specific set of computing applications is demonstrated. The approach stands in contrast to traditional reconfigurable computing techniques which focus on providing general purpose functionality at the expense of substantial overheads relative to fixed-purpose implementations.

Contents

Acknowledgments **iv**

Abstract **v**

Chapter 1 Introduction **1**

1.1 Background 2

1.2 State of the art 4

1.3 Thesis Statement 6

1.4 Approach 7

Chapter 2 Problem Definition **10**

2.1 Computation Model 11

2.1.1 Choosing an Abstraction Level 11

2.1.2 Formalism 12

2.1.3 Functionality 14

2.1.4 Multiplexers and the Supremum 16

2.2 Evaluation Metrics 18

2.2.1 Delay 19

2.2.2 Energy 20

2.2.3 Area 21
2.2.4 Leakage 23
2.2.5 Multiplexers 23
2.3 Placement and Wires 24
2.3.1 Placement 24
2.3.2 Wires 26
2.3.3 Rent's Rule and Wirelength Estimation 27
2.3.4 Realization Space Properties 30

Chapter 3 Theoretical Development **31**

3.1 Defining Programmability 32
3.1.1 Side-by-Side Realization 34
3.1.2 Resource Sharing 35
3.2 Generalized Programmable Interconnect 44
3.2.1 Definition 45
3.2.2 Multiplexer Elimination 45
3.2.3 Completeness 48
3.3 Programmability: *a priori* and *a posteriori* 50
3.3.1 Formal Definition 51
3.3.2 Naive Solution 52
3.3.3 Desirable Properties 52
3.4 Programmability Overheads 55
3.4.1 Multiplexer Insertion 56
3.4.2 Topological Costs 58
3.4.3 Shared Component Inefficiencies 75
3.5 Design Space Regions 76

3.5.1 Guideline Criteria 79
3.5.2 Region Discussions 85
3.6 Summary 88

Chapter 4 Experimental Validation **90**
4.1 Merging Design Tool 91
4.1.1 Overall Flow 91
4.1.2 Optimization Strategies 94
4.1.3 Noteworthy Complexities 95
4.2 FLWR Case-Study 98
4.2.1 FLWR Overview 98
4.2.2 Application Details 101
4.3 ASIC Implementation 102
4.3.1 Input Circuit Baselines 102
4.3.2 Output Circuit Unconstrained Synthesis 103
4.3.3 Pareto Synthesis Flow 106
4.4 FPGA Implemenation 108
4.5 Revisiting Asymptotic Region Boundary Predictions 111
4.6 *a posteriori* Progammability 113

Chapter 5 Prior and Related Work **121**
5.1 FPGA Modification Approaches 122
5.2 High-level Synthesis 124
5.3 Datapath Merging Problem 126
5.4 Multi-mode Synthesis 129
5.5 Reconfigurable Computing 132

5.6 The Totem Project 134
5.7 Virtual Reconfigurable Architectures 136

Chapter 6 Conclusion **138**
6.1 Contributions and Potential Impacts 139
6.2 Future Work 141
6.3 Recapitulation 143

Appendix A Case Study Source Code **147**
A.1 CIC 148
A.2 CORDIC 152
A.3 CORDIC 154
A.4 FIR 155
A.5 heterodyning 156
A.6 mcu 157

Appendix B Case Study Dataflow Graphs **163**

Bibliography **169**

Vita **182**

Chapter 1

Introduction

1.1 Background

Advances made in semiconductor fabrication technology have dramatically reduced the incremental cost of producing digital computing devices, resulting in non-recurring engineering costs forming a substantial component of device cost for many Application Specific Integrated Circuits (ASICs). Programmable computing devices, in which the functionality of the device is largely established after fabrication, serve to amortize the non-recurring engineering costs over many potentially unrelated applications. Traditional programmable computing devices, including Field Programmable Gate Arrays (FPGAs) and fetch/execute-style processors, are targeted at broad application domains and typically incur tremendous area, energy, and delay overhead relative to ASIC designs implementing the same functionality.

Contemporary literature roughly estimates this overhead at 8-88X in area, 2-14X in delay, and 12-500X in power for FPGAs relative to a standard-cell ASIC design[56] (shown in Table 1.1). For a variety of reasons, including differing programming models, comparisons between processor-style devices and ASIC implementations are less readily available. However, within the application domain of high-performance digital signal processing, literature is available demonstrating order of magnitude improvements in energy and delay when moving designs from microprocessor-style Digital Signal Processors (DSP) to FPGA implementations [15] [57] [65]. That is, evidence exists that applications implemented in processor-style DSP architectures incur an even greater overhead than those implemented in FPGAs. More recent work suggests that while optimizations on processor-style architectures (e.g. [23],[89]) can improve performance and efficiency dramatically, “the very low energy costs of actual core [operations] mean that over 90% of the energy used in these solutions is still ‘overhead’. Achieving ASIC-like performance

	Delay	Power	Area
FPGA	6-112	36-5000	116-1276
Standard Cell	3-8	3-10	14.5
Full Custom	1	1	1

Table 1.1: Relative performance measures for FPGAs, Standard Cell, and Full Custom Integrated Circuits derived from [56]

and efficiency requires algorithm-specific optimizations. [44]"

For a variety of device types, it appears that programmability imparts a substantial overhead to application implementations. For a research engineer, such substantial penalties lead naturally to the questions:

- What is the nature of this overhead?
- Is it avoidable, or is it fundamentally inherent in programmable devices?
- Can it be minimized?
- If it can be minimized, how close are modern programmable devices to a theoretical minimum?
- If it can be minimized, can one describe the degree of "programmability" of a device in a manner suitable for understanding the trade-offs between programmability and implementation efficiency?
- What practical applications immediately benefit from answers to these questions?

1.2 State of the art

The notion of programmability has theoretical roots in the study of computational reducibility. In this field, relationships between computations are studied, and one computation is said to be reducible to another if the former can be computed by computing the latter, followed by a *de minimis* computation specified by the particular type of reducibility. Notably, reducibility applied to “effectively calculable” functions led to the the concept of a Universal Turing Machine [86], which establishes the existence of an upper bound for computability. This powerful proof argues that even simple machines with suitable external storage can compute anything that can be computed.

The Universal Turing Machine (UTM) proof implies that the architectural requirements for a device to be fully programmable are quite weak (perhaps even a 2-state, 6-rule finite state machine is sufficient when connected to an external memory [77]). The universal properties of the physically unrealizable UTM are present in a more tangible analog: the stored-program, von Neumann-style [88, 5] (i.e. fetch-execute) computer. Despite the theoretical attractiveness of universal computability, a Turing-completeness definition of programmability ignores practical costs such as memory required, computational speed, and energy usage.

The physical costs of a realized computation are perhaps most comprehensively considered in the rich field of high-level synthesis, which has been characterized as the study of “automated generation of the hardware circuit of a digital system from a behavioral description.” [42]. This ambitious and difficult problem has been studied as a distinct field for over three decades, with “mixed success”[41], and has been characterized by a series of generations of researchers, each claiming “to have got it right” [41]. While a more detailed discussion is left for section 5.2, high-level

synthesis is often conceptually considered to have two related optimization subproblems: binding and scheduling. Loosely speaking, these are roughly the decision of where and when, respectively, each of the components of the overall computation is physically performed. As might be inferred from the earlier definition, much architectural freedom is typically assumed in high-level synthesis, which distinguishes the approach from the structure imposed in other fields. Also implicit in the high-level synthesis problem definition is the assumption of a behavioral description of a specific computational problem present as input. That is, while high-level synthesis may involve the use of reconfigurable hardware, the stated goal is to use said hardware to implement specific computations and not to generate general-purpose hardware.

In contrast, the field of reconfigurable computing is concerned with the design of hardware architectures that can be reconfigured after fabrication to implement various computations. Reconfigurable computing, discussed more completely in section 5.5, had its humble birth in the implementation of "low cost/marginal performance class" [35] of circuitry, as more mundane concerns of testability and designer effort began to outweigh the incremental fabrication costs of large scale integrated circuit fabrication. In recent times, reconfigurable computing has exploded to include large field programmable gate arrays (FPGAs), and a plethora of academic architectures [45]. The distinguishing characteristic of reconfigurable computing approaches is the desire to expose hardware structure into the programming model. For example, the "sea of gates" model of FPGAs is exposed directly to the compiler, which must locate these gates and find physical resources to connect them together. This stands in contrast to the von Neumann-style approach, which prefers to expend extensive hardware cost to provide the illusion to the program of

general-purpose functional units, large uniform memories (and registers), and serial execution.

Given their prevalence and importance to computing, von Neumann-style fetch/execute processors have not been immune from studies of their physical costs. Among the numerous and incredible developments in computer architecture over the last century, one innovation is particularly relevant to this effort: data-path accelerators. These accelerator modules are targeted at specific computational fragments, and are coupled to the fetch/execute computing core in order to dramatically reduce the cost of the targeted computations. By doing so, the accelerators are able to exploit the general purpose nature of the fetch/execute core to handle things such as control flow, in order to simplify the accelerator module. This approach is discussed more extensively in section 5.3.

1.3 Thesis Statement

Extensive prior work has focused on synthesizing fixed-function logic from behavioral descriptions (e.g. high-level synthesis), improving performance of fetch/execute style processors (e.g. datapath accelerators), and exposing hardware limitations to programmers (e.g. reconfigurable computing). The level of programmability of these platforms is an implicit part of their design, and are effectively fixed for all users of a given platform. In contrast, *this effort hypothesizes that programmability can be elevated to an explicit design parameter that (1) can be rigorously defined, (2) has measurable costs amenable to high-level modeling, (3) yields a design-space with distinct regions and properties, and (4) can be usefully manipulated using computer-aided design tools.*

1.4 Approach

To test this abstract hypothesis, a specific and tangible engineering problem has been selected to serve as a skeleton around which supporting arguments will be made. Specifically, consider that one

- has a fixed-function circuit implementation of a computation that exceeds a specific set of performance requirements by some margin,
- wishes to leverage said margin to improve the functionality and flexibility of the implementation, and
- wishes to do so in a mechanized way with minimal designer intervention.

The work that follows demonstrates an automated methodology for improving the flexibility of a fixed-function circuit, while minimally degrading performance. Specifically, it will be shown that one can mechanically introduce a useful level of programmability into fixed-function logic circuit without incurring substantial overhead for the added functionality. Furthermore, by doing so one can directly manipulate programmability to explore a design space between general-purpose and fixed-function logic, thereby gaining insight into costs of programmability.

To support this thesis, the remainder of this document is structured at a high-level as follows:

- a rigorous definition of the problem to be solved and brief discussion of relevant prior work (Chapter 2),
- proposal of varying levels of programmability and development of an analytical model that predicts the overhead for each (Chapter 3), and

- development of automated software and conduct a case-study in the application domain of high-performance signal processing (Chapter 4).
- detailed discussion of related and prior work (Chapter 5)

In more detail, chapter 2 will define a model of computation based upon a hardware logic circuit that will be used throughout the effort. Several physical costs of computation will be used to evaluate implementations will be evaluated, including:

- area - the physical size of the computing implementation
- delay - the time taken for the computation to complete
- energy - the energy expended per unit of computation
- leakage power - the energy expended per unit time that the device is powered, regardless of computations performed.

Chapter 3 will propose several levels of programmability in two major categories: *a priori* and *a posteriori*. In doing so, notions of programmability that are perhaps familiar, but previously ill-defined will be formalized. A resource sharing methodology will be examined which, in conjunction with common-sub-expression elimination, can yield a model for a generalized programmable interconnect. This generalized programmable interconnect forms a model of programmability based on the composition of fixed-function components. The overheads that are introduced when constructing programmable circuits will be examined, followed an argument for distinct design-space regions that arise from the relative dominance various components of these overheads. Techniques for optimal construction of programmable

architectures and their relative suitability for the various design-space regions will complete the chapter.

Chapter 4 will discuss a software tool developed for this effort that generates programmable circuits meeting the definitions established in chapter 2 and the techniques of chapter 3. A case study will be presented using computations extracted from a software defined radio application. The resulting costs of programmable circuits synthesized in a standard-cell ASIC flow will be used to measure overheads relative to fixed-functionality circuits synthesized in the same flow. Comparison to the FPGA implementation originally used in the application, including the results of generating a virtual reconfigurable architecture within the FPGA fabric, will be presented.

Chapter 5 will discuss relevant prior and related work, while chapter 6 will summarize the work, enumerate the contributions, and suggest avenues for future study.

Chapter 2

Problem Definition

In this chapter, the problem to be solved is defined, and a formalism that will be utilized in later chapters is described. It should be understood that none of the individual pieces of the framework presented in this chapter are particularly novel. To the contrary, it is hoped that they represent an aggregation of assumptions that are individually well-tested and uncontroversial. Instead, the goal for this chapter is to produce a model framework that requires a minimal number of assumptions of the underlying implementation technology. By doing so, an implicit argument is made that the techniques presented in later chapters are not reliant on the details of a particular implementation technology, but are instead inherent to a wide range of physically-realized computations.

2.1 Computation Model

As elaborated in section 1.3, the goal for this effort is to improve the flexibility of a fixed-function circuit in an automated way. To operate on circuits in an automated way, it is necessary to formalize the description of a circuit, and define the ways in which it can manipulated. Such formalization defines a model of computation in order to allow manipulation of a computation while respecting the semantics required to preserve behavior.

2.1.1 Choosing an Abstraction Level

Modeling computation is a long studied problem, used in disciplines ranging from discussion of computability to more pragmatic issues. In the pursuit of ever higher-level synthesis capability, numerous models of computation have been proposed, including: finite state automatons, register transfer language, data flow, discrete-event, and continuous time models. Notably, the purely coordination aspects com-

puting have been studied extensively under the umbrella of "coordination languages" [36]. A key argument underlying the high-level synthesis approach is that higher levels of abstraction open up larger opportunities for optimization. The reasoning behind this argument is relatively straightforward: higher levels of abstraction by definition have less lower-level information and therefore fewer lower-level restrictions on optimizations than can be performed. As a result, it is typically argued that optimization in high-level synthesis should occur at the highest levels of abstraction.

However the stated goal of this effort, to trade performance margin in a fixed function circuit to gain added flexibility, benefits from having low-level knowledge of the fixed-function circuit beyond a behavioral description. In particular, to avoid introducing undue overhead on the fixed function implementation, it is desirable to retain the low-level structure of said implementation and perform manipulations at a lower level. As a result, the model of computation chosen is that of a circuit graph.

2.1.2 Formalism

Definition A *circuit* is a graph G defined by the tuple consisting of a set of components (i.e. graph vertices) and a set of connections between them (i.e. "nets" or graph edges).

$$G = (V, E) \tag{2.1}$$

Definition Each component $v \in V$ has an associated set of *ports*, $ports(v)$. For later use, define a function *component* to be the component associated with a port. That is,

$$component(p) = v \Leftrightarrow p \in ports(v). \tag{2.2}$$

In practice each port is typically identified by a name and optional bit selection. Moreover, each port is defined to be either an input or output port.

$$\forall p \in ports(V) : isInput(p) \vee isOutput(p) \tag{2.3}$$

$$isInput(p) \Leftrightarrow \neg isOutput(p) \tag{2.4}$$

For simplicity, each component is defined to have one output port. That is,

$$\forall v \in V : |p \in ports(v) : isOutput(p)| = 1. \tag{2.5}$$

One can avoid loss of generality by modeling a multiple output component as a composition of components, specifically one component per output port that all receive the same set of inputs.

Definition An *edge* connects between the output port of a component to the input port of a component. The output port is known as the source, while the input port is known as the sink of the edge. That is, each edge is defined as a 2-tuple of the source and sink port.

$$\forall (source, sink) \in V : isOutput(source) \wedge isInput(sink) \tag{2.6}$$

A single output port may be associated with multiple edges, each connecting to differing input port. Each input port is the sink of at most one edge.

$$(source, sink) \in E \rightarrow ((source^{*}, sink) \Rightarrow source^{*} = source) \tag{2.7}$$

An example circuit is shown graphically in Figure 2.1.

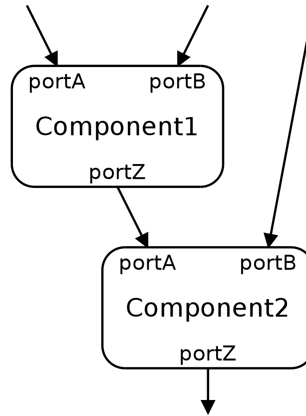

Figure 2.1: A simple circuit

2.1.3 Functionality

For this effort further knowledge about each component is required, notably a relation between components indicating that when connected appropriately, one component behaves equivalently to another. For example, one instance of an 8-bit adder component can be substituted for another instance of an 8-bit adder component without change in functional behavior. However, in place of an equality relation, a more useful comparison relation is defined instead.

Definition The *functionality* of a component (or circuit) a is said to be less than or equal to the functionality of a component (or circuit) b if and only if b can be substituted for a without change in functional behavior with appropriate port connections. Symbolically, this is written

$$a \leq b. \tag{2.8}$$

For example, an 8-bit unsigned adder might be said to be less than or equal to the functionality of a 16-bit unsigned adder, because by appropriately connecting the input ports (in some cases to a constant zero) and the output port (perhaps leaving the most significant bits unconnected), the 16-bit adder can be substituted for the 8-bit adder without altering the correct functional behavior of the circuit.

Definition Given a component (or circuit) a, the *functionality* of the component, written symbolically as $functionality(a)$, is defined to be the set of all components (or circuits) that implement strictly equivalent functionality to a. That is, the following are defined to be true:

$$a \in functionality(a) \tag{2.9}$$

$$a \in functionality(b) \Leftrightarrow (a \leq b) \wedge (b \leq a) \tag{2.10}$$

As a result, we can say that

$$functionality(a) = functionality(b) \Leftrightarrow (a \leq b) \wedge (b \leq a). \tag{2.11}$$

For convenience, we can also define a *functionality relation* on the set of functionalities such that

$$functionality(a) \leq functionality(b) \Leftrightarrow (a \leq b). \tag{2.12}$$

Theorem 2.1.1 *The functionality relation forms a partial order over the set of all functionalities F.*

Proof Under the definition above, the functionality relation is reflexive, as a com-

ponent can certainly be substituted for itself without change in functional behavior. It is also transitive, as if a can substitute for b and b can substitute for c without change in functional behavior, then certainly a can substitute for c without change in functional behavior. It is also anti-symmetric, in that if two units can be substituted for each other without any loss of functionality, their functionality must be the same. Thus the relation($\leq$) is reflexive, transitive, and anti-symmetric. Therefore $(F, \leq)$ forms a partial order. ∎

2.1.4 Multiplexers and the Supremum

Definition Because functionality relation $(F, \leq)$ is a partial order, a unique *supremum* (also known as "join" or "least upper bound") for a set of functionalities is defined. Following the traditional definition, the *supremum* of a set Z is an element s such that

- $\forall z \in Z : z \leq s$, and
- $\forall t \in T : (\forall z \in Z : z \leq t) \Rightarrow s \leq t$.

The former condition requires that s must be substitutable without loss of functionality for any element in Z. The later requires that if there exists any other element t that can substitute for all elements in Z, then it must also substitute for s. If one assumes that strictly increased functionality always incurs equal or greater costs, one can say that the supremum of a set of functionalities (if it exists) is the least expensive functionality that can substitute for any functionality in the set. While the uniqueness of the supremum is guaranteed by definition, the existence of the supremum is not. However, as will be shown below, there are conditions under which the supremum is guaranteed to exist.

Definition A *multiplexer* is a component that has a set of input ports referred to as data ports, an input port referred to as the selection port, and a single output port. The multiplexer has the property that for each data input port, there exists a connection to the control input port such that any device connected to the output port behaves functionally as if it were connected directly to the source of said data input port.

Theorem 2.1.2 *The existence of multiplexer functionality guarantees the existence of a supremum, and therefore makes $(F, \leq)$ a join semi-lattice.*

Proof For any set of components, instantiate a multiplexer and one instance of each component in the set. Now connect the outputs of each of the functional units to the data ports of the multiplexers. By definition of the multiplexer, there exists an appropriate connection to the control port of the multiplier such that components connected to the output port of the multiplexer can be made to behave as if they are directly connected to any one of the components in the original set. As a result, the resulting composition can behave identically to any of the original components, and thus the functionality of this composition is equal to or greater than the functionality of each of the original components. Thus a supremum is shown to exist for any set of input components, and $(F, \leq)$ is a join semi-lattice. ∎

Restated, provided a multiplexer element exists, it has been shown that for any set of functionalities it is possible to construct a circuit of equal to or greater functionality (the supremum) to any functionality in the set. This constructive proof is shown graphically in Figure 2.2. A curious reader may wonder about the existence and usefulness of the infimum. The infimum (or greatest upper board) of a set of functionalities would contain the most functional component possible that cannot

implement the functionality of any of the functions in the set. While the engineering usefulness of the infimum is not clear, it is perhaps useful in defining the topological characteristics of the functionality space and therefore the applicability of various optimization techniques based on traversing in that space. A cursory examination suggests that the existence of an infimum would need to be predicated on far more complex conditions than the existence of a single component, and this exercise is left for future work.

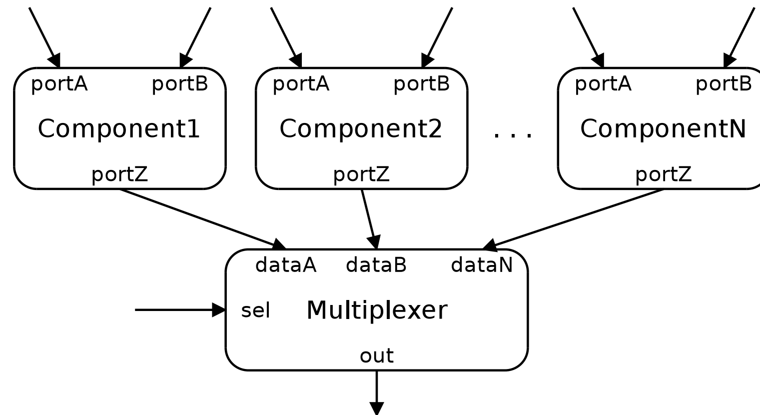

Figure 2.2: Construction style used to prove the existence of a supremum for any set of component (functionalities).

2.2 Evaluation Metrics

In the prior section, a formal definition for a circuit is given. In this section, several metrics on the set of possible circuits are evaluated, by which a circuit may be evaluated relative to others. Moreover, a set of properties for these metrics is derived, which will be used by later work. Particular attention is given to the derivation of rules for how the overall metrics for a composition relate to the metrics of each component. While the specific metrics given here are particularly relevant to metal

oxide semiconductor fabrication, the manner in which these metrics scale when components are aggregated into hierarchical structures are perhaps more universal.

Definition Formally a mathematical metric d is a function $d : X \times X \rightarrow \Re$, that satisfies the conditions

1. $d(p, q) > 0$ if $p \neq q$
2. $d(p, p) = 0$
3. $d(p, q) = d(q, p)$
4. $d(p, q) \leq d(p, r) + d(r, q)$, for any $r \in X$ [70].

In the context of measuring physical costs, it is desirable to relax the first constraint (the identity of indiscernibles), such that differing circuits may have identical physical costs. Such relaxed metrics are referred to as a pseudometric, and the condition of $d(p, q) = 0$ defines a metric identification equivalence relation of equally-costly circuits.

Definition Define the null circuit $\emptyset$ to be a circuit with no components and no edges. The metrics considered here are defined such that $d(\emptyset) \equiv 0$. This allows use of short-hand $d(x) \equiv d(x, \emptyset)$.

2.2.1 Delay

The first pseudometric considered is delay, defined to be the physical time taken to complete a unit of computation. Each component of a circuit is assumed to be modeled by an input-independent delay from the last input port being valid to the result being produced at each output port.

Definition The *delay metric* is defined as a function that maps from the domain of circuit onto the set of non-negative reals:

$$delay : V \to \Re^+. \tag{2.13}$$

Physical devices often have varying delays between each input/output port pair, which can be modeled without loss of generality via aggregations of components.

In collections of components, the overall delay depends on the topography of the connections between components. In a serial composition, the overall delay is the sum of the individual delays. In parallel composition, the overall delay is the greater of the composed components. Notably, is it possible for components to not contribute the overall delay, if they are effectively in parallel with a computation that takes a longer time. In contrast, components that effectively contribute to the overall delay (i.e. an incremental change in the delay of said component directly changes the overall delay) are referred to as residing on the critical-path of the circuit.

2.2.2 Energy

The second pseudometric considered is energy, defined to be the energy required to complete a unit of computation. It is assumed that each component of a circuit can be modeled as having an internal energy dissipation per operation. For the purposes of this architectural exploration technique, which is focused on how energy aggregates in collections of components, data-dependent effects on energy are ignored although the reader is advised that they may be substantial.

Definition The *energy metric* is defined as a function that maps from the domain

of circuits onto the set of non-negative reals:

$$energy : V \to \Re^{+}. \tag{2.14}$$

In collections of components, the overall energy is defined to depend solely on the sum of the individual components. Formally, the energy of a circuit is defined as

$$energy((V, E)) \equiv \sum_{v \in V} energy(V). \tag{2.15}$$

2.2.3 Area

The third pseudometric considered is area, defined to be the physical space required to complete a unit of computation. The term area, rather than volume, is adopted to improve readability in the context of dominant two dimensional fabrication technologies. It is assumed that each component of a circuit has an area requirement. However, unlike the other psuedometrics considered, area is not defined to map onto the set of positive reals. There are often multiple components of area that must be considered in an implementation, and an excess requirement in any one component effectively requires padding of the other components. An example of this can be found in the multiple layer construction in current semiconductor processes. If a circuit has dense wiring needs on a metal layer, it is commonly necessary to add additional padding space between circuits on the active layers to allow for this. Thus the set of non-negative reals does not adequately capture the aggregation effects of area. To capture this appropriately, the area metric maps to an arbitary metric space M as defined below.

Definition The *area metric* is defined as a function that maps from the domain of

circuits onto an arbitrary metric space $M = (m, d)$, where m is a set of points, and d is a metric on m :

$$area : V \to M. \tag{2.16}$$

Moreover the following conditions must be met:

- There must exist an element 0 in m, that represents the area of the null circuit. Symbolically

$$area(\emptyset) \equiv 0. \tag{2.17}$$

- Elements closer to 0 element are considered to be more desirable. Symbolically a is more desirable than b iff

$$d(0, a) < d(0, b). \tag{2.18}$$

- Each component is assumed to have an area represented by an element in m.
- There must exist an addition function that is valid and closed over all points in m.
- As a metric space, d must respect the triangle inequality.

Formally, then the total area of a circuit is defined as

$$area((V, E)) \equiv \sum_{v \in V} area(V). \tag{2.19}$$

For optimization, one can then map this total area onto the set of reals via the metric distance function d.

2.2.4 Leakage

The final pseudometric considered is leakage, defined to be the time rate of energy dissipation required while the circuit is powered.

Definition The *leakage metric* is defined as a function that maps from the domain of circuits onto the set of non-negative reals:

$$leakage : V \rightarrow \Re^{+}. \tag{2.20}$$

In collections of components, the overall leakage is defined to depend solely on the sum of the individual components. Formally, the total area of a circuit is defined as

$$leakage((V, E)) \equiv \sum_{v \in V} leakage(V) \tag{2.21}$$

2.2.5 Multiplexers

Based upon the metric definitions above, it is possible to make statements on the cost of the multiplexer element defined in section 2.1.4.

Definition A 2-input multiplexer, henceforth referred to as *MUX2*, is defined as to be a multiplexer as established in section 2.1.4 with 2 input ports.

Theorem 2.2.1 *Bounds on the costs of a multiplexer with an arbitrary number of inputs N can be defined in relation to the costs of a* MUX2*.*

Proof As shown in Figure 2.3, a N-input multiplexer can be realized as a hierarchical tree of 2-input multiplexer, and this provides us with a bound for the costs of the former in terms of the latter. The total delay through the multiplexer tree

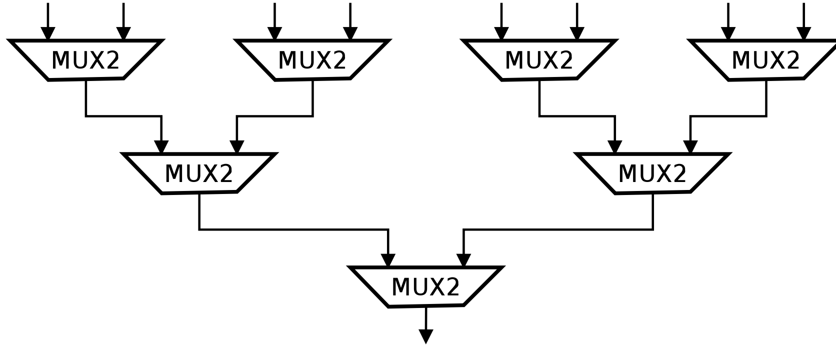

Figure 2.3: Realization of 8-input multiplexer using 2-input multiplexer elements

	2-input	N-input
delay	$delay(MUX2)$	$delay(MUX2)ceil(log_2(N))$
energy	$energy(MUX2)$	$energy(MUX2)(N-1)$
area	$area(MUX2)$	$area(MUX2)(N-1)$
leakage	$leakage(MUX2)$	$leakage(MUX2)(N-1)$

Table 2.1: Relative costs for a a 2-input multiplexer and resulting bounds on N-input multiplexer costs

scales as $log_2(N)$, while the area, energy, and leakage costs scale with number of multiplexers required as $N-1$. Table 2.1 captures these relative cost bounds.

2.3 Placement and Wires

Definition When computations are realized in a space each component must occupy a finite region in that space, and each component connected by an edge must be connected with components that may induce additional costs in terms of delay, area, energy and leakage. Such components are henceforth referred to as *wires*.

2.3.1 Placement

Definition A *realization space* P is a set of elements in which a computation is to be realized, upon which a metric distance function $dist : PxL \rightarrow \Re$ exists, thereby

creating a metric space $(P, dist)$.

Definition The *placement* for a circuit is a function which maps from the set of components in a circuit to a subset of the realization space. Symbolically

$$placement : V \to \{P' | P' \subset P\}. \tag{2.22}$$

Definition For a given circuit component, the subspace that it maps to is referred to as a *placement region.* It is required that the placement region for each component be disjoint, or symbolically

$$\forall a, b \in V : placement(a) \cap placement(b) \neq \emptyset \iff a = b. \tag{2.23}$$

From this it can then be shown that the placement function is injective. The placement region of each component is further restricted to be bounded. That is, for each component $v \in V$ there exists some size $s \in \Re$, such that

$$\exists b \in \Re : \forall placement(v) < b. \tag{2.24}$$

Definition Each port of a component also has a placement, defined by the *port placement* function

$$portplacement : \bigcup \{ports(v) | v \in V\} \to P. \tag{2.25}$$

Each port is defined to lie within the placement of its associated component. That is,

$$\forall v \in V : p \in ports(v) \Longrightarrow portplacement(p) \in placement(v). \tag{2.26}$$

2.3.2 Wires

Definition A circuit component w is said to be a *wire* if and only if it possesses all of the following properties:

1. *Two ports* - $ports(w) = \{i, o\}$ such that $isInput(i)$ and $isOutput(o)$ are true.
2. *Transparency* - The correct operation of the circuit is maintained if the ports sunk by o were directly connected to the port sourcing i.
3. *Length-dependent costs* - The cost of the is dependent solely on the type of wire and a single parameter $\ell(w) \in \Re$. For any given type of wire, the wire can be shortened or extended without limit and the cost of a given wire is dependent solely upon and monotonically[1] increases with the parameter $\ell(w)$.

