A HIGH LEVEL SYNTHESIS APPROACH FOR REDUCED INTERCONNECTS AND FAULT TOLERANCE

A HIGH LEVEL SYNTHESIS APPROACH FOR REDUCED INTERCONNECTS AND FAULT TOLERANCE

BY
DAVID LEMSTRA, B. ENG. & MGT. (COMPUTER)
JANUARY 2005

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

MASTER OF APPLIED SCIENCE (2005) McMaster University
(Electrical and Computer Engineering) Hamilton, Ontario

TITLE:	A High Level Synthesis Approach for Reduced Interconnects and Fault Tolerance
AUTHOR:	David Lemstra, B. Eng. & Mgt. (Computer)
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xiii, 109

Abstract

High Level Synthesis (HLS) is a promising approach to managing design complexity at a more abstract level as integrated circuit technology edges deeper into sub-micron design. One useful facet of HLS is the ability to automatically integrate architectural components that can address potential reliability issues, which may be on the increase due to miniaturization . Research into harnessing HLS for fault tolerance (FT) has been progressing since the early 1990's. There currently exists a large body of work regarding methods to incorporate capabilities such as fault detection, compensation, and recovery into HLS design.

While many avenues of FT have been explored in the HLS environment, very little work has considered the effectiveness and feasibility of these techniques in the context of large HLS systems, which presumably is the *raison d'etre* of HLS. While existing HLS FT approaches are often elegant and involve highly sophisticated techniques to achieve optimal solutions, the costs of HLS infrastructure in regards to scalability are not well reported. The intent of this thesis is to explore the ramifications of applying common HLS techniques to large designs.

Furthermore, a new HLS tool entitled RIFT is presented that is specifically designed to mitigate infrastructure costs that mount as greater parallelism is utilized. RIFT is named for its design philosophy of "Reducing Interconnects for Fault Tolerance". RIFT iteratively builds a logical hardware representation, which consists of both the components instantiated and their interconnections, one operation at a time. It chooses the next operation to be "mapped" to the burgeoning design based on scheduling constraints as well as the extra hardware and interconnect costs required to support a particular selection. Emphasis is placed on minimizing the delay of the datapath in effort to reduce the performance cost associated with the extra interconnects needed for FT. RIFT has been used to generate efficient solutions for FT designs requiring as many as a thousand operations.

Acknowledgments

I would like to thank the numerous people who helped and encouraged me during the time that this research was completed. First, I would like to acknowledge my supervisor, *Dr.* Nicola Nicolici. His support and enthusiasm is a major contributing factor to the completion of this thesis. His dedication to relevance and quality of work, as opposed to extraneous issues, is admirable. My colleagues, as we have moved from CRL to BSB and finally ITB, have been especially gracious in putting up with me, my distorted humor, and my pranks. Henry Ko has, in a sense, led the way for me, and often keeps Nicola occupied so I can work! He was also the designer of RIFT's Verilog parser (which has no bugs, of course). Thank you for insulating me from *lex* and *yacc*. Much appreciation goes to Dr. Shirani and Dr. deBruin for taking the time to review this (yet another) thesis. They suffered through it without a list of abbreviations. My apologies for this grievous oversight. My love and appreciation goes to my parents for their understanding and support over my entire academic "career". And finally, I wish to acknowledge all those friends with whom I've shared down-time during these last few years. I am indebted to you for my sanity.

Terms & Abbreviations

alter ego	a value produced on the other side of a control branch
ASAP	As Soon As Possible
ALAP	As Late As Possible
CAD	Computer Automated Design
CDFG	Control Data Flow Graph
CED	Concurrent Error Detection
CPU	Central Processor Unit
DFG	Data Flow Graph
DMR	Double Modular Redundancy
DSP	Digital Signal Process(or/ing)
EDA	Electronic Design Automation
FCR	Failover Control Register
FFSM	Failover Finite State Machine
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FT	Fault Tolerance
FU	Functional Unit
HDG	Hardware Description Graph
HDL	Hardware Description Language
HLS	High Level Synthesis
IC	Integrated Circuit
LO	Logical Operation

LUT	Look Up Table
lower buddy	redundant FU for upper buddy in failover mode
mux(es)	multiplexor(s)
redux	redundant multiplexor
RIFT	Reduced Interconnects and Fault Tolerance
SEU	Single Even Upset
STAR	Self Testing ARea
TMR	Triple Modular Redundancy
upper buddy	an FU protected by an adjacent lower buddy
VHDL	Very High Speed IC HDL
VLSI	Very Large Scale IC
XOR	exclusive OR

Contents

List of Tables

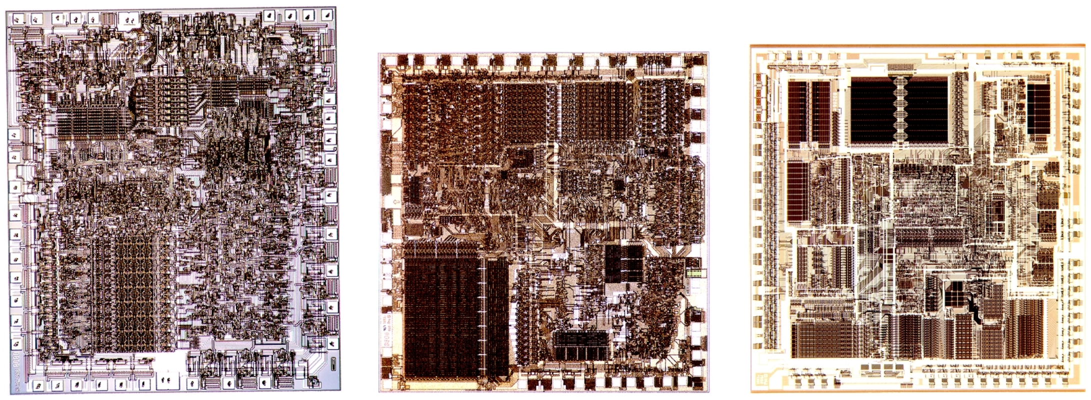
List of Figures

Chapter 1

An Introduction to High Level Synthesis

1.1 Introduction

As a consequence of rapid advances in semiconductor manufacturing, very large scale integrated (VLSI) circuit design is beginning to push conventional design methods to their limits. This trend is steadily motivating the case for VLSI design to be carried out at a higher level of abstraction than is currently the norm in an effort to mitigate rapidly growing complexity. Another consequence of rapidly decreasing process sizes is a greater exposure to long term reliability issues [13]. As such, much research has been focused on architectural solutions that create more robust circuits. The focus of this research is the efficient insertion of Fault Tolerance (FT) in VLSI designs at an abstract level in such a way as to transparently identify and compensate for latent defects. After this chapter, current approaches to automated FT are surveyed and then a new FT approach, the subject of this research, will be presented. First, however, this chapter will review some of the historical forces that have brought us to the current level of abstraction. Then the principles underlying High Level Synthesis (HLS), and the general HLS techniques on which most work is based will be introduced.



(a) 1974: 8080 (b) 1978: 8088 (c) 1982: 286

Figure 1.1: Early Intel ICs: A progression of EDA induced organization[a]

[a]Intel Microprocessor Hall of Fame (intel.com/intel/intelis/museum/online/hist_micro/hof)

1.2 History of Abstraction in IC Design

The beginning of Electronic Design Automation (EDA) commenced soon after the emergence of the integrated circuit in the 1960's. The very first use of EDA was essentially as electronic drawing aides for interactive design [41]. Then, in the 1970's, forays were made into the automation of interconnect insertion, an especially tedious task, and the placement of transistors. The first level of abstraction achieved was the introduction of the standardized cell. Before this, transistors had been manually placed as needed, but the use of cells allowed for logic design to be abstracted from transistor layout. About this time, the first significant conflict originating from the use of EDA arose. Standardized cells required interconnects to be placed in channels between columns of abutting cells. Because this is a well characterized problem, standardized cells and channel routing was much more amendable to place and route automation, especially considering the computing resources available at that time. EDA critics correctly pointed out, however, that the resulting circuits were inferior in performance and area to unstructured Integrated Circuits (IC) designed manually by skilled designers. Two factors ushered in the acceptance of automated placement and routing. First, the construction of cell libraries did have a greater initial cost. This, however, was mitigated by the fact that having performance characteristics of the cells readily available made for easier simulation. Secondly, as process technologies shrank and complexity grew, the manual design process simply could not keep up with the demand for new products. The result was a rapid shift to place and route EDA methodologies in 1980 [27].

The next jump in abstraction was achieved with Hardware Description Languages (HDL). The creation of first HDLs in the early 1970s was driven by the need to represent gates at a higher level as IC capacity rose past tens of thousands of gates. At first these tools were not so much used for design, but for the simulation of designs [5]. HDLs allowed for more efficient simulation through use of multi-bit operations, busses, and registers and also supported the use of programming control constructs. Collectively, these began to be referred to as Register Transfer Level (RTL) models. Many different HDLs were developed, but in the mid 1980s, Verilog HDL and Very High Speed IC HDL (VHDL) became predominant. While originally most HDLs modeled ICs using concurrent modeling structures, the concept of sequential assignments was eventually introduced. This in turn eventually led to products, starting in the late 1980s, that could synthesize a gate level net-list from the original RTL representation [41]. From there, a design would go through an increasingly automated design flow that, from the mid 1980's onwards, commonly included separate stages for floor planning, global routing, detailed cell placement, local routing, and finally transistor level layout. The RTL level of abstraction is the starting point from which most mainstream digital IC development is done in the current day.

Designing at the RTL level constitutes a significant time savings over less abstract methodologies. There are several key components that allow for abstract design. Multi-bit values, or vectors, can be specified as logical variables, much as in software programming. Assignment statements generally describe how a desired result is obtained by manipulating input arguments. In general, if a statement result is stored to a register, it is thought of as a variable, otherwise it is thought of as a continuous signal. Concurrent constructs can be used to represent concurrently executed hardware components. Outside these constructs, statements are generally used to create combinational logic networks, or signals. Inside the constructs, which are generally triggered by a clock or other signal, a series of sequential statements can be made using variables and signals to generate results that are stored with a latch or flip-flop. Thus the term RTL reflects a specification of how values are to be manipulated as they are transfered from one combinational signal or storage register to the next. Control structures, such as 'If-Then-Else' statements, make it much easier to abstractly define Finite State Machines (FSM), which can be used to control a datapath. Various automated tools exist to convert these abstract FSMs into efficiently encoded control logic in hardware. Loops can also be used to iteratively define hardware for synthesis, as opposed to iteratively processing data at runtime as is the case in the context of software programming. Designs can also be parameterized in conjunction with vector notation and 'For' loops. Once a component has been designed, it can be reused in a hierarchal manner, which is possible only because it is an abstract, logical representation. The advantages of RTL allow the designer to more easily specify the behavior of a circuit, as opposed to the circuit itself, and utilize the EDA tool flow to work out the lower level details such as timing, placement, and routing. This level of abstraction and automation is essential as the design space moves into the era of ICs with hundreds of millions of transistors.

1.3 The Next Level

At the time of this writing, RTL synthesis has been used for well over a decade and a half. The exponential nature of Moore's law, which states that gate density will double every 18 months [44], suggests that capacity has increased three orders of magnitude in that time. RTL design principles are being strained and having difficulty coping on their own and several new paradigms are gaining currency as a result. In 1980, Paul Russo noted in [54] that increased capacity was allowing the different subsystems of what is now the Central Processor Unit (CPU) to be brought onto a single piece of silicon. The idea of "Systems on a Chip" is much the same, however, now the idea involves moving multiple specialized computing "cores" onto a single chip, such as microprocessors, Digital Signal Processing (DSP) cores, encryption cores, and the like [53]. A further extension of this idea is the

”Network on a Chip” concept, which involves a standardized communication network between cores [7]. These approaches allow for utilization of increased capacities while avoiding the recurrent engineering costs of building those functionalities from scratch. Indeed, leading CPU manufacturers Intel, Sun, and AMD are not using the next iteration of Moore’s law to make further architectural advancements as per usual. Instead they are instantiating multiple instances of their previous designs onto a single die [3, 26, 57]. It is true that this change in direction is partly due to heat management issues because leakage current has not scaled down as well as feature size and clock frequency. However, this could also be construed as a obvious indication that production capacities are beginning to eclipse design capabilities by a large margin.

1.4 High Level Synthesis

Another approach to utilizing capacity is called either Behavioral Synthesis or High Level Synthesis (HLS). HLS describes a broad range of methods to achieve different design goals, but most HLS systems accept a behavioral description of the work to be executed. Then, using predefined components, the HLS tool assembles a structural description of a circuit in RTL that satisfies area and timing constraints and that can be synthesized. Most approaches create a design that completes the required processing over multiple clock cycles, which allows components to be reused. Some research, however, has explored the development of micro-architectures as well [18]. It is commonly the minimization of either area, latency, clock period, or a combination thereof that drives the HLS process. By changing the constraints, multiple design variations can quickly be generated and compared on the basis of area or latency and sometimes the clock period. If a particular design has at least one metric that is better than in all other designs, it is said to be Pareto optimal. Because IC design is often about making tradeoffs between several metrics, automated system level design can be used to quickly identify Pareto optimal designs from which the designer can select the most appropriate. HLS thus not only accelerates IC design, it can also be considered as an automated design space exploration tool that is built on top of the

RTL design flow, much the way RTL design was built upon the previous generation of placement and routing tools. The rest of the chapter is a brief survey of issues and methods of HLS systems. Much is based on De Micheli's authoritative text, "Synthesis and Optimization of Digital Circuits" [42], and should be referred to for a deeper treatment HLS fundamentals.

1.5 Behavioral Description

There are different perspectives on which format HLS should interpret the behavioral description given to it. Some promote abstract extensions based on current HDLs, such as SystemVerilog. Others have promoted the use of system level description tools, such as SystemC, which are derived from programming languages [17, 39]. Some see no reason to try to express hardware and its idiosyncrasies in terms of a computer science language. The other perspective is that such languages or HDL combinations would ease the burden of design for those unfamiliar with hardware principles, or perhaps allow hardware software co-design tools to more easily partition the two in a continuous fashion [43]. Irrespective of where the behavioral description originates, it can be parsed into what is known as a Data Flow Graph (DFG) as seen in Figure 1.2. The vertices depict the flow of data into and out of nodes, which are called Logical Operations (LO) and represent the processing or manipulation of the input data. The DFG is also useful in determining the precedence of data operations in that a node cannot begin execution until all of its ancestor nodes have generated their outputs. The DFG can be extended to a Control Data Flow Graph (CDFG) which also depicts control structures such as 'If-Then-Else' branches or 'For' loops. These graphs represent the processing that will need to be accomplished by the HLS design when completed, but do not represent the hardware implementation of the final design.