Definition A circuit is said to be *realized* if and only if all of the following conditions are met:

1. All non-wire components are connected solely via wire components

$$(source, sink) \in E \Longrightarrow \\ isWire(component(source)) \vee isWire(component(sink)). \quad (2.27)$$

2. All connected sources and sinks are placed together

$$\forall(source, sink) \in E : portplacement(source) = portplacement(sink). \quad (2.28)$$

[1]In many cases, the cost of a wire increases linearly. However, in contemporary fabrication technologies, wires are often separated by distances well below a wavelength of the signal transmitted, and therefore wires act as physically distributed capacitors in which propagation delay increases quadratically with length.

3. There exists a placement into a realizable space $(P, dist)$ such that the length of a wire is defined by the placement

$$\forall v \in V : isWire(v) \implies$$
$$\ell(v) = dist(portplacement(input(v)), portplacement(output(v))). \quad (2.29)$$

2.3.3 Rent's Rule and Wirelength Estimation

In the prior section, the notion of a wire and physically realizable circuit were formally introduced. Notably, in recent fabrication technologies, the cost of wires has become an increasingly large portion of the overall cost of circuits [47]. As a result, this cost has come under increasing study, including statistical estimation. One particular insight developed in the course of this study, Rent's rule, yields particularly useful insights.

Hagen, et al [43] note that Rent's rule is an empirical relation first observed by E.F. Rent in the late 1960's, that

> *reflects a power-law scaling of the number of external terminals of a given subcircuit with the number of modules in the subcircuit. Specifically,*
>
> $$T = k \cdot C^p \quad (2.30)$$
>
> *where T is the average number of external terminals (pins) in a subcircuit or partition; k is* Rent's constant*, a scaling constant which empirically corresponds to the average number of terminals per module; C is the number of modules in the subcircuit (or partition); and p is the* Rent parameter *or* Rent exponent*, with $0 \leq p \leq 1$.*

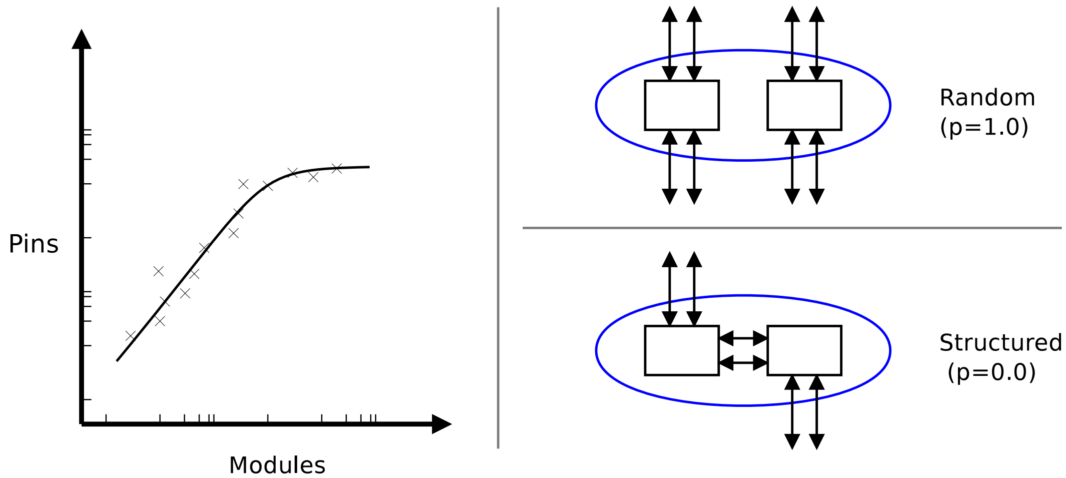

Figure 2.4: Graphical Representation of Rent's Rule. The left shows pin count plotted logarithmically against number of modules, where the slope respents the Rent exponent p. The right shows an example of circuits with random ($p = 1$) and highly structured logic ($p = 0$).

This is captured graphically in Figure 2.4. Notably, when the number of pins (T) is plotted against the number on a logarithmic scale, as in the left side of Figure 2.4, the slope of a fitted line is the Rent exponent p for the circuit. On the right side, one can see that each of the square modules have 4 pins, thereby resulting in a corresponding point of $(T, C) = (4, 1)$. In the top half, indicative of random, unrelated logic, the aggregate of two modules has 8 external pins, yielding a corresponding point of $(T, C) = (8, 2)$. In contrast, when related logic is grouped together, as is typical in highly structured circuits, the aggregate of two modules again has 4 external pins, yielding a point of $(T, C) = (4, 2)$. By examining these two pairs of points in the context of equation 2.30, one can find that Figure 2.4 illustrates the two extremes of *Rent exponent*, $p = 0$ and $p = 1$.

In practice, the *Rent exponent* p is likely to be a value in between these extremes. A regular two dimensional array such as a RAM or tiled array logic

is likely to have an exponent of 0.5, as the number of external pins (along the perimeter) increases as the square root of the number of cells (or area). This holds more generally, with the Rent exponent related to a fractional dimensionality for circuits.

Landman and Russo [58] observed that the Rent parameter depends both upon the circuit structure and the partitioning algorithm used. Hagen, et al [43] extended this concept to yield the notion of an "intrinsic Rent parameter" defined to be the lowest Rent parameter possible for any given circuit. Thus, the Rent parameter can be used as an evaluation metric both for partitioning, and as a metric for circuit placement difficulty.

Donath [29] formulated an upper bound on expected average interconnection length on both linear and square placement topologies based upon partitioning techniques. He noted that for large circuits, the average wire length $\overline{R}$ is related to the *Rent exponent* p and the number of circuit components C, as follows

$$\overline{R} \sim \begin{cases} C^{p-0.5} & \text{for } p > 0.5 \\ \log C & \text{for } p = 0.5 \\ f(p) & \text{for } p < 0.5, \end{cases} \tag{2.31}$$

where $f(p)$ is an unspecified function of only p.

This upper bound establishes a statistical relationship between the number of gates in a circuit, the circuits Rent exponent, and a wirelength. This work has been expanded to include expectations for the statistical wirelength distribution [30]. Notably, Stroobandt [82] provided an explanation for the observation of Donath that average wirelength deviates from the upper bound by approximately a factor of 2.

If one considers that a 2D plane assumed by Donath has an effective Rent

exponent of 0.5 (i.e. $perimeter = area^{0.5}$), the three sections of the Donath equation above correspond to the cases of when the circuit to be implemented has greater, equal, and lesser asymptotic connectivity needs than the realization space. This relationship has been noted by other authors and applied in other contexts. For example, DeHon [27] has used this observation to optimize for area efficiencies in FPGA interconnect designs

2.3.4 Realization Space Properties

Two properties of a realization space as defined in section 2.3 are defined here for later use.

Definition A realization space is said to have the property of *translation-invariant costs* if the cost of a realized (i.e. placed) circuit is invariant to the position in the realization space. That is, provided the realization space is large enough, multiple copies of a realizable circuit can be placed in the space, and the costs of each copy are identical. Note that many implementation technologies possess this property, including 2D lithographically produced integrated circuits and tile-based FPGAs.

Definition A realization space is said to have the property of *2D Rent's rule wiring costs* if the average interconnect length R between aggregated components with an intrinsic Rent parameter of p follows the relationship of equation 2.31.

Chapter 3

Theoretical Development

This chapter begins by arguing for a specific definition of programmability. From this definition a model of programmability based on the assembly of fixed-function components with multiplexer "glue" will be proposed and rigorously defined. This model and the resulting design-space will be explored theoretically, with formal mechanisms for traversing between points within the design-space. From these mechanisms the extent of the design-space will be explored, and points and properties of interest will be identified. The chapter concludes by examining the interactions between the proposed model and the fabrication models developed in Chapter 2. From this interaction, distinct design-space regions are identified based on the relative magnitude of various cost components. Each design-space region will be discussed and optimization strategies suitable for each will be suggested.

3.1 Defining Programmability

In the author's experience, the term programmability is widely understood but used only informally. To avoid ambiguity and confusion in the remainder of this work, a more rigorous definition for this term will be developed and justified. As the question at hand is one of definition of a word with use beyond a specific technical field, a dictionary is a useful authoritative starting point. The Oxford English Dictionary describes the term "programmable" as "capable of being assigned a function by the user" [2]. Albeit pedantic, in this author's experience ambiguity arises when ascribing meaning to the word "a" of this definition. The Oxford English Dictionary provides several definitions for the indefinite article "a", two of which are particularly relevant:

- "referring to something not specifically identified [...] but treated as one of a class: one, some, *any*[1]", and

- "in a more definite sense: one, a certain, *a particular*[1]." [2]

Correspondingly, in the author's experience, two lines of reasoning prevail about the definition of programmability:

- a device is programmable if it is capable of being assigned *any* function by the user, or
- a device is programmable if it is capable of being assigned *a particular* function by the user.

The former definition is perhaps an implied reference to the universality of Universal Turing Machine discussed in Section 1.2. Under this definition a device would be programmable if and only if it is Turing-complete. However appealing, this definition is unsatisfying for this author, as it both contradicts common usage of the term, and is unsuitably rigorous for engineering usage due to the impracticality of constructing the infinite-length tape that is critical to Turing-equivalence. Certainly one would consider early 4-bit microprocessors with kilobits of RAM programmable, yet such machines would not qualify under a rigorous application of this definition. Carrying out this definition to its logical conclusion, one is faced with unsatisfying questions of the form: how much memory is required for a device to be programmable? Under this definition, it would seem that no physically realizable device can be said to be programmable.

In contrast, the latter definition permits realizable devices and is therefore arguably more useful in an engineering context. Rather than requiring an implied form of Turing-completeness, a device is said to be programmable if it can be assigned a particular function, and is distinguished from a fixed-function device by the

[1] Emphasis added

ability to be assigned more than one function. It is this definition that will serve as the basis for this document, and upon which stronger properties will be defined.

Definition A device is said to be *programmable* if and only if it is capable of being assigned more than one function.

This definition necessarily relies upon a contextual definition of the term "function". Thus the property of being programmable is not an intrinsic property of a device, but is instead a property of how a device is used.

3.1.1 Side-by-Side Realization

In section 3.1, it was argued that a device is programmable if it can be assigned more than one function. Based upon this definition, and the definitions of Chapter 2, one can show that given a set of circuits and the multiplexer element defined in Section 2.1.4, one can create a single programmable circuit capable of being assigned to any of the input circuits tasks via a construction of the form shown in Figure 3.1.

Theorem 3.1.1 *(Side-by-side Realization) Given any set of circuits, and the existence of a multiplexer element, one can construct a programmable circuit capable of being assigned the function of any of the input circuits.*

Proof Proof follows in a similar form to that of Theorem 2.1.2, as each individual circuit is merely an aggregation of components. ∎

Moreover, one can bound the overhead of this construction. Based on the rules of section 2.2, provided the multiplexer select signals change infrequently, one can say that the delay of the programmable circuit is equal to the cost of the original circuit plus the delay needed to reach the repositioned inputs, the multiplexer delay,

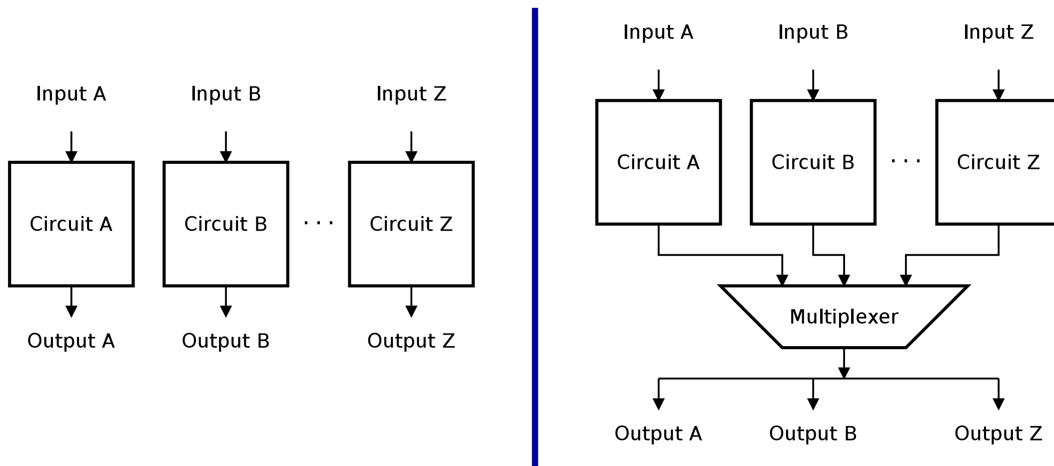

Figure 3.1: Conceptual diagram of naive side-by-side realization. The circuit on the right can be said to be programmable because it can be assigned to any of the functions of the circuits A..Z on the left

and the delay incurred in the wire needed to reach the multiplexer. The area incurred is the sum of the area of each of the input circuits, plus that of the multiplexer and the wires needed to reach the multiplexer. Naively, the energy is also equal to the sum of the energy of each of the input circuits plus that of the multiplexer and wires. However, with proper operand isolation techniques, the unused circuitry can remain idle. As a result, the overall energy is that of the selected circuit plus that of the multiplexer and wires. A similar argument applies to leakage and power gating.

3.1.2 Resource Sharing

The side-by-side realization of the prior section is inefficient when, as is common, there are portions of circuitry that can be shared between the various circuits. Because only a single circuit of Figure 3.1 is active at a time, one may reduce the area by sharing various components between circuits.

Definition Consider that each circuit is composed of individual components. By

merging similar components from differing input circuits into a single component capable of implementing the functionality of either (see section 2.1.3), one can share the component thereby potentially saving area and reducing wiring costs. This manipulation is known as *resource sharing*.

Resource sharing has the effect of pushing multiplexers from the extrema deeper into the circuits. For example, consider the two circuits shown in Figure 3.2. They can be implemented into a programmable circuit in a side-by-side realization as shown in Figure 3.3. However, one can also consider merging component *Mul_2* and *Mul_B* into a shared component. As shown in Figure 3.4, by introducing appropriate multiplexers into the circuit, and broadcasting the result of the shared unit to all possible destinations, one can continue to implement the same functionality, but with a single multiplier (*Mul_2B* in this case). This example highlights general rules for component sharing while preserving functional correctness that are captured in the following theorem.

Theorem 3.1.2 *Functionally equivalent behavior is preserved across a resource sharing manipulation if and only if all of the following conditions are met:*

- *each input port of the resulting shared component must have connectivity, if needed via an inserted multiplexer*[2]*, to the driver of the corresponding input ports on each original component participating in the sharing,*
- *each output port of the resulting shared component must have connectivity to the components driven by the output port on each original component participating in the sharing, and*

[2]connected in such a manner that the appropriate input can be selected

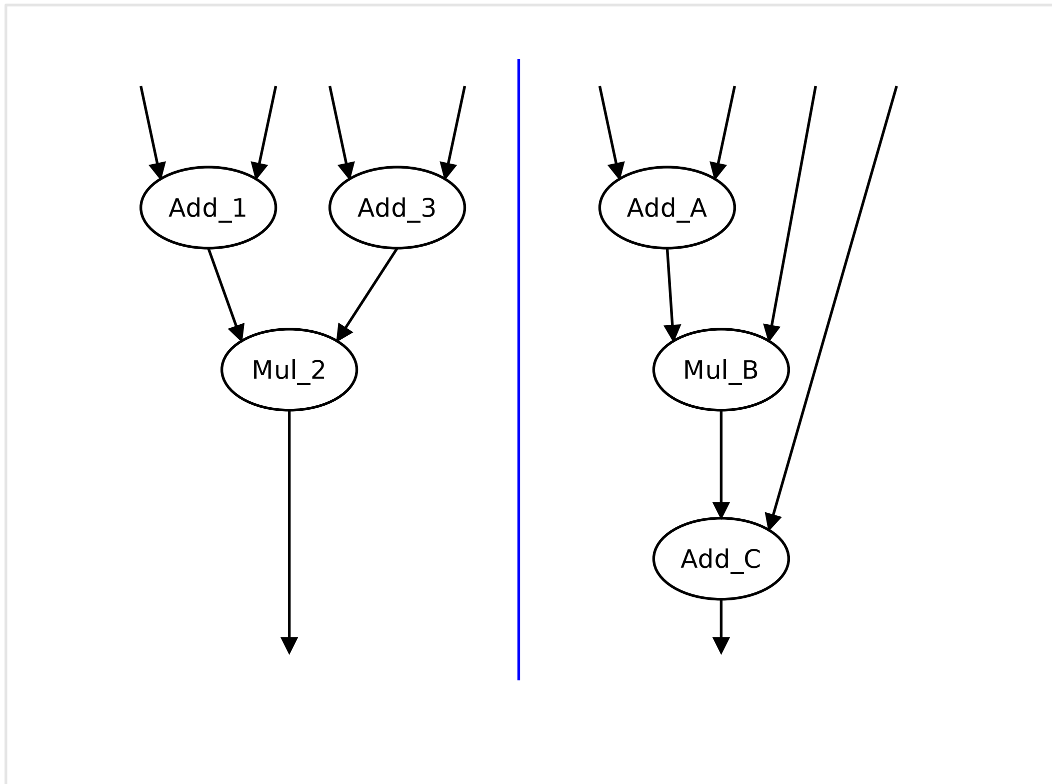

Figure 3.2: Programmable Interconnect Example: The input circuits

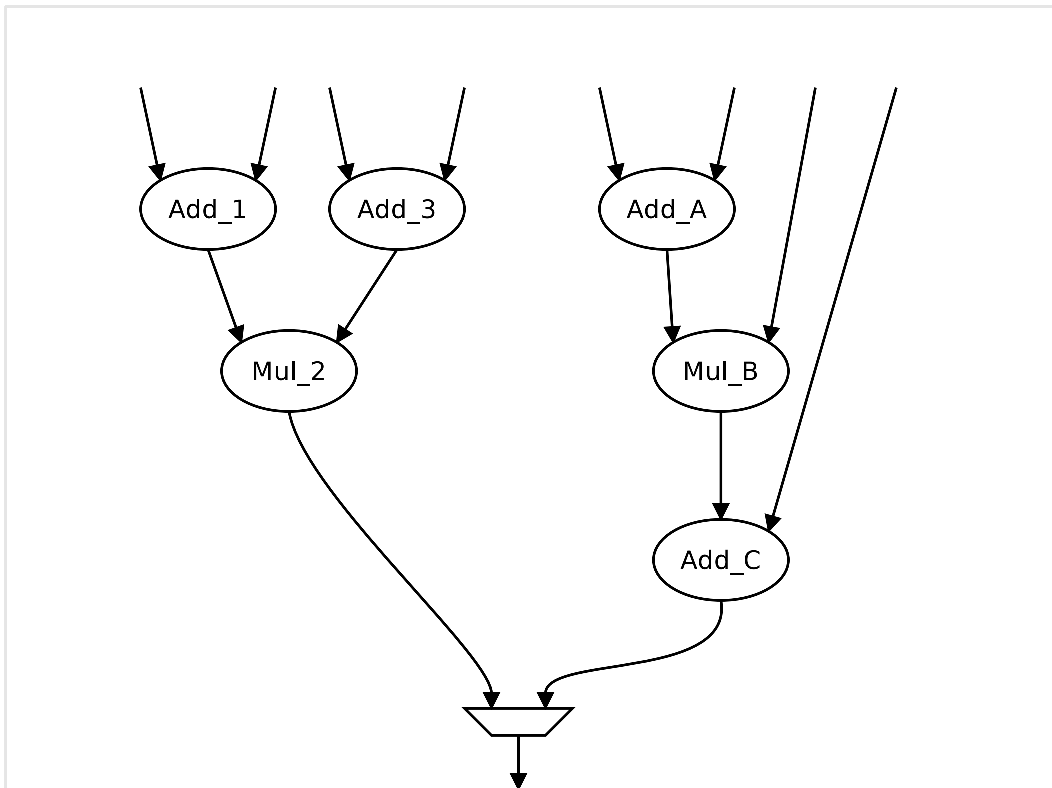

Figure 3.3: Programmable Interconnect Example: Side-by-Side Realization

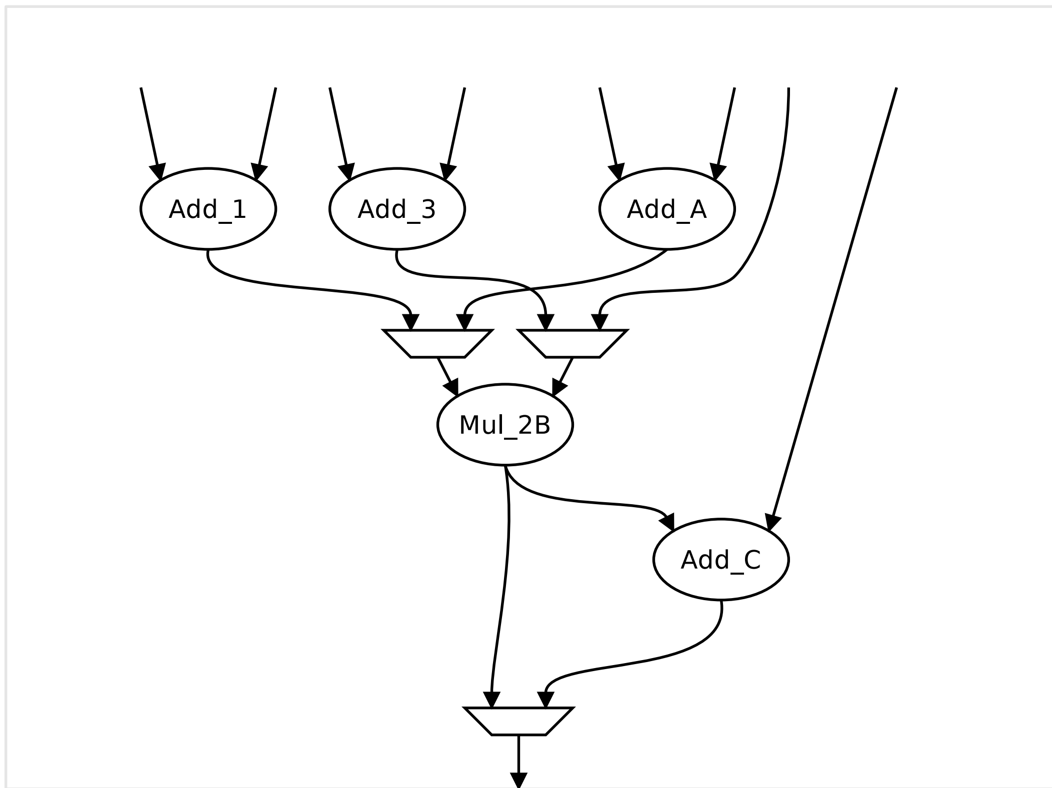

Figure 3.4: Programmable Interconnect Example: Merge *Mul_2* and *Mul_B* into a shared *Mul_2B*

- *the resulting shared component has functionality equal to or greater than the supremum of the set of resources to be shared.*

Proof It will first be shown that functionally equivalent behavior is preserved if the conditions are met, then the converse will be shown.

If the first condition is met, then by the definition of a multiplexer, there must exist some connection to the multiplexer selection ports such that the circuit has equivalent behavior to the original circuit. If the third condition is met, the definition of equal to or greater functionality ensures that there exists some port connections to the shared component that provides equivalent behavior to any of the original unshared components. Finally, downstream components must behave equivalently, because they are required to be connected to the shared component by the second condition. As a result, there must exist a set of connections to the multiplexer control port such that the overall circuit has equivalent behavior.

The inverse will now be proven by showing that if any one condition is not met, then functionally equivalent behavior is not preserved. For the first condition, if there exists an input port of the resulting shared component that isn't connected to the driver of a corresponding input port, then there can exist a circuit in which that input was needed by the shared component for equivalent behavior, and therefore equivalent behavior cannot be preserved. For the second condition, a similar proof applies to the equivalent behavior of downstream devices if an output port is not connected. For the third condition, if there exists a component in the set to be shared that has greater functionality than the resulting shared device, then there must exist a circuit whose behavior the shared circuit is not equivalent to.

Because both the implication and inverse were shown to be true, the biconditional theorem must be true. ∎

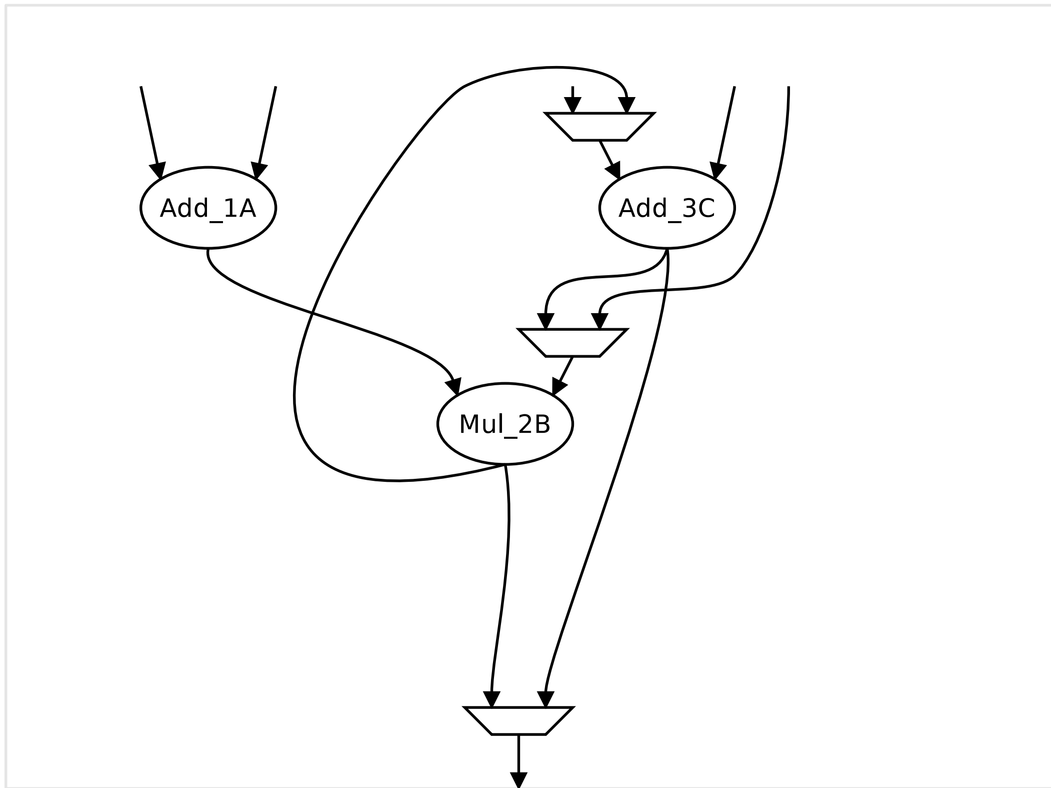

Figure 3.5: Programmable Interconnect Example: Circuit resulting from merging multiple components

The first rule implies that given a component with a set of input ports N that is to be shared between a set of circuits K, one can state that one needs at most $|N|$ $|K|$-input multiplexers to preserve functional correctness. This merging process can be applied multiple times, as demonstrated in Figure 3.5. The binding, or selection of which components to merge can be described as a collection of sets, in which components in each set, at most one from each original circuit, are to be shared. Based upon the example of Figure 3.5, several definitions will be made.

Definition The selection inputs to the multiplexer elements collectively define the task description within the context of a particular programmable device. In this

sense, the inputs to the multiplexer selection elements define a *program* to be executed by the device.

Definition A device's *programming function* maps from the set of functionalities it can be assigned to the set of programs that configure it to execute those functionalities.

If N circuits were merged, the multiplexer selection inputs can be encoded in $log_2(N)$ bits. However, often the number of multiplexers exceeds $log_2(N)$, as is the case in Figure 3.5. Notice that while the two circuits to be implemented were of the form $(A+B)*(C*D)$ and $((A+B)*C)+D$, by appropriately configuring the three multiplexers, one can also implement circuits of the form $(A+B)$, $(C+D)$, $(A+B)*C$. In this sense, one might view the co-domain as larger than the image of the intended programming function. However, the domain of the programming function is defined to be the set of functionalities that can be assigned, which leads to the following definition.

Definition The set of functionalities realizable by a device is referred to as the device's *programmability domain.*

Notice that the ability to execute additional unintended functionalities is not present in the side-by-side circuit of Figure 3.3. In the terminology of Section 2.1.3, the circuit of Figure 3.5 is strictly greater in functionality than the circuit of Figure 3.3. That is, resource-sharing in this example has the effects of (1) increasing the number of multiplexers, (2) decreasing the overall area[3], and (3) increasing programmability. This last observation leads to the following theorem and corollary.

[3]provided that multiplexers are smaller than the adders and multiplier eliminated via sharing

Theorem 3.1.3 *Resource sharing manipulations may result in an increase in functionality, but never in a decrease in functionality.*

Proof Because behavior is preserved by definition in a resource sharing manipulation, the functionality of the resulting circuit must be greater than or equal to the original circuit. Thus resource sharing manipulations never decrease functionality. Moreover, as the example in Figure 3.5 shows, the functionality of the resulting circuit may increase.

Corollary 3.1.4 *In the absence of subgraph isomorphisms between original circuits, resource sharing manipulations will tend to increase functionality.*

Proof Based on the conditions of Theorem 3.1.2, multiplexers are required to be inserted in resource sharing manipulations to preserve functional correctness in the absence of subgraph isomorphisms. Moreover, in the absence of subgraph isomorphisms, the number of these multiplexers is likely to scale asymptotically linearly with the total number of input ports on components that are shared and thereby linearly with the number of shared components (M) times the average number of input ports (N). For K merged circuits, K-input multiplexers will be required, and thus the number of configuration options scales as K^{MN}, while the number of configuration options (and similarly the number of distinct functionalities) for the side-by-side realization scales as only K. While not all configuration options are valid or distinct, in the absence of subgraph isomorphisms, as the number of shared resources (M) rises, it is apparent that so too does the functionality of the resulting circuit. ∎

Based upon the allowance in Theorem 3.1.3 for both strictly preserving and strictly increasing functionality across a resource sharing manipulation, two distinct

inverse manipulations can be defined.

Definition *Resource replication* is a transformation in which a component whose output fans out to multiple sinks is replaced by replicated components, one per sink. This manipulation is the inverse operation of resource sharing. Much as multiplexers may be inserted during resource sharing, any interconnect multiplexers sourcing the component's inputs that are redundant after the transformation may optionally be eliminated. Such elimation may reduce the functionality of the circuit. Similarly, much as resource sharing allows the shared component to be strictly greater in functionality of any original components, in resource replication each of the replicated components may be specialized and have strictly less functionality than the original.