1.6 HLS Hardware Resources

Components used in HLS fall into several classes. Functional resources are perhaps the most important class, consisting of different types of data manipulation units, such as adders, multipliers, other mathematical operators, or custom logic modules. These generally are provided as part of a library or are supplied by the designer. Specifics such as the number of operands, commutativity of the operands, the number of clock cycles the unit requires, and so forth are assumed to be known. An instantiation of a particular functional resource is referred to as a Functional Unit (FU) henceforth and is thought of as both a data consumer and a data producer.

Registers are a second class of resources. They are used to store values from any data source, be they inputs, FUs, or other registers. When they are suppling data to FUs, other registers, or outputs, they are thought of as sources. When they are accepting data, they act as sinks.

Inputs and outputs allow new data into the system and calculated data out. They tend to be a trivial aspect in the realization of a complex HLS design.

Finally, when an HLS system is completely specified, wiring is needed to connect inputs, registers, and FUs together. Multiplexors (muxes) allow a data value to be selected from among several sources. In many HLS systems, FU related costs are assumed to be much greater than the costs associated with wiring and muxes. This is because adding infrastructure to share a particular FU would not be worthwhile if several instances of that FU required less area than the shared HLS solution. Thus the costs of interconnects are often ignored.

1.7 HLS Timing

Execution time in an HLS design system is usually described in terms of clock cycles. If time is to be either optimized or constrained, it is most often done in terms of the number of clock cycles. A time constrained approach will generally add FUs only as needed to meet that timing constraint while minimizing the number of FU instances, and thus area, needed. In some systems, a technique called "chaining" is used whereby operations with short delays are "chained" together such that they process data sequentially within a single clock step. This is particularly useful if FUs with long delays are being used concurrently with shorter operations. In this case, the measure of the clock period is also important and usually included in performance measures. Wiring and muxes will also affect the clock period, but is often considered to a lesser extent.

1.8 HLS Metrics

There are many different strategies that fall under the scope of HLS. There are several different criteria for HLS metrics, the most common being area, latency, and the clock period. Different approaches will limit one factor or more and seek to optimize the remaining metrics. At the HLS abstraction level, the absolute characteristics of different hardware components may or may not be known, however, the relative characteristics should be available. For instance, use of a carry look ahead adder should be faster than a ripple carry adder, though the latter will require more area. Frequently, area estimation relies simply on how many instances of a particular resource unit is needed by comparative designs: more instances of a unit type will require more area. Thus many HLS strategies limit or optimize the number of instances needed, as opposed to actual area. Some HLS systems may try to alter the original behavioral specification to reduce the number of FUs required [10]. For instance, using associativity and commutativity laws may result in a variation that requires fewer expensive units. The following transform would require both less operations and less multiplications, which usually is relatively expensive compared to addition.

$$a * c + b * c \longrightarrow (a + b) * c$$

Timing, in terms of the number of clock cycles, is usually a very well defined aspect of a problem and can be given as a direct constraint. The period, however, is difficult to estimate. The period depends, not only on the components, wiring, and muxes utilized, but also on the what type of technology the final design will be compiled on. Because the HLS CAD tool's final output is an RTL output, the actual period timing is not known until the RTL is compiled using an external tool flow. For these reasons, it can be convenient to ignore clock period and instead make comparisons of work based on the number of resources and clock cycles needed.

1.9 HLS Methodology

The transformation of the CDFG into hardware model is considered to be an NP-hard problem [52], and thus is often segregated into stages [25]. Each stage discussed here is depicted with a small 5-Tap Finite Impulse Response (FIR) filter example in Figures 1.2 to 1.5. Additions require one clock cycle, while multiplications require two. Scheduling is the process of determining the clock cycle in which each LO will be executed (Figure 1.5). Determining which FU each LO will be executed on is called binding (Figure 1.3). For convenience, the steps together will be referred to as mapping. Once mapping has been completed, the lifetime of each value produced by each FU in each clock cycle is fully defined. Register mapping determines when and to what physical register each value will be mapped. Register multiplexing is often used to reduce the actual number of physical registers required (Figure 1.4). Once FU mapping and register mapping is completed, the data-flow of the entire system is fully specified. At this stage wiring and muxes can be added as needed to facilitate the specified data-flow. The completely specified solution is then also used to construct an FSM to control data-flow, register enables, and conditional executions, where the main state is equivalent to the clock cycle count.

1.9.1 Scheduling

Much research has been done into various HLS scheduling algorithms and related optimizations. Scheduling is the determination of when each LO will be executed. The simplest may be the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) algorithms, which is overlaid onto the CDFG in Figure 1.2. ASAP scheduling sets each LO with no precedent values to clock 1. Each subsequent node is scheduled in the clock after its latest dependency is generated. The longest path from a starting

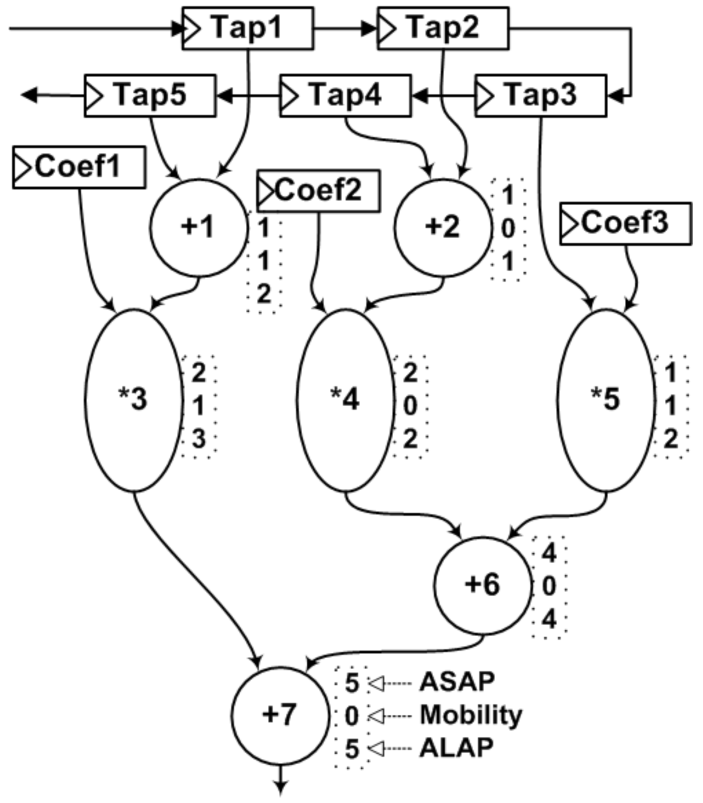


Figure 1.2: A CDFG for 5 tap FIR with ASAP, Mobility, & ALAP

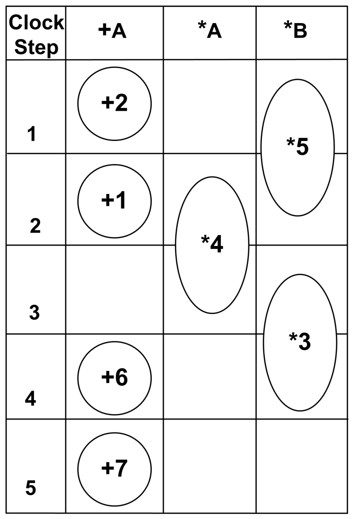


Figure 1.3: A binding table with 1 adder and 2 multipliers

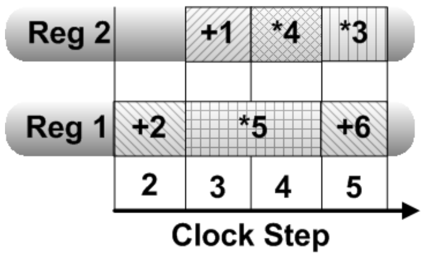


Figure 1.4: Assignment of values to registers by Left Edge Sort

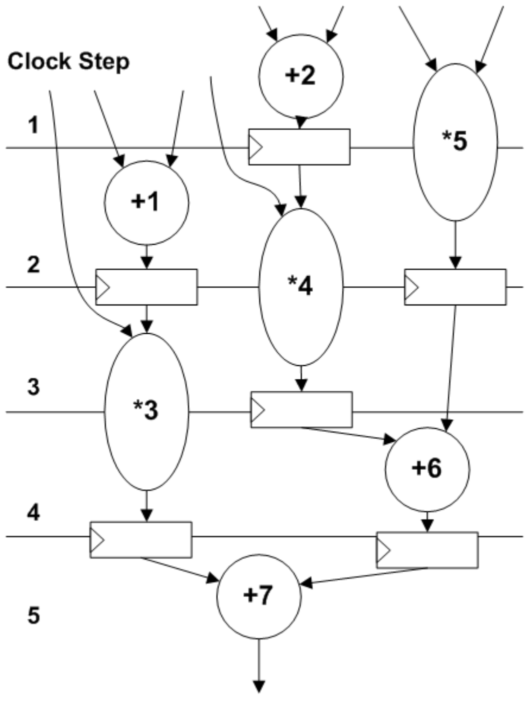


Figure 1.5: Scheduling LOs to clock steps

LO scheduled in the first cycle to the final output determines the minimum number of clock cycles needed and thus the critical path, which is the lower latency bound. ALAP starts by scheduling the last LO with either the latency constraint, or, if not given, the critical path latency found by ASAP. Each LO is then scheduled "backwards" such that it produces its value immediately before the earliest child needs it. As with ASAP, LOs that require multiple clock cycles to execute must also be accounted for. The difference between the ASAP and ALAP scheduling for each LO is called the slack or mobility, and reflects the different clock cycles that may be scheduled. The ASAP and ALAP algorithms are used as a starting point for several other scheduling algorithms [25].

If the goal is strictly the minimization of latency, only ASAP scheduling is needed. More elaborate approaches are required to satisfy latency constraints while also minimizing the total number of FUs required. List scheduling is a classic approach that iteratively schedules LOs one clock cycle at a time [14]. In each clock step, a list of "eligible" LOs, those with satisfied precedences, is chosen from based on a secondary ranking which depends upon the goals of the List algorithm variation being used. As many LOs are chosen as there are available FUs. If LOs with zero slack times exist, they must be scheduled in that clock cycle to avoid violation of the latency constraint. If the number of LOs requiring scheduling exceeds the number of FUs available, then an extra FU of the required class must be instantiated to accommodate the LO with zero slack. For this reason, LOs with the nearest ALAP time are usually scheduled first to make a best effort in avoiding the need for an extra FU instantiation. It is also possible to use List scheduling as a latency minimized, resource constrained problem by restricting the FUs and selecting LOs on the basis of ASAP only.

A prevalent heuristic is Force Directed scheduling [50], which can be implemented as the secondary selection algorithm in List scheduling. It makes use of measurements of the "demand" for each resource and determines how each possible LO scheduling alters the demand for the entire system. The change in demand is thought of as a force, where larger forces result in a "stretched" scheduling. LO selections that have the smallest or even negative force associated are scheduled first. The rationale of the heuristic is that it will work to minimize concurrency required of each FU class using broader considerations than merely choosing LO with the nearest ALAP value. It is regarded as superior to List scheduling for most applications, but is of cubic complexity and thus less useful for larger problems.

1.9.2 Binding

Binding is the spatial assignment of LOs to FUs. In some instances, binding can be done before scheduling, usually for timing related approaches such as FU chaining. However, to deal with complications arising in part from the implementation of control logic, binding is often completed in step with scheduling [20]. Without control logic, the number of LOs that can be mapped is equivalent to the number of FUs of that class that exist, less those that are multi-cycle and still in use from the previous clock cycle. Multiple LOs in mutually exclusive (mutex) branches, however, can be bound to the same FU, which complicates scheduling a great deal. By scheduling and binding each LO in the same step, it is assured that every scheduled LO will in fact also have a binding in that clock cycle. Conversely, the integrated scheduling binding algorithm can continue to possibly schedule mutex LOs in FUs that might otherwise be considered occupied by segmented algorithms. When performing integrated binding, there is little opportunity to utilize binding to optimize for secondary factors. What can be done is to give preference to bindings that place mutex LOs in the same FU and clock cycle in an effort to fit as many LOs as feasible into the available FUs.

1.9.3 Register Mapping

Once LO mapping has been completed, the source, start time, and end time of the value produced by each LO is known. In most cases, an effort is made to reduce the number of registers needed to store the values during their lifespans by multiplexing them. Minimizing the number of registers reduces to a interval graph that can be solved optimally using a left edge sort algorithm [35], of which a very brief example is shown in Figure 1.4. Mutex LOs can be taken advantage of by mapping their values to the same physical register. More complications are discussed in Section 3.7.2 in conjunction with the discussion on implementation.

1.9.4 Insertion of Interconnects

Once scheduling, binding, and register mapping have been completed, most of the components and data flow requirements are completely specified. A model of the physical hardware can then be built. Inputs, outputs, physical FUs, and registers are all added to the model. Each register and FU operand are wired to input muxes which will be used to select from appropriate sources as required by the schedule and controlled by the FSM. Every LO is compared to the FU to which it is assigned, and, if needed, the appropriate wires are added from the required inputs or registers to the requisite operand's input mux. If the FU in question possesses the commutative property, operands can be rearranged to minimize the number of separate inputs the largest mux has in order to reduce size and delay. Much more elaborate reduction strategies are discussed in Chapter 3. A connection must exist from the FU output to the required register's input mux as well. Once this hardware model is complete, it will fully describe the datapath and can be converted into an RTL description.

1.9.5 FSM

For the datapath to operate as required, control information must be properly orchestrated by a FSM. Especially in the case where there are no control statements, it might be possible to create a separate FSM for each component. However, a conventional FSM that can be properly compiled later by an RTL compiler benefits greatly when control information originally associated with each LO is aggregated into one FSM entity. This FSM has states that often correspond to the clock cycle count. In each state the required configuration of each mux and possibly an enable for each register is stored. It is possible that each of these signals be contingent on conditional logic dictated by the control statements in the original behavioral description. When this information has been collected and aggregated, it can be converted to an RTL description, which completes the HLS design process.

1.10 Summary

HLS has been investigated since at least 1990 in many different applications. Some work involves adapting software compiler concepts such as transformations and re-organization for more efficient scheduling. CPU design has also been an influence, suggesting things like insertion of predictive branch techniques. Others have discussed strategies to improve scheduling in control oriented systems by moving LOs into or out of branches [19]. Other techniques involve the principles of HLS but with different aims and different procedures discussed here. The work discussed here represents what might be considered mainstream HLS. The next Chapter will examine different research completed into extending HLS advantages for the purpose of adding various FT capabilities. Chapter 4 presents a new FT system, called RIFT, that provides cost effective, transparent, online defect detection and compensation meant specifically for large, highly parallel systems. But first, Chapter 3 will present work on a method to reduce interconnect costs, which is an essential component of RIFT.

Chapter 2

Use of HLS for Fault Tolerance

As advancing technology allows further advances into deep submicron design, the long term reliability of integrated circuits are increasingly coming to question. Issues discussed in [13], such as electromigration and high frequency resistance make it increasingly likely that an IC may pass manufacturing test only to develop defects prematurely while in use. These defects may first be expressed as intermittent faults before progressing to a permanent fault. Single Event Upsets (SEU), sometimes referred to as transient faults, are nonrecurring errors that are caused by energized particles, environmental noise, electromagnetic interference, and the like. These errors are becoming more common as feature size shrinks and stored charges become smaller and are thus more easily influenced [15]. Intermittent faults differ from SEUs in that they are caused by physical, possibly worsening, defects. A principle cause is electromigration, which can create frequency dependent opens and shorts. Electron tunneling is another cause, as it can eventually breakdown the gate oxide in transistors. Intermittent faults can lead to permanent faults as damage accumulates. Permanent faults as a result of manufacturing are on the decline, however the incidence of permanent faults occurring while in use is becoming increasingly problematic [13].

While increasing susceptibility to faults, shrinking process technologies are also allowing greater capacity for design complexity. This in turn is allowing for many architectural advancements. Both the emergence of core centered designs and platforms, and the staggering amount of logic that can increasingly be fit onto a chip tend to back the case for HLS. It would make sense, then, to exploit the advances enabled by miniaturization, to mitigate the very consequence of that miniaturization, namely, long term reliability. This argument is compellingly supported by the large amount of research that has gone into HLS in general, and for Fault Tolerance(FT) related HLS work in particular. A survey of this work is presented in this chapter.

A complete FT system may require several different capabilities, including fault detection, isolation, compensation, and recovery, depending on the the level of FT required. HLS techniques have been applied in each of these areas. Of the first two, fault isolation is really a subset of detection. Isolation identifies what hardware component is at fault, whereas detection can only report that an error has occurred. Compensation is the ability to avoid errors due to the fault once it has been detected. Recovery is the ability to catch and correct every error as they occur. Techniques surveyed use either hardware or time redundancy or a combination thereof to determine the presence of and possibly the location of a fault. Care must be taken in selecting FT methods to ensure protection is achieved for the faults that are of concern, as not every method deals with every type of fault. Temporal duplication may detect intermittent and transient faults, but be susceptible to permanent faults. The converse may be true for other methods. Few methods can protect against all fault classes, and those that do tend to be expensive. In addition, most research, including that presented in Chapter 4, assumes that only one fault is present at a given time. Most approaches guarantee treatment of one fault and handle each subsequent fault with decreasing assurance.

2.1 Fault Detection and Isolation

The capability of detecting and isolating faults while an IC is in use is often called Concurrent Error Detection (CED). This is different from the more common case of off-line testing, which is usually reserved for manufacture related testing. A conventional CED system generally needs two distinct sets of hardware to detect errors and three to isolate the faulty unit. This is called Double and Triple Modular Redundancy(DMR, TMR). A similar method uses temporally derived detection by using double or triple recomputation techniques. HLS methodologies use high level knowledge of the system to insert detection or isolation capabilities in a manner that requires less time or resources than required by standard spatial DMR and TMR. Fault Security is an attribute of a system that guarantees that either the result is correct, or that any observable error will be reported. In an early paper, [31], Karri and Orailoğlu duplicate the CDFG and try to map the second onto the same hardware as the first, adding FUs as needed. The technique uses the algebraic properties of associativity, distributivity, and commutativity to aid mobility in scheduling the duplicate CDFG and thus take better advantage of idle resources. In [32], they again use duplication, but in this method the CDFGs are split into regions defined by a chosen LO and its ancestors. The original region can be ”secured” by storing, if necessary, and comparing its result to the duplicate region, which can now be scheduled at different times, ideally on preexisting idle FUs. It is also possible for the duplicate region to break its input dependencies by using data from the original CDFG before it is produced and verified by the duplicate. This ”delineation” gives greater mobility to duplicated LOs and allows greater mapping flexibility. In order to ensure that a fault doesn’t corrupt both results, all LOs within the two regions must be bound to disjoint FUs. By utilizing previously idle resources, this method is able to detect faults with as much as 37% less area than DMR on small examples. However, these are theoretical results in that they only account for the extra FUs required and do not represent actual synthesized gains.

In [36], Jha *et al.* consider the possibility of fault aliasing in the absence of a strict requirement for disjoint hardware between regions. By requiring LOs to be bound on distinct FUs, but allowing an FU to be used in both the original and duplicate regions, hardware requirements can be reduced. This, however, leads to the possibility that a fault in such a shared FU could result in a error on each distinct LO in such a way that the final result produced by the regions are equal but erroneous.

This concept is referred to as aliasing and can result in undetected faults. In [32] above and other work, aliasing was avoided by always using disjoint hardware for each original and duplicate region pair. In [36], additional algorithms are used to relax this constraint in an effort to achieve more hardware reuse. They identify possible faults and calculate the probability aliasing will occur. If acceptably low, the possibility is ignored. Otherwise the arrangement is retracted or an explicit check is scheduled to catch errors. Synthesized average results are as much as 22.3% less than DMR.

In [61], Wu and Karri also address the issue of error aliasing in the context of temporal re-computation. Instead of adding extra FUs for fault detection, they use re-computation on the same hardware using different allocations. They calculated that the risk of aliasing is reduced when the number of times an FU is used changes as much as possible between the original and the re-computation. They implement a system that minimizes the chance of missing an error. Area overhead for a FIR filter is reported to be 18% with a maximum chance of missing an error at 27%. By partitioning the CDFG into smaller recomputing regions, the probability of missing a fault is reduced to 4% with an area overhead of 30%. In [60] they also introduce to HLS a data diversity method first proposed in [48, 49]. It utilizes operand shifts in the re-computed CDFG so that errors are propagated to different bits in the compared results. They found that data diversity aliasing can be reduced by increasing the shift amount and increasing the defective FUs usage in a CDFG region. However, the data path width of the system must be increased by the same amount as the shift desired. Data diversity anti-aliasing efforts result in 12-25% area costs with a probability of missing an error less than 2.5%. Using partitioned CDFGs increases the area to 14-29% but reduces the chance for false positives to less than 1%. The re-computing method does, however, require twice as many clock cycles. Operand shifting may not work for non arithmetic FUs. In [62], Wu and Karri add the ability to break data dependencies to partly overlap re-recomputation to reduce the 100% clock cycle increase.

Isolation of a fault to a particular hardware unit, as opposed to merely determining one has occurred, is somewhat more involved. Many of the previous detection methods can be extended to their equivalent of TMR. In [22], Hamilton and Orailoğlu use an error coding system to correlate errors in multiple regions to isolate a common faulty FU. They report synthesized results of 43% and 11% less area overhead than standard TMR and DMR, respectively. This technique, however, is vulnerable to transient faults and to errors that are not necessarily expressed for every calculation. This problem tends to limit this system to isolating permanent faults.

2.2 Fault Compensation

The other half of fault tolerance is recovery from defects, also known as built in self repair (BISR). The classic approach, used in a broad array of disciplines, is the $N + 1$ fail over system with one redundant unit for every N FUs to be protected [55]. A common $N + 1$ design has the single redundant unit take over the role of any failed unit. Thus it requires the input connections from all of its protectorate. An example is [34], by Kumar and Lach, which uses a reconfigurable unit so that the N protected FUs may be of different classes.

Most BISR HLS research, however, tends to depend on rescheduling to existing hardware instead of using an $N + 1$ based design, in part because it is not obvious how to leverage HLS advantages otherwise. Guerra *et al.* takes an approach in [16] whereby an alternate mapping of LOs to FUs is used for each different possible FU or register failure. The alternate mapping can include algebraic transformations to favour usage of surviving modules. When no other options exist, extra hardware is added. Reported chip area overhead results are 2.3-61% and 4.4-19.3% for scheduling and transformation based reconfiguration, respectively. However, the examples used are different and the number of clock cycles used are not reported. Karri *et al.* have experimented with rescheduling to deal with faulty units as well in [30]. They use specifically designed multi-function FUs to add more flexibility to scheduling. They report synthesized results of 5-11% area cost to meet single fault requirements. Double fault coverage ranges from 14-68%. It is also noted that as more distinct schedules are needed to accommodate possible faulty units, the area required for interconnects grows rapidly.

Graceful degradation is a second class of redundancy proposed by Chan and Orailoğlu that uses the remaining intact modules, but uses alternate mapping with an allowance for the use of extra clock cycles. In [11], the scheduling table, which is a grid of FUs and clock cycles, is split into upper and lower triangles for each class of FUs. If a fault in an FU occurs, the lower triangle is delayed by one cycle and the operations to the right of the faulty FU, inclusive, are remapped to the FU on the right. In the upper triangle, operations including and to the left of the faulty unit are rescheduled to the FU on the left. LOs are scheduled into these triangles normally, except for the additional constraint that LOs in the upper triangle with precedents generated in the lower triangle must be scheduled at least one cycle after they are produced. Otherwise a data hazard occurs in compensation mode as when the lower triangle is delayed, values might be produced too late for upper triangle LO consumption. The reported performance degradation in terms of clock cycles is 7-60% when the worst possible FU suffers a failure. Area costs of using a reconfigurable architecture over a non-FT static design are not reported. The work is extended in [45] to support register faults in the same manner. The reconfigurable design requires 25% more registers. Performance degradation when reconfigured is 7-55%. Interconnect complexity, as measured by the number of wires needed, goes up by 66%, 54%, and 114% when support for FU, Register, and both types of failover are added, respectively.

A separate approach to graceful degradation is advanced by Karri *et al.* in [29] and is called Phantom Redundancy. It is similar to [11] in that separate schedules with relaxed timing constraints are used to accommodate failure of an FU. No spare modules are required. In contrast with [45], Karri *et al.* recognize the need to manage the design process to minimize the additional interconnects needed to support reconfigurable datapaths. To this end, a genetic algorithm based routine is used in an effort to reallocate LOs in failed FUs *to the same* alternate FU. Phantom Redundancy is reported to require 11-79% more clock cycles and a relatively cheap estimated area overhead of 0.7-5%. In addition, it is acknowledged that extra muxes cause an approximate additional 10% performance cost as a result of an increased period.

2.3 Fault Recovery

The FT method embodied by Fault Recovery might be considered the most comprehensive of the different levels of FT, as it aims to detect and correct faults, as well as the errors produced, immediately. The common approach is to insert checkpoints in the schedule where all internal values are stored until the next checkpoint is reached. The CDFG is duplicated as in DMR. Between checkpoints, the duplicated results are compared and, if a discrepancy is found, the calculation is restarted, or "rolled back" to the last checkpoint. The costs associated with this method are the duplication, voting circuitry, and extra registers which hold the values of the previous checkpoint. There is also a performance issue as the rollback re-calculation will require extra time and delay subsequent results. [46] addresses three major issues in designing such systems. The first is in determining, given the quantity of checkpoints required, between which clock steps they should be added. Because all the existing values at a checkpoint must be saved, effort is made to schedule LOs such that their produced values, and thus registers, are minimized at the checkpoints. Values with longer lifespans are targeted to cross checkpoints, as they will already occupy a register for more of a checkpoint cycle regardless. [47] expands on these methodologies by using a multi-dimensional force directed scheduling algorithm. Algebraic transformations of the duplicate CDFG are incorporated to allow better resource allocation, similar to the HLS DMR work presented earlier. A synthesized recoverable 16 point FIR filter design with fault recovery is reported to be 2.7 times larger than a non FT design. It is also reported that 85% of the area is dominated by interconnect. It should be noted, also, that this rollback design is subject to a nondeterministic delay during recalculation due to the possibility of a transient fault that lasts multiple checkpoint cycles. In the case of a permanent fault, rollback is unable to recover unless a TMR approach is used.

In [9], Blough *et al.* approach the fault recovery in a more formal framework, first using specialized algorithms to determine lower bounds on the number of FUs and registers needed. Two approaches are then used to determine schedules with rollback checkpoints inserted. The first is the prioritized cost function method where the number of FUs is constrained to the lower bound, and scheduling then seeks to minimize register cost using branch and bound search. The weighted cost method loosens the FU constraints in a stepwise fashion in an effort to extract cost savings from better register use. The best costing solution based on the relative register to FU cost then determines the best solution. Although synthesized results are not presented, [9] claims optimal solutions and 10-30% improvement over [47].

Hamilton and Orailoğlu present in [23] and [56] an interesting method for adding recovery that also supports permanent as well as transient faults. It uses the concept of checkpointing and DMR, as before, and also assumes that there is more than one value stored at each checkpoint, which implies parallel computation ”strings”. When an error is detected, the subsequent checkpoint iteration abandons duplication and recalculates the erroneous calculation on completely different FUs. The remaining FUs can be used to perform, in duplicate, the calculations that would have been run had there not been an error. If they depend on the erroneous result, each half of the duplicate computation pair will use one of the two results, even though it is known that one data input is incorrect. At the end of the second checkpoint cycle, the redundant calculation will determine which of the original duplicate computations was originally at fault, information which can be used to select the correct dependent duplicate of the current cycle as well. Care must be taken that the dependent computations use the same disjoint FU subsets as the original erroneous subsets and use the corresponding suspect results. Thus it can be assumed if the fault caused another error, it will only corrupt the duplicate that already has the incorrect input value anyways. Execution then proceeds in normal DMR mode until another error is detected. Obviously, this scheme relies on a single fault assumption, and also has some fairly stringent scheduling requirements. The advantage is that computation is not interrupted or delayed in the event of a fault, in contrast to the rollback methods, regardless of whether the fault is transient or permanent. The authors also discuss a variation that is less hardware intensive and uses at most one checkpoint cycle to correct transient and permanent faults. They report results for this technique combined with graceful and sparing recovery methods at 47-50% and 35-43% area less than TMR, respectively.

2.4 FPGA Specific FT Approaches

In recent years, field programmable gate arrays (FPGA), because of their flexibility and growing capability, are increasingly being used in critical systems. Typical HLS FT compensation systems work by reorganizing how LOs are mapped onto a hardware configuration. FPGA approaches to FT can be enlightening in that they typically work towards the opposite: reorganizing the hardware in an effort to support the logically mapped design requirements. Another parallel is that both FPGA FT and generic HLS FT are interested in providing diagnosis or alternate configurations which can be pressed into service when needed without the use of CAD tools. This section is a brief survey of FPGA FT concepts and methodologies that may prove useful when contemplating FT for the generic architecture.