If during the manipulation, functionality is strictly preserved, this is referred to as *complete resource replication*. If functionality is strictly decreased during the resource replication, thereby corresponding to a strict increase in the associated resource sharing manipulation, this is referred to as *incomplete resource replication*.

3.2 Generalized Programmable Interconnect

While the techniques of the prior section were described ostensibly for the purposes of resource sharing, the result of the manipulations is a set of multiplexers and wires that connect fixed-function components in a manner that results in a programmable circuit. This connectivity will be referred to as a programmable interconnect, and will now be generalized to create methodology for modeling programmable devices via the composition of fixed-function components.

3.2.1 Definition

Definition A *programmable interconnect* is a collection of components that provide programmable connectivity between components. Notably, no computation is done in the interconnect. Instead, the sole function of the interconnect is to route data values from the output ports of components to the input ports of components.

Theorem 3.2.1 *Given a programmable interconnect, equivalent connectivity can be constructed solely out of multiplexers and wire components. Such a programmable interconnect is refereed to as a* multiplexer-based programmable interconnect*.*

Proof By definition, the sole function of a programmable interconnect is to route input values to output values. Given a set of potential connectivity for each sink, one can envision constructing a connected multiplexer with an input connected via a wire to each potential source. Because of the properties of multiplexers and wires, there must exist some connection to the selection pins of the multiplexer such that resulting sink will behave as if it were connected directly to a specified source. By repeating this for all sinks, one can yield equivalent behavior of the overall circuit.

■

3.2.2 Multiplexer Elimination

Section 3.1.2 describes a resource sharing manipulation and demonstrates that this manipulation tends to increase functionality. In this section, a manipulation will be described that decreases functionality. Consider the case of merging the two circuits in Figure 3.6. Note that they are isomorphic to more clearly illustrate the functionality reduction, however strict isomorphism is not required.

Figure 3.7 shows a potential merging of the two circuits of Figure 3.6. In this circuit, pairs AddA/Add1 and SubB/Sub1 have been merged (or bound) together.

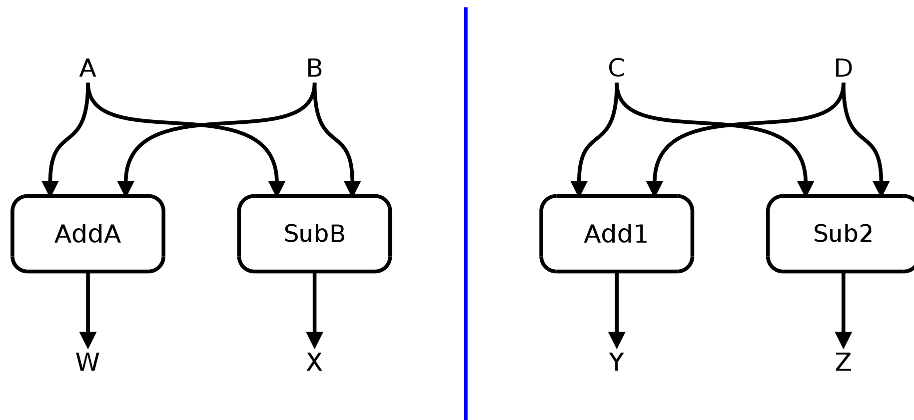

Figure 3.6: Two example (isomorphic) circuits to be merged.

However, note that Mux1 and Mux3 are redundant, as are Mux2 and Mux4. To improve resource usage, one might consider removing these redundant pairs, as shown in Figure 3.8. However, this decreases functionality. For example, the circuit of Figure 3.7 is capable of implementing the functions $W = A + D, X = C - D$, but this cannot be configured in the circuit of Figure 3.8.

Theorem 3.2.2 *Multiplexer elimination within a multiplexer-based programmable interconnect does not increase functionality, but may decrease it.*

Proof Multiplexer elimination strictly decreases the number of multiplexers, thereby reducing the number of configuration options, and thus potentially the functionality of the circuit. Because after sharing the configuration options present multiplexers is a subset of those present prior, functionality will not increase. However, as shown in the example of Figure 3.8, functionality may decrease. ∎

Definition Multiplexer elimination induces multiple fan-out from multiplexers within a programmable interconnect. The inverse manipulation, termed *multiplexer replication*, reduces fan-out. An interconnect composed exclusively of multiplexers with

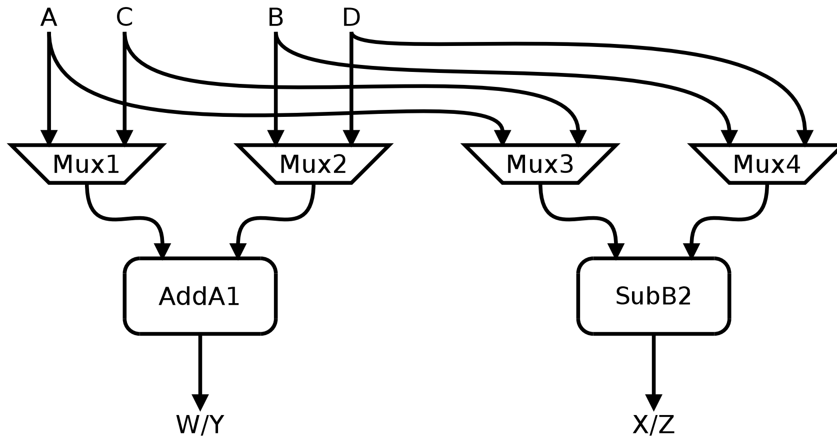

Figure 3.7: A merging of the circuits in Figure 3.6. Pairs AddA/Add1 and SubB/Sub1 have been merged together. Observe that Mux1 and Mux3 are possibly redundant, as are Mux2 and Mux4.

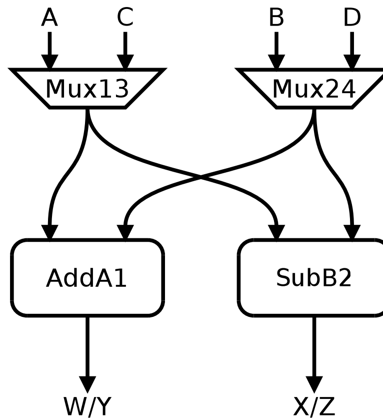

Figure 3.8: Circuit of Figure 3.7, except with redundant pairs Mux1/Mux3 and Mux2/Mux4 eliminated

a fan-out of one is said to be *fully replicated*, and via inductive application of Theorem 3.2.2 can be shown to have the greatest possible functionality for a given binding of application components.

Theorem 3.2.3 *Resource sharing among the fixed-function units preserves or increases functionality, while resource sharing amongst the programmable interconnect strictly decreases functionality.*

Proof Note that multiplexer elimination is a form of resource sharing distinguished by sharing interconnect resources, instead of the application components. Given this observation, the stated duality follows by simple aggregation of Theorems 3.2.2 and 3.1.2. ∎

3.2.3 Completeness

In prior sections, two invertible manipulations have been examined: resource sharing amongst application components, and resource sharing amongst interconnect components. This section will show that the complete design space of multiplexer-based interconnects can be traversed via these two invertible manipulations alone.

Theorem 3.2.4 *For any given multiplexer-based programmable interconnect connecting application components, there exists a series of multiplexer replication manipulations followed by a series of resource replication manipulations that yield a side-by-side realization of the programmability domain of the original circuit.*

Proof A constructive proof will be given. For each multiplexer in the programmable interconnect with a fan-out greater than one, perform a multiplexer replication. Repeated application yields a fully replicated design point, with a (possibly trivial) multiplexer tree driving every input of every application component. For each

n-input multiplexer present in the programmable interconnect, construct n side-by-side circuit realizations, each connecting the respective input of the multiplexer to the output of the multiplexer. By repeated application of this resource replication manipulation, a side-by-side realization of the programmability domain of the original circuit will be constructed.

Note that functionality is preserved during each manipulation and therefore inductively the resulting circuit also preserves functionality. Note further that the resource replication manipulations move multiplexers from the circuit internals to the output. Through exhaustive application, the only interconnect multiplexer remaining will be at the output, thereby forming a side-by-side construction as shown in Figure 3.1. Thus it has been shown that the circuit resulting from the described manipulations is of side-by-side format and preserves the functionality of the original circuit. ∎

Corollary 3.2.5 *For any given multiplexer-based programmable interconnect connecting application components, there exists a set of resource sharing manipulations followed by a set of multiplexer elimination manipulations that can yield the specified programmable interconnect from a side-by-side realization of each of the circuits within its programmability domain.*

Proof Because each individual manipulation is invertible and does not decrease functionality, proof follows via the reverse sequence of Theorem 3.2.4. ∎

While Theorem 3.2.4 as written is only valid for side-by-side realization of all of the circuits within a given programmability domain, by incompletely (see definition 3.1.2) expanding the combinatorial interactions between multiplexers, it is possible to selectively cull various options within the side-by-side realization.

Such a selective expansion decreases the functionality of the resulting side-by-side realization. Thus, it can be shown that a series of manipulations exists between any programmable interconnect and side-by-side realization, without requiring that the side-by-side realization cover the entire programmability domain.

Theorem 3.2.6 *Any possible multiplexer-based programmable interconnect can be yielded by manipulating a side-by-side realization of the desired functionality via a series of resource sharing manipulations followed by a series of multiplexer elimination manipulations.*

Proof As explained above, this follows from Theorem 3.2.5 and the selective application of incomplete resource replication. ∎

Corollary 3.2.7 *For any cost function, the optimal multiplexer-based programmable interconnect can by yielded by manipulating a side-by-side realization via a series of resource sharing manipulations followed by a series of multiplexer elimination manipulations.*

Proof Since the optimal multiplexer-based programmable interconnect is by definition a multiplexer-based interconnect, then by Theorem 3.2.6, the series of manipulations must exist. ∎

3.3 Programmability: *a priori* and *a posteriori*

While the data path merging approach yields a circuit capable of implementing the functionality of any of the input circuits specified prior to circuit synthesis, this provides only a subset of the benefits traditionally associated with a programmable device. Often, the benefit of a programmable device is that it can be used to

implement functionality not envisioned at the time the device was designed (e.g. bug fixes, or feature enhancements). It is convenient to study these two design problems separately. The former problem, that of designing a device to implement a specific set of functions known prior to design time will be referred to as the *a priori* design problem. The latter problem, that of designing a device to be capable of being assigned a function not envisioned until after design time, will be referred to as the *a posteriori* design problem.

The *a priori* problem is addressed via data path merging techniques of Sections 3.1.2, and optimization techniques of section 5.3. In contrast, the *a posteriori* problem will be addressed in this section.

3.3.1 Formal Definition

The distinguishing characteristic of the *a posteriori* problem is to design for the implementation of functionality which cannot be precisely described at design-time. However, often some information about this functionality is available as design-time, and so it is useful to use stochastic terminology to formally define the *a posteriori* problem.

Consider the set of all possible functionalities that one may want to assign to a programmable device. Assume further that one can assign a probability of implementation to each of those functionalities. We can then define the programmability domain of the *a posteriori* problem as the set of functionalities with non-zero probability of being assigned to the device. The *a posteriori* design problem is then that of finding a circuit capable of implementing each of the functionalities in the programmability domain.

One may also consider for each possible solution to the *a posteriori* design

problem, there exists a function, referred to as the cost profile function, that maps from the programmability domain to the cost metric space of section 2.2, and yields the cost of implementing the functionality of point in the particular solution circuit. One could then consider that there is an optimization cost function mapping from the set of cost functions to the set of reals. The *a posteriori* optimization problem is then that of finding the circuit capable of implementing each of the functionalities in the problem domain, with the cost profile function having the minimum cost.

3.3.2 Naive Solution

Given the formal definition of the prior section, a straightforward approach to solving the *a posteriori* problem is to enumerate circuits that will implement each of the required functions and merge them using the *a priori* techniques of of Sections 3.1.2 and 5.3. This will generate a programmable circuit capable of implementing any of the functionality in the programmability domain via the configuration of appropriate multiplexers, thereby creating a solution to the *a posteriori* problem.

While this approach is simple, it may not be practical in the case of a programmability domain with high cardinality. Higher-level approaches are therefore advantageous and will be examined in Section 3.3.3.

3.3.3 Desirable Properties

When one thinks of a programmable architecture capable of implementing arbitrary computations, one typically envisions particular characteristics. Several of these characteristics are now formalized for later use.

We begin with properties of the computational primitives:

- *Compositional Universality* - The property that any computable function can

be computed by appropriate composition of the available computational primitive elements. This condition is easily satisfied, for example, via the availability of a NAND primitive.

- *Function Coverage* - The property that, via suitable composition of primitives, one can implement any function mapping from the set of possible data values to the set of possible data values. As an example, one could envision that a machine with operating on multi-bit words, but only bit-wise NAND operators would not satisfy this condition, as one could not propagate the carry bits needed for addition within words. In that case, each bit position would be an separate partition.

- *Standard-completeness* - Decades of experience have resulted in programmers expecting specific sets of computational primitives to be available. Notably, the C language standard [50] defines no less than 14 distinct primitives: (e.g. bitwise negation, logical negation, multiplication, division, modulo, addition, subtraction, bitwise-shift left, bitwise-shift right, arithmetic shift-right, equality, bitwise and, bitwise or, and bitwise xor). While a subset of these primitives are necessary to implement arbitrary binary functions (i.e. and/or/xor/not), and a subset are necessary to implement bitwise connectedness (shifts), others are arguably convenience functions expected to be present.

We now consider properties of the interconnect between computational primitives:

- *Instance Connectedness* - The property that the output from any primitive instance can be used as the input to any other primitive instance. This corresponds to the property of strong connectedness in the interconnect.

- *Class Connectedness* - The property that arbitrary connections can be made between classes of primitive components. That is, for any pair of primitives a and b, there exists a path in the programmable interconnect from a primitive of equal or greater functionality of a to the input of a primitive of equal or greater functionality b. This is a related, but weaker version of *Instance Connectedness*, as it is satisfied even if only a single instance of each class of primitives is strongly connected.

- *No Coupling* - The property that the output of any primitive instance can be connected to any other primitive instance, without requiring the connection of any other pair of primitive components. For a more familiar analogy, in a microprocessor, one can typically multiply registers A and B together and place to result in register C without being required to corrupt register D in order to do so.

- *Bounded Coupling* - The property that any primitive instance can be connected to any other primitive instance, without requiring more than a finite and bounded number of additional connections. For a familiar analogy, in a microprocessor, if we need to do addition, we may be required to alter the (carry) flag register, but it is possible to do addition without modification of unrelated registers.

- *Constant access* - The property that arbitrary data values defined *a posteriori* can be encoded and later used as the input to any computational primitive.

These properties will be used in Chapter 4 to evaluate the programmability of circuits produced experimentally.

3.4 Programmability Overheads

In the prior sections, a model of programmability has been developed in which a collection of fixed-function components are connected by a programmable interconnect. This model, while certainly not unique, underlies the construction of a large body of programmable devices: FPGAs, the datapath of microprocessors, and coarse-grained reconfigurable logic.

This section is intended to provide insight into the costs of programmability within a multiplexer-based programmable interconnect model, especially the asymptotic scaling of costs as the number of implemented functionalities increases. To do so in a manner independent of the peculiarities of specific circuits, a parametrized stochastic approach is pursued. The specific approach selected, based upon "Rent's rule" [58], is well known in the EDA community [16] and has proved useful in quantifying circuit characteristics in order to estimate features including wirelength distributions[30, 24] and average wirelengths [29, 82]. These results have in turn been used to estimate critical-path lengths, dynamic-power dissipation, and die areas [25]. Among the parameters used are:

- C - the number of components in a circuit (or subset thereof),
- p - the Rent exponent,
- k - the average number of terminals per component,
- α - the fraction of terminals that are inputs (related to the average fanout f by the equation $\alpha = f/(f+1)$ [24]), and
- n_{cp} - the number of components in the critical path.

Further, several parameters are defined specific to this effort:

- N - the number of circuits being merged. In effect, this is the number of distinct functionalies required in the programmability domain of a generated device.

- β - the fraction of multiplexers that need to be inserted to maintain functional correctness. Recall from section 3.1.2 that, in the case of topological similarities, multiplexers may not be required in certain resource sharing manipulations.

Theorem 3.2.6 shows that given any set of functionalities, there exists a multiplexer-based programmable interconnect constructed via a series of resource sharing and multiplexer elimination manipulations from a side-by-side realization of those functionalities. Theorems 3.1.2 and 3.2.2 provide restrictions on how these two types of manipulations impact the programmability domain and the construction of these devices. Notably, Theorem 3.1.2 provides three conditions for preserving equivalent behavior across resource sharing manipulations. The overheads inherent in programmable interconnect in terms of these three conditions will now be considered.

3.4.1 Multiplexer Insertion

The first condition of Theorem 3.1.2 states that for a resource sharing manipulation to be functionality-preserving "each input port of the resulting shared component must have connectivity, if needed via an inserted multiplexer, to the driver of the corresponding input ports on each original component participating in the sharing."

This condition implies the insertion of multiplexers, potentially one for each input port. From the parameters defined in the introduction to this section, we can compute the number of multipliers needed as the average number of input

ports (αk) per component, times the fraction of multiplexers needed (β), times the number of components is the resulting programmable. Thus the number of N-input multiplexers needed can be estimated by $\alpha k \beta C$. However, this model is difficult to utilize because the number of inputs needed for each multiplexer increases as the number of required functionalities increases (N). This effect causes the average number of terminals per gate (k) and fraction of input terminals (α) to vary with N. Theorem 2.2.1 allows an alternate path allowing simpler computations: in place of introducing each N-input multiplexer, consider the introduction of assemblies composed of $(N-1)$ 2-input multiplexers. Based on this derivation, the number of 2-input multiplexers inserted design can be computed as:

$$M_{programmable}(\alpha, k, \beta, C, N) = \alpha k \beta C(N-1). \tag{3.1}$$

Based on Table 2.1, the worst-case overhead due solely to the inserted multiplexer components can be estimated as follows:

- $\Delta delay = delay(MUX2) n_{cp} ceil(log_2(N))$,
- $\Delta energy = energy(MUX2) M_{programmable}$,
- $\Delta area = area(MUX2) M_{programmable}$, and
- $\Delta leakage = leakage(MUX2) M_{programmable}$.

In practice, the energy and leakage costs of the multiplexers can be substantially reduced through proper operand isolation and power gating techniques.

3.4.2 Topological Costs

The second condition of Theorem 3.1.2 states that for a resource sharing manipulation to be functionality-preserving, "each output port of the resulting shared component must have connectivity to the components driven by the output port on each original component participating in the sharing." This requirement induces overhead relative to a fixed-function equivalent due towires needed implement alternative functionalities. As wiring costs are intimately related to component placement, discussion of placement will be performed first. Two cases will be considered: 1) a placement derived from an existing privileged functionality placement, and 2) free placement.

First, consider a placement derived from a privileged functionality (i.e. prescribed placement). In the application of adding functionality to an existing fixed-function design with minimal impact, one might choose to largely reuse the placement of the fixed-function design. One might further envision placing components for alternative functionalities entirely adjacent to the existing fixed-function design. The only components needed to be inserted topologically inside of the fixed function design being 2-input multiplexers to inject alternative signals when needed, and buffers to extract needed signals from the fixed function circuit. By doing so, provided that the buffers and 2-input multiplexers are small relative to the fixed function circuitry, both the topology and size of the fixed function circuit remain largely unchanged. As a result, impact to the privileged fixed-function circuit is minimal at the expense of the performance of the alternate functionality. Such a case is theoretically straight-forward, and of limited academic curiosity.

In contrast, consider the case where the placement of resulting programmable circuit is allowed to vary. Such flexibility provides the potential for greater perfor-

mance of the alternate functionality, and integrates easily with existing EDA tool flows at the unplaced netlist level. A disadvantage of this flexibility is the potential for negative impact on the original fixed-function circuit, and so these overheads will be considered here in greater detail.

To understand wiring overheads, Rent's rule will be utilized. The reader is referred to section 2.3.3 for background information on Rent's rule and its applicability to placement and wire-length estimation.

Intrinsic Rent Exponent for Side-by-Side Realizations

As noted by Hagen, et al [43], partitioning-derived Rent parameters for a circuit provide means for estimating the number of wires that will cross between recursive partitions of a given circuit. This knowledge can be used to estimate wirelength distributions[30], including average wirelengths [29, 82] without circuit placement. However, it will be utilized here to estimate the wiring overhead caused by merged disinct designs via resource sharing manipulations.

Theorem 3.4.1 *For side-by-side realizations, neglecting the influence of the output multiplexer, the intrinsic Rent parameter of the overall circuit is bound to less than the maximum intrinsic Rent parameter of any individual circuit.*

Proof Since the intrinsic Rent parameter is defined as "the minimum Rent parameter attained over all hierarchical decompositions of a given circuit" [43], the existence of an intrinsic Rent parameter for each of the circuits to be merged guarantees the existence of a corresponding hierarchical decomposition following the statistical relationship of equation 2.30. Conversely, to demonstrate an upper bound on the intrinsic Rent parameter of the merged circuit, it is sufficient to demonstrate

the existence of hierarchical decomposition following the statistical relationship of equation 2.30.

Such a decomposition can be made by first partitioning the overall circuit into its composing circuits. Because the circuit is a side-by-side realization, neglecting this output multiplexer, this partitioning can be made with zero edge cuts. Further, because zero edges cross the partition, this partition cannot increase the Rent parameter of the overall circuit. Recursive partitioning can then proceed for each circuit individually that must necessarily exist to satisfy the intrinsic Rent statistic given for each circuit individually.

Now consider the Rent parameter of the specified decomposition. The Rent parameter is a statistical relationship between the number of external terminals and size of the partition when sampled over a hierarchical decomposition. In the specified decomposition of the side-by-side realization, the original partitions (i.e. samples) from each circuit have been preserved and aggregated together. As a result, for a given partition size, the number of external terminals can in no case be larger than any of the original circuits. As a result, the intrinsic Rent statistic for the side-by-side realization cannot be any larger than that of the original circuits. ∎

Note that the theorem above neglects the influence of the output multiplexer. To include the effects of the output multiplexer, one can first partition the output multiplexer into a separate partition, and then proceed with the decomposition as described above. In this case, the output multiplexer partition becomes a single sample in determination of the intrinsic Rent parameter statistic. Application of the law of large numbers yields the following corollary.

Corollary 3.4.2 *For sufficently large input circuits, the intrinsic parameter of the side-by-side realization will approach a value no larger than the maximum intrinsic*

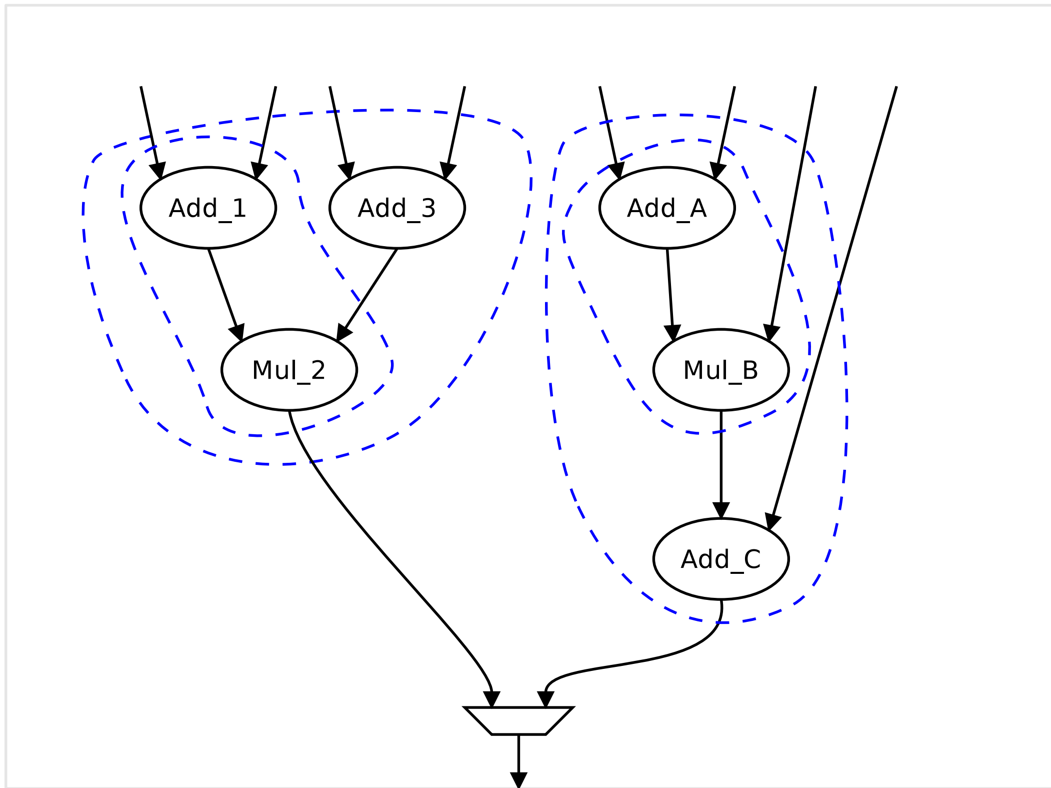

Figure 3.9: Circuit of Figure 3.3 with a partition boundaries for an example hierarchical decomposition shown as dashed lines

Rent parameter of any individual circuit.

Maximal Resource Sharing

In the prior section, it shown that a side-by-side realization has an intrinsic Rent exponent no larger than the intrinsic Rent exponent of any input circuit. In this section, a similar upper bound will be shown for realizations that incorporate resource sharing and multiplexer elimination manipulations to achieve the property of *maximal resource sharing.*

Definition For any set of circuits composed of components with homogeneous func-

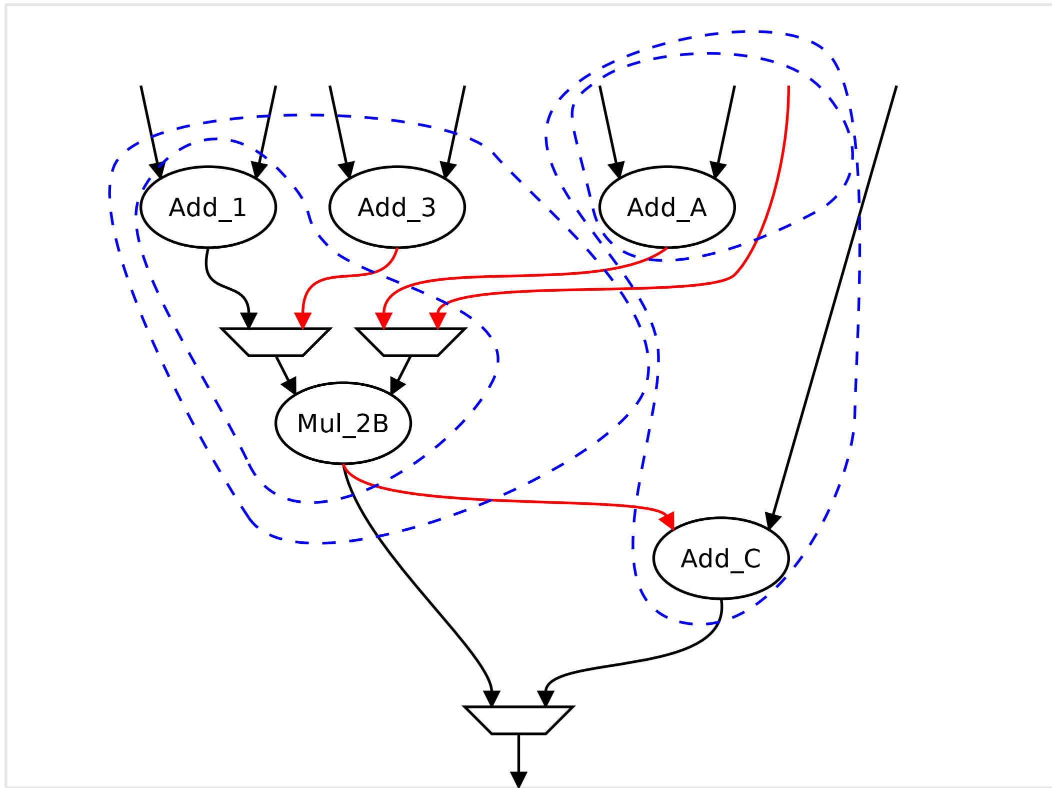

Figure 3.10: Circuit of Figure 3.9, following a resource sharing manipulation involving *Mul_2* and *Mul_B*. Partition boundaries for the example hierarchical decomposition of Figure 3.9 are shown as dashed lines. Boundary-crossing connectivity needed for the merge has been highlighted.

tionalities, a programmable implementation of these circuits is said to have the property of *maximal resource sharing* if the cardinality of components in the realization is no greater than that of the largest circuit in the set.

To simplify the discussion, the restricted problem of homogeneous components will be considered first. In this restricted problem, all components of all circuits to be realized have identical functionality and therefore can be bound indiscriminately. While this simplified problem is of theoretical use in this section, it also has direct practical applications for traditional look up table-based FPGA architectures.

Theorem 3.4.3 *For any set of circuits composed of components with homogeneous functionalities, there exists a multiplexer-based programmable interconnect that possesses an intrinsic Rent exponent no greater than the greatest intrinsic Rent exponent of any single circuit in the set. Moreover, if the partitions of the hierarchical decompositions associated with the intrinsic Rents exponent for each circuit are balanced in size, then the resulting interconnect possesses the maximal resource sharing property.*

Proof Consider a set of circuits I, that that individually satisfy Rent's equation 2.30 as follows

$$T_i = k_i \cdot C_i^{p_i} \text{ for all } i \in I. \tag{3.2}$$

Here for circuit i, T_i is the average number of external terminals (pins) in a subcircuit or partition; k_i is *Rent's constant*, a scaling constant which corresponds to the average number of terminals per module; C_i is the number of modules in the subcircuit (or partition); and p_i is the *Rent parameter* or *Rent exponent*, with

$0 \leq p_i \leq 1$. For simplicity in notation, this derivation assumes that the average number of terminals per module k_i is constant for each circuit[4], or symbolically, $k_i = k$ for all $i \in I$.

By definition, the existence of an intrinsic Rent exponent for a circuit implies the existence of a hierarchical decomposition of that circuit that follows the relationship of equation 3.2. Based upon the existence of this decomposition for each circuit in I, a multiplexer-based programmable implementation can be constructed that satisfies equation 2.30 with a p no greater than

$$\max_i p_i. \tag{3.3}$$

To construct this implementation, first construct a binding by simultaneously traversing the hierarchical decompositions of each circuit and binding the partitions at each level with partitions from corresponding levels from each circuit. For example, consider the hierarchical decompositions of the circuits shown in Figure 3.9. In this example one might initially bind *Add_3* with *Add_C*, and the set {*Add_1*, *Mul_2*}, with the set {*Add_A*, *Mul_B* }. Proceeding down a hierarchy level, one might then bind *Add_1* with *Add_A* and *Mul_2* with *Mul_B*. If the partitions are balanced at each level[5], then the resulting binding satisfies the maximal resource sharing property.