2.4.1 Fault Detection and Isolation

In [58], Tahoori *et al.* discuss a simple offline test method to only identify faults in the interconnects and Look Up Tables (LUT) that are to be used by a particular design on a FPGA. This allows for the idea of *application-specific* FPGAs which can distinguish and ignore faults that are outside the required FPGA resources. For high volume FPGA designs, yields can be improved by not testing parts of the FPGA that are not used. Abramovici *et al.* propose a method capable of testing FPGAs online, however, the method only works with FPGAs capable of partial online reconfigurability [2]. The technique uses reserved groups of blocks called ”STARs”, for self-testing areas. Within each group, one block is configured as a test pattern generator, which feeds two other blocks under test. Another block in the group analyzes the responses. As the test progresses the role of the blocks rotates. The STARs are able to swap places with functional parts of the FPGA and thus test the whole chip in a ”roving” fashion. A method published by Lala and Burress in [37] decomposes logic expressions such that they can be mapped into a FPGA LUT along with a redundant LUT that outputs the complement. The LUTs can be cascaded with others to allow for larger logical functions. The two outputs are compared and indicate an error if they disagree. The result is an online self-checking FPGA circuit.

2.4.2 Fault Compensation

A method for using HLS to add redundancy to FPGA designs is suggested in [4]. The CDFG is broken into "detectable subgraphs", similar to the secure CDFG regions discussed earlier. The idea is to try and map each subgraph into a "tile" on the FPGA in a manner that minimizes interconnect requirements between subgraphs. Then spare tiles can be reserved in the FPGA which will be less costly to use as a replacement for any subgraph found to contain a defect. Hanchek and Dutt present work concerning how to reconfigure FPGAs to use pre-allocated spare configurable logic blocks without having to use CAD tools to generate a new design that circumvents a fault [24]. They propose that FPGAs incorporate switches that can bypass logic blocks while loading the FPGA configuration. During the generation of the original FPGA configuration, extra wiring is reserved in anticipation of using a redundant logic block. The method mostly amounts to merely extending wires one block. Once a faulty resource has been located, a switch in the FPGA programming bus is used to bypass the defective logic block and add the spare. Then the original CAD generated configuration can be loaded onto the FPGA for normal operation. The authors also suggest that spare interconnects could be added in parallel with used interconnects. Defective wire could be bypassed by using switches that redirect signals over the adjacent spare path. Unfortunately, these methods must be built in by the fabricator. The actual fault identification could be coupled with an online isolation scheme such as [37] or an offline approach as described in [51].

2.5 Motivation

Originally, the goal of this research was to determine methods by which FPGA designs could be made to have FT capabilities without the need for reconfiguration. The methods mentioned for FPGAs in the previous section are promising, but none to our knowledge, with the exception of STAR, are capable of compensating for faults without offline reconfiguration. STAR is reconfigurable online, but relies on a partial reconfiguration technique only available on a small subset of FPGAs, as well as an external controller. It is also not immediately clear how the STAR technique can transparently relocate state and run-time data when moving the testing area. Because most FPGA architectures are not disclosed, it becomes necessary to add FT at an abstraction level higher than the physical layout. The work presented in the next two chapters still satisfies this original goal and the results are generated on FPGAs. However, the methodology itself and the following motivation is generic enough that it is equally applicable to ASIC design as well.

Many ideas and approaches to FT systems have been advanced. However, they all suffer to some degree from two major problems introduced by HLS infrastructure:[1]

1. An increasingly complex and deep FSM
2. A substantial amount of extra interconnects are introduced.

When compared to an *optimal* non-fault tolerant HLS produced design, it is obvious that the failure of any unit cannot be tolerated, as this would preclude that optimality. Thus to accommodate a single failure, either the number of execution steps must be increased, or one or more additional units must be incorporated. Both options then imply the first problem listed to be true, as both require alternate arrangements to redistribute the work, and therefore different mappings. When a component fails, the physical wiring inputs which did feed it must be redirected to an alternate unit, thus requiring extra wiring. This may not be so bad in itself, but the alternate FU may require extra routing in the form of a new or larger mux. If not considered, this could, in aggregate, considerably affect the critical path of the HLS infrastructure and consequently increase the period and thus reduce the performance of the design. Fault detection and isolation also requires extra routing and control and thus also incurs these two costs, although possibly to a lesser extent.

Presumably as chip size grows, HLS is supposed to harness the increased capacity by implementing greater parallelism through the use of more modules, ideally without further burden on the designer. It is submitted that the promise of HLS is in how it can be used to leverage preexisting modules as capacities increase by easily and

[1]Henceforth, the term "arbitrary" should be understood to mean without due consideration of these two factors.

efficiently increasing parallelism. However, when HLS systems become increasingly parallel, even massively parallel, the potential exists for the two HLS issues listed above to increasingly dominate performance and area considerations if not adequately managed. Consider that FPGA products currently available (*circa* 2005) can contain as many as 96 DSP oriented blocks that can be configured as 36x36 bit multipliers [1]. If HLS arbitrarily maps fault tolerant LOs to this many multipliers, it is conceivable that an input consumed by one multiplier might also need to be redirected to many of the other multipliers for redundant configurations. Or a redundant unit might have as inputs the complete set of the protected units' inputs, depending how redundancy is configured. The ramifications concerning the HLS FSM are not insubstantial either. Indeed, it would be difficult for an HLS FSM to accommodate separate configurations for the continuum of a hundred units without protracting the critical path, especially if the work involves hundreds or thousands of clock cycles.

Much work has been done towards optimally packing operations, fault checks, and redundancy into the mapping grid. Yet for large and massively parallel systems, the cost of adding or reserving a few units for FT will likely be inconsequential compared to the importance of managing overall complexity. Some research has already moved in this direction. For instance, Phantom Redundancy is a variation of graceful degradation where emphasis is in moving all displaced operations to a common replacement unit in an effort to reduce interconnect complexity, though with the requirement of more clock cycles [29]. In their conclusion in [28], the authors acknowledge that the largest cost of HLS FT designs is "interconnection complexity". It is observed in [47] that rollback and recovery techniques can require as much as 85% of chip area. Unfortunately, most FT oriented HLS research, referenced here or otherwise, report performance in terms of clock cycles and added interconnect complexity, while rarely are the consequences on clock period reported. Given this is an important determinant of actual performance cost, it could be surmised that either the experiments were not carried out to this level, or that this metric is not favourable. Furthermore, although classic benchmarks such as the 16-tap FIR filter help to serve as a basis for comparison, it seems somewhat preposterous to utilize such elaborate HLS algorithms and then demonstrate them on small examples that have already been redesigned *ad nauseam.* In any case, the effects of HLS augmented FT on actual circuit performance is relatively unexplored, especially as concerns large parallel systems for which HLS should prove most conducive. The intent of the work presented in the remaining chapters is to explore interconnect complexity and FT as it pertains to large systems, and to demonstrate a method by which these costs can be reduced.

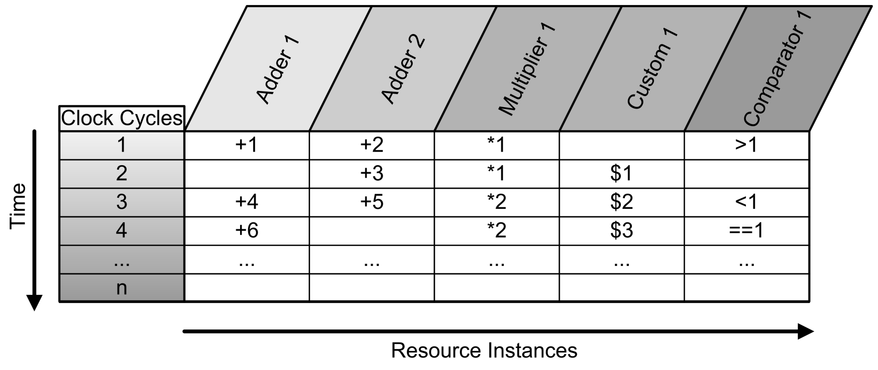
Chapter 3

Reducing Interconnects

As should be somewhat evident from Chapter 1, there are many differing methods to approach the generic HLS problem. The criteria for which improvements, or even optimality, is desired will greatly serve to determine the approach taken to HLS. The addition of FT invariably necessitates greater interconnect complexity and the costs associated with it. Thus, before FT is examined in Chapter 4, this chapter will examine the nature of interconnect costs and methods to mitigate them. The chapter concludes by presenting experimental results of HLS designs implemented on FPGAs that compare interconnect ignorant and aware approaches.

3.1 Common HLS Criteria

Research in HLS has been conducted in areas varying from standard timing and control oriented optimization to various fault detection, isolation, and recovery methods. Generally, these approaches explore performance and resource cost tradeoffs in effort to find Pareto optimal designs. However, in most cases, performance is measured in the somewhat narrow terms of the number of clock cycles required to complete the work. Similarly, the number of FUs required is often considered as equal to area, though in fact, it is only proportional. However, this definition of area and performance is appealing when dealing with mapping algorithms, as the final result can be shown on a mapping grid as in Figure 3.1, where one axis reflects FU resources,



Clock Cycles	Adder 1	Adder 2	Multiplier 1	Custom 1	Comparator 1
1	+1	+2	*1		>1
2		+3	*1	$1	
3	+4	+5	*2	$2	<1
4	+6		*2	$3	==1
...	...	...	...	...	...
n					

Figure 3.1: Resource & time tradeoff results in an HLS system

and the other the available clock cycles. Competing scheduling approaches can be easily evaluated by comparing the resources and clock cycles used. This can be useful when dealing with complex algorithms, particularly for scheduling control structures. Unfortunately, in the final implementation, other issues can significantly affect the final real world performance of an HLS designed system.

3.2 Interconnect Cost Components

The cost of interconnects has always been acknowledged in HLS design [20]. However, it is often dealt with as an unavoidable or perhaps trivial cost as compared to the main area and "performance" objectives. As the HLS design problems under consideration increase in size, complexity, and parallelism, the costs associated with interconnects will become more important.

Before a more complete treatment of interconnect associated costs can be given, it is important to more precisely define what is meant by the term "interconnects" . In the context of this work, interconnects are the components of a chip that direct and transport data between inputs, outputs, FUs, and registers. This definition thus includes both the *wiring* and the *steering logic*, which usually consists of muxes, but can also consist of tri-state buffers when busses are under consideration. Because FPGAs are the focus in this work, busses and tri-state buffers are not considered. Most useful IC designs require interconnects. If a CDFG were to be directly converted into an IC, all the vertices would become simple wires and steering logic would be unneeded. Most such designs, however, would need substantially more FUs because there would be no FU reuse. HLS methods aim to lessen the number of FUs required by reusing them, but to do so must add extra wires and steering logic when an FU requires data from different sources. These extra interconnect requirements are part of HLS's infrastructure costs, along with the registers now needed to store data until it is used. Registers add to the number of data sources and so also add to interconnect costs. If registers are used to store data from multiple sources, then even more steering logic will be required, further increasing HLS cost. Let I, O, FU, OP and R represent the number of inputs, outputs, FUs, operands per FU, and registers that exist in a system, respectively. Then the theoretical maximum number of connections needed can be expressed as follows:

$$\begin{aligned} Interconnect\,Complexity &= Sources \cdot Sinks \\ &= (I + FU + R) \cdot (O + R + (FU \cdot OP)) \\ &\quad - FU \cdot (FU \cdot OP) \\ \lim_{I,O \to 0} &= FU \cdot (R + R \cdot OP) + R^2 \end{aligned}$$

The $FU^2 \cdot OP$ term is removed because values produced by FUs must be registered before they are used as FU inputs. The R^2 is a result of allowing register to register transfers. When parallelism is increased and the inputs and outputs become trivial, the remaining complexity is a strong function of the number of FUs and registers.

There are at least two costs associated with interconnects. The first and obvious one is that both wiring and muxes will require area on the chip. Wiring area is difficult to estimate in HLS circumstances because the floor plan and thus the length of the wires cannot be known until lower in the design flow. Thus, at the HLS level, wiring area, at best, can only be though of as proportional to the number of point to point connections. The area required by muxes is somewhat more defined, as the size, relative to a two input mux, is proportional to the number of inputs less one.

The second cost is that interconnects add extra delay. Wire delay is proportional to length, which again is unknown. In the case of FPGAs, wires are fixed and their routing depends on statically configured switches which can considerably increase the propagation delay. Muxes also can add considerable delay. Like area, a muxes' delay can be estimated based on the delay of a two input mux. The delay is proportional to $\lceil log_2 (N) \rceil$, where N is the number of inputs. When a mux is used on an FPGA, it is usually instantiated with a LUT. Because a LUT has a limited number of inputs, larger muxes require several LUTs and thus usually several "logic blocks" which must be connected together by wires. This means that muxes on an FPGA are distributed in nature and can be quite costly.

3.3 Interconnect Specific Work

From the beginning of HLS research, it has always been recognized that the HLS infrastructure adds some interconnect related costs. An early paper by Tseng and Sieworek orients the HLS procedure around the data paths "for the minimization of the number of storage elements, data operators, and interconnection units" [59]. They map the sources, FUs, and sinks to separate graphs and use clique partitioning to determine the number of components needed and to partition data transfers to busses. The approach does try to reduce area by reusing interconnects, however, it does not consider the area requirements of steering logic nor delay.

Cloutier and Thomas present in [12] an approach whereby scheduling and binding are integrated into a single step that also considers interconnects. It is based on a force directed method that has been extended. Although the algorithm considers the interconnect savings to be gained by reordering commutative operands, it does not consider wiring outside this consideration. For this reason, and because force directed scheduling uses a weighted average of several other considerations, emphasis is not placed on minimizing interconnect costs.

A simultaneous scheduling and binding algorithm using simulated annealing is presented by Kollig and Al-Hashimi in [33]. One of the changes that can randomly be made to the model during the simulated annealing process is to swap the inputs of a LO if they are commutative. The cost function includes FUs, registers, and also the equivalence in input muxes. As such, the area of muxes are considered directly and the area of wires connected to muxes are indirectly considered, but delay is not.

In [52], Rim *et al.* propose both an Integer Linear Programming and a heuristic method for binding FUs and registers. It specificly considers the cost of point to point wiring and also mux area. While the work is comprehensive, it does not consider delay introduced by muxes. Furthermore, because the methodology does not include scheduling, there is no freedom to minimize the costs by managing the schedule.

A binding algorithm is presented by Bhattacharya *et al.* in [8]. It is focused on minimizing the critical path of combinational networks used to implement conditional execution in control oriented designs. It does not, however, address period reduction for registered datapath architectures common to HLS. Opportunities to reduce the period when scheduling and allocating resources are not addressed either.

3.4 Delay Components

In the prior work presented in the previous section, the area costs of interconnect are well considered. However, the manner in which interconnects are constructed can considerably affect the critical path. The clock period is proportional to the critical path of a circuit and is comprised of several components. The largest contribution to delay would most likely be from the FUs. Because the design of the FUs are supplied by the designer, HLS can do nothing to address this delay. If the HLS infrastructure allowing reuse of FUs is more expensive in terms of area and delay than the actual FUs themselves, it would be more effective to simply use extra FUs. Not only would the delay cost of HLS be avoided, but the greater parallelism (if not already maximized) would allow for execution in fewer clock cycles. Therefore there is some point were the ratio of the mux to the FU delay and area would favour abandoning HLS techniques. If the the ratio was high because the mux is supporting a large number of inputs, it may be better to consider using more FUs than strictly necessary rather then inordinately increase mux size and delay for greater reuse. An example of these principles can be seen in Figure 3.2. In this case, because the cost of the FU is small compared to the two operand muxes, it can be replaced with three

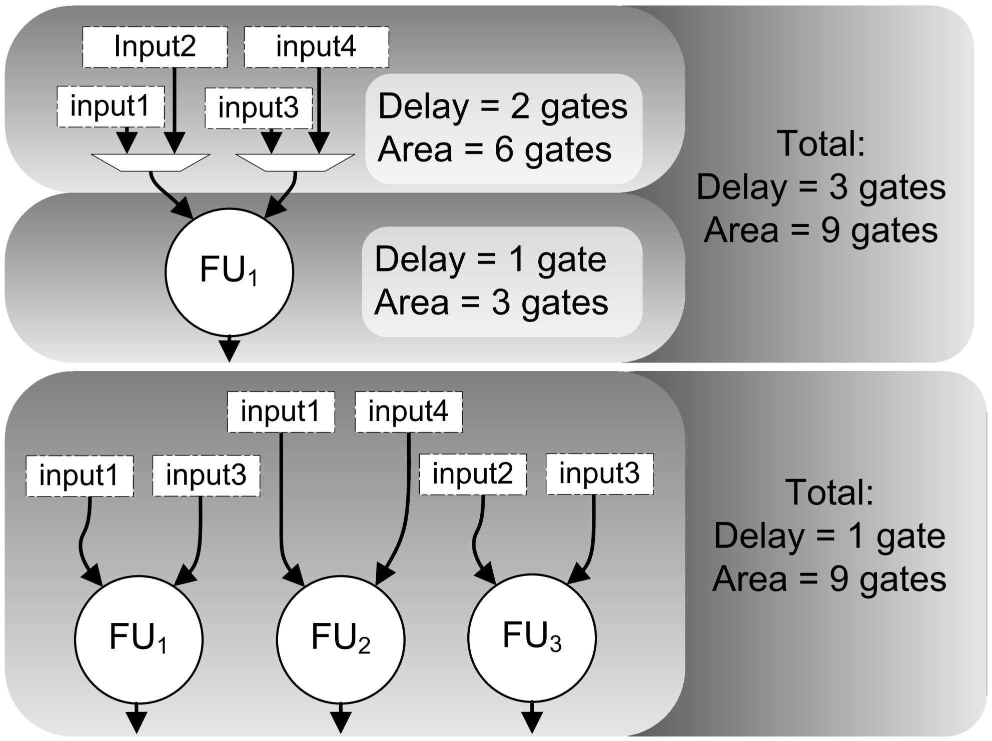


Figure 3.2: HLS cost considerations

FUs and still have less delay. This tradeoff would be worthwhile if only three or less of the four possible input combinations are needed. The configuration using muxes would be worthwhile if the area cost of the FU were much greater than the muxes combined. Generally, for HLS, this is assumed to be the case.

A second delay source could, in some cases, be the control logic. Generally the finite state machine is regarded as a smaller and more simple than the rest of the design as well as "adjacent" to the datapath, as depicted in Figure 3.3, and thus unlikely to impact the critical path. However this assumption may not be justified for designs that are predominantly control oriented, especially if some manner of FT mechanism is added.

The final major component of delay in an HLS design is the routing infrastructure added by the HLS CAD tool itself. The central tenant of HLS is to reuse expensive FUs by executing multiple operations on each in succession. This requires each FU

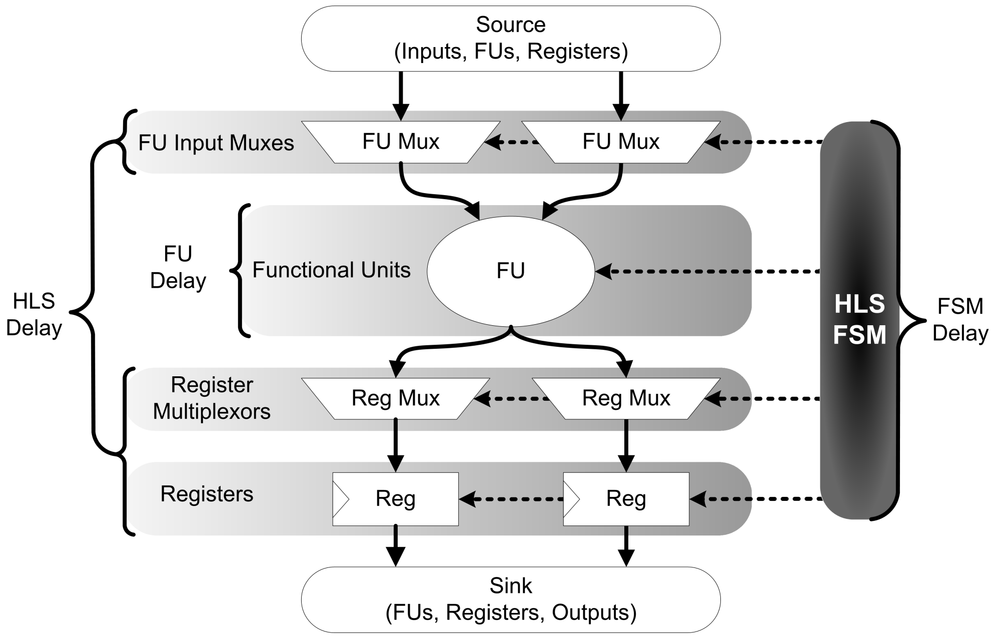


Figure 3.3: Delay components of an HLS system

to be able to read multiple values on each of its operand inputs as dictated by binding. Therefore, muxes must be inserted to allow values to originate from different input ports or registers to be consumed by the FU. The more distinct input sources required, the larger the required mux, which increases the delay of the mux proportional to $\lceil log_2 \rceil$ of the number of inputs. If using a homogenous set of FUs, the critical path length is determined by the set of muxes with the largest number of distinct inputs, and is again proportional to $\lceil log_2 \rceil$ of that amount. The same is also true of multiplexed registers, which also contributes to the critical path. Therefore, the component of clock delay where HLS has the greatest opportunity to improve the critical path is the two layers of muxes that feed FUs and registers, as seen in Figure 3.3.

3.5 Difficulties in Considering Delay

In a broader sense, it is obvious that to minimize the critical path, the muxes at the FU and register level need to be balanced. Traditional HLS, however, is unsuited to these considerations, and in fact, often ignores them for several reasons. The classic design procedure first schedules and binds LOs from the CDFG onto the available physical FUs with the aim of attaining the highest possible utilization rate. In other words, concentrate the work into the smallest amount of time and resources possible. Once this step is completed, the results produced by each LO have a well defined start time and lifespan, the end of which is determined by the start time of the consuming LOs. This allows a left edge sort to be used to pack the produced values into the fewest number of physical registers. Once these steps have been completed, the wiring requirements are almost completely specified. The only exception is that commutative FU have some flexibility regarding the ordering of their operands. Thus it is difficult to add delay considerations to the standard HLS procedure.

The problem can be further explained as one of cyclical dependencies. If it were known during LO mapping which FU had an existing wire to an available register, LO binding could be done in a way that reduces the number of distinct register inputs. During LO mapping, only the original input wires are concretely defined. Other inputs will come from values stored in registers, but which register in particular is not known until register mapping has been completed. Thus, managing connections to FUs during LO binding can only be done with values, which are not yet bound to specific registers. Thus there is no knowledge of how different binding choices will affect FU input mux size. As such, it is difficult for classic HLS CAD tools to continue to work at a more abstract level and still deal with the lower level details of how the components are actually wired. For these reasons, HLS approaches have been more focused on improving mapping against a clock cycle benchmark as opposed to the more complex endeavor of improving a design based on a clock period x clock cycles measurement of performance.

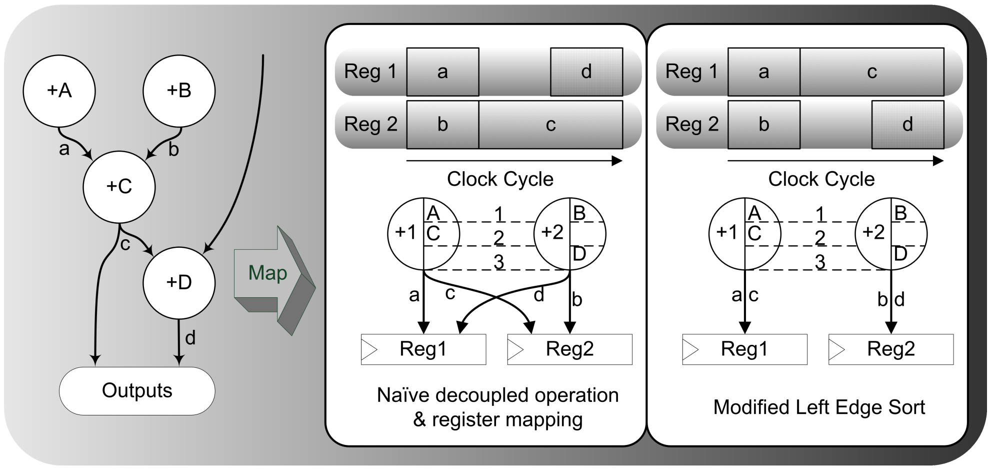


Figure 3.4: Improvements possible using an enhanced Left Edge Sort

3.6 Delay Centric Approaches

3.6.1 Enhanced Left Edge Register Sorting

It would be advantageous for a FU with an existing connection to a register to reuse that register as much as possible. However, this is difficult to take advantage of with the current model. The LO mapping must be completed before the left edge packing algorithm can be used to determine the specific register to which a produced value will be stored. It is possible to modify the left edge sort routine to consider the source of a value when packing onto physical registers (as LO mapping has been already completed). However, depending on the aggressiveness of the new routine, this may lead to some degree of sub-optimal register packing. A delay aware register packing algorithm would capture an extra degree of freedom, the choosing by source, in reducing delay as depicted in Figure 3.4. However, gains available by manipulating the binding and scheduling of LOs, as demonstrated in Figures 3.6 and 3.7, cannot be realized with this post-scheduling approach.

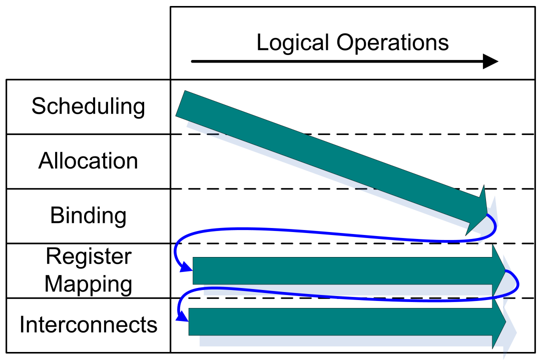
3.6.2 Heuristic Approach

A second possible way to approach the problem could be to use a heuristic. This method has been used by several others for various aspects of the mapping problem, such as mapping problem solved with simulated evolution [40] and simulated annealing in [33]. Use of heuristics to optimize delay could be computationally costly. The first difficulty would be the basis on which to measure improvements of a design. It is possible to complete the design flow down to placement and routing for each iteration and use the resulting frequency performance as feedback for the next iteration. However, adding this step to a heuristic loop would necessarily imply an expensive iterative cost, and consequently, a high overall computational cost.

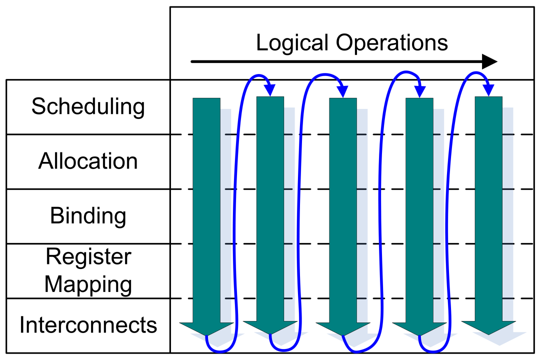
The heuristic approach could be improved by instead treating the size of the largest FU and register multiplexors as proportional to the delay introduced by the HLS procedure, thus avoiding the synthesis step. Even so, the scalability of such an approach is questionable. There are several degrees of freedom, which contribute to a fairly large design space:

- The number of instances of each resource type.
- The timing options for each LO.
- The binding options for each LO.
- The binding options for inputs of commutative FUs.
- The number of registers to use.
- The binding of values to registers.

The timing for register scheduling is fixed by the mapping of LOs to FUs, and is thus not a free variable, unless one considers esoteric tricks such as moving values around to open registers over the value's lifetime. Several of these variables are essentially graph colouring problems with extra constraints and considerations, making them NP-hard. It should be possible to use such a brute force approach in designing delay aware HLS systems. However, given the rapid growth in the size of the design spaces



(a) List: Breadth first



(b) RIft: Depth first

Figure 3.5: Breadth vs. Depth methods

under consideration, a design driven approach capable of making rational tradeoffs instead of random guesses is likely to be more computationally efficient. Furthermore, it may be more difficult to explore nuanced architectural enhancements in a heuristic framework.