With the binding established, the construction of a programmable implemen-

[4]This assumption can be made without loss of generality. Variation of k_i, the average number of terminals per module, between circuits corresponds to differing levels of granularity implicit in the term "module". By pre-clustering "modules" together in circuits with low k_i, one decreases the effective granularity of those circuits, thereby increasing k_i to match the other circuits. Moreover, in the datapath merging application even this accommodation is unlikely to be needed, as good candidates for datapath merging will tend to be built from the same library of underlying modules, and will therefore tend to have a similar k_i.

[5]In the case of an odd number of components, this requires matching the larger partitions together and smaller partitions together.

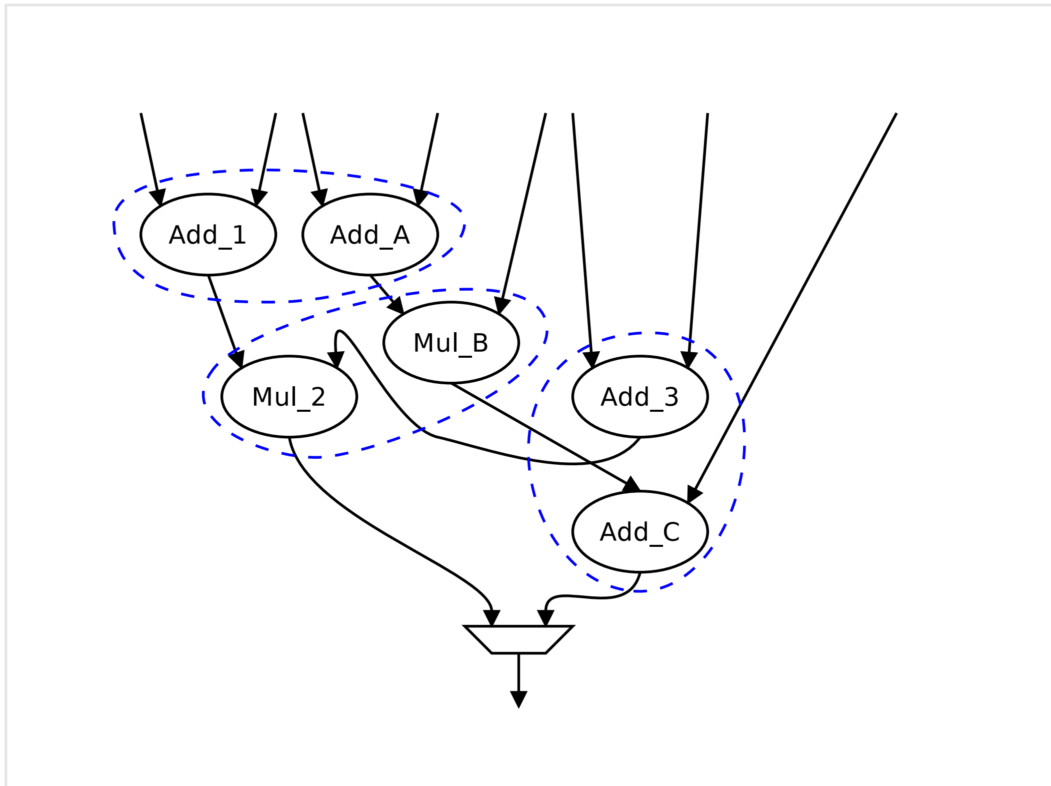

Figure 3.11: Circuit of Figure 3.9, shown with bound components grouped by regions bordered with dashed lines.

tation will now be described. Begin with the side-by side realization of the circuits as shown in Figure 3.9. Bound components for the ongoing example are shown grouped with dashed lines in Figure 3.11. Perform resource sharing manipulations between bound components, sharing inputs and multiplexing outputs as described in section 3.1.2. As shown explicitly in Figure 3.12, the maximum number of connections (i.e. terminals) needed to a set of shared components is bound to be no more than the maximum connections needed for any single original component.

By recursively mapping the partitions of the original circuits matched during the binding process, a hierarchical decomposition of the resulting circuit is created.

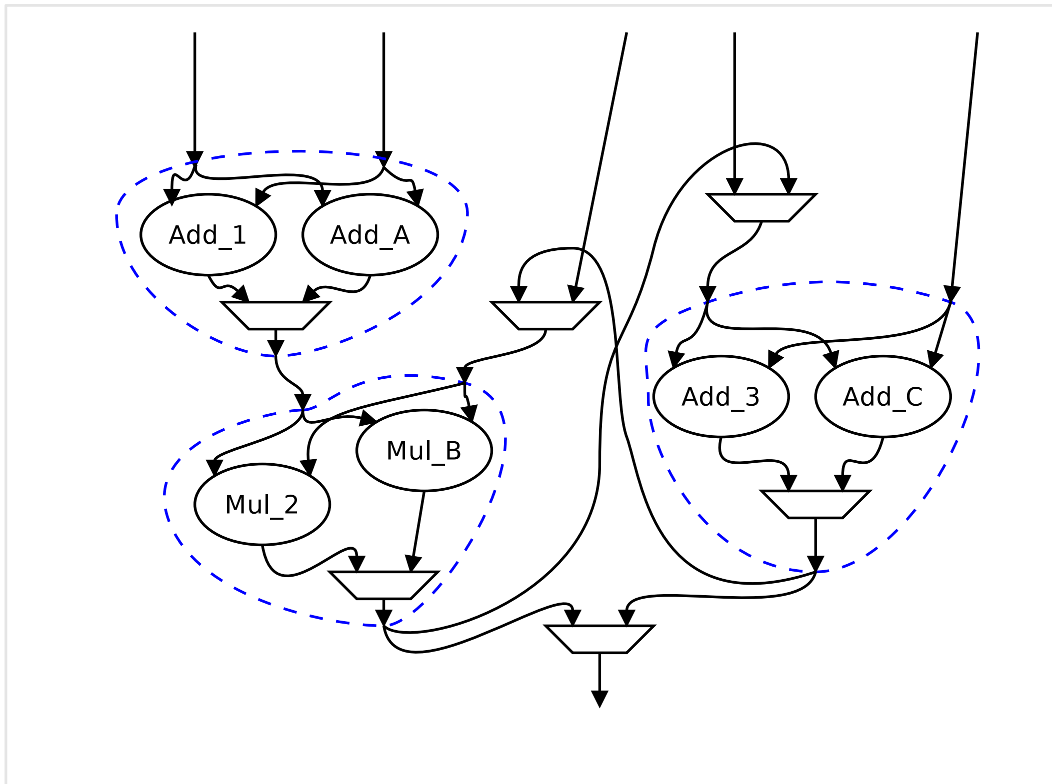

Figure 3.12: Circuit of Figure 3.11 with resource sharing manipulations shown partially complete. The multiplexers and required fan-outs have been inserted, but the components themselves have not been merged.

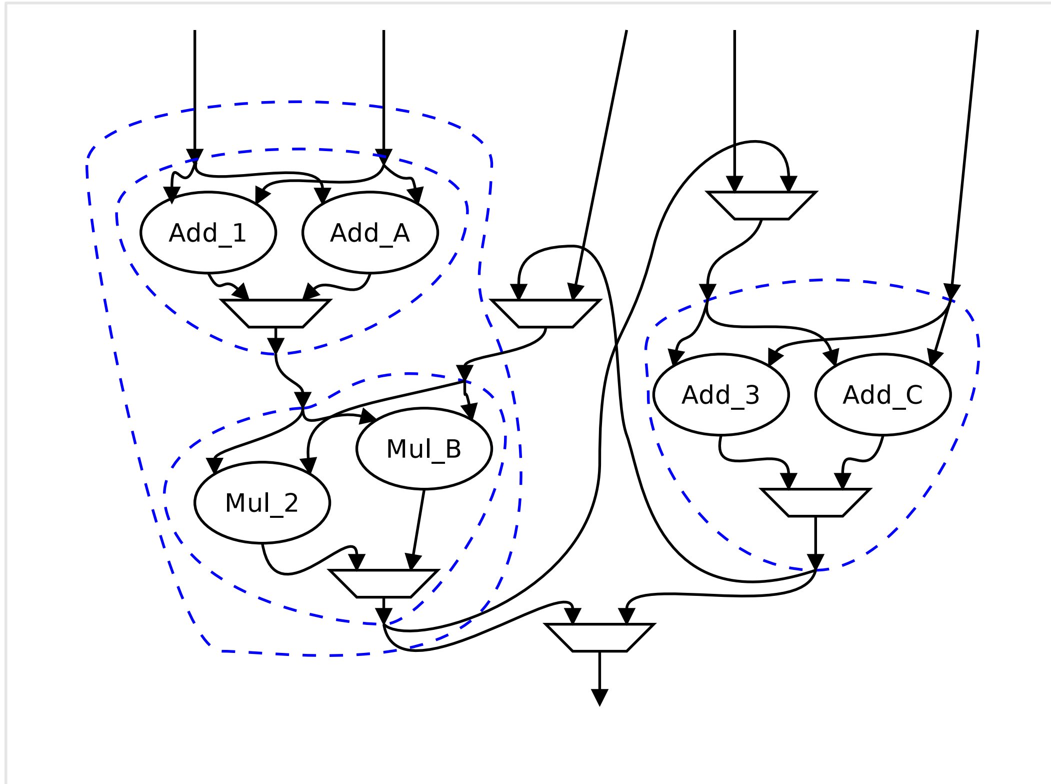

Figure 3.13: Circuit of Figure 3.11, shown with hierarchical decompositions from Figure 3.9 mapped as described in the text.

For each partition at each hierarchical level, the overall number of required external terminals is no greater than that required for any single circuit in the set I[6]. See Figure 3.13 for the mapping of partitions in the case of the example.

With the hierarchical decomposition established, consider individual partitions in the context of equation 2.30. For every partition, the number of terminals in the shared realization T_{shared} has been shown to be less than or equal to $\max_{i \in I} T_i$.

[6]The necessity of bidirectional connections to achieve this bound is acknowledged. In many Rent's rule treatments, this mismatch between inputs and outputs is handled via an α parameter that is the fraction of terminals that are sink terminals and is related to the average fanout f by the equation $\alpha = f/(f+1)$ [24]. Donath notes that since "the range of variation is relatively small for α, it is a good approximation to assume that variations of α are insignificant." [29]

Moreover, the number of components in each partition must be at least as great as that of any one of the input circuits. That is, for any partition, the equation $C_{shared} \geq \max_i C_i$ holds. The potential inequality results from the potential addition of multiplexers.

By solving equation 2.30 for p we find that

$$p = \log_C \left(\frac{T}{k} \right) \tag{3.4}$$

By considering that $C_{shared} \geq \max_i C_i$. and $T_{shared} \leq \max_i T_i$, it is a simple algebraic excercise to show that

$$\log_{C_{shared}} \left(\frac{T_{shared}}{k} \right) \leq \log_{C_i} \left(\frac{T_i}{k} \right) \text{ for all } i \in I, \tag{3.5}$$

and therefore that

$$p_{shared} \leq \max_{i \in I} p_i. \tag{3.6}$$

∎

There is a conceptual similarity of the prior proof to the multi-level partitioning strategy for the placement problem studied by Breuer [9], Dunlop, and Kandernighan [31] among many others. In this strategy, a circuit is recursively partitioned, with the resulting partitions mapped onto partitions of the placement space. Ultimately a placement of a circuit is derived. From a practical perspective, the proof above suggests a conceptually similar multi-level partitioning algorithm for the binding subproblem of datapath merging. From a more abstract perspective, by creating a mapping between two (or more netlists), the proof also suggests that multi-level partitioning strategies need not be limited to mapping netlists onto

Cartesian spaces. Instead, one can generalize the partitioning-based placement problem by making the recursive partitioning of the placement space explicit. This, while useful, should not be surprising as both the placement problem and the wirelength-minimization binding subproblem of datapath merging are two instances of a more general topological mapping problem.

Impact on Average Wirelengths

To gauge the impact of introducing programmability into a fixed-function circuit, consider the case of merging N random circuits, each with an intrinsic Rent exponent of p_n and a number of components C_n for $n \in 1..N$. Individually, when placed into a realization space with *2D Rent's rule wiring costs* as defined in section 2.3.3, each circuit would be expected to have an average interconnect length $\overline{R_n}$ given by

$$\overline{R_n} \sim \begin{cases} C^{p_n - 0.5} & \text{for } p_n > 0.5 \\ \log C & \text{for } p_n = 0.5 \\ f(p_n) & \text{for } p_n < 0.5, \end{cases} \tag{3.7}$$

where $f(p_n)$ is an unspecified function of only p_n.

Theorem 3.4.3 provides that, in the homogeneous component case, there exists an resource sharing implementation such that the overall Rent exponent of the circuit $p_{programmable}$ is bounded by

$$p_{programmable} \leq \max_n p_n. \tag{3.8}$$

A resource sharing implementation, particularly one with maximum resource sharing property, will possess both additional wires and components (e.g. multiplex-

ers) relative to any of the realizable functionalities implemented alone. If one defines the number of 2-input multiplexers needed as $C\alpha\beta k(N-1)$, where α is the fraction of terminals that are sinks [29], β is the fraction of multiplexers possible that are actually inserted, and k is the average number of terminals per component, then one would expect the number of components in the shared realization $C_{programmable}$ is the sum of the maximum number of components in any single functionality, plus the added multiplexers, or

$$\begin{aligned} C_{programmable} &= (\max_n C_n) + (\max_n C_n)\alpha\beta k(N-1) \\ &= (\max_n C_n)(1+\alpha\beta k(N-1)). \end{aligned} \tag{3.9}$$

Thus we would expect the average wirelength of the resulting circuit to be bound by

$$\overline{R_n} \sim \begin{cases} [(\max_n C_n)(1+\alpha\beta k(N-1))]^{(\max_n p_n)-0.5} & \text{for } (\max_n p_n) > 0.5 \\ \log\left[(\max_n C_n)(1+\alpha\beta k(N-1))\right] & \text{for } (\max_n p_n) = 0.5 \\ f((\max_n p_n)) & \text{for } (\max_n p_n) < 0.5. \end{cases} \tag{3.10}$$

For a better intuition, consider the case where every functionality has the same number of components ($C_n = C$), Rent exponent ($p_n = p$), and has an average wirelength exactly as predicted by equation 2.31. Define the *average wirelength overhead* ($\overline{R}_{programmable}/\overline{R}_{fixed}$) as the ratio of the resulting average wirelength to that of input functionalities. Algebraic simplification of the equation above yields

$$\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}} \leq \begin{cases} [(1+\alpha\beta k(N-1))]^{p-0.5} & \text{for } p > 0.5 \\ 1+\log_C[1+\alpha\beta k(N-1)] & \text{for } p = 0.5 \\ 1 & \text{for } p < 0.5. \end{cases} \quad (3.11)$$

Notably, in the case of $p < 0.5$, which may be encountered in "highly serialized circuitry [that] may be designed with low exponents (as little as 0.47)" [29], there is no expected overhead in average wirelengths. However the total number of wires, and therefore total wirelength, will certainly increase. In the case of $p = 0.5$, which is typical of regular, planar memory topologies, the average wirelength scales approximately logarithmically with the number of functionalities to be implemented. In the common case of $p > 0.5$, the average wirelength scales in a more complex manner. Because by definition$\alpha \leq 1$, $\beta \leq 1$, and $p \leq 1$, for larger circuits ($C \geq 6$), a simplified but looser bound can be derived from equation 3.11:

$$\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}} \leq \sqrt{1+k(N-1)}. \quad (3.12)$$

Because multiplexers with a fixed number of inputs were assumed, k is approximated as being independent of the number of circuits implemented. With this assumption the average wirelength in this case scales asymptotically with the number of functionalities no worse than $\sqrt{N}$, but often much better. For example, even in the rather pessimistic case of "highly parallel circuitry" with "large exponents p (as much as 0.75)" cited by Donath [29], an arbitrary average terminal count of $k = 4$, an arbitrary average fanout of 4, no multiplexer elimination due to topological similarities, and 100 merged functionalities, equation 3.11 estimates an average

wirelength of only 3X relative to a fixed-function circuit:

$$(1 + (0.2)(1.0)(4)(100 - 1))^{(0.75-0.5)} = 2.99. \tag{3.13}$$

Number of Wires

Donath [29] and Davis et al [24] show that the expected total number of wires for a circuit W with a Rent parameter p, average terminals per component of k, fraction of terminals that are sinks α, and C number of components is given by

$$W = \alpha k C(1 - C^{p-1}). \tag{3.14}$$

This derivation is commonly used with the average wirelength $\overline{R}$ to estimate a total wirelength. Following a strategy similiar to that used in the prior section to estimate *average wirelength overhead* ($\overline{R}_{programmable}/\overline{R}_{fixed}$), a derivation for the *wire cardinality overhead* ($W_{programmable}/W_{fixed}$) will now be shown.

Consider the case of merging N random circuits, each with an intrinsic Rent exponent of p and a number of components C_n for $n \in 1..N$. As in the prior section, equation 3.14 can be restated using definitions for $C_{programmable}$ and $p_{programmable}$ from equations 3.9 and 3.8 to arrive at the following bound:

$$W_{programmable} \leq \alpha k(\max_n C_n)(1 + \alpha\beta k(N-1)) \left[1 - \left((\max_n C_n)(1 + \alpha\beta k(N-1))\right)^{(\max_n p_n)-1}\right]. \tag{3.15}$$

For a better intuition, again consider the case where every functionality has

the same number of components ($C_n = C_{fixed}$), same Rent exponent ($p_n = p_{fixed}$), and has an average wirelength exactly as predicted by equation 2.31. Then

$$\frac{W_{programmable}}{W_{fixed}} \leq \frac{\alpha k C_{fixed}(1+\alpha\beta k(N-1))\left[1-(C_{fixed}(1+\alpha\beta k(N-1)))^{(p-1)}\right]}{\alpha k C_{fixed}(1-C_{fixed}^{p_{fixed}-1})}. \tag{3.16}$$

Or following algebraic simplification,

$$\frac{W_{programmable}}{W_{fixed}} \leq \frac{(1+\alpha\beta k)\left[1-(C_{fixed}(1+\alpha\beta k(N-1)))^{(p_{fixed}-1)}\right]}{\left[1-C_{fixed}^{p_{fixed}-1}\right]}. \tag{3.17}$$

In the limit of large C_{fixed}, this simplifies to

$$\frac{W_{programmable}}{W_{fixed}} \leq 1+\alpha\beta k(N-1). \tag{3.18}$$

Moreover, emperically this remains a fairly good approximation, typically better than 50% for reasonable values of α, β, and k. From this equation it can be seen that the total number of wires in a circuit grows asymptotically linearly with the number of distinct functionalities implemented. Note however, that even in the pessimistic case, this does not create atypical wiring demands, as the number of components in the resulting circuit grows linearly as well due to added multiplexers. Careful examination of equation 3.14 shows that the total number of wires is expected to grow linearly with the number of components in the limit of large numbers of components. That is, the resulting programmable circuit should not be expected to be unusually difficult to place and route.

Heterogeneous Component Case

As a part of the resource sharing manipulations, components from each circuit will be bound together. Theorem 3.4.3, targeted at circuits with homogenous components, selects this binding without regard to the functionality of the components. Instead, component bindings are selected with regard to satisfying the bound on the Rent's rule. It is important to note that such a binding is still feasible, as Theorem 2.1.2 provides that it is always possible to create a multiplexer-based circuit that implements any set of functionalities. Figure 3.12 shows how such a circuit might look. Note that in this figure, adders have only been matched with adders, and multipliers with multipliers. However, the inserted multiplexers network is such that this need not be true to maintain correct function of the circuit.

Therefore even in the heterogeneous case, it is possible to obtain a programmable circuit with Rent exponent bounded similarly to Theorem 3.4.3. However, such a *functionality-agnostic* binding is likely to result in binding of components with differing functionalities, and there may be a potentially high overhead for binding them together using the provisions of Theorem 2.1.2. This style of binding may be advantageous if wires are comparatively more expensive than components. Moreover, in the opposite limit where components are significantly more expensive than wires, it is preferable to select a binding that minimizes component costs, and the wiring length to do so is of negligible importance. As a result, in both extrema of the wire/component cost balance it is not necessary to explicitly handle the heterogeneous component case.

While a full treatment of the heterogeneous component case is left as future work[7], note that the derivation of Theorem 3.4.3 remains valid if the partitions at

[7] This is partially because the author's available experimental test cases fall in the design region of expensive components with comparatively inexpensive wires, making experimental verification

each level are made to have balanced numbers of each type of component. Thus, it would seem plausible that merging circuits composed of small numbers of distinct component types may yield results similar in form to the homogeneous case. Moreover, this type of partitioning constraint would seem ideally suited for leveraging multi-objective partitioning strategies familiar to the EDA community [51, 74]. To capture this distinction for the purposes of reasoning about the asymptotic features of the design space, a separate Rent exponent parameter $p^{\star}$ is proposed.

Definition $p^{\star}$ is defined as the Rent exponent that arises from the parallel hierarchical decompositions of a set of circuits restricted to have at level of decomposition, balanced numbers of each component type across each of the parallel partitions. That is, it is the minimal possible effective Rent parameter of the multipexer-based programmable circuit capable of implementing the functionalities of each of the circuits in the set.

If we reasonably assume that $p^{\star} \geq p$, then we can compute an additional wiring overhead due to the heterogeneous component types. Based upon equation 2.31, the following ratio on average wirelength can be found for the common case of $p > 0.5$:

$$\frac{\overline{R}_{heterogeneous}}{\overline{R}_{homogeneous}} = \left[C_{programmable}\right]^{(p^{\star}-p)} . \tag{3.19}$$

3.4.3 Shared Component Inefficiencies

The third condition of Theorem 3.1.2 states that for a resource sharing manipulation to be functionality-preserving, "the resulting shared component has functionality equal to or greater than the supremum of the set of resources to be shared."

difficult.

This condition implies the existence of overhead due to the shared components that must implement multiple functionalities. Fortunately, for any given shared component, an upper bound on this overhead can be established by invoking the side-by-side construction technique of Figure 2.2. In the case of a restricted component library, this upper bound may fact represent the only valid construction.

3.5 Design Space Regions

In the prior section, the overheads induced in a multiplexer-based programmable interconnect were decomposed into three contributory sources based upon the conditions in Theorem 3.1.2. Concisely, these three overheads can be attributed to multiplexers, wires, and components. In this section, this decomposition will be used to sketch out the gross regions of the design space based on the relative contributions of the various components to the overall cost of a programmable circuit.

These design space regions can be visualized in Figure 3.14. In this visualization, the axes are total multiplexer and wire costs normalized to total functional unit costs. For this figure, costs here are deliberately left independent of any particular cost metric (e.g. delay, area, energy, and leakage), as the concept applies generally to any cost metric.

In is important to keep in mind that the design space of Figure 3.14 is composed of points, each representing a specific programmable implementation. However, the region containing Pareto optimal design points may be limited. For instance, given a set of circuits composed of components (e.g. microprocessor cores) substantially more costly than either wires or multiplexers, Pareto optimal design points may be restricted to the component-dominated region of the design space. Restrictions may also come from a specific physical realization technology. For

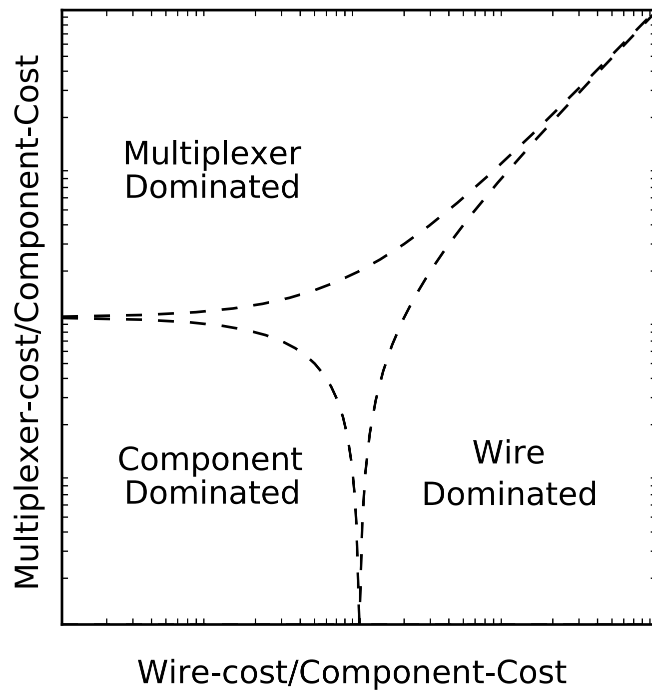

Figure 3.14: Graphical representation of proposed design-space regions for multiplexer-based programmable circuits.

instance, a technology with switching components substantially more costly than wires (e.g. super-micrometer CMOS VLSI) may not have Pareto optimal design points in the wire-dominated region.

Such restrictions are useful because they can serve to guide optimization strategies. The general co-optimization problem is NP-hard, as it can be shown to reduce to subgraph isomorphism. Thus, simplification of the problem is of clear computational benefit. If a particular application is known to have Pareto optimal solutions restricted to one (or more) design-space regions, appropriate simplified optimization strategies can be employed that strategically neglect aspects of the design. Each of these three design regions will now be examined, and for each:

- a targeted optimization strategy to optimizate the dominenant cost will be suggested, and
- closed-form guidelines for when a design will reside in a particular region will be provided.

Compiling the various individual cost estimates from prior sections, one can summarize expected costs of a multiplexer-based programmable circuit as the sum of:

- the component cost portion of the fixed-function equivalent circuit,
- the costs of added multiplexer components, and
- the wire cost fixed-function equivalent circuit times the ratio of programmable wire costs to fixed-function wire costs.

Translating this into symbolic form for delay and energy costs yields

$$
\begin{aligned}
delay(programmable) = & delay(fixed_{components}) \\
& + n_{cp} ceil(log_2(N)) delay(MUX2) \\
& + delay(fixed_{wires}) \frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}
\end{aligned} \tag{3.20}
$$

and

$$
\begin{aligned}
energy(programmable) = & energy(fixed_{components}) \\
& + M_{programmable} energy(MUX2) \gamma \\
& + energy(fixed_{wires}) \gamma \frac{W_{programmable}}{W_{fixed}} \frac{\overline{R}_{programmable}}{\overline{R}_{fixed}},
\end{aligned} \tag{3.21}
$$

where γ is the circuit-dependent fraction of dynamic energy dissipated as a result of efficiencies gained from operand isolation.

3.5.1 Guideline Criteria

Consider now the relative magnitudes of the types of costs composing equations 3.20 and 3.21. The costs for the delay metric will be considered first in detail, followed by analogous results for the energy metrics. A similiar derivation will then be made for the area metric.

Delay

Because there are three types of costs, and three regions, there are three comparisons to be made. First, compare the fixed-function component cost to the cost of

added multiplexer components, testing for the dominance of fixed-function component costs:

$$delay(fixed_{components}) \overset{?}{>} n_{cp}ceil(log_2(N))delay(MUX2)^8. \tag{3.22}$$

Rearranging this equation by dividing both sides by $n_{cp}delay(MUX2)$ yields the following equivalent comparision:

$$\frac{delay(fixed_{components})}{n_{cp}delay(MUX2)} \overset{?}{>} ceil(log_2(N)). \tag{3.23}$$

Note that the $delay(fixed_{components})/n_{cp}$ term is the average delay of components on the critical path, and dividing by $delay(MUX2)$ effectively states this average delay in units of a 2-input multiplexer delay. The equivalent comparison is then whether the average component delay along the critical path in units of a 2-input multiplexer is greater or less than $log_2(N)$, where N is the number of explicit functionalities in the programmable circuit. If the result of this comparison is much greater, then fixed-component costs dominate multiplexer costs. If it is much less, then multiplexer costs will tend to dominate fixed-component costs. Thus this equation predicts a bound on the number of merged components before multiplexer costs become dominant over useful component costs.

Second, consider a test for determining if fixed-function component cost dominates the cost of wiring:

$$delay(fixed_{components}) \overset{?}{>} delay(fixed_{wires})\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.24}$$

[8] Throughout this section, the symbol $\overset{?}{>}$ will be used to denote a magnitude comparison test between two costs. Equal magnitude costs, and therefore the location of the asymptotic region boundary, would be indicated be equality of the sides of the equation.

Rearranging this equation yields

$$\frac{delay(fixed_{components})}{delay(fixed_{wires})} \overset{?}{>} \frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.25}$$

Recall equation 3.12, which bound on $\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}$ as:

$$\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}} \leq \sqrt{1+k(N-1)}.$$

Substitution of this bound, followed by algebraic manipulation yields the bound

$$\left[\frac{delay(fixed_{components})}{delay(fixed_{wires})}\right]^2 - 1 \overset{?}{>} k(N-1). \tag{3.26}$$

For better intuition, consider the case for asymptotically large N, and an average of three terminals per component ($k = 3$) typical for synthesized RTL with many two-operand components:

$$\left[\frac{delay(fixed_{components})}{delay(fixed_{wires})}\right]^2 \overset{?}{>} 3N. \tag{3.27}$$

Or equivalently in prose, component costs will be dominant over wiring costs if the square of the ratio of component delay to wiring delay in the fixed-function circuits is greater than three times the number of explicit functionalities present in the circuit.

Finally, consider the case of comparing the cost of added multiplexers to the costs of wiring:

$$n_{cp}ceil(log_2(N))delay(MUX2) \overset{?}{>} delay(fixed_{wires})\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.28}$$

Rearranging this equation by dividing both sides by $n_{cp}delay(MUX2)$ yields the following equivalent comparison:

$$ceil(log_2(N)) \overset{?}{>} \frac{delay(fixed_{wires})}{n_{cp}delay(MUX2)}\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.29}$$

Substituting the bound on $\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}$ from equation 3.12 yields

$$ceil(log_2(N)) \overset{?}{>} \frac{delay(fixed_{wires})}{n_{cp}delay(MUX2)}\sqrt{1+k(N-1)}. \tag{3.30}$$

Grouping N-dependent terms on a single side, yields

$$\frac{ceil(log_2(N))}{\sqrt{1+k(N-1)}} \overset{?}{>} \frac{delay(fixed_{wires})}{n_{cp}delay(MUX2)}. \tag{3.31}$$

Unfortunately, as the left-hand side is not monotonic with N, this does not yield a simple bound as do equations 3.23 and 3.26.

Energy

Similar derviations for energy yields a test for the dominance of fixed-function component costs over added multiplexers:

$$\frac{energy(fixed_{components})}{Cenergy(MUX2)} \overset{?}{>} \alpha k\beta(N-1)\gamma. \tag{3.32}$$

Now assume several bounds guaranteed by definition, specifically that the number of input terminals is equal or less than the total number of terminals ($\alpha \leq 1$),

the fraction of multiplexers inserted is less than one ($\beta \leq 1$), and no gains from operand isolation ($\gamma = 1$) techniques. The equation then simplifies to a bound related to the minimum extent of the component-dominated region:

$$\frac{energy(fixed_{components})}{Cenergy(MUX2)} \overset{?}{>} k(N-1). \tag{3.33}$$

In prose, this equation suggests that the fixed-function component energy is dominant if the average energy per component in units of energy dissipated by a *MUX2* component exceeds the product of the averge number of terminals per component and the number of designs implemented less one.