3.6.3 An Interconnect Driven Greedy HLS Approach

The approaches discussed so far had the disadvantages of either having cyclical dependencies, or required the use of brute force with little capacity for nuance. The final approach proposed here is to flatten the HLS procedure such that LO and register mapping are completed concurrently *one LO at a time* in a greedy fashion. This would mean mapping a LO to a clock cycle and FU as well as selecting a register to store the output value in, all in one step. The process is then repeated for subsequent LOs until all are scheduled. This changes the way in which the hardware is designed, as shown in Figure 3.5. Whereas before hardware was built up in stages of first all the FUs and then as many registers as needed, this method builds the hardware incrementally. If first mapping all the FUs and then mapping the registers is thought of as a ”breadth first” routine, the new approach would be a ”depth first” method. Alternatively, this method could be described as a datapath focused design effort, as opposed to resource based. The main disadvantage to this approach is increased

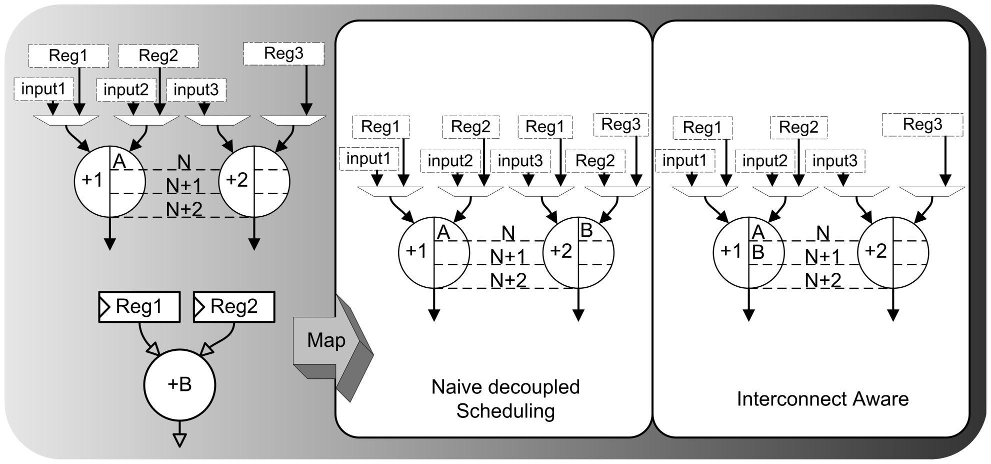


Figure 3.6: Advantage of considering delay while scheduling

complexity in having the LO and register scheduling and binding problems all open at once and being solved incrementally, without complete knowledge of the previous "stage". Adding wiring as a consideration only adds to this complexity. Nevertheless, the advantage is substantial. The incremental approach allows for various opportunities to be taken advantage of in a greedy fashion, such as use of current knowledge of the circuit to reuse existing wiring. This advantage is demonstrated in Figure 3.6, where the ability to consider wiring while manipulating LO scheduling can lead to a less complex wiring configuration. In this example, an input wire is reused when the '+B' is deferred a clock cycle. In Figure 3.7, an example is shown where wiring knowledge is used during binding to reuse existing register wires. This datapath based approach also more easily allows for additional architectural additions, such as fault tolerance, addressed in the next chapter, or buffered multiplexors. This new method is named thus after its advantages: "Reduced Interconnects and Fault Tolerance" or RIFT.

3.7 Implementation of RIft

The remainder of this chapter will concern itself with the details and performance of the first component of RIFT, namely the reduction of interconnects. FT is an

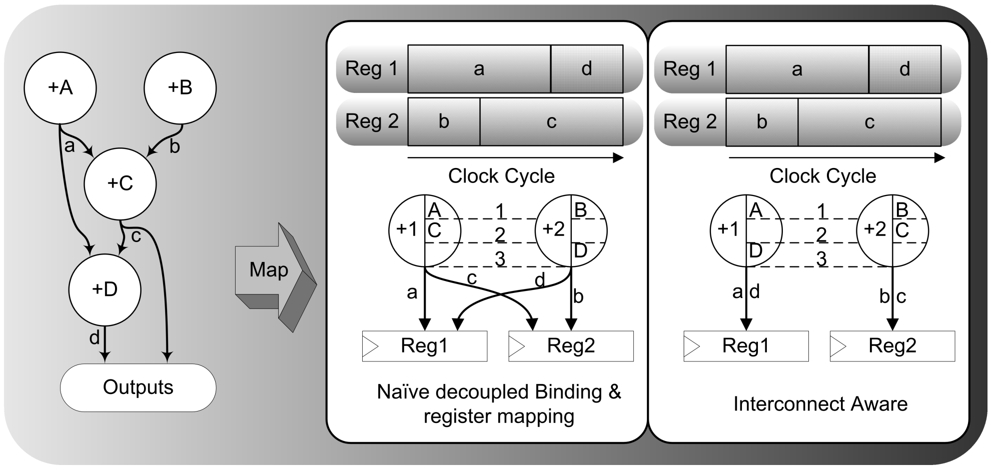


Figure 3.7: Advantage of considering delay while binding

extension of this topic and will be discussed in the next chapter. To distinguish between the two versions, the version that only reduces interconnects will henceforth be written as RIft, with a lowercase ”ft”. When the version with FT is referred to, all capitals, as in RIFT, will be used.

3.7.1 A RIft Cost Structure

The basic premise of how RIft works is to incorporate into HLS the incremental cost of adding the hardware required to accommodate the mapping of subsequent LOs. It follows then that the first step in deciding which LO to map next is to determine the cost of mapping every eligible LO on every available and compatible resource. This is determined by examining the possible variations of the following decision variables:

- The physical FU to be mapped to.
- The mapping of LO inputs to physical FU input muxes
- The existing or new register the produced value is to be mapped to.

The mapping of logical inputs to physical FU input muxes is done in a greedy fashion for each possible mapping of a commutative LO to an FU. The selection of an appropriate register is directed by run time specified options and further discussed in the next subsection. When all possible allocations have been determined, the following cost components for each possible LO to FU mapping are described in the following list. Their formal cost is also stated in the respective line of Equation 3.1.

Input Wire Cost: The number of wires that would be added to the FU input muxes.

Register Cost: The number of new registers that would need to be added.

Register Wire Cost: The number of input wires required by the register input mux.

Marginal Cost: The increase in size of the largest FU mux of a class.

$$\begin{aligned} Cost_{i \to j} = C_{input} \cdot & \sum_{k=1}^{|operands|} e_j^k \\ & + C_{reg} \cdot f_j \\ & + \overline{f_j} \cdot C_{reg\_wire} \cdot g_j \\ & + C_{marginal} \cdot T_j \end{aligned} \tag{3.1}$$

$Cost_{i \to j}$ denotes the total cost of assigning operation i to the physical FU j. The C_x variables are all cost coefficients that can be set by the designer. The first line is the cost incurred by the addition of inputs to the mux of FU j. e_j^k is a boolean variable that is 1 if for the jth FU the kth operand requires the addition of a new distinct input. C_{reg} is the cost of a new register, and f_j is a boolean variable indicating if one is needed. C_{reg_wire} is the cost of adding a wire to a register input, and g_j indicates if a new wire is needed for the chosen register. The final segment is the marginal cost multiplied by T_j which is '1' if this allocation results in a new maximum operand mux size for this FU class. This is meant to penalize any mapping that could increase the critical path.

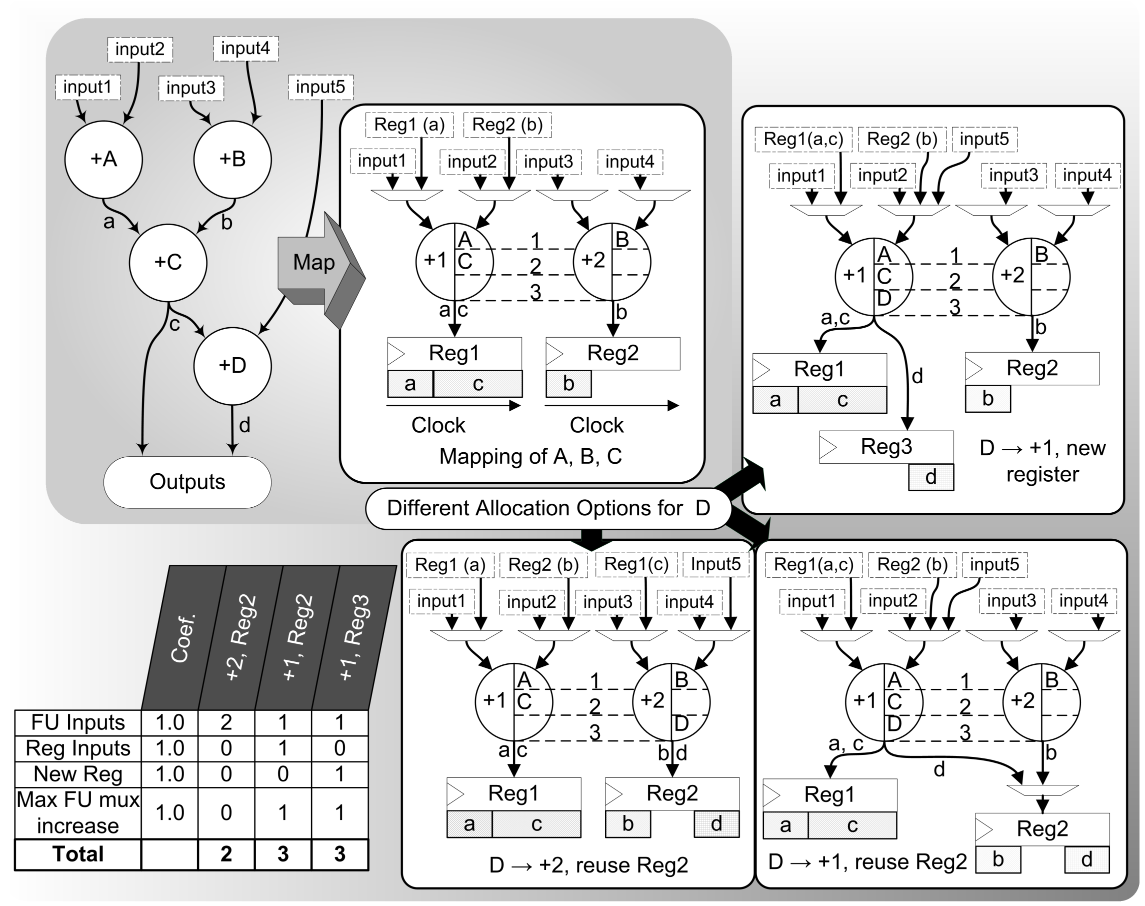
As an example, the use of the cost calculation in selecting the next mapping can be seen in Figure 3.8. The first panel demonstrates how operations 'A, B, & C' have already been mapped from the CDFG to the Hardware Description Graph (HDG). The HDG has been drawn with timing information superimposed. Between FUs '+1' and '+2', the clock steps 1, 2, and 3 are denoted in descending order on the dashed lines. An accounting of the value stored in a register is immediately beneath that register, where time increases to the right. The three remaining white panels depict some possible ways in which the single remaining unmapped operation 'D' could be mapped. The cost is determined for the three possible allocations and presented in the included table. In this case, it is least expensive, and most logical, to bind 'D' to FU '+2' and reuse register 2. As LOs are iteratively mapped, the options available to the remaining unscheduled LOs will change, as will their associated costs, and thus the cost and compatibility information for remaining LOs must be updated after every new LO mapping. It is important to note that a classic "breadth first" multi-stage mapping algorithm would not have the interconnect information shown in 3.8 and thus would not be able to consider the routing costs. RIft is able to do so only because of its integrated scheduling, binding, and register allocation routine.

3.7.2 Register Mapping

For every LO to FU allocation that is considered, a particular register is chosen to store the produced value based on several rules which are selected by the designer. The simplest case is that of partitioning. When this directive is specified, every FU is connected to its own registers and does not share with other FUs. The exception is for the implementation of conditional logic. If in one conditional branch, a value is produced on a different FU type than in a subsequent branch, the register that stores that value will need to accept connections from each FU of different types. The following code segment would lead to such a situation:

```
if condition then $c = a + b$;
else $c = a * b$;
```

If the different values for 'c' are not stored in the same register, then subsequent operations needing to access value 'c' would not know what register to look in without some manner of elaborate and costly setup. The two example assignments each are



	Coef.	+2, Reg2	+1, Reg2	+1, Reg3
FU Inputs	1.0	2	1	1
Reg Inputs	1.0	0	1	0
New Reg	1.0	0	0	1
Max FU mux increase	1.0	0	1	1
Total		**2**	**3**	**3**

Figure 3.8: Different allocation costs for 'D'

represented by LOs and it is convenient to euphemistically refer to them as "alter egos". If the conditional value is produced by the same class of FU, RIft will attempt to use the same FU for each branch. However, the cost coefficients for register and FU input wires may yet dictate a mapping to separate FUs. The advantage of partitioning is that, excepting conditional requirements, no register muxes and their associated control circuitry are needed. The disadvantage is that there likely will be more registers, and thus also more sources which the FU muxes must potentially accept values from, which could complicate routing at the FU level.

Without the partitioning directive, RIft will attempt to share registers wherever possible. Compatible registers are selected in the following order:

1 If already instantiated, an alter ego must be connected to the appropriate register regardless of wiring.
2 Any available and compatible connected register that has a mutex compatibility, excluding those reserved for an alter ego.
3 Any other available connected register, excluding those reserved for an alter ego connection.
4 Any available register with a mutex compatibility, by smallest input amount.
5 Any other available register, by smallest input amount first.
6 A new register is created.

The first rule is dictated, as discussed above, by the need for alter egos to store their values to the same register. Thus if a register is being selected for an alter ego, RIft must check to see if the alter ego's counterpart already has been assigned a register, and if so, that one must be used. Otherwise, the selection proceeds normally. In order to ensure a register is available to the alter ego counterpart when it is needed, an "alter ego hold" is placed on that register to ensure no other values, including mutually exclusive values. This is removed when all the counterparts have been mapped. If the required selection does not involve an alter ego, the next prospective group of registers are those that are already connected to the targeted FU. Registers that already hold values that are mutually exclusive with the value to be stored are preferred since it is less likely that any other produced value will be able to use that register.

If no connected register is available, RIft will proceed based on the register and the register input wire coefficient costs, C_{reg} and C_{reg_wire}. If a wire is less costly than a register, RIft will try to select a register used by other FUs, again trying to find a mutex compatibility first. Of the eligible register candidates, that with the smallest number of existing inputs will be chosen. This is because a wire will need to be added between the FU and the selected register, and by selecting the smallest, increase of the critical path caused by register mux expansion is avoided when possible. Finally, if an available register still has not been found, a new one will be created. However the register is selected, only the cost of using that one register for that particular candidate LO to FU allocation is considered.

Finally, there is one more register selection directive that is meant to give RIft some ability to accommodate different target platforms. A size threshold can be set that will limit the number of inputs that register muxes may have. This threshold is ignored when alter ego requirements must be satisfied, however, RIft tries to anticipate alter ego requirements and use a new register where appropriate. On an FPGA, large muxes can be relatively expensive, yet every register essentially has a built in two-input mux, due to the cellular architecture. Therefore, it may be advantageous to set the threshold to two inputs. It would be expected that the number of registers needed would be significantly reduced, as compared to partitioning and with only a trivial increase in routing complexity. For ASIC targeted designs, large mux instantiations are relatively more compact, and thus larger muxes may be less costly to implement. Even so, the threshold can be used to control how many mux levels will be allowed in the critical path according to

$$critical\ path \propto \lceil log_2\,(threshold) \rceil$$

For instance, an input threshold of 5 to 8 would be specified to limit register muxes to 3 levels of two-input muxes.

3.7.3 RIft Scheduling

The addition of RIft changes the manner in which LO mapping is carried out. List and Force Directed scheduling are mainly concerned with choosing the next LO to be scheduled in the current clock cycle and RIft can be added as an extension to either. Because classic List scheduling only uses different levels of eligibility when selecting LOs to map, the RIft cost can be used more directly in selecting both the next LO to map and also the FU for that LO to be mapped to. Since Force Directed scheduling already uses a cost, that of the ”stress” removed by scheduling a particular LO, it would be more ungainly to incorporate a RIft based cost as well. However, RIft cost could still be used in determining which FU a chosen LO should be bound to. For purposes of investigating the advantages of the RIft approach, it is probable

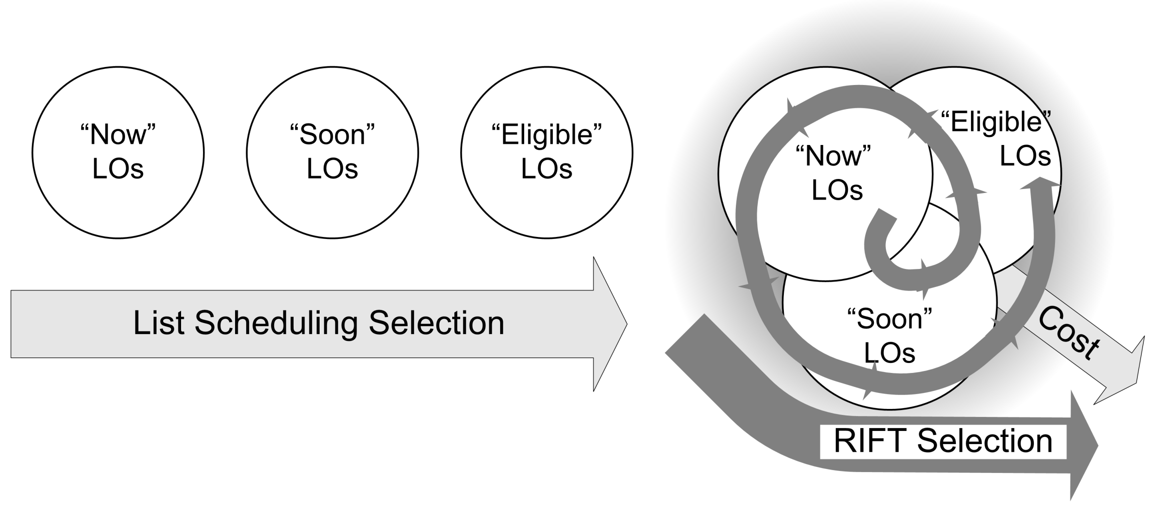


Figure 3.9: LO selection methods

that the more complex Force Directed scheduling would be more likely to obscure the results achieved by the addition of RIft algorithms. For these reasons, RIft has been implemented as an extension to the List Scheduling algorithm.

The algorithm for RIft is less intuitive than that of List Scheduling. The version of List scheduling implemented splits up LOs that have their precedents satisfied into three distinct groups:

***now*:** Now LOs are those that must be scheduled in the current clock cycle in order to meet the systems latency constraints.

***soon*:** Only multicycle LOs can be in the soon category. They do not need to be scheduled in the current clock cycle, but they will need to be scheduled within the number of cycles this LO type takes to execute.

***eligible*:** All other LOs that have their precedences satisfied but do not fall into the previous two categories.

This leads to two separate conflicting LO selection criteria. The first is to ensure that *now* nodes are scheduled without the need for additional FUs where possible, and that *soon* nodes preempt *eligible* LOs. The second criteria is to select the cheapest LOs from the largest possible pool of *eligible* LOs, as this constitutes the greedy method.

If the *now* LOs are scheduled first, the cheapest LOs from the *soon* and *eligible* sets may be blocked or overlooked. However, if the cheapest LOs are scheduled first, they could block *now* LOs that have no other compatible FUs, which would force the addition of a costly FU. This problem can arise whenever a lower priority LO is selected while a higher priority group contains an LO that has less allocation options than there are members in that higher priority group. By selecting the lower priority LO in these cases, it is possible that the higher priority LO would eventually be unnecessarily blocked in the current or some subsequent clock cycle, which would lead to the addition of an unnecessary FU.

A RIft selection algorithm is also constrained by the need to accommodate conditional logic. This means there can be mutually exclusive LOs that can be scheduled in the same FU and clock cycle. Thus the amount of LOs that can be scheduled to free FUs is not known until completed. Thus, selection cannot be treated as simply examining the $\binom{N}{R}$ possible selections, where N is the number of LOs and R is the number of FUs, and choosing that with the lowest cost.

Thus in summary, a solution must:

- not rely on knowing how many LOs can fit into the available FUs.
- continue until no other LO can be mapped.
- ensure that *now* LOs are never blocked.
- ensure that *soon* LOs are never blocked by *eligible* LOs.
- as much as possible choose from the largest possible pool of LOs to ensure selection of the lowest cost LO available.

The algorithm described in Algorithm 1 was developed and implemented to solve these mutually conflicting requirements. First, for every available LO, it is determined how many compatible binding opportunities exist, which is described in steps 1 to 3. The cost of mapping each compatible LO to FU combination is determined in steps 4 and 5. The selection process starts with the *now* LOs, which are initially the only LOs in what can be thought of as the selection pool $LO_{Selection}$ (Step 7). If a *now* LO

Algorithm 1: Selection of next LO to schedule

```
Input  : $LO_{Now}$, $LO_{Soon}$, $LO_{Eligible}$, $FU_{type}$
Output: $LO_{Best}$
1  $LO_{All} = LO_{Now} \cup LO_{Soon} \cup LO_{Eligible}$
2  foreach ($LO_{Candidate}$ in $LO_{All}$) do
3      $FU_{Compatible}[LO_{Candidate}]$ = DetemineCompatibileFUset($LO_{Candidate}$,
       $FU_{Type}$)
4      foreach ($FU_{Candidate}$ in $FU_{Compatible}[LO_{Candidate}]$) do
5          $Cost[LO_{Candidate}, FU_{Candidate}]$ = DetemineCost($LO_{Candidate}$,
           $FU_{Compatible}$)
       end
   end
6  $TooSmall = FALSE$
7  $LO_{Selection} = LO_{Now}$
8  foreach ($LO_{Candidate}$ in $LO_{Selection}$) do
9      if ($0 < |FU_{Compatible}[LO_{Candidate}]| \leq |LO_{Now}|$) then
10         $TooSmall = TRUE$
       end
   end
11 if ($TooSmall \neq TRUE$) then
12     $LO_{Selection} = LO_{Selection} \cup LO_{Soon}$
13     foreach ($LO_{Candidate}$ in $LO_{Selection}$) do
14         if ($|FU_{Compatible}[LO_{Candidate}]| \leq |LO_{Now}|$) then
15             $TooSmall = TRUE$
           end
       end
   end
16 if $TooSmall \neq TRUE$ then
17     $LO_{Selection} = LO_{Selection} \cup LO_{Eligible}$
   end
18 Return LowestCostLO($LO_{Selection}$)
```

exists that has a number of mapping options greater than zero and equal or less than the amount of *now* LOs still to be scheduled, there is a situation where the given now LO's mapping options could all be blocked. Thus a lowest cost LO selection must be constrained to the set of *now* LO's until this condition is removed (steps 8-10). If every *now* LO has sufficient mapping options, then the selection pool of LO's can be expanded to include the set of *soon* LOs in addition to the *now* LOs as in steps 11 to 15. The same conditions are then applied to the selection set to determine if the *eligible* LOs can be added to the selection set (Steps 16, 17). Once the selection set is determined, the lowest cost LO is chosen for allocation, and the whole selection process is restarted. In this way the lowest costing LO possible is selected in a manner that guarantees higher priority LOs will not be blocked in the current clock cycle and makes a best effort to avoid blocking in the future. It is important that steps 2 to 5 be completed before every single LO selection, as compatibility and cost can be affected by the allocation of the previously selected LO. The algorithm will continue until all the remaining unallocated LOs have no remaining allocation options. Not mentioned in Algorithm 1 is that if at this point there is a *now* LO still unscheduled because it did not have any compatible options to start with, it will force the creation of a new FU. RIft is not expected to better or match the resource allocation produced by the conventional mapping algorithms, however, it should be able to satisfy the same latency constraints. How well RIft actually compares, as well as the effect of several extra optional modes of operation, is discussed in the next section.

3.8 Results for Reducing Interconnects

This section presents the first half of results achieved by the RIft system for implementing an HLS design with an interconnect minimizing strategy. However, it is important to realize that the results are expected to be strongly affected by the characteristics of the test cases. For instance, a test case that uses only one type of FU should be expected to have a differing outcome than a test case having several classes of FUs. Likewise, a simple data processing design will result in different characteristics than a control oriented design. Thus a discussion of the pertinent aspects of test cases is presented in SubSection 3.8.1.

It is also important to note that the goal for RIft is to advance the state of HLS with regards to realistic cases for which it would be useful to use such a tool. This implies the use of test cases that work on more than the trivial "validation" examples, usually consisting of less than 40 operations, but nonetheless commonly used for benchmarking. The premise of RIft is to explore HLS in the context of design problems that are large enough for an HLS system to be *useful*. Thus the empirical studies to be presented utilize hundreds of operations, which is a significant departure from most prior work that typically uses small "validation" examples. The larger results presented are, unfortunately, literally without compare, and little in the way of useful observations to previous work can be made in this regard. For this reason an interconnect unaware HLS CAD tool, based on classic List scheduling, has been developed as a fair reference to RIft. For the sake of protocol, a trivial benchmark, the commonly implemented symmetric 16-tap FIR filter has been included to in some way compare RIft to other approaches. Rather than use common small benchmarks, emphasis is placed on how well RIft works for designs of differing attributes. Understanding of how the fundamental attributes of a proposed design will affect RIft performance allows for those expectations to be extrapolated to the arbitrary design. This seems appropriate since HLS is meant to streamline custom designs, rather than redesign that which is already well established.

3.8.1 RIft Context: The Test Cases

The test cases developed can be categorized in several ways. The first has strictly to do with size. As can be seen in Table 3.1, the first part of the naming scheme uses either "large" or "huge". In general, the large cases consist of more than four hundred logical operations, while the huge cases typically contain approximately a thousand LOs. Table 3.1 lists some of the attributes, including a breakdown of the number of instances of each type of LO. The number of clock cycles each test case is constrained to is also listed. This number is typically set to match the critical path so as to maximize the parallelism of the design. Each of the large and huge test cases

Test Case	Initial Design(cycles)[Instantiated]				Operation Allocation		
	+(1)	-(1)	*(2)	C. Cycles	Rounds	Ops/Round	History
Sym16_FIR	15		8	10	-	-	-
large_plus	428			13	15	30	3
large_minus		428		13	15	30	3
large_mult			428	13	15	30	3
large_plus_minus	201	227		13	15	30	3
large_mixed	198	169	73	31	25	18	2
large_long	202	202	94	58	50	10	2
large_wide	186	145	77	9	6	80	2
huge_plus	958			20	20	50	3
huge_minus		958		20	20	50	3
huge_mult			958	20	20	50	3
huge_plus_minus	484	474		20	20	50	3
huge_mixed	425	430	323	56	40	30	2
huge_long	404	442	342	79	60	20	2
huge_wide	312	314	282	16	10	100	2

Table 3.1: Test case characteristics

were randomly generated. The third set of attributes concerns the characteristics of the dependencies within a design. Each test case has a number of consecutively constructed "rounds" which contain a fixed number of LOs. All the LOs within a round are constrained to only use values that have been generated in previous rounds. This negates the possibility of cyclical dependencies. The final parameter is the number of rounds of history. This determines from how many previous rounds a given LO may select values for operand inputs. As the rounds of history are increased, consumed values will likely need to be stored for longer durations, which should result in less register sharing, possibly necessitating a larger amount of registers. When the number of operations per round is increased, more parallelism is introduced, and RIft can use more instances of FU classes to speed execution of LOs. Finally, modifying the number of rounds changes the number of operations in a test case. A test case with a larger number of rounds will have more internal dependencies, a longer critical path, and should be able to attain greater hardware reuse. In the following synopsis of each type of test case, "*" is used to denote both large and huge instances.

***_minus:** This is the simplest case, consisting only of subtraction operations. The dependency structure is intended to be moderate.

***_plus:** Similar to *_minus, except as a commutative operation, addition operations result in greater complexity due to the greater flexibility of input binding.

***_mult:** Same as *_minus, except a multicycle multiplication is substituted for addition.

***_plus_minus** Combination of two FU classes should likely be more difficult due to the possibility of more distinct input sources.

***_mixed:** Similar to *_plus_minus except multiplication, a two cycle operation, is also included.

***_long:** The same as *_mixed, however the dependency structure has been made "longer". The number of rounds has been greatly increased and the number of LOs per round greatly reduced. This should increase the critical path and reduce concurrency. This is expected to be the worst performing test case under RIft.

***_wide:** Same as *_mixed, but on the opposite side of the spectrum as *_long. Concurrency is greatly enabled by a larger number of operations per round but with fewer rounds.

These test cases are used extensively in this chapter to demonstrate the abilities of RIft, and also in the next chapter where FT is presented.

3.8.2 Testing Procedure

The test cases described in the previous section are the starting point for RIft. The format of these examples consists of a standard Verilog text file with the normal input and output definitions and optionally standard Verilog sections that are not parsed by the RIft CAD tool. Within this file, multiple HLS sections can be inserted. Each section must list the FU classes available and their characteristics, such as their symbol, number of operands and clock cycles, and commutativity property. The main part of this section are the equations that are listed by the designer. They are assumed to be written in a sequential, blocking manner. Variables that are to be outputs can be explicitly indicated. It is also possible to specify which clock cycle an input becomes valid. Stored values that are not consumed by any other operation are assumed to be outputs. Inputs are assumed to be unregistered, however, registered behavior can be specified by using an input to register assignment at the beginning of the equation section.

In order to provide a baseline upon which to compare RIft, a form of List scheduling has also been implemented which can act on the same input file. To be fair, this implementation does not incorporate many of the improvements introduced since List scheduling's inception, such as more advanced treatment of control structures and so forth. However, many of these features are not present in RIft either, although it should be possible to incorporate them. Thus, the use of List scheduling represents a reasonable and realistic reference point from which to compare the advantages and disadvantages of RIft.