A similar derviations yields a test for the dominance of fixed-function component costs over the cost of wiring:

$$\frac{energy(fixed_{components})}{energy(fixed_{wires})} \overset{?}{>} \gamma \frac{W_{programmable}}{W_{fixed}} \frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.34}$$

By incoporating definitions for $\frac{W_{programmable}}{W_{fixed}}$ and $\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}$ from equations 3.12,3.18 and the bounds described earlier ($\alpha \leq 1$, $\beta \leq 1$, and $\gamma = 1$), one arrives at a second bound related to the minimum extent of the component-dominated region

$$\frac{energy(fixed_{components})}{energy(fixed_{wires})} \overset{?}{>} [1 + k(N-1)]^{3/2}. \tag{3.35}$$

Consider the case of comparing the cost of added multiplexers to the costs of wiring:

$$M_{programmable} energy(MUX2)\gamma \overset{?}{>} energy(fixed_{wires})\gamma \frac{W_{programmable}}{W_{fixed}} \frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}. \tag{3.36}$$

By incoporating definitions for $\frac{W_{programmable}}{W_{fixed}}$ and $\frac{\overline{R}_{programmable}}{\overline{R}_{fixed}}$ from equations 3.12,3.18 and the bounds described earlier ($\alpha \leq 1$, $\beta \leq 1$, and $\gamma = 1$), one arrives at an equation of the form

$$kC(N-1)\left[1+k(N-1)\right]^{2/3} \overset{?}{>} \frac{energy(fixed_{wires})}{energy(MUX2)}. \tag{3.37}$$

Much as for delay, this bound is not as a convenient a form as the bounds in equations 3.33 and 3.35.

Area

In section 2.2.3 the area metric is not defined to map onto the set of positive reals. Thus, simple relationships of the form previously found for delay and energy will not be presented for area. However, if one restricts attention to a single component of area, for instance the active area as might be reported in a standard cell library, and ignores wiring area, simple relations can be expressed.

Within this context, the area used by fixed-function components can be compared to that used by multiplexers and the following test for dominance derived:

$$area(fixed_{components}) \overset{?}{>} area(MUX2)M_{programmable}. \tag{3.38}$$

Substituting the definition of $M_{programmable}$ from equation 3.1 and rearranging terms yields

$$\frac{area(fixed_{components})}{area(MUX2)} \overset{?}{>} \alpha k \beta C(N-1). \tag{3.39}$$

Now assume several bounds guaranteed by definition, specifically that the number of input terminals is equal or less than the total number of terminals ($\alpha \leq 1$),

and the fraction of multiplexers inserted is less than one ($\beta \leq 1$), yielding the simplied relation

$$\frac{area(fixed_{components})}{Carea(MUX2)} \overset{?}{>} k(N-1). \tag{3.40}$$

In prose, this implies that the fixed function component area is dominant if the average fixed-function component area expressed in units of *MUX2* components is greater than the product of the average number of terminals times one less than the number of explicit functionalities. Thus this equation establishes a bound for the number of functionalies that can be implemented surely within the component-dominated region.

3.5.2 Region Discussions

Because equations 3.23, 3.26, 3.31, 3.35, 3.33, 3.37, and 3.40 depend solely on high-level parameters, they are capable of providing guidance early in the design process as to which design-space region the resulting design is likely to reside, thereby guiding optimization. For example, if the comparisons in equations 3.23 and 3.26 are true, then the design is likely to reside in the component-dominated design-space region. The remainder of this section considers scaling behavior and optimization within each of the three design-space regions.

Component Dominated

Equations 3.23, 3.26, 3.35, 3.33, and 3.40 establish a minimum size for the component-dominated region based solely upon high-level parameters. For example, consider a 90nm standard cell library with characteristics shown in Table 3.1. Using this library, a 32-bit adder and 16x16-bit multiplier, as might be found in a typical Dig-

Cell	MUX2 Cell	Full Adder Cell	32-bit Adder	16x16 multiplier	
Delay	31	60 (A/B to CO)	2100	2500	ps
		32 (CI to CO)			ps
		33 (CI to S)			ps
Energy	15	31	3100	5800	fJ
Area	7.1	19	1800	4200	μm^2
Leakage	1900	7600	24000	69000	nW

Table 3.1: Assumed 90nm standard-cell performance characteristics.

ital Signal processing application, were synthesized with the costs also shown in Table 3.1. Further assume that for an (unbuffered) wire of length ℓ delay can be approximated as $\left(\frac{2.08fs}{\mu m^2}\right)\ell^2$, and switching energy dissipation can be modeled as $(0.11fJ/\mu m)\,\ell$[9]. For clarity, all numeric values are treated as having 2 significant digits.

To gain intution, several simplifying assumptions are made:

- the fixed-function component has costs approximated by those of a 32-bit adder,
- the average number of terminals (k) per component is 3, and
- the average wirelength in the critical path is approximately equal to the square root of the area of a 32-bit adder times the number of components (C).

From the last assumption and Table 3.1, one can approximate the average wirelength on the critical path as $\sqrt{1800C}$ in units of μm, with a corresponding modeled delay of $3.7C$ ps, and switching energy of $4.7\sqrt{C}$ fJ. In this context, consider now the limits on the number of functionalities (N) within the component-dominated region predicted by equations 3.23, 3.26, 3.35, 3.33, and 3.40.

[9] These wire performance values are derived from parameters provided by Ho, et al. for mid-layer wires [47].

In the context of equation 3.23, the ratio of delay between the 32-bit adder component and *MUX2* of 67.7 suggests that for up to 2^{67} functionalities (N), the fixed-function component delay will be greater than the added multiplexers. In the context of equation 3.26, for the number of programmable functionalities (N), the fixed-function component delay will be greater than the added wiring up to $\frac{10^5}{C^2}$. For a design with 10 components, this represents a 1000 functionalities bound, while for 100 components, this represents up to 10 functionalities.

In the context of equation 3.33, the ratio of dynamic energy from the 32-bit adder to a 32-bit multiplexer is approximately 6.5, suggesting that up to $N = 3$ functionalities will be in the component-dominated region for energy. Equation 3.35 predicts up to $N = 25C^{-1/3}$ will be in the component-dominated region for energy. For a design with 10 components, this represents a bound of $N = 12$, while for 100 components, the predicted bound is $N = 6$. However, as will be studied experimentally, this conservative analysis neglects the substantial benefits of operand isolation that are captured by the γ parameter.

Similarly, equation 3.39 predicts that up to $N = 5$ functionalities will be located within the component-dominated region for area.

In the component-dominated region, the costs of computing components is dominant when compared to both wiring and multiplexer costs. Thus it is advantageous to minimize the number of components instantiated, which implies satisfying the conditions of *maximal resource sharing*. Beyond this, optimizing the insertion of multiplexers and wiring is likely to have little impact on the overall performance of the circuit. Therefore within the component-dominated region, one might say that added functionality is comparatively inexpensive.

Multiplexer Dominated

In the multiplexer-dominated region the cost of the multiplexers dominates the implementation cost, and as a result insertion of multiplexers should be minimized. Optimization criteria is the subject of the Datapath Merging Problem of high-level synthesis, and is well-addressed by existing academic literature. A full discussion and summary of prior work is provided in section 5.3.

Wire Dominated

In the wire-dominated region, the cost of wires dominates the implementation cost. As a result, optimization should focus on minimizing the resulting wiring complexity.

Desirable optimization strategies for this region include those deriving from recursive bisection strategy followed in the proof of theorem 3.4.3. A case study of this type of strategy is provided in section 4.1.2. Other than those provided within this paper, binding strategies for this region are ill-addressed in current academic literature for on-chip networks, but are well studied within networking domain.

3.6 Summary

In this chapter, a model of programmability based on the assembly of fixed-function components with multiplexer "glue" was proposed and rigorously defined. Within this model, design-points of theoretical interest were identified and defined. Transformations for traversing within the design-space of programmable implementations were defined along with an examination of the properties preserved and lost through the various transformations. The programmability problem itself was refined to have both *a priori* and *a posteriori* cases, and qualitative evaluation metrics for reasoning about the amount of functionality provided was provided for both.

Further, the implementation cost for circuits created within this programmability model were examined, and decomposed according to the need to meet each of three criteria for a functionality-preserving transformation. The relative cost of these three components were used to define design-space regions. Guidance was provided to predict which of the three design-space regions identified a design was likely to fall within. For each region, discussion as to relative amount of overhead present and the best optimization strategies were suggested.

Chapter 4

Experimental Validation

4.1 Merging Design Tool

One of the key claims underlying this effort is that applications implemented in traditional programmable devices incur a tremendous overhead relative to those implemented in ASIC devices. Based upon this assertion, it is hypothesized that one can automatically introduce programmability into fixed-function devices, thereby yielding devices that incur substantially less overhead than fully programmable devices and yet maintain a limited, but still useful, degree of programmability. The validity of this hypothesis can be demonstrated by existence proof, and therefore, this effort will include the creation of a tool that will accept a set of data-flows as input and produce a programmable architecture capable of implementing those flows as output. This tool is known as the Reconfigurable Architecture Synthesis Tool (RAST). The input of the RAST is a set of netlists while the output will be a Verilog module that is capable of implementing each of the input netlists' functionality when provided with the correct configuration bits.

4.1.1 Overall Flow

The tool functions by reading RTL (in this case, Verilog) representations of the input designs. The designs are synthesized into technology independent macro-models using a commercially available synthesis tool, in this case Synopsys™ DesignCompiler™. The macro-model netlist is then read into a custom tool and several local optimizations are performed. Notably, bit-sliced components are aggregated into single components. Wire for individual bits are clustered into buses.

The resulting netlists are merged by our custom tool that identifies common cell instantiations between the input netlists and outputs a single netlist that shares these components. Appropriate multiplexers are inserted at the input of shared

cells, such that the inter-cell connectivity for any of the input netlists can be implemented by setting the multiplexer select lines appropriately. The select lines to the multiplexers are exposed at the interface of the generated circuit such that the functionality of any of the input circuits can be configured. A simple binary selection scheme can be used so that for N input designs, $ceil(log_2(N))$ configuration bits are required.

The technique presented here is a netlist-based approach in which, when configured properly, the output circuit behaves identically in each cycle to the selected input circuit. That is, no retiming or scheduling flexibility is allowed of the tool. As a result, for each type of component in the input circuits, the output circuit must contain at least the largest number of components found in any one of the input netlists.

To reduce dynamic power, operand isolation gating circuitry is inserted at the boundary between circuitry that will be active for a particular configuration and circuitry that will not be active for that particular configuration. As a result, inactive cells see constant operand inputs. In the studied cases, this approach adds an average of 3% in area to the combined circuit, but decreases dynamic energy consumption by 7%-65% (mean: 39%) depending on which circuit is selected. Power gating of inactive cells will be considered in future work, but is omitted in this paper. Eliminating power gating enables theoretical reconfiguration times of a single cycle, but are practically limited to the pipeline length of the circuit reconfigured.

Following the merging of the input netlists, the resulting reconfigurable netlist is exported to Verilog. As a result, the merging tool is "Verilog-in/Verilog-out", such that the merged netlist can be incorporated into a larger overall design. This gives the design engineer the ability to explicitly control the reconfigurabil-

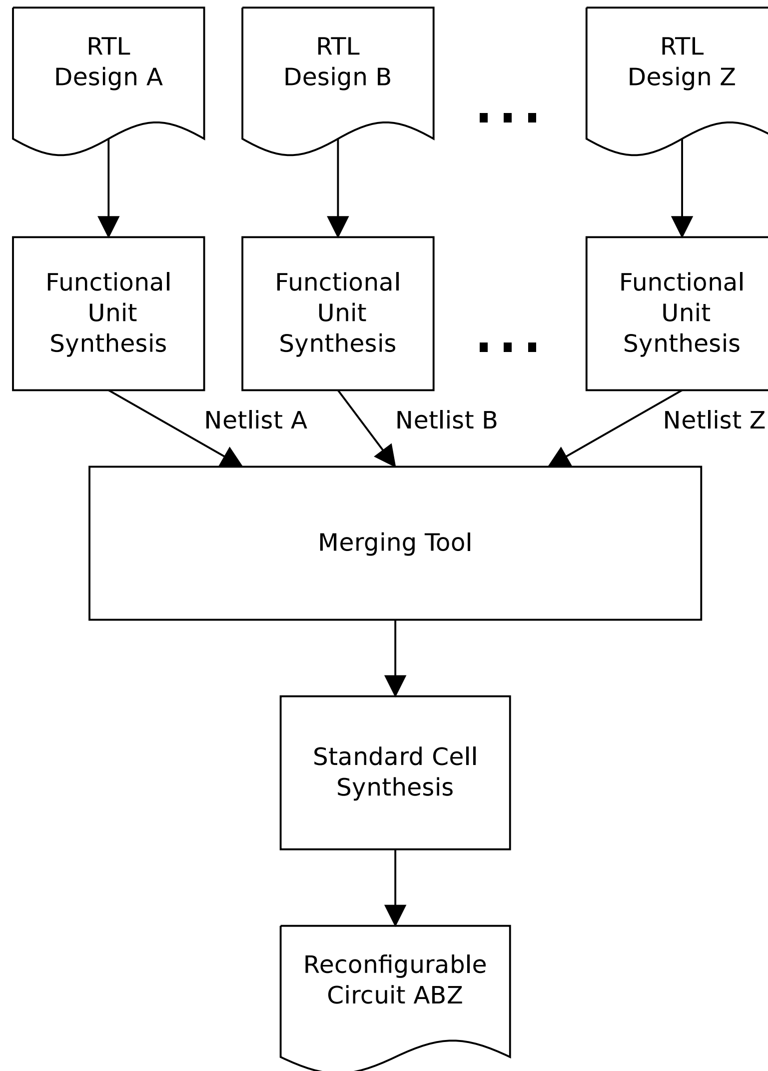

Figure 4.1: Conceptual design flow

ity present within the final design. One can envision that segments of the overall application be made reconfigurable, while others remain application-specific depending on design objectives. More importantly, the reconfiguration boundaries can be tailored to the specific application.

4.1.2 Optimization Strategies

Clearly the selection of components to merge (i.e. “binding”) forms a critical portion of the optimization problem for this design flow. Ongoing work has enumerated multiple reasonable optimization objectives for which the underlying optimization problem is often NP-hard. For instance, one may be concerned with area-minimization and thus require a “maximum-binding” objective. One may also consider a “minimum-multiplexer” binding that requires the addition of the fewest multiplexers. One may also wish to optimize delay for specific paths or input circuits at the expense of others. Alternately, one may consider minimizing various interconnect parameters of the generated circuit, such as total wirelength (or proxies thereof, such as Rent exponents [67]).

However, it is important to note that the substantial energy and delay improvements of the proposed reconfigurable ASIC technique relative to more general purpose reconfigurable circuitry (e.g FPGAs) are not dependent on solving the optimization problem. Instead, the improvements flow directly from providing only the reconfigurablity required by the application.

As discussed in section 3.5, the optimal binding problem is computationally intractable in practice but, for many cases, the computational burden can be eased by selecting a strategy based on the design-space region the solution is located in. The RAST tool allows binding strategies to be plugged-in, and several strategies

have been implemented and for conciseness are identified by a mnemonic:

- *mg_null* - a side-by-side realization with no binding,
- *mg_bmh* - a bipartite matching heuristic scheme intended to be effective within the multiplexer-dominated region,
- *mg_greedy* - a simple greedy scheme, and
- *mg_random* - a random scheme for comparative purposes.

4.1.3 Noteworthy Complexities

Beyond achievement of any explicitly stated goals, an implicit contribution in any research effort is the identification and charting the inevitable unexpected difficulties, and recording the arbitrary decisions made in the resolution of those difficulties. Such documentation often serves a cartographic function for later researchers, facilitating comparisons with alternative approaches, and suggesting avenues for future research. This effort is no exception, and several minor complexities worthy of brief discussion will now be discussed.

Partial-port netlists

In many highly optimized signal processing circuits, extensive use is made of bit-level manipulations. Modeling this in the context of coarse-grained reconfigurable logic operating on many-bit words is possible in multiple ways. One method, common in software-rooted approaches, is to model the bit-level manipulations (e.g. bit-wise shift) as distinct operations in the dataflow graph. For computations to be synthesized as software, this makes a great deal of sense, as these distinct operations

are ultimately realized as distinct instructions, each with a distinct cost in execution time and scheduling restrictions.

However, for standard-cell level netlists, this model is not as appropriate. To first order bit-shift operations are effectively cost-free to realize, as connections are typically made solely at the bit-level without regard to word-level alignments. As a result, distinct bit-level manipulation nodes in the dataflow graph work are obstacles to extracting the true costs of connectivity in the graph. In the case of generating multiplexer-based interconnects, it is desirable to avoid this masking. To handle this case in the RAST, the named ports of each component are further split into buses by connectivity. For example, given a 16-bit port "A" on a component in which bits 0-7 are routed to one set of destinations and bits 8-15 are routed to another set, the RAST tool will create two ports labeled "A[7:0]" and "A[15:8]". These two ports will then be treated as independent for further processing. In this manner, the RAST tool internal data structure design prioritizes connectivity over explicit identifiers extracted from the RTL source code.

Memories

Many signal processing circuits require the use of memory, often in two forms: read-only constants (e.g. filter coefficients) and writable temporary working space. Those in this case study are no exception. Much like other components, such memories benefit from resource sharing. Read-only constants from differing functionalities can be stored in disjoint areas of a single read-only memory (ROM), perhaps aligned on power-of-two boundaries to ease address signal generation. Temporary working areas that are disjointly active can similarly be stored in disjoint regions of a shared random access memory (RAM).

In a non-trivial subset of commonly used signal processing algorithms, the potential state transitions of the temporary working memory is self-stabilizing, in that proper output can be assured after a specified number of clock cycles regardless of the initial state of the temporary working memory. Two broad classes of commonly used applications have this characteristic: those with acyclic data flow graphs (e.g. the heterodyning circuit of Figure B.5), and those with cyclic but provably finite impulse response (e.g. the cascaded integrating comb filter of Figure B.4). In these cases, it is possible to share temporary working areas for functionalities that are disjointly active in the same regions of a random access memory.

Much as memories warrant specialized synthesis tools (i.e. memory compilers), memories are handled explicitly in the RAST tool. Ports that would be connected to internal memories are instead exposed as external ports. That is, in the current version of the RAST tool, memories are handled explicitly by the user. As each of the designs tested assumes at most a single RAM or ROM, for the test cases studied, it is assumed that a single external memory is available to be connected. The costs of such a memory are not considered extensively in this document, as the sharing memory between tasks is relatively straightforward and high-level estimation of memory costs are well developed [83].

Cycles in Output Netlists

Certain bindings can create cycles in the resulting netlist that have the potential to cause problems for a synthesis tool. For example, in Figure 3.2, there is a cycle involving nodes *Mul_2B* and *Add_3C* and two additional multiplexers. This has the potential to cause loops in static timing analysis. These loops are false timing paths, in that any useful configuration of the multiplexers avoids such a loop. However,

the falseness of such path may not be apparent to the synthesis tool and therefore may cause difficulties in both timing optimization and characterization.

In this case study, the RAST tool is capable of generating such loops. The difficulties are handled instead through careful control of the synthesis and static timing tools. As a part of its output for a merged design, the RAST tool outputs metadata for the configuration lines. This metadata is encoded as specially formatted Verilog comments that the synthesis TCL script parses. Using this information, appropriate "case-analysis" parameters are setup for the synthesis tool and static timing tool. These case analysis parameter convey to the synthesis tool that the configuration ports will be used in only a select logical configurations and that it should consider each value of the configuration line separately.

4.2 FLWR Case-Study

In order to validate the predictions of chapter 3, a case-study was undertaken using a Software Defined Radio (SDR) platform known as the Flexible Low-Power Wideband Receiver (FLWR). This section details the FLWR platform, its applications, and the intended benefit of applying the techniques of chapter 3 to the FLWR platform.

4.2.1 FLWR Overview

The University of Texas at Austin Applied Research Laboratories (ARL:UT) has developed a FPGA-centric Software Defined Radio (SDR) platform called the Flexible Low-Power Wideband Receiver (FLWR). As shown in Figure 4.2, the FLWR hardware includes a 100 megasample per second (MSPS) analog-to-digital (ADC) converter, a low-cost FPGA, 32 MB of SDRAM, and an USB 2.0 interface into a

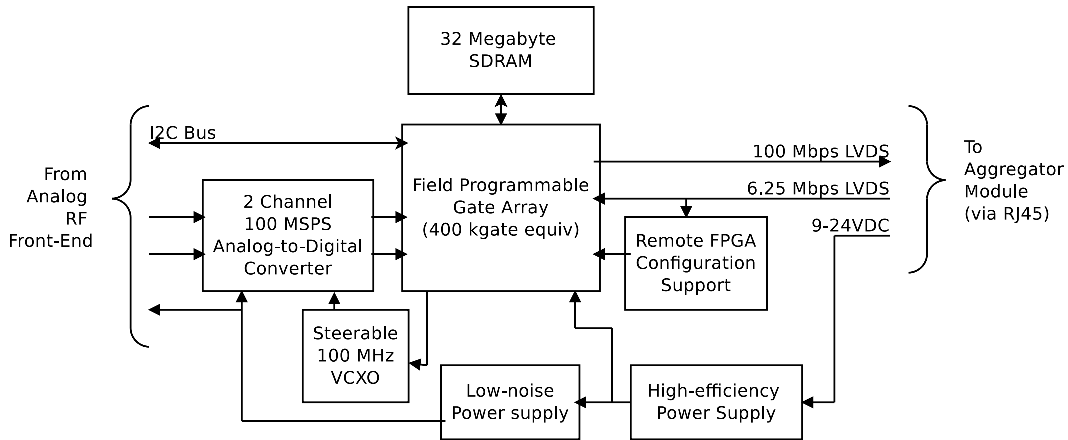

Figure 4.2: Block Diagram of the Flexible Low-Power Wideband Receiver

PC. The FLWR has been used in a variety of applications, including the Long Wavelength Demonstrator array (LWDA), a prototype of the Texas Ionospheric Ground Receiver (TIGR), and a lunar bistatic radar experiment. In one sample application, the FLWR collects 100 MHz of input RF bandwidth from the onboard ADC, tunes and filters approximately 12.5 MHz of this bandwidth, then utilizes a polyphase filter bank structure to produce 1024 independent frequency channels for further processing. While in this mode of operation, the FLWR performs nearly 1 billion multiply-accumulation operations per second, and the entire platform dissipates approximately 1.8 Watts of input power. A photograph of the hardware is shown in Figure 4.3.

Currently, applications for the FLWR are hand-coded in a subset of Verilog suitable for efficient implementation in the onboard FPGA. While this development methodology is suitable for a number of applications, interest has arisen in being able to rapidly reconfigure the Digital Signal Processing code running in the FPGA from a stand-alone low-power embedded CPU. While an embedded CPU is adequate for downloading bit-streams into the FPGA, it is not sufficiently capable of executing the Computer Aided Design (CAD) tools needed to generate new FPGA bit-streams

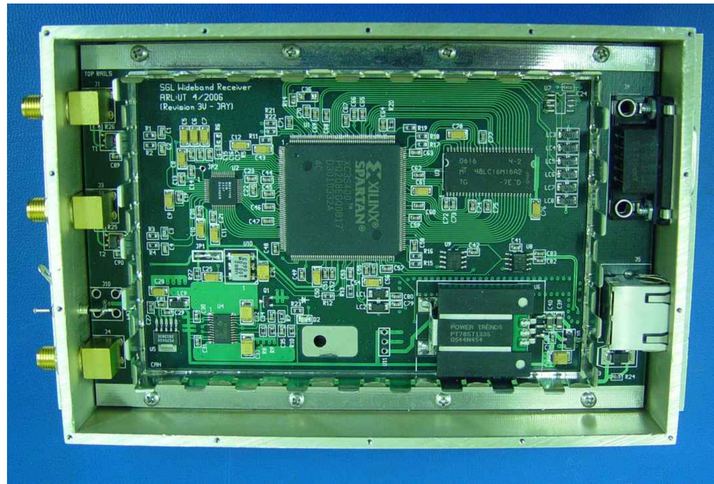

Figure 4.3: Image of the Flexible Low-Power Wideband Receiver

within reasonable time and energy budgets. This is due to the computational complexity of the FPGA mapping/placement/routing problem, which is NP-hard even for simplified routing architectures in the absence of timing constraints [61].

However, the FLWR is only intended to operate over a limited set of algorithms within a limited application domain and therefore the reconfiguration abilities required by the FLWR are far less general than that of an FPGA. For example, it is desirable for a FLWR to have multiple identical, independently-reconfigurable channels. Each of these should capable of performing several functions in a time-multiplexed manner, such as: 1) Spectral estimation via FFT followed energy detection, 2) Broadband digital down conversion for signal identification, and 3) Narrow-band digital down conversion followed by demodulation.

One might envision each of the parallel channels initially in the spectral estimation mode looking for the emergence of received RF energy. This would be followed by a transition to a broadband digital down conversion mode in order to identify the type of signal detected. Afterwards, a transition to a narrow-band

digital down conversion mode would allow recovery of the data from the signal detected. Each of the operating modes uses a significant fraction of the FPGA resources such that they cannot be implemented side-by-side in an area-efficient manner. However, as is common in the signal processing problem domain, many of the operators within each mode (e.g. adders, multipliers) are common between the modes allowing the proposed merging tool is able to efficiently share resources between circuitry implementing each mode.

4.2.2 Application Details

To demonstrate this principle, five algorithms commonly implemented on the FLWR platform were identified:

- coordinate rotation digital computer (CORDIC) sine/cosine generator,
- complex multiplication heterodyning stage (hetero),
- Cascaded Integrating Comb (CIC) decimating filter,
- Finite Impulse Response (FIR) filter, and
- Fast Fourier Transform (FFT) butterfly.

Verilog implementations of these algorithms have been extracted from working FLWR-based systems and processed through the flow outlined in Section 4.1.1. The RTL for these designs, originally targeted at the Xilinx synthesis tool XST, were trivially augmented with explicit reset logic in order to synthesize under the Synopsys™Design Compiler suite, but are otherwise unmodified.

4.3 ASIC Implementation

To evaluate the suitability of this approach for an ASIC design-flow, netlists are imported to Design Compiler, and synthesized and mapped into a 90nm standard-cell library. Critical path delay, dynamic energy dissipation, static power dissipation and area are estimated based on Design Compiler's "topographic mode". While in topographic mode, interconnect parasitics are estimated based upon a global placement and process parameters, although a legalized placement and detailed routing are not performed. Since the proposed technique is an architectural design exploration technique, the relative fidelity of metrics derived from a global placement were deemed sufficient for this initial exploration.

4.3.1 Input Circuit Baselines

To provide a baseline from which performance on the generated circuits can be compared, the RTL for each circuit were synthesized. In the initial synthesis run, each circuit is synthesized with an effectively infinite delay constraint. It is important to note that the optimal synthesis is an NP-hard problem and that RTL synthesis tools often use potentially unstable heuristic algorithms during optimization stages. Without constraint, the use of potentially unstable optimizations (e.g. simulated annealing) within the synthesis tool is minimized and these tools generally produce stable, repeatable results. Thus, unconstrained results are useful for verifying expected behaviors separately from optimization effects and will therefore be discussed first.

Because the RAST tool must parse the RTL netlist output by the synthesizer, and then output RTL to be read by the synthesizer, a useful test is to feed the RTL for a single design through the RAST tool and then synthesize the resulting

output. If everything works correctly, the synthesis results should be approximately the same. Table 4.3.1 shows the results of both the original RTL (*source* column) and the RAST-generated output (*vcv* column). For each of the five designs described in section 4.2.2, and the *mcu* design to be described in section 4.6, four costs are considered: delay, energy, leakage, and area. These are reported in units of picoseconds, picoJoules, nanoWatts, and square millimeters. In the tables, these values are scaled by a power of ten selected to yield 2-3 significant digits for easy visual inspection. The selected scaling value is reported in the table and, within a single cost metric, is consistent across all designs to facilitate comparison.

The values are approximately the same in most cases, and simulation using a suite of test-vectors yielded no discrepancies between the RAST-generated and source RTL. The lone outlier is the energy for the CORDIC_serial design, the cause of which remains elusive after investigation. The CORDIC algorithm performs extensive bit-level manipulations (e.g. shifts and adds), and it is suspected the processing of the netlist affected the heuristics used by the synthesis tool to derive placement constraints for datapath components. That the simultation results and area values remain effectively unchanged is consistent with this hypothesis. Note that such an aberration does not directly bias later results, as future comparisons will be made to the original RTL values, not the VCV.

4.3.2 Output Circuit Unconstrained Synthesis

Initially each evaluated circuit is synthesized without critical-path constraints. Because the synthesizer is unconstrained in this case, detailed cell-level logic optimization is not performed. As a result, unlike more realistic constrained scenarios examined in following sections, the unconstrained results tend to be stable and

delay ($\times 10^2 ps$)	source	vcv
CIC	24	24
CORDIC_serial	28	30
fftbutterfly32	29	29
fir32	28	28
hetero	25	25
mcu	33	31
energy ($\times 10^{-1} pJ$)	source	vcv
CIC	105	102
CORDIC_serial	65	75
fftbutterfly32	370	369
fir32	208	208
hetero	346	343
mcu	29	32
leakage ($\times 10^4 nW$)	source	vcv
CIC	11	11
CORDIC_serial	15	16
fftbutterfly32	32	32
fir32	19	19
hetero	29	29
mcu	12	12
area ($\times 10^{-4} mm^2$)	source	vcv
CIC	76	76
CORDIC_serial	97	96
fftbutterfly32	178	178
fir32	113	113
hetero	170	170
mcu	89	90

Table 4.1: Unconstrained synthesis results for original source (*source* column) and the RAST-generated passthrough output (*vcv* column)

predictable, and are therefore more comparable and interpretable.

In the results to follow, the RAST tool generates a programmble circuit capable of implementing the functionality of each of the five target circuit detailed in section 4.2.2. The costs of the resulting programmable circuit are then estimated through static analysis and compared to the orginal fixed-function equivalent. The

static analysis cost estimation process is repeated for each of the five target circuits with the circuit control lines driven to the appropriate state to implement the functionality of the selected target circuit. The unconstrained results for the five target circuits detailed in section 4.2.2 are shown in Table 4.2.

As shown in Table 4.2, the generated multiple personality circuit is smaller than the two largest input designs (hetero & fft), and represents a 1.9X reduction in area over implementing all five designs side-by-side. In effect, implementing the three smaller designs is free. Similarly, leakage power is smaller than the sum of the largest two designs (hetero & fft) and represents a 2.3X reduction relative to implementing all 5 designs side-by-side. Critical path delay varies from 0-24% worse (mean: 12%) than a side-by-side implementation, depending on which circuit is configured. Dynamic energy varies from 8.5-40.5% (mean: 21%) worse than an side-by-side implementation, depending on which circuit is configured. The 40.5% outlier in dynamic energy (CORDIC) suffered from having its many 16-bit adders promoted to 32-bit adders in order to minimize area when merged with the 32-bit adders prevalent in other designs.

To provide context for the area metric, the individual components constituting the input designs were synthesized and the resulting area of each measured. Because a multiple personality circuit must have at least as many instances of each type as the maximum number in any input circuit, one can establish a minimal component constitution given a set of input circuits. The areas of this minimal component constitution were summed to estimate a lower area bound for merged circuitry. Note that this does not include any required multiplexers, nor inefficiencies due to placement/routing concerns and thus represents an loose, unobtainable lower bound. For the five sample designs studied, such an unobtainable lower bound

of $260 \times 10^{-4} mm^2$, or approximately 11% below a solution with the random binding. Inverting the ratio and comparing to the highest area of five random binding trials, it can be concluded that in the studied case, even a random binding strategy repeatedly achieves within 13% of this lower bound. While not initially intuitive, this result is consistent with the analysis of section 3.5.2. Only the minimum of programmable interconnect required by the targeted circuits are inserted, and the total area of the multiplexers required is small relative to the required functional components regardless of binding choices.

4.3.3 Pareto Synthesis Flow

The post-merge synthesis flow outlined in section 4.1.1 is subject to bias induced by the synthesis constraints. While Table 4.2 lists the resulting performance constraints for the canonical unconstrained design-point, they are not realistic. To explore the impact of delay synthesis constraints, each synthesized design is repeatedly resynthesized for a variety of critical path delay constraints. An automated process was used in which the design is initially synthesized unconstrained, followed by a resynthesis using an unachievable (10ps in 90nm) critical path delay constraint. With the upper and lower bounds established, the automated process proceeds to resynthesize at logarithmically spaced intervals between the bounds. At each synthesis step, the performance measures discussed in section 2.2 are estimated and recorded, including:

- achieved critical path delay,
- dynamic energy utilization per cycle,
- static power dissipation per unit time, and

delay ($\times 10^2 ps$)	source	mg_null	mg_random	mg_greedy	mg_bmh
CIC	24	25	25	26	26
CORDIC_serial	28	32	34	34	33
fftbutterfly32	29	30	37	36	37
fir32	28	30	36	37	36
hetero	25	28	30	29	29
energy ($\times 10^{-1} pJ$)	source	mg_null	mg_random	mg_greedy	mg_bmh
CIC	105	110	137	147	143
CORDIC_serial	65	84	137	113	117
fftbutterfly32	370	395	485	460	459
fir32	208	231	295	284	277
hetero	346	366	428	407	399
leakage ($\times 10^4 nW$)	source	mg_null	mg_random	mg_greedy	mg_bmh
CIC	11	107	43	59	43
CORDIC_serial	15	107	44	60	44
fftbutterfly32	32	107	45	61	44
fir32	19	107	44	60	43
hetero	29	107	44	60	43
area ($\times 10^{-4} mm^2$)	source	mg_null	mg_random	mg_greedy	mg_bmh
CIC	76	647	292	395	300
CORDIC_serial	97	647	292	395	300
fftbutterfly32	178	647	292	395	300
fir32	113	647	292	395	300
hetero	170	647	292	395	300

Table 4.2: Resulting costs for unconstrained synthesis experiment. The source column shows the costs of a fixed-function implementation, while the columns to the right indicate the performance of a merged circuit (capable of implementing any of the five functionalities) when configured to implement the functionality in question. The various columns indicate various binding strategies.

- utilized cell area.

By examining the results for a variety for synthesis constraints, it is possible to derive the set of Pareto optimal design points (the "Pareto frontier"). These points each represent a unique design trade-off between delay and other performance characteristics. Through experimentation with the standard cell synthesis flow used for this exercise, it was found that dynamic energy, static power, and area metrics

are positively correlated. That is, when one of the three variables increased, the others tended to increase as well. This unsurprising dependency is indicative that the commercial synthesis flow we used is unable to expose trade-offs between these three parameters, perhaps indicating that the trade-off does not exist within the achievable design space of the flow. Therefore the results presented here focus on Pareto optimal trades of critical path delay against dynamic energy, static power, and area.

Figure 4.4 plots the achieved dynamic energy and area vs. delay Pareto curves. For each of the five input circuits, the figure shows both the achievable design points for the application-specific implementation and the automatically generated reconfigurable implementation. Recall that the **mp_greedy** and **mp_random** traces represent a multiple personality implementations, each capable of behaving identically to any of the five input circuits.

4.4 FPGA Implemenation

For comparison, each of the five sample circuits under consideration in this study were also synthesized for a Xilinx Spartan-3 90nm FPGA. The standard Xilinx ISE tool flow was used, and power and timing analysis were executed on the results. In contrast to the standard-cell flow, where critical pieces of the data-path are built out of smaller components and much room exists for trade-offs between energy and delay, FPGA synthesis tools generally map and place standard multi-bit operations in a specific manner efficient to the underlying FPGA fabric. In particular, in modern commercial FPGAs multiplications are typically mapped to dedicated multiplication units, and addition is mapped so as to utilize dedicated carry-chains within the device. As these operations dominate the signal processing application domain, and

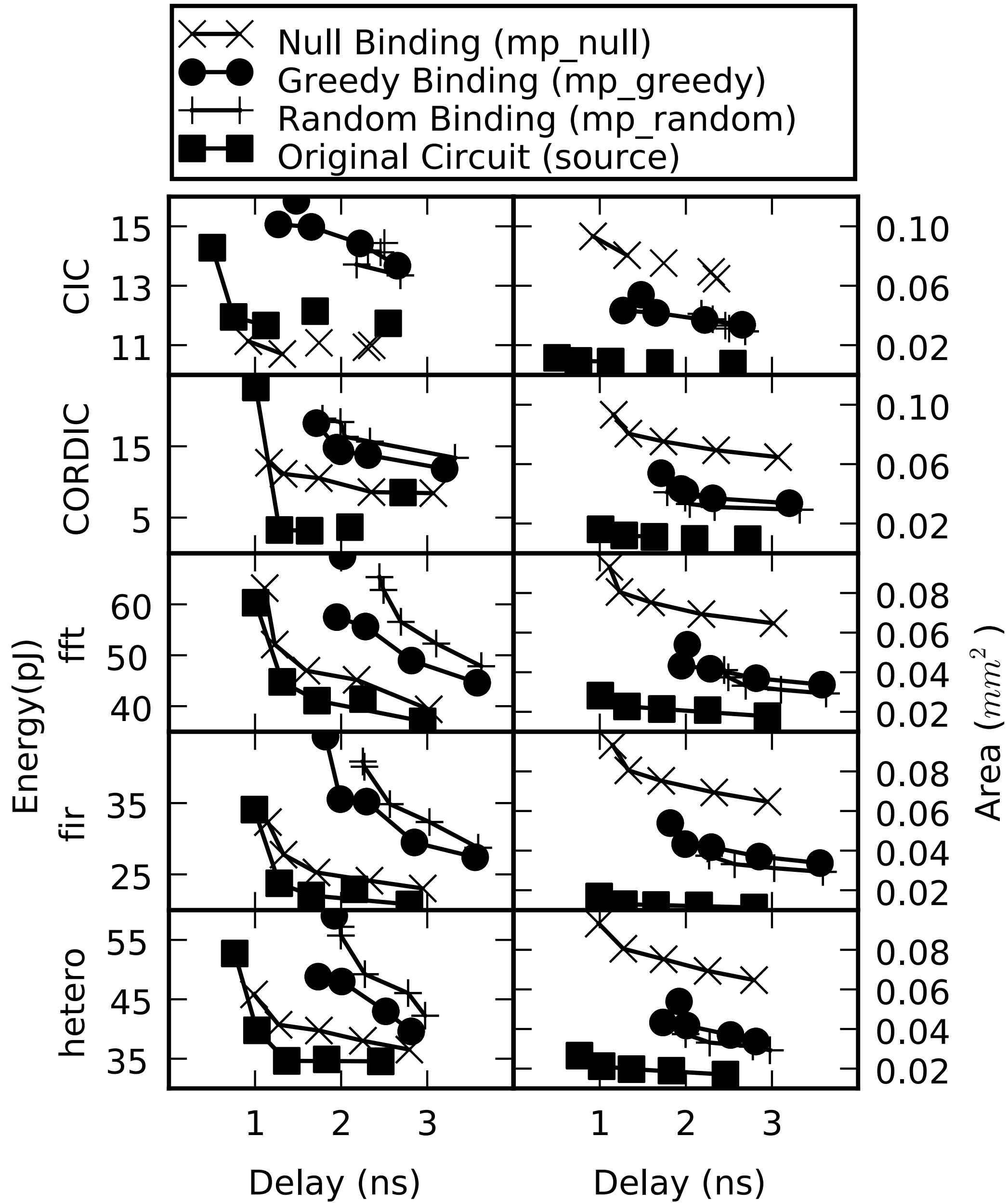

Figure 4.4: Achievable dynamic energy and area design points for each input circuit and its three reconfigurable implementations as a function of critical path delay. The points forming the Pareto Frontier for each circuit are connected with lines.

	Standard Cell ASIC					FPGA
delay ($\times 10^2 ps$)	source	mg_null	mg_random	mg_greedy	mg_bmh	
CIC	24	25	25	26	26	118
CORDIC_serial	28	32	34	34	33	122
fftbutterfly32	29	30	37	36	37	181
fir32	28	30	36	37	36	130
hetero	25	28	30	29	29	91
mcu	33					143
energy ($\times 10^{-1} pJ$)	source	mg_null	mg_random	mg_greedy	mg_bmh	FPGA
CIC	105	110	137	147	143	1400
CORDIC_serial	65	84	137	113	117	3200
fftbutterfly32	370	395	485	460	459	9000
fir32	208	231	295	284	277	3400
hetero	346	366	428	407	399	3400
mcu	29					6000

Table 4.3: 90nm FPGA Implementation Results compared to RAST flow with 90nm standard-cell implementation.

may become very inefficient if mapped in any other manner, only a single design point is shown for the FPGA synthesis results.

The resulting performance figures for the FPGA are shown in Table 4.3 and graphically in Figures 4.5 and 4.6. Only critical path delay and dynamic energy figures are shown, as reputable die size information for the FPGA was not available in the searched public literature. Moreover, despite the favorable light that such results would likely paint the proposed multiple personality approach, it is arguably misleading to present area and static power figures based upon a partially utilized FPGA, as "design to fit" FPGA fabrics are often neither attainable, nor desirable [27]. In order to reflect a comparison to the standard cell flow, power and energy figures reflect only the V_{core} component of the device, and ignores the other power supplies within the FPGA.

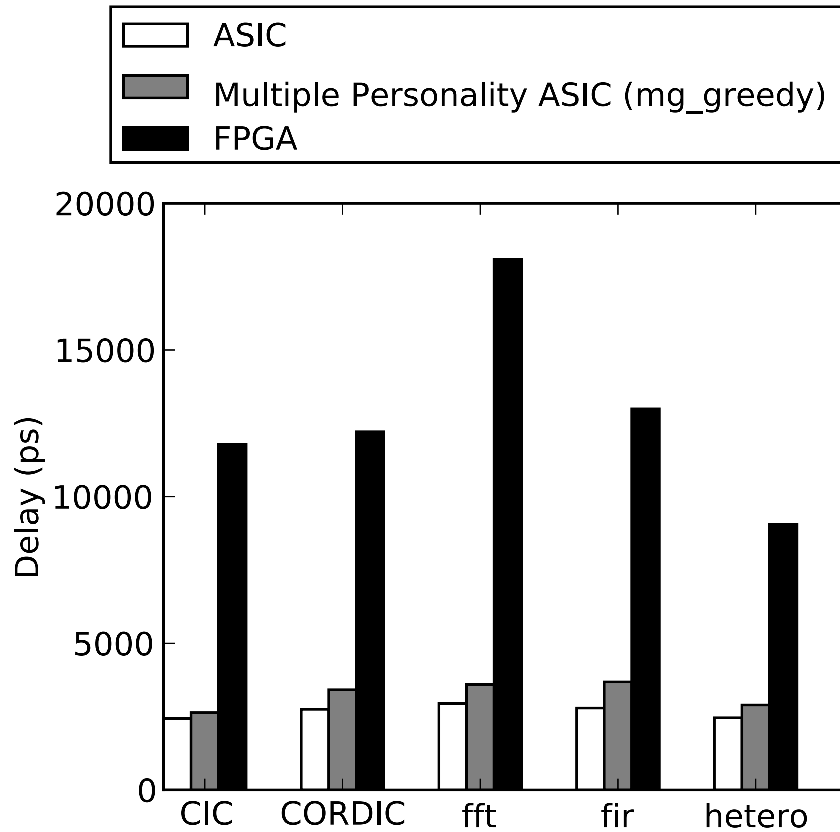

Figure 4.5: Comparative critical path delay for a fixed-functionality standard-cell ASIC, the proposed multiple personality approach, and an FPGA.

4.5 Revisiting Asymptotic Region Boundary Predictions

From the RAST case study, estimates can be made for several of the model parameters used in section 3.5.2. In section 3.5.2, very conservative values were used for parameters α, β, and γ, to predict a lower-bound on the number of functionalities (N) present at the boundary of the component-dominated asymptotic region. In this section, crude estimates for these parameters are made from experimental data, and the predictions revisited to produce a more typical estimate.

In the output of the RAST tool using binding algorithm, there were 101

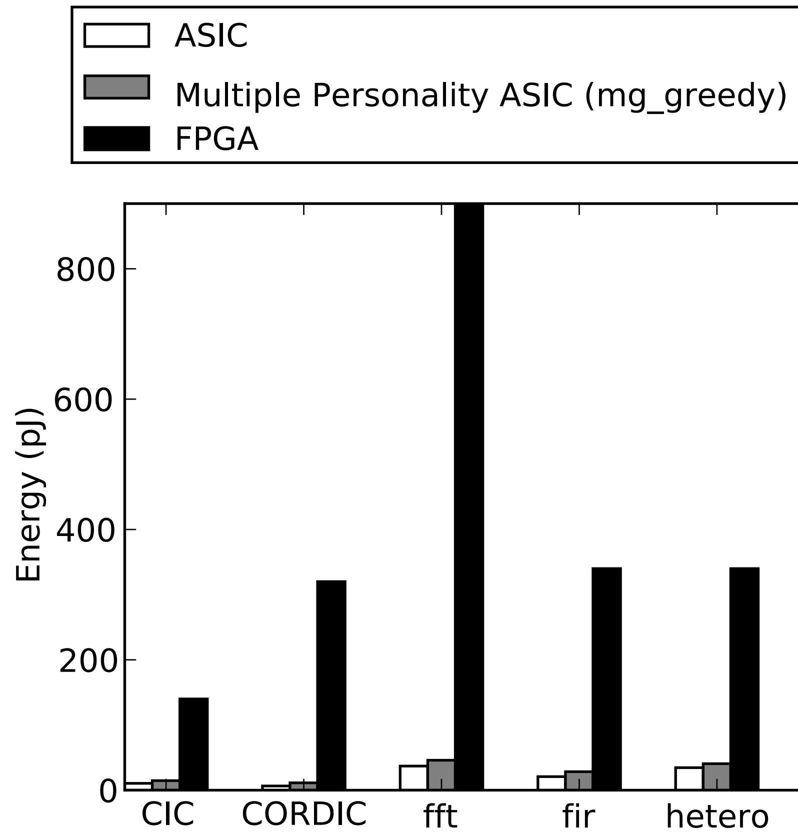

Figure 4.6: Comparative dynamic energy per cycle for a fixed-functionality standard-cell ASIC, the proposed multiple personality approach, and an FPGA.

fixed-function components, each with 3 terminals ($k = 3$) and 2 inputs. This allows estimation of the fraction of terminals that are inputs at two-thirds ($\alpha = 0.67$). Similarly, the number of components C can be fixed at the average number of components in each functionality of 27.

Furthermore, with two inputs per functional unit, each of which may need one or more two-input multiplexers inserted to achieve the desired functionality. Of the minimum 202 potential multiplexers, only the equivalent of 51 two-input multiplexer elements were needed. Thus a conservative estimate for the fraction of

Asymptotic Region Boundary	Delay	Energy	Area
Component/Mux	2^{80}	40	38
Component/Wire	208	20	-

Table 4.4: Typical number of functionalities (N) predicted to riside within the component-dominated asymptotic region. See section 3.5.2 for more detail.

potential multiplexers actually inserted (β) is

$$\beta \approx \frac{51}{202} \tag{4.1}$$

As noted in section 4.1.1, operand isolation techniques produced a reduction in energy overhead of mean 39%, yielding an estimate for the fraction of dynamic energy overhead actually incurred of $\gamma \approx 0.61$.

Using these parameters in the methodology of section 3.5.2 yields expected values for the size of the component-dominated region which are summarized in Table 4.4. That is, depending the particular cost function selected, one should expect to be able to produce programmable circuits with between 20 and 2^{80} functionalities before the cost of programmability dominated the cost of the fixed-function components. Thus for future work, the component-dominated region is interestingly large.

4.6 *a posteriori* Progammability

The RAST toolchain provides a computer aided technique producing a programmable circuit capable of implementing the functionality of any of the input circuits known as design-time. This *a priori* programmability is sufficient to directly satisfy the definition of programmability justified in section 3.1. However, as noted in section 3.3, a useful feature commonly associated with programmability is that of implementing

functionality not envisioned at design-time. Such *a posteriori* programmability is useful for feature improvements and bug-fixes. It is often acceptable for efficiency to drop as the desired functionality moves beyond that envisioned at design-time, but the ability to gracefully degrade by implementing unintended functionality in existing hardware is valuable.

As argued in section 3.3, merging techniques, such as those implemented by RAST, are theoretically sufficient for *a posteriori* programmability provided that the set of envisonable funtionalities can be enumerated. While theoretically attractive, such a strategy has the substantial practical flaw: it requires enumeration and merging of an potentially intractable number of envisonable functionalities. That is, such a brute-force solution problem is likely to be computationally intractable in practice. As a result, section 3.3 argues for and proposes higher-levels of abstraction.

One intriguing technique is to utilize multiplexers inserted during the merge to implement circuitry beyond the original targeted set. Specifically, the higher-level *a posteriori* programmability goals and metrics of section 3.3 can be improved by strategically adding "extra" multiplexers during the binding process, thereby directly exposing an efficiency vs. generality trade space. Such exploration is left for future work, but it is the author's belief that the properties enumerated in section 3.3.3 provide both an evaluation criteria for such work and directly suggest approaches for mechanically introducing multiplexers to meet said criteria.

For this work, a more straightforward leveraging of the RAST design flow to provide general-purpose programmability is explored. Two observations motivate this strategy:

- The Universal Turning machine proof suggests that, except for the infeasible infinite memory, the hardware requirements for a device to be fully pro-

grammable are quite weak[1].

- The incremental cost of implementing an additional functionality falls with the total number of designs. This arises because (a) additional designs share the resources of the others, and (b) key multiplexer costs (e.g. delay) scale logarithmically with the number of inputs (and therefore number of desired functionalities).

These properties can be exploited to gain general-purpose programmability by merging an additional microcontroller design (MCU) alongside the original *a priori* designs. To evaluate this strategy within this case study, a simple microcontroller has been implemented in Verilog. This microcontroller includes:

- a 32-bit datapath,
- multiplication, add, subtract, xor, or, and, not, and shift instructions,
- an accumulator-based architecture,
- multiple condition-codes for branching,
- a 3 cycle pipeline without interlocks, and
- the capability to connect to an external RAM up to 512 entries x 32 bits.

The intent is for the instruction set to be simple, while sufficently resembling contemporary digital signal processor instruction sets to support a judgement that the design is indeed programmable for general purpose tasks. For reference, the instruction set is shown in Table 4.5. An architectural diagram is shown in Figure 4.7. The instruction encoding, similar in spirit to Very Long Instruction Word

[1] See section 1.2

Mnuemonic	Description
NOP	No operation (for filling delay slots)
LD	Load from memory
ST	Store value to memory
IN	Load value from I/O port
OUT	Output value to I/O port
ADD	Add accumulator and memory operand
SUB	Subtract memory operand from accumulator
RSB	Subtract accumulator from memory operand
MUL	Multiply accumulator and memory operand (16x16-bit signed)
SHL	Shift accumulator value left 1 bit
SHR	Shift accumulator value right 1 bit
NOT	Bit-wise inversion of accumulator value
AND	Bit-wise AND of accumulator value
XOR	Bit-wise XOR of accumulator value
ZERO	Zero accumulator
XCH	Exchange most and least significant words of accumulator
DEC	Decrement memory operand
JMP	Unconditional jump to address
JB	Jump if below
JNB	Jump if not below
JZ	Jump if zero
JNZ	Jump if not-zero

Table 4.5: MCU instruction set

(VLIW) designs is shown in Table 4.6 and supports a limited amount of parallelism within a cycle. For instance, one may store a value to memory, add two values, and conditionally jump within a single instruction word. To support this, pipeline interlocks are not present, and it is assumed that the programmer will accommodate these features. An assembler was written to support this MCU, and a test program implementing a FIR filter is shown in Figure 4.8. Note that this MCU approach satisfies many of the abstract properties outlined in section 3.3.3 that are desired in a programmable system, including compositional universality, function coverage, class connectedness, bounded coupling, and constant access.

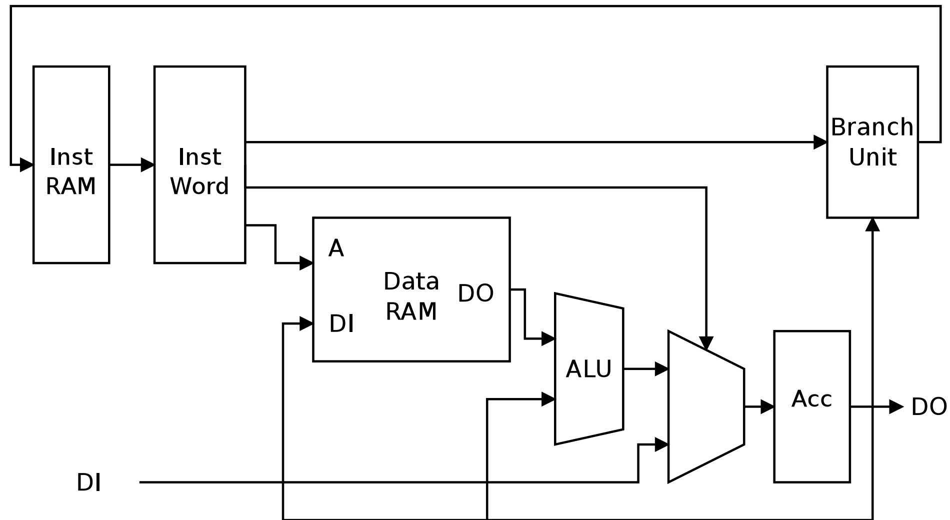

Figure 4.7: Block Diagram of the Microcontroller Unit (MCU)

Bits	Description
27-25	Branching opcode
26-16	Branch address
15-12	ALU operation
11	Store accumulator value to memory
10	Output accumulator value to I/O port
8-0	Memory address

Table 4.6: MCU instruction encoding

Consistent with the discussion of section 4.1.3, for the purposes of this case study, it is expected that the RAM will be shared with other circuitry and so RAM costs are not considered in the figures that follow. Obviously, the validity of this assumption is application-specific. If none of the *a priori* circuits require sufficient RAM, the costs incurred for having RAM may be substantial. Such a penalty appears to be unavoidable: general purpose *a posteriori* programmability is expensive. However, the need for RAMs of this size are common in the signal

```
        NOP
LOOP:
        IN   0x2
        IN   0x3
        ST   _X            # _X=IN(0x2)
        MUL  _X
        IN   0x4
        ST   _ACC
        MUL  _X
        ADD  _ACC
        IN   0x5
        ST   _ACC
        MUL  _X      ; JMP LOOP
        ADD  _ACC    # JMP Delay slot
        OUT  0x1     # JMP Delay slot


_X:     CONST 0
_ACC:   CONST 0
```

Figure 4.8: MCU Assembly source for a simple 3-tap FIR filter

processing domain[2], and this case is assumed for the remainder of the analysis.

The processor here is not intended to be particularly fast. As noted earlier, it is often acceptable for functionality unenvisioned at design-time to run slower than functionalties that were optimized at design time. As an example, when the FIR design was implemented within the MCU, the same functionality was obtained at $1/13^{\text{th}}$ of the *a priori* design throughput. While not efficient, the added flexibility comes with little penalty. As shown in Table 4.7, the incremental cost of implementing this MCU as an additional functionality alongside the five other test cases varies according to which of the five circuits is configured, but averages only 1.3% in critical path delay, 3.5% in dynamic energy per cycle, 7.8% in leakage power, and

[2]Commercially available "signal processing"-optimized FPGAs such as the Virtex-5SX line provide extra dedicated RAMs for this reason

10.1% in area.

Given the low impact, this configuration may be viewed as a "dual" of a hardware accelerator module attached to a general-purpose microprocessor. Instead of a small limited-purpose accelerator module integrated into a much larger microprocessor, this configuration integrates a small microprocessor into the very fabric of a much larger limited-purpose design.

	w/o MCU				with MCU			
delay ($\times 10^2 ps$)	mg_null	mg_random	mg_greedy	mg_bmh	mg_null	mg_random	mg_greedy	mg_bmh
CIC	25	25	26	26	24	26	28	31
CORDIC_serial	32	34	34	33	32	35	33	35
fftbutterfly32	30	37	36	37	30	36	35	37
fir32	30	36	37	36	30	39	36	36
hetero	28	30	29	29	28	29	29	29
mcu					33	33	32	31
energy ($\times 10^{-1} pJ$)	mg_null	mg_random	mg_greedy	mg_bmh	mcu_null	mcu_random	mcu_opt	mcu_bmh
CIC	110	137	147	143	110	132	147	148
CORDIC_serial	84	137	113	117	85	140	126	133
fftbutterfly32	395	485	460	459	407	493	468	473
fir32	231	295	284	277	237	303	292	285
hetero	366	428	407	399	367	434	424	401
mcu					48	122	113	111
leakage ($\times 10^4 nW$)	mg_null	mg_random	mg_greedy	mg_bmh	mcu_null	mcu_random	mcu_opt	mcu_bmh
CIC	107	43	59	43	119	46	68	46
CORDIC_serial	107	44	60	44	119	47	69	46
fftbutterfly32	107	45	61	44	120	47	70	47
fir32	107	44	60	43	119	47	69	47
hetero	107	44	60	43	120	47	69	46
mcu					119	46	68	46
area ($\times 10^{-4} mm^2$)	mg_null	mg_random	mg_greedy	mg_bmh	mcu_null	mcu_random	mcu_opt	mcu_bmh
CIC	647	292	395	300	740	320	463	332
CORDIC_serial	647	292	395	300	740	320	463	332
fftbutterfly32	647	292	395	300	740	320	463	332
fir32	647	292	395	300	740	320	463	332
hetero	647	292	395	300	740	320	463	332
mcu					740	320	463	332

Table 4.7: Comparative results for designs with and without a merged MCU

Chapter 5

Prior and Related Work

This work draws from several areas, each solving a related, but fundamentally different problem. These problems include: 1) modification and/or run-time configuration of FPGA architectures, 2) high-level synthesis of application-specific data-path accelerators, and 3) reconfigurable computing architecture design. In addition, a substantial body of directly relevant prior work has developed out of The Totem Project [46] at The University of Washington.

5.1 FPGA Modification Approaches

A number of prior efforts involve using FPGAs in atypical ways to achieve goals related to this effort. The approaches can be categorized into two general: 1) pruning of unneeded FPGA resources, 2) run-time reconfiguration of existing FPGAs.

FPGA vendors have long offered reduced functionality versions of their components in order to reduce costs. For example, Xilinx™ offers an approach called EasyPath™ [84], which relaxes production-line device testing to include only those resources actually utilized by a particular customer design. This allows devices that are not completely functional, and would normally be scrapped, to be sold to suitable customers at a reduced cost. In contrast, Altera™ offers a "HardCopy" service, in which FPGA designs are mapped onto "common partially fabricated units" [1] of similar construction to an FPGA. Instead of using FPGA-like programmable circuits, the operation of the device is customized via production of a subset of the metal layers. In this manner components not needed in the FPGA fabric are simply not connected, yielding improvements in both performance and energy dissipation [1].

Parvez et al. [66] have extended this approach to allow a set of applications to be mapped to an FPGA in such a manner that the total resources used in the FPGA

are minimized. The unused resources can then be pruned, leaving a "Application Specific Inflexible FPGA (ASIF)" device that can implement any of the applications in the set, but only limited subset of others. Re-synthesis of the resulting ASIF into standard cell technology yielded 85% reduction in FPGA area usage in one case. An overview of early work on a new CAD flow capable of mapping designs into the ASIF is also presented.

An alternate approach that yields rapid reprogrammability is run-time configuration of existing FPGA architectures. This work has historically focused on several key problems:

- software libraries to abstract the complex (and often proprietary) nature of FPGA architecture internals [40],
- minimizing the storage needed for target FPGA architecture descriptions [80] [53] [81],
- minimizing the execution time of routing algorithms [52] [11] [28], and
- integrating circuit-descriptive software libraries [68].

While these efforts have been successful to varying degrees, run-time synthesis and configuration of complete data-paths is largely an unachieved goal due to the NP-hard computational complexity of the FPGA technology mapping problem. Moreover, the commercial environment is such that if the mapping problem could be simplified, traditional FPGA vendor tool-chains would be among the first to adopt the techniques. As a result of the computational complexity, the biggest benefactors of the run-time configuration work are problems that can be addressed without attempting to solve the NP-hard mapping problems, including multi-layer

floorplanning [76], radiation induced soft-fault detection/correction in FPGAs [39], and limited debugging probe readout.

5.2 High-level Synthesis

High-level synthesis (HLS) is generally defined as the "automated generation of the hardware circuit of a digital system from a behavioral description" [42]. High-level synthesis techniques have been well-known for over two decades [62], and have been used successfully in numerous commercial products.

For conceptual reasons, the high-level synthesis problem is traditionally broken down into two optimization subproblems: (i) scheduling and (ii) binding (also known as allocation or assignment in the literature). The scheduling subproblem is to determine the order in which operations are executed, typically with the goal of optimizing execution time under constraints derived from operation dependencies and hardware limitations. The binding subproblem consists of mapping operations in the input behavioral description to physical computing resources in the target hardware circuit, often with the goal of minimizing required hardware resources. While the binding and scheduling subproblems are often treated as conceptually distinct, these two optimization problems are not separable under many, if not most, interesting conditions [62]. While extensive work has been done in the area of high-level synthesis, the problem is so broadly defined and computationally complex that much of its promise remains unfulfilled in practice. As a result, interest and work on various aspects of high-level synthesis continues.

While the high-level synthesis problem arguably assumes scheduling flexibility by definition, the binding subproblem for a fixed schedule has been studied independently. Notably, Haung et al. [48] examines a problem defined as follows:

> We assume that the data path allocation is performed after the scheduling. [...] Our inputs include: (1) the scheduled data flow graph and (2) the set of available functionunits with known capability (ie. types of operations each function unit is capable of performing). Our goal is to allocate a minimal set of registers for variables, assign operations into function units, and connect registers to and from function units using minimal number of multiplexers (measured as the number of multiplexer inputs).

Weighted bipartite matching is used for two independent and successive steps of register allocation and operation assignment. By assuming register connections between operations and allocating them in a separate step based on lifetimes, this work avoids much of the higher order potential mux elimination steps (and thereby potential optimization) exploited by the works discussed in section 5.3.

van der Werf et al. [87] examine the case of merging data flow graphs (DFGs) into a "multi-functional processing unit" which can execute one DFG at a time. The goal of the exercise is to "find an operator assignment that minimizes the silicon area that is occupied the [...] interconnection consisting of multiplexers and wires." A local search algorithm based upon iterative and simulated annealing using an assignment 2-exchange primitive is used to minimize the number of edges in the interconnect.

As noted by Geurts et al.[37], an alternative approach is to expand the high-level operators into Boolean logic, and then use logic synthesis techniques to minimize redundancies (e.g. subexpressions). While conceptually pure, and perhaps optimal in a general way, the general logic synthesis problem is known to be in the class NP, and the reduction from high-level operators to Boolean expressions only

increases the computational complexity (exponentially).

Fan et al. [34] demonstrate an approach for generating "reconfigurable interconnection network" components, which are switch blocks that "yield routing solutions for a pre-given set of applications." This work focuses on the generation of multi-stage switch boxes that have minimal complexity, and therefore presumably low costs of implementation.

5.3 Datapath Merging Problem

The Datapath Merging (DPM) problem can be viewed as a restricted subset of the high-level synthesis problem. The general DPM problem accepts as input any number of DFGs, and produces as output a "single reconfigurable datapath" [63], with the goal being to "design a reconfigurable datapath which incorporates all the [...] datapaths and has [the fewest] functional units and interconnections as possible." [63] Experimental work on this subject has primarily focused on using unscheduled DFGs obtained from intermediate compiler representations of software implementations, although manual examples of the technique exist in the context of FPGA run-time reconfiguration [75].

A substantial portion of work on the DPM problem are related to techniques for accelerator synthesis [37], [4]. A common use case for such accelerators is to implement Application-Specific instruction set extensions for otherwise conventional microprocessors. For instance, Zuluaga and Topham [91] propose a technique that considers latency constraints during the merging process between multiple instruction set extensions. In this technique each functional unit type is synthesized using Synopsys' DC Ultra synthesis tool into a 130nm standard-cell library under a range of synthesis constraints, yielding a Pareto frontier of area vs. delay curves for each

component. The Pareto front for each component is then used in a relaxation process [38] to produce a Pareto frontier of merged solutions using modified versions of the resource-sharing techniques of Brisk et al. [10]. Notable other examples of application-specific instruction set generation are listed in [90] and [10]. Brisk et al. [10] propose a DPM heuristic solution restricted to directed acyclic graphs (DAGs), and focuses on reducing functional unit area at the expense of interconnection sharing. This heuristic solution relies on enumerating the operation type along all paths in the input DFGs into a set of strings, followed by finding the longest common-substrings (appropriately weighted for component area), and using them to construct an output merged DFG.

In the chapter "Application-Specific Unit (ASU) Synthesis" Geurts et al.[37] outlines a methodology for the DPM problem, focused on merging clusters of operations into a single unit. Much as in earlier work [87], the sole goal is to minimize the area cost. Two techniques are presented, differing primarily in the modeling of interconnect cost: 1) local estimation based on linear programming, and 2) a global modeling based on integer linear programming. Discussion of the problem of false-combinatorial cycles introduced by binding together operators with dependency-order conflicts is provided, along with mechanisms to reject solutions containing these cycles. Also discussed are approaches for iteratively merging $n > 2$ graphs using pair-wise iterative techniques, along with a conclusion that "the pairwise merging technique can be used to generate solution which are close to optimal".

Huang and Malik [49] discuss the DPM problem as a component of a methodology to minimize run-time reconfiguration overhead in Systems-on-a-Chip(SoC). Specifically, a design flow is described in which a software application is profiled to find inner-loops (i.e. kernels), which are then extracted as DFGs and merged to

form a reconfigurable datapath. The interconnect for the reconfigurable datapath is optimized through matching functional units from the input DFGs via maximum bipartite mapping. The ability to use bi-partite matching in the DMP problem relies on the restriction of merging only 2 DFGs at a time (i.e. 2-DPM). To merge more than $n > 2$ DFGs (i.e. n-DPM), the additional graphs are iteratively merged into the final graph one at a time. Because of this, the algorithm is a greedy heuristic in that binding choices made early on are made without regard to all the input graphs and may therefore be suboptimal.

Moreano et al. [63] further elaborate on the DPM problem by defining a "compatibility graph" in which each node corresponds to a pair of arcs from differing input DFGs. In this case, a compatibility graph node symbolizes the mapping of the head and tail of one arc to the head and tail, respectively, of the other arc during the merging process. Compatible graph nodes are those in which the mappings implied by the node are not conflicting (i.e. a single DFG node is conflicting iff it is mapped to more than one node in another DFG). By defining the existence of an edge between nodes in the compatibility graph if and only if they are compatible, the problem of minimizing the number of interconnection edges becomes that of finding a maximum clique in the compatibility graph. As the maximum clique problem is NP-complete, the authors propose a specific polynomial-time heuristic algorithm by Battiti and Protasi [6]. Similar to [49], a greedy iterative approach is proposed for the merging of more than 2 DFGs. The overall approach described is referred to as "Moreano's heuristic (MH)" or the "Compatibility Clique" algorithm in later works.

Various algorithmic aspects of the DPM problem were elaborated upon by de Souza, et al. [79]. Again, minimization of interconnect edge cardinality was proposed as a goal, and under this condition the DPM problem was shown to be NP-hard.

Deciding the feasibility of a specific edge cardinality was shown to be NP-complete in general via reduction to the subgraph isomorphism problem. Interestingly, in this analysis a feasible solution is restricted to use as few functional unit of each type (or "label" in the terminology of [79]) as is possible.

Moreano, et al. [64] expand on more practical aspects of the DPM problem by defining the "DFG merge problem", which they define as minimizing the total area of the merged graph, including modeled interconnect, multiplexers, and wires. The authors note that minimizing functional unit area cost is not a difficult task in the case of 2-DPM, as one can leverage polynomial time solutions to the maximum weight bipartite matching problem. However mapping so as to minimize interconnect area is an NP-hard problem. Experimentally, 9 applications taken from compiler intermediate representation of the MediaBench suite [59] are examined in a case study. For each application up to 8 kernels were extracted as DFGs, merged by both the compatibility clique technique, and synthesized with the Synopsys design compiler tool. The limited applicability of the resource allocation technique built-in to Design Compiler was discussed. The resulting designs were compared in terms of both synthesized area, and merging tool computational run time to 3 other binding techniques (specifically bipartite matching, iterative improvement, and integer programming). Quantitatively, area reductions averaging 19.6% over an unmerged datapath were reported, while energy and delay are not reported.

5.4 Multi-mode Synthesis

More recent work has applied a classic high-level synthesis argument that scheduling and binding are best jointly-optimized to the DPM problem [12, 7]. Specifically Chavet et al. [12] argue that among prior work (e.g. [87, 37, 49, 63, 64]), "four

distinct approaches can be identified" based upon which of the steps in a conceptual high-level synthesis design are modified to be merge-aware:

1. Graph merging - merging occurs outside of the classic HLS steps prior to scheduling and binding,
2. Joint scheduling - HLS scheduling is aware of the merge,
3. Sharing during binding - merging occurs during traditional HLS binding, and
4. Datapath merging - merging occurs outside of the classic HLS steps after scheduling and binding.

Note that Chavet et al.[12] use a post-scheduling, post-binding definition for the term "datapath merging" that differs from the pre-scheduling, pre-binding definition used by Moreano et al. [63]. The prior work argues the need for a distinct "multi-mode" synthesis design flow that co-optimizes scheduling and binding, and suggest that new "scheduling, binding and register merging algorithms have to be proposed". The authors go on to propose such an algorithm and implementation strategy based upon a "strong semantic memories" (e.g. FIFO buffers), a FSM controller and synthesized datapaths. Experimental work was conducted by merging variants (e.g. differing tap-length or matrix-size) of the same algorithm for several common numerical algorithms (e.g. FFT, FIR, IFFT, LMS, DCT). Very limited attention was paid to merging different algorithms. However, 3 pairs of DFGs for differing algorithms were merged. In all cases, area and operation count results were provided for this approach, and compare favorably to the result discussed below ([14]). Andriamisaina et al. [3] furthered this work by presenting a methodology that uses multi-mode aware scheduling and binding steps during the synthesis of each input DFG.

Chiou, Bhunia and Roy [13, 14] presented a multi-mode synthesis flow based upon a SPA-tially Chained Transformation (SPACT) in which the input DFGs are scheduled individually with estimated resource constraints, then concatenated, then bound, and then synthesized into HDL code. Three heuristic approaches are proposed with varying trade-offs between area and power. In the minimal resource approach (SPACT-MR), the input DFGs are scheduled based upon minimal available functional resources. In the resource constrained approach (SPACT-RC), some additional functional resources are made available during scheduling to potentially improve power at the expense of area. In the signal similarity-based approach (SPACT-SIM), the binding cost function is tuned to minimize switching power based upon data value statistics. In each cases, binding is performed with modified version of the weighted bipartite matching algorithm proposed by Huang [48]. Experimental data is provided based upon merging several variants (e.g. differing filter tap lengths, constraint lengths) of a single DSP algorithm into a single module, synthesizing into a 0.25 μm standard cell library and then measuring total power, area, and critical path delay overheads for the merged design relative to a fixed-purpose design. This experiment is repeated for three common DSP algorithms (Verterbi-style add-compare-select networks, finite impulse response (FIR) filters, and infinite impulse response (IIR) networks. Area savings of 14 to 44% over the sum of individual designs were reported, along with delay overhead from 2-12%, and power overhead of 2 to 30%. The same three algorithms were compared to a fixed-purpose implementations in a 0.22 μm Virtex FPGA, with the reported critical-path delay improvements of about 3.5X, and power consumption improvement of about 53-66X. Note that in this work, only variants of the same algorithm were merged with each other.

Kumar and Lach [55] present an approach for incorporating flexible arithmetic components (FAC) into multi-mode synthesis. The example FAC given is a functional unit that can both multiply and add, but is smaller than the combined areas of an adder and multiplier. It is argued that the utilization of FACs in multi-mode synthesis will further reduce interconnect costs by by allowing DFG subgraph isomorphisms without exactly matching functional units to be merged. Smith et al. [78] propose a similar approach and claim an average improvement of 1.67X in delay and 5.55X in area for nine small (2-4 operation) common subgraphs implemented in 90nm standard cell logic relative to a Xilinx Virtex 4 FPGA.

Although not directly applicable, multi-mode strategies have been incorporated into system-level modular hardware-software co-design frameworks, such as the Multi-Objective Co-Synthesis Framework for Multi-mode Embedded Systems, termed "CHARMED" [54]. Notably, this framework claims to use multi-objective genetic algorithm to produce a non-dominated set (Parteto front) of implementations. While experimental results are provided for 10 randomly generated DFGs, this author was unable to locate any more rigorous results from CHARMED in later literature. Schmitz et al. [73] provide another example of the energy-efficient multi-mode system-level co-design that considers the probability of occurrence for each mode to minimize expected costs.

5.5 Reconfigurable Computing

Reconfigurable logic arrays had their humble birth in the implementation of "low cost/marginal performance class" [35] of circuitry, as more mundane concerns of testability and designer effort began to outweigh the incremental fabrication costs of large scale integrated circuit fabrication. As fabrication abilities improved, gate

arrays became field programmable, and began to be used for larger applications [85]. Today Field Programmable Gate Arrays (FPGAs), capable of emulating millions of gates, are frequently used to implement entire applications, and may be the only computing device in a piece of hardware.

Modern reconfigurable computing architectures can be categorized in many dimensions, including:

1. functional unit granularity (e.g. gate array or object array),
2. functional unit regularity (homogeneous or heterogeneous),
3. interconnect style (e.g. mesh or hierarchical),
4. overhead per instruction word (e.g. fetch/execute loop or FPGA-style bitstream reconfiguration), and
5. global synchronization style (e.g. asynchronous message passing/synchronous clocking).

Academic literature in this field is largely filled with the exploration of single "island" design-points. Typically a unique architecture is developed, hard-compiled demos are generated, benchmarks are published, and the numerous unique features of the architecture are cited as responsible for the remarkable performance. Reference [45] alone cites 18 distinct architectures first published within a single four-year period, including the RaPiD architecture discussed in section 5.6. Similarly, gate array architecture design studies [60] [69] in academic literature are dominated by exploration of design parameters of a single synthesis tool (Versatile Packing, Placement and Routing for FPGAs [8]). Comparatively little effort has been invested in rigorously defining the space of reconfigurable computing architecture, although [26] provides a useful framework for analyzing several dimensions.

5.6 The Totem Project

The Totem Project at The University of Washington has the stated goal of providing an "automatic path for the creation of custom reconfigurable hardware, targeted for use in Systems-on-a-Chip (SoCs)" [46]. This ambitious project intends to span from high-level architecture generation, through layout of the programmable chip, ultimately including CAD suites customized for each generated architecture. To date, the Totem project has focused on two distinct architecture styles: 1) a course-grained architecture based upon the Reconfigurable Pipelined Datapath (RaPiD) architecture [21], and 2) a crossbar-based architecture similar to Programmable Logic Arrays (PLAs). While a substantial amount of work on the latter style has been performed, the former represents the work most relevant to this document and so will be discussed in greater detail.

The RaPiD architecture [33][22] leveraged by the Totem project is a linear array of compute components that operate on full 16-bit words of data. A single cell from the RaPiD architecture is shown in Figure 5.1, which contains compute components including 16-bit ALUs, a 16x16 multiplier, 16-bit RAM, and 16-bit registers. Interconnection is performed at word-level granularity by segmented routing tracks spanning the length of the array. The RaPiD architecture is quite specific in form, allowing for 5 local routing tracks, and 10 tracks with longer segments. Bus connectors are provided to allow adjacent segments in a single routing track to be connected in order to provide for longer distance communication. A large portion of Totem work is based heavily upon the RaPiD [21] infrastructure, including use of the RaPiD-C [32] compiler to generate netlists from a C-like language, and use of RaPiD area models to gauge success.

Building upon the RaPiD framework, in [18] a two stage algorithm was devel-

oped for combining multiple RaPiD netlists into an application-specific RaPiD-like structure. In the first stage, simulated annealing approach is used to bind compute elements from input netlists to physical units in a RaPiD-style linear array. In the second stage, one of three algorithms (Greedy, Clique Partitioning, or Maximum Sharing) is used to generate the routing between the components. The generated architectures preserve the linear functional unit layout structure from RaPiD. However, they abandon the strict RaPiD routing structure and instead only contain point-to-point routing paths and multiplexers as required to implement the specific input designs. This paper considered area optimization as a sole metric and demonstrated that a custom architectures can achieve area efficiencies of only 1.5 times a lower bound based on the minimum number of functional units able to implement each of the input netlists.

The concepts in [21] were further elaborated upon in the Ph.D dissertation of Katherine Compton [17], which introduced the term "configurable ASIC" for the generated architectures. Among the contributions of the dissertation is a comparison of sample configurable ASIC designs against a traditional FPGA implementation. The comparisons with traditional FPGAs were limited to area-efficiency, but were quite favorable, with improvements ranging from 4-12X. Because the area model for the generated architectures used custom-layout RaPiD functional units, overall factors of 2X improvement in area were possible over purely standard cell implementations. Follow-on publications, most recently [20], have further discussed this idea, although comparisons have thus far been limited to area-efficiency.

In [19], a novel method for measuring the flexibility of domain-specific reconfigurable hardware is proposed. The methodology relies upon generating synthetic netlists that mimic the characteristics of netlists within a specific domain. By gen-

erating a domain-specific reconfigurable architecture based upon these synthetic netlists alone, the flexibility of the architecture can be quantitatively measured by its ability to place and route the original domain-specific circuit.

5.7 Virtual Reconfigurable Architectures

In an interesting application of datapath merging problem to existing FPGAs, Rullmann and Merker have developed a technique for development of virtual architectures on top of FPGAs using datapath merging [72]. In another paper, the datapath merging technique (including a novel Ant Colony Optimization algorithm) is used to generate placement constraints to force the FPGA synthesis tool to place similar logic in similar placement between multiple designs, thereby maximizing redundant configuration bits between DFGs[71].

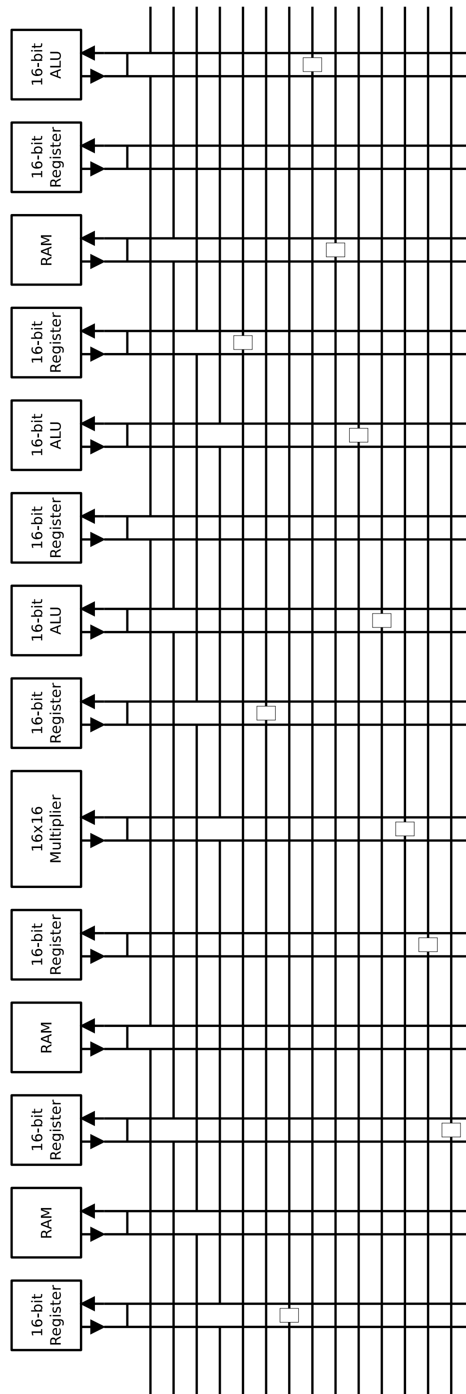

Figure 5.1: A single cell from the RaPiD architecture [33] [22] [18]

Chapter 6

Conclusion

6.1 Contributions and Potential Impacts

In exploring the thesis of chapter 1, this work provides a number of contributions with direct implications both to applied engineering and future academic research, including:

- A rigorous definition of programmability - In this author's experience, programmability is an informally used term, resulting in much confusion. As a result, programmabilty is often a neglected aspect in comparisons between computing architectures. Section 3.1 argues for a definition of programmability suitable for engineering used based upon how a device is used, rather being an intrinsic feature of a device. Such a definition serves as a foundation upon which programmability can be treated as an explicit design parameter.

- Formal division of programmabilty into *a priori* and *a posteriori* aspects - Section 3.3 argues for a division of programmability into *a priori* and *a posteriori* aspects, in which the intended functionality of a device is fixed either before or after fabrication time. As chapter 4 shows, *a priori* knowledge yields distinct advantages in implementation efficiency. Moreover, the uncertainty inherent in the *a posteriori* problem demands distinct approaches from those of the *a priori* problem.

- An interconnect-based framework for construction of programmable devices - While a programmable device can be implemented in perhaps infinite ways, section 3.2 suggests a construction technique that separates the composition of a device into fixed-function components that are connected by an interconnect responsible for implementing the programmable aspects of the devices. Such a construction facilitates explicit manipulation of the programmability of a

device within a well-defined framework with many useful properties.

- Identification of design-space regions for programmable interconnect-based devices - The properties of a device with fixed function components embedded in programmable interconnect naturally give rise to a number of distinct cost classes: those from the fixed function components, those from the active components of the interconnect, and those from the wires composing the interconnect. One of these classes of cost may dominate the others, and this gives rise to distinct design-space regions for programmable interconnect-based devices. As established in 3.5, the scaling properties and optimization strategies within each design-space region are distinct and can be predicted.

- Predictions of the extent and usefulness of the design-space regions - Section 3.5.2 suggests that the extent of the efficient component-dominated region can be predicted based upon high-level design parameters. Such predictions are useful for system-level planning of both the size of programmable blocks (as programmable regions that are too large become overwhelmed by wire costs) and the amount of functionality to build into a single programmable block (as large numbers of function become overwhelmed by multiplexer costs).

- A tool for exploring *a priori* programmability in a computer-assisted design flow - Chapter 4 documents the creation of a computer assisted design tool for exploring *a priori* programmability that is easily integrated into a standard-cell design flow. This tool facilitates the control of programmability with a design. For instance, designs from entirely different design teams may be efficiently combined into a single programmable design. Because the output of the tool is easily integrated with other logic, the tool allows designers to incoporate

programmability only to the extent that is desirable.

- Demonstration of a "dual" for the general-purpose processor accelerator module - As discussed in section 5.3, the performance of large general-purpose microprocessors can be greatly improved through the addition of a small special-purpose accelerator module. Section 4.6 demonstrates the opposite idea: the introduction of a small general-purpose design into a larger fixed-function design with minimal added cost. Such a design has the potential to allow the introduction of general-purpose programmability into designs that may not otherwise tolerate the overhead of a conventional microprocessor.

This work is a foray into the treatment of programmability as an explicit design parameter. By formalizing terms, approaches, and properties, it both suggests and provides a foundation for a number of future academic research avenues. Moreover, the experimental validation conducted has resulted in computer-aided design tools for explicitly controlling programmability within computing designs.

6.2 Future Work

Several areas of future work were identified within this document:

- exploration of usefulness of the infimum in the functionality relation (Section 2.1.4),
- derivation of a Rent's rule bound for the heterogeneous component merging case (Section 3.4.2),
- automatic insertion of multiplexers to better address the *a posteriori* problem (Section 4.6),

- experimental identification of design points in regions other than the component-dominated region (Section 3.5), and
- experimental verification of predicted boundaries (Section 3.5).

Notably, a number of investigations suitable for future doctoral dissertations follow directly from observations made in this document:

- Firstly, a largely theoretical work is suggested to explore better classification criteria for programmable devices. While section 3.1 defines a minimal criteria for programmability, further (stronger) classifications are warranted. Several avenues for developing such classification criteria are suggested within this document. The infimum discussion of section 2.1.4 suggests a method for rigorously constraining devices via the absence of specific functionality. Such a device might be said to be as programmable as possible without possessing certain threshold capability. The high-level properties enumerated in section 3.3.3 suggest an alternate approach for rigorously characterizing programmable devices that could be further developed. A key part of this investigation would be the study of cost implications for various criteria.
- A second theoretical work is suggested to identify and exploit unintended functionality within programmable devices. Corollary 3.1.4 notes that resource sharing manipulations tend to increase the potential functionality of programmable circuits beyond those intentionally specified. Section 3.1.2 further discusses how these unintended functionalities can be exploited by appropriately configuring multiplexers with unintended configurations. This work would consider how these functionalities can be enumerated and appropriately exposed to a programmer to enable better performing *a posteriori* programma-

bility than the microprocessor-based approach outlined in section 4.6. Further work might consider how to appropriately insert additional multiplexers at design-time to enhance the ability to later implement functionalities *a posteriori*. The properties outlined in section 3.3.3 may serve as guidance on how to carry out this effort.

- Finally, an experimentally-focused work might carry out further development of the RAST tool of section 4.1.1 in order to perform experimental verification of principles in this document. This work would enhance the binding optimizer to support an optimization algorithm targeted at the wire-dominated region. The guiding principles of such an algorithm are suggested in the proof of theorem 3.4.3. With this improvement, the RAST tool would possess optimizers targeted at each of the three regions discussed in section 3.5. The remainder of the work would identify case studies to experimentally verify the predictions of section 3.5 relating to cost scaling, optimization strategies, and asymptotic region boundaries.

6.3 Recapitulation

Chapter 1 hypothesized that programmability can be elevated to an explicit design parameter that

- can be rigorously defined,
- has measurable costs amenable to high-level modeling,
- yields a design-space with distinct regions and properties, and
- can be usefully manipulated using computer-aided design tools.

A specific engineering problem was proposed in which one wishes to introduce a useful degree of programmability into a fixed-function circuit. This application was used as a contextual vehicle by which the thesis is discussed in the remainder of this document.

Chapter 2 provided a formalism in which the thesis of chapter 1 can be discussed in a rigorous manner. A computation model appropriate for the application was defined formally. Four cost metrics relevant to the implementation of computations in contemporary semiconductor technology were discussed: delay, energy, area, and leakage. A review of practicalities related to such implementations, including placement, wiring and the high-level modeling thereof concludes the chapter.

Chapter 3 began by proposing and defending a formal definition of programmability, noting that programmabilty is not an intrinsic trait of a device, but instead is associated with how it is used. Specifically, a device is said to be programmable if it can be assigned a particular function, and is distinguished from a fixed-function device by the ability to be assigned more than one function. Based upon this definition, a simple side-by-side realization of a programmable device was discussed that hinges on the availability of a multiplexer component. A model of programmability was then defined based upon the composition of fixed-function components and multiplexers. This class of multiplexer-based interconnects forms a design-space, of which the side-by-side realization is but one point. Two manipulations (resource sharing and multiplexer elimination) were then introduced, which are used to transform local regions of multiplexer-based interconnects. By modifying the interconnects, these two manipulations and their inverses provide a means to traverse the design-space of all multiplexer-based interconnects. The connectedness of this design-space via these manipulations was shown, in that any

multiplexer-based interconnect can be transformed via these manipulations to or from a canonical side-by-side realization.

The problem of generating programmable devices was then refined by distinguishing the *a priori* and *a posteriori* subproblems. The former concerns the generation of a device that can fulfill any of a set of functionalities enumerated at design-time. In contrast, the *a posteriori* subproblem arises out of a practical consideration – directly enumerating the set of desired functionalities may not be feasible in practice and therefore higher levels of abstraction are desirable. For example, programmable devices may be called to implement functionalities not envisioned at design-time in order to implement bug-fixes and feature improvements. The *a posteriori* subproblem concerns the development of properties and techniques to allow this without a full enumeration of possible functionalities.

The overheads due to programmability within the design-space were then discussed. Three regions of the multiplexer-based interconnect design-space were identified based upon the source of dominant cost: multiplexers, wires, or the fixed-function components. Each region possesses different scaling characteristics and desirable optimization criteria. A derivation was performed, based upon Rent's rule, to predict the boundaries between these regions based upon high-level fabrication parameters. Among other uses, these predictions were used to provide guidance for how many functionalities can be implemented with minimal overhead in a given implementation technology.

Chapter 4 documented the development of a computer-aided design tool that addresses the *a priori* subproblem. This tool is then applied to a software-defined radio application. The resulting programmable designs were then synthesized and overheads compared to fixed-function implementations in both standard-cell and

FPGA implementation technologies. A simple *a posteriori* capability was provided via the development of a simple microcontroller unit that can be programmed after the design is completed. The resulting overheads were then examined in the context of the predictions from Chapter 3, and bounds were extrapolated.

Chapter 5 provided a detailed discussion of prior work for the interested reader.

Appendix A

Case Study Source Code

A.1 CIC

```
module CIC(clk,rst,q,a,csync); // Cascaded Integrating-Comb
    filter
input clk;
input rst;
output signed [15:0] q;
input signed [15:0] a;
input csync;

reg csce;



/* Sign-extend input */

reg signed [32:0] i;

reg csync_delayed;



/********** Integrate ***********/

reg signed [32:0] iA0, iA1, iA2, iA3;

always @(posedge clk) begin
    if(rst) begin
```

```
      iA0<=0;
      iA1<=0;
      iA2<=0;
      iA3<=0;
      i<=0;
    end else begin
      i<=$signed({a,1'b1});
      iA0<=iA0+i;
      iA1<=iA1+iA0;
      iA2<=iA2+iA1;
      iA3<=iA3+iA2;
    end
end


/************ Comb *************/

reg signed [32:0] csacc;
wire signed [32:0] csin=csync_delayed?iA3:csacc;
reg signed [32:0] cs1,cs2,cs3,cs4;

always @(posedge clk) begin
  if (rst) begin
     csacc<=0;
     cs1<=0;
```

```
    cs2<=0;
    cs3<=0;
    cs4<=0;
  end else if (csce) begin
    cs1<=csin;
    cs2<=cs1;
    cs3<=cs2;
    cs4<=cs3;

    csacc<=csin-cs4;
  end
end


wire signed [32:0] qq;

assign qq=csacc+65536;
assign q=qq[32:17];


/********** Timing circuits ******/

srlecycleR #(4,17'h007) Icsce(clk,csce_or1,csync|csce_or1,
   rst);
```

```
always @(posedge clk) begin

  if(rst) begin
    csce<=0;
    csync_delayed<=0;
  end else begin
    csce<=csync|csce_or1;
    csync_delayed<=csync;
  end
end


endmodule
```

A.2 CORDIC

```
module CORDIC_serial(clk, rst, xo, yo, xi, yi, zi, ki);

input  clk;
input  rst;
output signed [15:0] xo, yo;
input  signed [15:0] xi, yi, zi, ki;



reg signed [15:0] x, y, z;


reg signed [14:0] r;


reg signed [31:0] xs, ys;


reg signed [15:0] zs;


always @(posedge clk) begin
  if(rst) begin
    r<=15'b010000000000000;
  end else begin
    r<={1'b0, r[0], r[13:1]};
    xs=r[13]?xi:((x*r)>>13);
    ys=r[13]?yi:((y*r)>>13);
    zs=r[13]?zi:z;
```

```
      if(r[13]) begin
        z<=zi;
        x<=xi;
        y<=yi;
      end else if(zs[15]) begin
        z<=zs+ki;
        x<=x+ys;
        y<=y-xs;
      end else begin
        z<=zs-ki;
        x<=x-ys;
        y<=y+xs;
      end
    end
  end

    assign xo=x;
    assign yo=y;

endmodule
```

A.3 CORDIC

```
// Implements a pipelined, complex, radix-2 FFT butterfly
module fftbutterfly32(clk,Cr,Ci,Dr,Di,Tr,Ti,Ar,Ai,Br,Bi);
  input clk;
  output signed [31:0] Cr,Ci,Dr,Di;       // Outputs
  input signed [15:0] Ar,Ai,Br,Bi;        // Inputs
  input signed [15:0] Tr,Ti;              // Twiddle factors

  wire signed [31:0] Mr,Mi;  // intermediate value

  reg signed [31:0] Cr,Ci,Dr,Di;

  assign Mr=Br*Tr-Bi*Ti;
  assign Mi=Br*Ti+Bi*Tr;

  always @(posedge clk) begin

    Cr<=Ar+Mr;
    Ci<=Ai+Mi;
    Dr<=Ar-Mr;
    Di<=Ai-Mi;
  end
endmodule
```

A.4 FIR

```
// Direct-form 3-tap FIR filter
module fir32(clk,z,x,h0,h1,h2);

input clk;
input signed [15:0] x, h0,h1,h2;
output signed [31:0] z;

reg signed [15:0] a,b;

reg signed [31:0] z;

always @(posedge clk) begin
  // Tapped-delay line
  a<=x;
  b<=a;

  // Compute output
  z<=x*h0+a*h1+b*h2;
end

endmodule
```

A.5 heterodyning

```
module hetero(clk,IN_I,IN_Q,LO_I,LO_Q,OUT_I,OUT_Q);
  input clk;
  input signed [15:0] LO_I,LO_Q,IN_I,IN_Q;
  output signed [31:0] OUT_I,OUT_Q;

  reg signed [31:0] P_II,P_QQ,P_IQ,P_QI;
  reg signed [31:0] OUT_I,OUT_Q;

  always @(posedge clk) begin
    /* Products */

    P_II<=LO_I*IN_I;
    P_QQ<=LO_Q*IN_Q;
    P_IQ<=LO_I*IN_Q;
    P_QI<=LO_Q*IN_I;

    /* Sum */
    OUT_I<=P_II-P_QQ;
    OUT_Q<=P_IQ+P_QI;
  end

endmodule
```

A.6 mcu

```
module mcu(input clk,
           input rst,

           output [8:0] ram_addra, ram_addrb,
           output  [31:0] ram_dia, ram_dib,
           input   [31:0] ram_doa, ram_dob,
           output ram_ena, ram_enb,
           output ram_wea, ram_web,

           input [31:0] io_di,
           output [31:0] io_do,
           output [8:0] io_addr,
           output io_os,
           output io_is
                  );



  assign ram_wea=0;
  assign ram_ena=1;
  assign ram_enb=1;
  assign ram_dia=0;
```

```
reg [31:0] p2_inst_word;
reg [31:0] p3_acc;


/* Pipeline stage 0 - Compute PC */

reg [8:0] p0_pc;

always @(posedge clk or posedge rst) begin
  if(rst) begin
    p0_pc<=0;
  end else begin
    p0_pc<=p0_pc+1;
    case(p2_inst_word[26:25])
      0: if(p2_inst_word[27])                  p0_pc<=
         p2_inst_word[24:16];             /* NOP/JMP */
      1: if(p2_inst_word[27]^(p3_acc[31]))  p0_pc<=
         p2_inst_word[24:16];             /* JB */
      2: if(p2_inst_word[27]^(p3_acc==0))   p0_pc<=
         p2_inst_word[24:16];             /* JZ */
    endcase
  end
end

/* Pipeline stage 1 - Instruction fetch */
```

```
wire [31:0] p1_inst_word;


assign ram_addra=p0_pc;  assign p1_inst_word=ram_doa;
//  always @(posedge clk) p1_inst_word<=ram[p0_pc];


/* Pipeline stage 2 - Load/Store */

wire [31:0] p2_operand_x;

always @(posedge clk or posedge rst) begin
  if(rst) begin
      p2_inst_word<=0;
  end else begin
      p2_inst_word<=p1_inst_word;

      /*
      p2_operand_x<=ram[p1_inst_word[8:0]];
      if(p1_inst_word[11])
        ram[p1_inst_word[8:0]]<=p3_acc;    // Note the 2
            cycle offset
      */
  end
```

```
end

assign ram_addrb=p1_inst_word[8:0];     assign p2_operand_x=
    ram_dob;
assign ram_dib=p3_acc;
assign ram_web=p1_inst_word[11];


/* Pipeline stage 3 - Execute */


always @(posedge clk or posedge rst) begin
   if(rst) begin
        p3_acc<=0;
   end else begin
         case(p2_inst_word[15:12])
             0:   p3_acc<=p3_acc;

                  /* NOP */
             1:   p3_acc<=p2_operand_x;
                                                          /* LD
                  */
             2:   p3_acc<=p3_acc+p2_operand_x;
                                                   /* ADD */
```

```
3:    p3_acc<=p3_acc-p2_operand_x;
                                              /* SUB */
4:    p3_acc<=p2_operand_x-p3_acc;
                                              /* RSB */
5:    p3_acc<=$signed(p3_acc[15:0])*$signed(
    p2_operand_x[15:0]);     /* MUL */
6:    p3_acc<={p3_acc[30:0],1'b0};
                                              /* SHL */
7:    p3_acc <={1'b0,p3_acc[31:1]};
                                              /* SHR */
8:    p3_acc<=~p3_acc;

    /* NOT */
9:    p3_acc<=p3_acc&p2_operand_x;
                                              /* AND */
10:   p3_acc<=p3_acc^p2_operand_x;
                                             /* XOR */
11:   p3_acc <=0;

   /* ZERO */
12:   p3_acc<={p3_acc[15:0],p3_acc[31:16]};
                                     /* XCH */
13:   p3_acc<=p2_operand_x -1;
                                                 /* DEC */
```

```
                14:    p3_acc<=io_di;

                   /* IN   */
                15:    p3_acc<=0;

                   /* ZERO */
              endcase
        end
      end

      /* IO Bus */

      assign io_addr=p2_inst_word[8:0];
      assign io_do=p3_acc;
      assign io_os=p2_inst_word[10];
      assign io_is=(p2_inst_word[15:12]==14);



endmodule
```

Appendix B

Case Study Dataflow Graphs

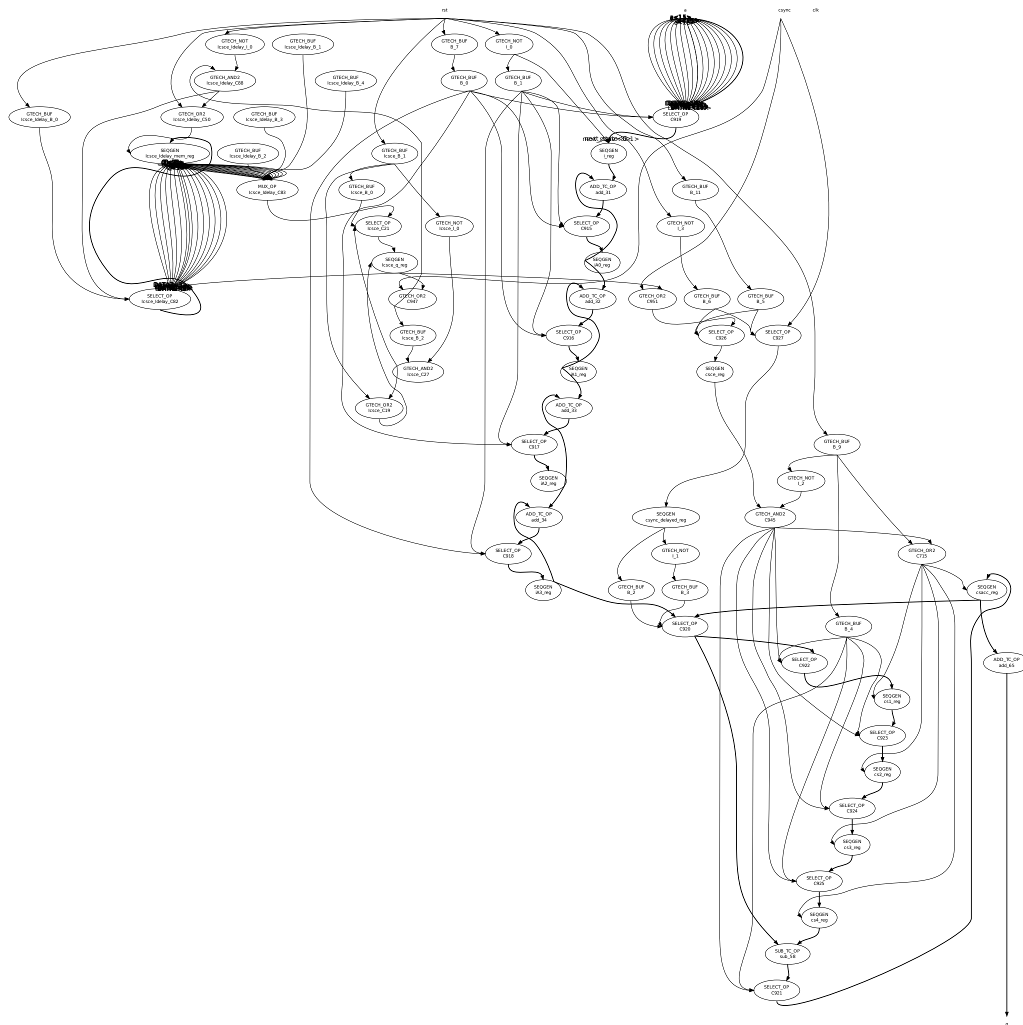

Figure B.1: CIC Data-flow Graph

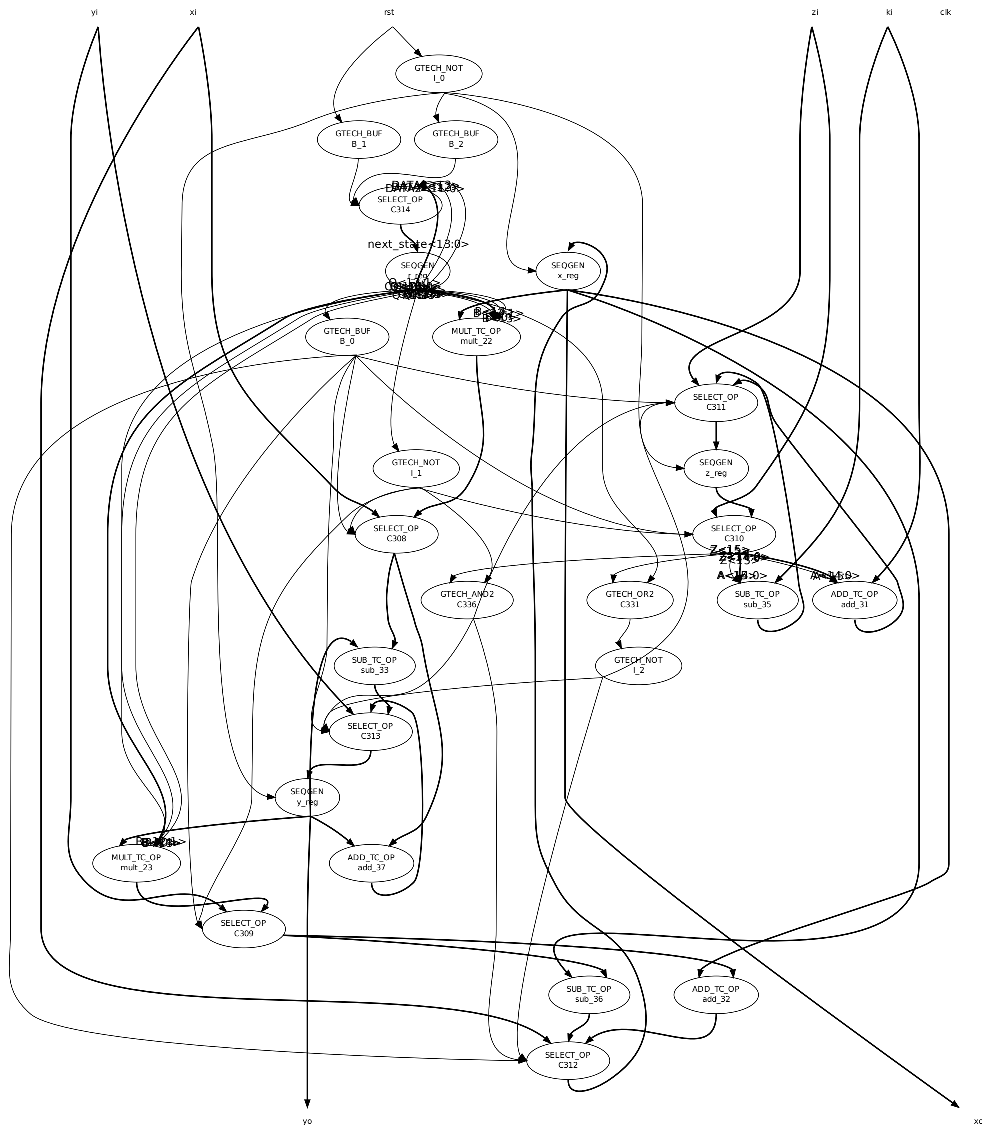

Figure B.2: CORDIC Data-flow Graph

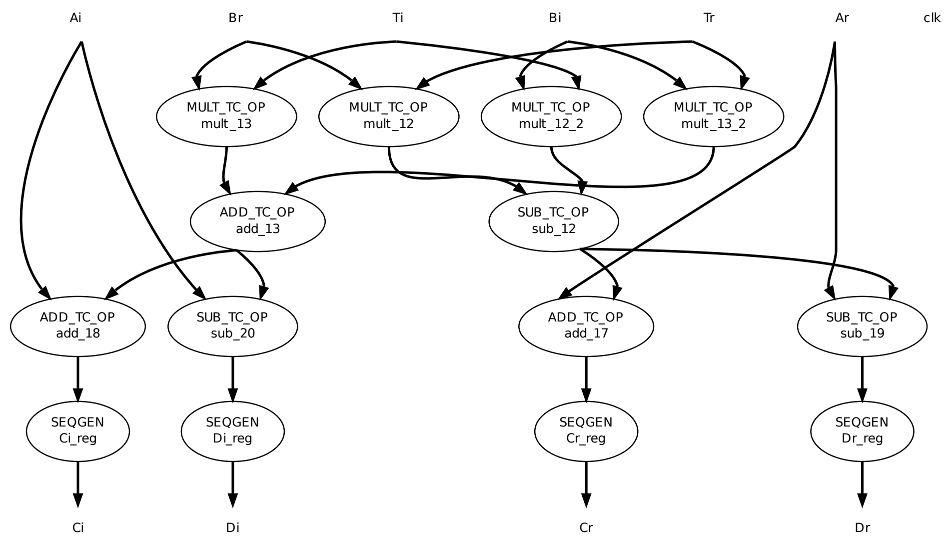

Figure B.3: FFT Data-flow Graph

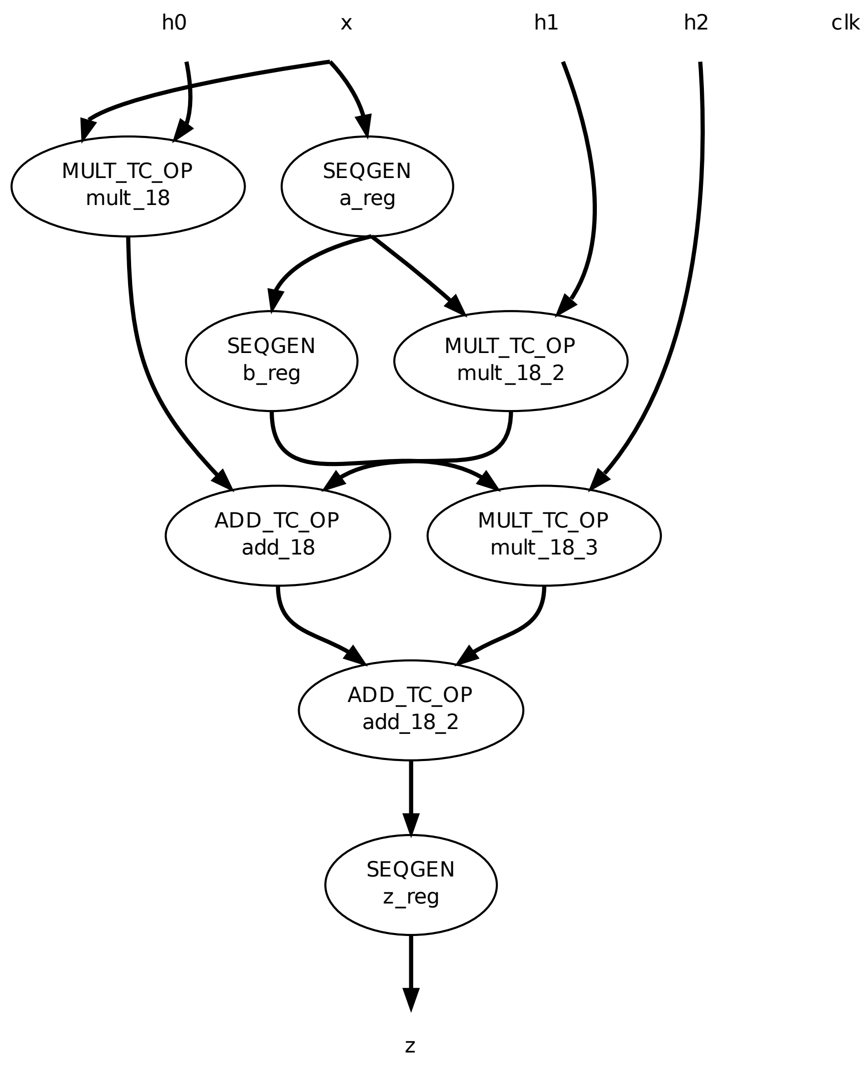

Figure B.4: FIR Data-flow Graph

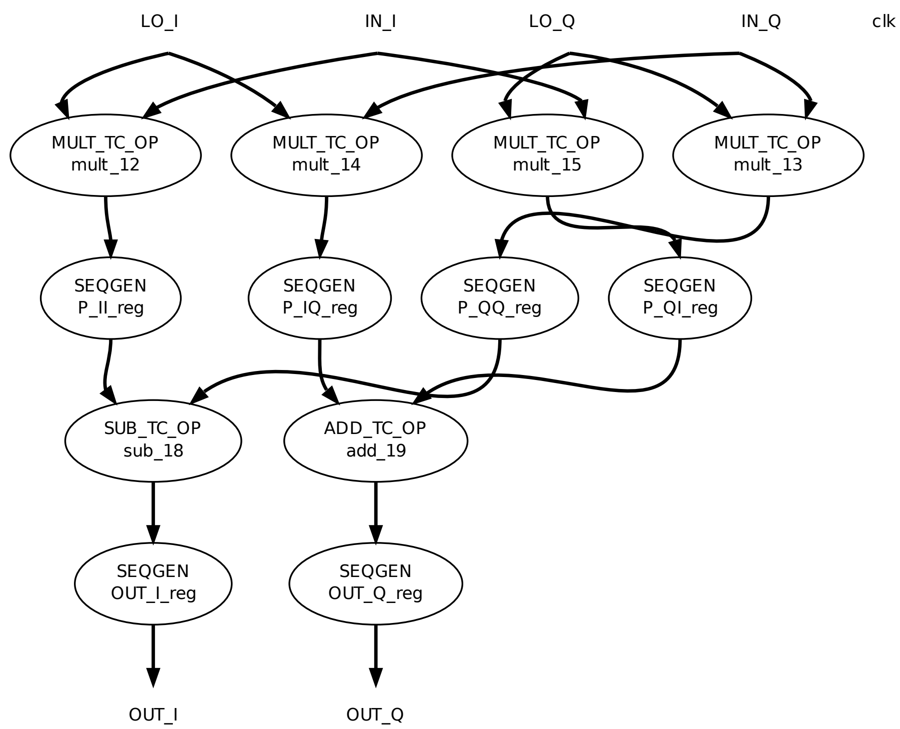

Figure B.5: Heterodyning Data-flow Graph

Bibliography

[1] Generating functionally equivalent FPGAs and ASICs with a single set of RTL and synthesis/timing constraints. Technical report, Altera, Inc., 2009.

[2] Oxford english dictionary, 2010.

[3] C. Andriamisaina, E. Casseau, and P. Coussy. Synthesis of Multimode digital signal processing systems. In *Proceeding of Adaptive Hardware and Systems NASA/ESA Conference on Adaptive Hardware and Systems*, page 7, Edinburgh Royaume-Uni, 2007. AHS.

[4] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. CHIPS: Custom hardware instruction processor synthesis. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 27(3):528, 2008.

[5] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[6] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. Technical report, Algorithmica, 2001.

[7] L. Bertrand and E. CASSEAU. Automated multimode system design for high

performance DSP applications. In *Proceedings of the 17th European Signal Processing Conference (EUSIPCO 2009)*, pages 1289–1293, 2009.

[8] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[9] M. Breuer. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*, pages 284–290. IEEE Press, 1977.

[10] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st annual conference on Design automation*, pages 395–400. ACM New York, NY, USA, 2004.

[11] P. K. Chan and M. D. F. Schlag. Acceleration of an FPGA router. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 175, Washington, DC, USA, 1997. IEEE Computer Society.

[12] C. Chavet, C. Andriamisaina, P. Coussy, E. Casseau, E. Juin, P. Urard, and E. Martin. A design flow dedicated to multi-mode architectures for DSP applications. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 604–611. IEEE Press, 2007.

[13] L.-Y. Chiou, S. Bhunia, and K. Roy. Synthesis of application-specific highly-efficient multi-mode systems for low-power applications. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10096, Washington, DC, USA, 2003. IEEE Computer Society.

[14] L.-y. Chiou, S. Bhunia, and K. Roy. Synthesis of application-specific highly

efficient multi-mode cores for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 4(1):168–188, 2005.

[15] S. Choi, R. Scrofano, V. Prasanna, and J. Jang. Energy-efficient signal processing using FPGAs. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 225–234, New York, NY, USA, 2003. ACM.

[16] P. Christie and D. Stroobandt. The interpretation and application of rent's rule. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(6):639 –648, dec 2000.

[17] K. Compton. *Architecture Generation of Customized Reconfigurable Hardware.* PhD thesis, Northwestern University, 2003.

[18] K. Compton and S. Hauck. Totem: Custom reconfigurable array generation. *IEEE Symposium on FPGAs for Custom Computing Machines*, 00:111–119, 2001.

[19] K. Compton and S. Hauck. Flexibility measurement of domain-specific reconfigurable hardware. *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 155–161, 2004.

[20] K. Compton and S. Hauck. Automatic design of area-efficient configurable ASIC cores. *IEEE Transactions on Computers*, 56(5):662–672, 2007.

[21] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *ARVLSI '99: Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, page 23, Washington, DC, USA, 1999. IEEE Computer Society.

[22] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40, 1999.

[23] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, 2008.

[24] J. Davis, V. De, and J. Meindl. A stochastic wire-length distribution for gigascale integration (GSI)Part I: Derivation and validation. *IEEE Transactions on Electron Devices*, 45(3), 1998.

[25] J. Davis, V. De, and J. Meindl. A stochastic wire-length distribution for gigascale integration (GSI)Part II: Applications to clock frequency, power dissipation, and chip size estimation. *IEEE Transactions on Electron Devices*, 45(3), 1998.

[26] A. DeHon. *Reconfigurable architectures for general-purpose computing.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1996.

[27] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 69–78, 1999.

[28] A. DeHon, R. Huang, and J. Wawrzynek. Hardware-assisted fast routing. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 205, Washington, DC, USA, 2002. IEEE Computer Society.

[29] W. Donath. Placement and average interconnection lengths of computer logic. *Circuits and Systems, IEEE Transactions on*, 26(4):272–277, Apr 1979.

[30] W. Donath. Wire length distribution for placements of computer logic. *IBM Journal of Research and Development*, 25(2-3):152–155, 1981.

[31] A. Dunlop and B. Kernighan. A procedure for placement of standard cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4(1):92–98, 1985.

[32] C. Ebeling. Rapid-c manual. *University of Washington Technical Report UW-CSE-02-07-06*, 2002.

[33] C. Ebeling, D. Cronquist, and P. Franklin. RaPiD–reconfigurable pipelined datapath. *6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, 1996.

[34] H. Fan, Y. Wu, and C. Cheung. Design Automation for Reconfigurable Interconnection Networks. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 244–256, 2010.

[35] H. Fleisher and L. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19(2):98–109, 1975.

[36] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992.

[37] W. Geurts, F. Catthoor, S. Vernalde, and H. De Man. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications.* Kluwer Academic Pub, 1997.

[38] S. Ghiasi, E. Bozorgzadeh, P. Huang, R. Jafari, and M. Sarrafzadeh. A unified theory of timing budget management. *IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 25(11):2364, 2006.

[39] M. Gokhale, P. Graham, E. Johnson, N. Rollins, and M. Wirthlin. Dynamic reconfiguration for management of radiation-induced faults in FPGAs. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 26-30 April 2004.

[40] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 261, 1999.

[41] R. Gupta and F. Brewer. High-Level Synthesis: A Retrospective. *High-Level Synthesis*, pages 13–28, 2008.

[42] S. Gupta. *Spark: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[43] L. Hagen, A. B. Kahng, F. Kurdahi, and C. Ramachandran. On the intrinsic rent parameter and spectra-based partitioning methodologies. *IEEE Trans. on Comput.-Aided Des., Integrated Circuits and Systems*, 13:27–37, 1994.

[44] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[45] 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, Piscataway, NJ, USA, 2001. IEEE Press.

[46] S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips, and A. Sharma. Totem: Domain-Specific Reconfigurable Logic. *submitted to IEEE Transactions on VLSI*, 2008.

[47] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[48] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu. Data path allocation based on bipartite weighted matching. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 499–504, New York, NY, USA, 1990. ACM.

[49] Z. Huang and S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. *Design, Automation and Test in Europe Conference and Exhibition*, 0:0735, 2001.

[50] A. ISO. IEC 9899: 1999, Programming Languages–C. *International Organization for Standardization*, 1999.

[51] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, page 13. IEEE Computer Society, 1998.

[52] E. Keller. JRoute: A run-time routing API for FPGA hardware. *7th Reconfigurable Architectures Workshop*, pages 874–881, 2000.

[53] E. Keller and S. McMillan. An FPGA Wire Database for Run-Time Routers. In *Proceedings on the 2002 International Conference on Military Applications of Programmable Logic Devices (MAPLD02)*. Citeseer, 2002.

[54] V. Kianzad and S. Bhattacharyya. CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP04)*. Citeseer, 2004.

[55] V. V. Kumar and J. Lach. Highly flexible multi-mode system synthesis. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 27–32, New York, NY, USA, 2005. ACM.

[56] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM Press.

[57] P. Kwan and C. T. Clarke. FPGAs for improved energy efficiency in processor based systems. *Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005, Singapore, October 24-26, 2005: Proceedings*, 2005.

[58] B. Landman and R. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20:1469–1479, December 1971.

[59] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *In International Symposium on Microarchitecture*, pages 330–335, 1997.

[60] F. Li, Y. Lin, L. He, D. Chen, and J. Cong. Power modeling and characteristics of field programmable gate arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(11):1712–1724, 2005.

[61] Y. W. S. T. M. Marek-Sadowska. On computational complexity of a detailed routing problem in two dimensional FPGAs. *VLSI, 1994. 'Design Automation of High Performance VLSI Systems'. GLSV '94, Proceedings., Fourth Great Lakes Symposium on*, pages 70–75, 4-5 Mar 1994.

[62] M. McFarland, A. Parker, and R. Camposano. Tutorial on high-level synthesis. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 330–336, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[63] N. Moreano, G. Araujo, Z. Huang, and S. Malik. Datapath merging and interconnection sharing for reconfigurable architectures. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 38–43, New York, NY, USA, 2002. ACM.

[64] N. Moreano, E. Borin, C. D. Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. In *in IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 969–980, 2005.

[65] K. Parnell and R. Bryner. Comparing and contrasting FPGA and microprocessor system design and development. Technical report, Xilinx, 2004.

[66] H. Parvez, Z. Marrakchi, and H. Mehrez. Application Specific FPGA Using Heterogeneous Logic Blocks. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 92–109, 2010.

[67] J. Pistorius and M. Hutton. Placement rent exponent calculation methods, temporal behaviour and FPGA architecture evaluation. In *SLIP '03: Proceedings of the 2003 international workshop on System-level interconnect prediction*, pages 31–38, New York, NY, USA, 2003. ACM Press.

[68] A. Poetter. JHDLBits: An open-source model for FPGA design automation. Master's thesis, Virginia Polytechnic Institute and State University, 2004.

[69] K. Poon, S. Wilton, and A. Yan. A detailed power model for field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(2):279–302, 2005.

[70] W. Rudin and J. Cofman. *Principles of mathematical analysis*. McGraw-Hill New York, 1964.

[71] M. Rullmann and R. Merker. Maximum edge matching for reconfigurable computing. In *Reconfigurable Architectures Workshop at 13th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), Rhodes, Greece*. Citeseer, 2006.

[72] M. Rullmann, R. Merker, H. Hinkelmann, P. Zipf, and M. Glesner. An Integrated Tool Flow to Realize Runtime-Reconfigurable Applications on a New Class of Partial Multi-Context FPGAs. In *Proc. 19th Intl. Conf. on Field Programmable Logic and Applications*, 2009.

[73] M. Schmitz, B. Al-Hashimi, and P. Eles. Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(2):153–169, 2005.

[74] N. Selvakkumaran and G. Karypis. Multi. Objective Hypergraph Partitioning Algorithms for Cut and Maximum Subdomain Degree Minimization. 2003.

[75] N. Shirazi, W. Luk, and P. Cheung. Automating production of run-time reconfigurable designs. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:147, 1998.

[76] L. Singhal and E. Bozorgzadeh. Multi-layer floorplanning on a sequence of reconfigurable designs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL06)*, pages 605–612. Citeseer, 2006.

[77] A. Smith. Universality of wolframs 2, 3 turing machine. *Submitted for the Wolfram 2, 3 Turing Machine Research Prize http://www.wolframprize.org*, 2007.

[78] A. Smith, G. Constantinides, and P. Cheung. Fused-Arithmetic Unit Generation for Reconfigurable Devices using Common Subgraph Extraction. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 105–112, 2007.

[79] C. C. d. Souza, A. M. Lima, G. Araujo, and N. B. Moreano. The datapath merging problem in reconfigurable systems: Complexity, dual bounds and heuristic evaluation. *J. Exp. Algorithmics*, 10:2.2, 2005.

[80] N. Steiner. A standalone wire database for routing and tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs. Master's thesis, Virginia Polytechnic Institute and State University, 2002.

[81] N. Steiner and P. Athanas. An alternate wire database for Xilinx FPGAs. *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 336–337, 2004.

[82] D. Stroobandt. Improving Donath's technique for estimating the average interconnection length in computer logic. 1996.

[83] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi. Cacti 5.1. Technical report, Hewlett-Packard Development Company, 2008.

[84] F. Toth. The Easy Path to Cost Reductions. *Xcell Journal*, pages 72–74, 2004.

[85] S. Trimberger, X. Inc, and C. San Jose. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, 1993.

[86] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[87] A. van der Werf, M. J. H. Peek, E. H. L. Aarts, J. L. van Meerbergen, P. E. R. Lippens, and W. F. J. Verhaegh. Area optimization of multi-functional processing units. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 292–299, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[88] J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[89] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: anytime anywhere anyway signal processing. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 128–139. ACM, 2009.

[90] C. Wolinski, K. Kuchcinski, and E. Raffin. Automatic design of application-

specific reconfigurable processor extensions with upak synthesis kernel. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1):1–36, 2009.

[91] M. Zuluaga and N. Topham. Resource sharing in custom instruction set extensions. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors.(Jun. 2008)*, 2008.

Vita

Johnathan Andrew York attended Stratford High School in Houston, Texas. In 1999, he entered the University of Texas at Austin. He received a Bachelor of Science in Electrical Engineering with honors. He continued his study at the University of Texas at Austin, receiving a Master of Science in Engineering in 2004. Johnathan has been employed in various positions at the University of Texas Applied Research Laboratories since 2001.

Permanent Address: 801 Stevenage Drive
Pflugerville, TX 78660

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.