From the starting point of the Verilog HLS designs, the RIft and List CAD tools are used to produce a finalized design in legitimate Verilog form. In order to evaluate the area and performance of a finalized design, it must be compiled by a Verilog compiler. For the purposes of this testing, Altera Corporation's Quartus II version 4.0 software was used. Part of the reason for this was the ability to use Altera's proprietary LPM modules as the basis for FUs, the basic units that are built upon by HLS. The time required for each step has not been recorded for each example, but generally, on a 2.8GHz Pentium 4 PC with 512MB of RAM, RIft takes roughly 1-100 seconds, whereas subsequent compilation by Quartus requires approximately 10-180 minutes, depending on the size of the initial design.

It is possible, and even likely, that different commercial tools would result in better or worse final results, the different target technologies notwithstanding. However, the effectiveness of various synthesis tools at finalizing RTL designs is outside of the scope of this work. By using the same Verilog compiler tool for both the List and RIft approaches, the relative differences in compiler performance are assumed to cancel out.

Because of the size of the examples under consideration, the number of clock cycles

Test Cases		# Instances		FU Muxes				Reg Muxes			
	Clo	List	RIft	List		RIft		List		RIft	
	cks	$+,-,*,R$	$+,-,*,R$	#	>	#	>	#	>	#	>
Sym16_FIR	10	3,0,5,23	3,0,5,28	12	7	10	5	6	3	0	0
large_plus	13	34,0,0,85	34,0,0,127	68	13	68	9	85	8	0	0
large_minus	13	0,34,0,85	0,34,0,122	68	13	68	10	85	8	0	0
large_mult	26	0,0,34,85	0,0,35,127	68	13	70	8	85	8	0	0
large_plus_minus	13	18,18,0,86	18,18,0,132	72	13	72	11	82	8	0	0
large_mixed	31	7,6,6,50	7,8,8,72	38	25	46	16	43	11	0	0
large_long	58	5,4,7,25	5,6,6,50	30	23	34	19	24	12	0	0
large_wide	9	22,17,20,163	22,17,20,212	118	9	118	8	134	4	0	0
huge_plus	20	49,0,0,161	51,0,0,222	98	20	102	12	155	10	0	0
huge_minus	20	0,49,0,161	0,49,0,229	98	20	98	14	155	10	0	0
huge_mult	32	0,0,62,155	0,0,62,225	124	16	124	11	142	10	0	0
huge_plus_minus	20	25,25,0,164	27,25,0,243	100	20	104	14	158	11	0	0
huge_mixed	56	8,8,15,74	8,9,16,117	62	44	66	29	72	16	0	0
huge_long	79	7,6,11,49	7,9,12,92	48	41	56	25	48	16	0	0
huge_wide	16	21,21,39,230	21,21,39,333	162	16	162	14	221	7	0	0

Table 3.2: List vs. RIft: clock steps and FU usage

needed to complete the work, also referred to as the latency, is not a complete measure of performance. Interconnect complexity is expected to have a considerable affect on clock delay. Thus the period of a design that has been completely synthesized must be considered as well. The actual performance of a circuit is the latency $\times$ the period, which, for the purposes of this research, will be called throughput. For all test case results reported here and in the next chapter, the List and RIft or RIFT versions of each are both constrained to the same number of clock cycles. Thus a realistic performance comparison can be made from the reported frequency alone.

3.8.3 Experimental Results

A comparison of the results achieved by List and RIft scheduling is available in Table 3.4 while a breakdown of HLS related hardware components is given in Table 3.2. To make it easier to discern the difference between the structure of List and RIft, Table 3.3 presents the difference in the number of components (RIft - List). In each table, the test case name is listed in the first column. The second column of Table 3.2 states the number of clock cycles needed, while the third and fourth columns detail

	FUs Instantiated		FU muxes		Reg muxes	
Test Case	(+,-,*)	# Registers	#	>	#	>
Sym16_FIR	0,0,0	5	-2	-2	-6	-3
large_plus	0,0,0	42		-4	-85	-7
large_minus	0,0,0	37		-3	-85	-7
large_mult	0,0,1	42	2	-5	-85	-7
large_plus_minus	0,0,0	46		-2	-82	-7
large_mixed	0,2,2	22	8	-9	-43	-10
large_long	0,2,-1	25	4	-4	-24	-11
large_wide	0,0,0	49		-1	-134	-3
huge_plus	2,0,0	61	4	-8	-155	-9
huge_minus	0,0,0	68		-6	-155	-9
huge_mult	0,0,0	70		-5	-142	-9
huge_plus_minus	2,0,0	79	4	-6	-158	-10
huge_mixed	0,1,1	43	4	-15	-72	-15
huge_long	0,3,1	43	8	-16	-48	-15
huge_wide	0,0,0	103		-2	-221	-6

Table 3.3: The difference in component requirements (RIft - List)

how many instances of each operation are implemented by the respective algorithm ('R' designates registers). It should be noted that, unless otherwise stated, all the RIft test cases use register partitioning, as discussed in SubSection 3.7.2. In every test case, RIft either matches or exceeds List in the amount of instances that are required, and by as much as three subtraction FUs in the case of huge_long. While earlier it was stated that RIft will always be able to make the same latency constraints as List, the significantly different selection processes are not guaranteed to use the same amount of FUs. Although RIft has an elaborate method of dealing with possible blocking scenarios once they are identified, the results suggest that RIft is not as competent as List in avoiding those scenarios in general.

Nonetheless, RIft achieves a fairly substantial gain in *both* area and performance, as detailed in Table 3.4. The reduction in area ranges from 10% to 50%. Area is measured in terms of Logic Elements (LE). These are the smallest resource type based on a LUT in most of Altera's FPGAs. Performance, which should typically be measured in terms of throughput, can in this case be measured directly from the maximum frequency, as the number of clock cycles used by List and RIft are the same. This is in direct contrast to most previous research that considers performance only in

	Area(FPGA LEs)			Performance (MHz)		
Test Cases	List	RIft	%Less	List	RIft	%Gain
Sym16_FIR	2,550	2,537	0.5	25.44	33.11	30.1
large_plus	6,594	3,942	40.2	17.76	25.36	42.8
large_minus	6,865	4,203	38.8	17.55	23.5	33.9
large_mult	9,178	6,593	28.2	34.13	47.72	39.8
large_plus_minus	6,662	4,487	32.6	17.76	22.74	28.0
large_mixed	5,744	4,730	17.7	18.25	24.46	34.0
large_long	4,868	4,341	10.8	22.12	28.25	27.7
large_wide	8,094	6,738	16.8	18.67	25.58	37.0
huge_plus	13,925	8,200	41.1	13.71	22.04	60.8
huge_minus	14,381	9,013	37.3	13.59	19.97	46.9
huge_mult	18,920	12,910	31.8	30.54	40.13	31.4
huge_plus_minus	14,102	9,542	32.3	13.33	20.23	51.8
huge_mixed	14,625	10,232	30.0	15.72	22.46	42.9
huge_long	11,963	9,301	22.3	16.66	23.64	41.9
huge_wide	17,200	13,239	23.0	15.47	20.94	35.4
Average (of large_*, huge_*)			28.8			39.6

Table 3.4: List vs. RIft: area and performance

terms of clock cycles and neglects entirely the effect of the period on throughput. The greatest consistent gains are achieved with the *_plus and *_minus test cases, and the reasons for this can be determined from columns 5 to 12 in Table 3.2. Columns 5 and 7 list, for each test case, the number of muxes that are needed to direct input wiring into FU operands for List and RIft, respectively. Columns 6 and 8 list, in terms of number of inputs, the size of the largest mux for List and RIft, respectively. Columns 9-12 hold data in the same format, except this data concerns the second tier muxes that feed the registers. The number of FU muxes is obviously not the source of the improvements, as RIft utilizes a similar amount of muxes or more than List for every case, which is expected since RIft tends to use more FUs. The difference is that RIft manages to reduce the maximum mux size, which should reduce the critical path if the number of inputs is reduced by enough to remove a level. To be fair, since FPGAs cannot implement muxes as compactly as with an ASIC platform, the performance gains of up to 60% would probably be somewhat less in a ASIC implementation.

However, for both technologies, the number of two input muxes, and thus the area, used by the macro muxes should be roughly proportional to the number of inputs, and thus both would yet stand to gain.

The greatest difference between List and RIft concerns the Register muxes. Because List scheduling is forced to arbitrarily bind values to registers, it *must* instantiate muxes to control the flow of values to the registers. RIft, however, tends to partition registers based upon the FUs that feed them. It gives a best effort to first reuse these registers before adding wires to a register in another FU's "register partition". There are two reasons the RIft test cases without control statements have no register muxes. The first is that RIft is aware when selecting an FU if there is a register available to hold the produced value. The second reason is that the cost coefficient for registers is, for this example, the same as a new wire, which causes RIft to use a new register instead of crossing FU register partitions. Because of this, the RIft approach, as documented in Table 3.2, can use as many as twice the number of registers. Because the design is implemented on an FPGA, the extra registers do not tend to add to the area. This is due to the fact that the LEs used to construct the muxes also contain an unused register. The savings only escalate for larger register-mux combinations. When targeted to ASIC implementations, the register is no longer "free", however the area consumed by ASIC muxes should be less already, therefore achieving a reduction in area by way of the "free" register is less important for an ASIC implementation. However, the benefits of a reduced critical path are equally relevant for either technology. The effects of using different register packing directives are discussed later in this section.

The performance and area gains are reduced for *_minus, *_plus_minus, *_mixed, *_wide, and *_long, in roughly that order. The reasons for this are instructive. Although the largest mux size plays a large part in determining performance, it does not seem to be the only consideration. Large_minus looks to have a better mux structure than large_plus, and they both use the same FU, where the '-FU' class is actually an Altera adder LPM that is configured for subtract mode. This configuration was determined, by examining the chip layout, to require two extra LEs and an increase in the FU's critical path of 0.584ns. Thus, by substituting subtraction FUs for addition

FUs in large_plus, the area of large_minus would be expected to increase by 68 LEs (2 LEs x 34 FUs) as opposed to 261 LEs. The extra delay would suggest that the frequency should be reduced from 25.36MHz to 24.99MHz, all things being equal, instead of the observed 23.50MHz. The unaccounted for difference is likely due to the extra routing required to satisfy the additional constraints imposed by commutativity. The same can be noted for huge_plus and huge_minus. Area increased by 426 LEs and frequency dropped to 19.97MHz, as opposed to the 98LEs and 21.76MHz expected due to the use of the subtracting FU's. Therefore, it can be concluded that RIft's ability for improvement can be limited by extra constraints on how values are routed.

A similar conclusion can be drawn from the reduced improvements obtained for the *_plus_minus and *_mixed cases. Improvements in performance are somewhat erratic, as large_mixed is on par with large_minus, and huge_plus_minus compares well to huge_plus and huge_minus. Area improvements, however, are consistently less than for the single FU class counterparts. Because RIft attempts to partition LOs based on datapath requirements, it follows that the use of more FUs allows for more partitions which then can support smaller datapath sets, which improves RIft's routing gains. Although the number of LOs are roughly the same across the large* and huge* test cases, the heterogenous cases use less FUs of each type. Furthermore, source registers now may be fed by an FU of a different class and interclass value transfers only add to the number of distinct path sets that must be accommodated. Thus there is less opportunity for data path partitioning within a class of FUs, and less opportunity for improvements to routing. This fact also explains another larger trend in the data set. As the number of FUs used increases from the large* to the huge* cases, so too does RIft's performance. This is supported by the fact that all huge* cases have both better area and performance improvements than their large* counterparts, especially for the *_mixed, *_long, and *_wide cases. The exception is the timing improvements of *_mult, as large_mult has an outstanding area reduction of 53%, the reasons for which are not fully explained, but may be due to the details concerning the specialized multicycle setup of the multiplier.

A final factor in RIft's performance is the effects of the dependency structure for

	# Components Instantiated								Register "Fan Out"	
	List				RIft					
Test Case	+	-	*	Reg	+	-	*	Reg	List	RIft
large_long	5	4	7	25	5	6	6	50	1.56	2.94
large_mixed	7	6	6	50	7	8	8	72	2.63	3.13
large_wide	22	17	20	163	22	17	20	212	2.76	3.59
huge_long	7	6	11	49	7	9	12	92	2.04	3.29
huge_mixed	8	8	15	74	8	9	16	114	2.39	3.45
huge_wide	21	21	39	230	21	21	39	333	2.84	4.11

Table 3.5: Effects of parallelism on register to FU ratios

particular test cases. Recall that *_mixed, *_long, and *_wide all share the same FU classes (+,-,*) and have a similar number of operations. *_long has a high level of dependency and is thus expected to use fewer units and have a much larger critical path, which has been verified in Table 3.2. Because there are fewer FUs than in *_mixed, it should be expected that RIft will not be able to improve routing as well, which is supported by the results in Table 3.4, where *_long has a lower frequency. What may be somewhat counter-intuitive is RIft's inability to improve huge_wide's performance more than huge_mixed. This contradicts the earlier suggestion that RIft should perform better as the number of FUs per class increases. The factor that reduces RIft performance with increasing parallelism is suggested by Table 3.5, which sorts large_* and huge_* by increasing parallelism. It presents calculations of the ratio of registers to FUs. This measure is not a legitimate measure of "fan out", however it does demonstrate that as dependency is reduced and parallelism increased, the average FU will on average likely be required to accommodate more inputs, and that RIft will suffer from this factor more than List, which has randomly allocated inputs. Thus, as parallelism increases, the growth in number of input sources feeding FUs tends to reduce RIft's ability to reduce interconnects and thus performance. The net result is the suggestion that RIft's performance as a function of parallelism is akin to a bell curve: decreasing from some optimum with both less and more parallelism.

As noted previously, RIft tends to eliminate routing at the register level because, by default, RIft partitions registers under FUs so that each register has only one input, and thus does not need a mux. The test cases were rerun twice with partitioning

				FU muxes						Reg muxes					
	# Registers			#			>			#			>		
Test Case	B	∞	2	B	∞	2	B	∞	2	B	∞	2	B	∞	2
Sym16_FIR	28	-4	-4	10	1		5			0	6	7	1	2	1
large_plus	127	-36	-24	68			9			0	87	93	1	4	1
large_minus	122	-33	-18	68			10	-1	1	0	86	90	1	4	1
large_mult	127	-38	-24	70	-2	-2	8			0	89	94	1	4	1
large_plus_minus	132	-40	-25	72		2	11		-1	0	91	93	1	4	1
large_mixed	72	-29	-18	46	-6	-6	16	-1	-1	0	43	50	1	3	1
large_long	50	-25	-15	34	-2	2	19	-3	-3	0	25	34	1	5	1
large_wide	212	-47	-37	118			8			0	133	124	1	2	1
huge_plus	222	-57	-36	102		-4	12	-1	-1	0	157	166	1	4	1
huge_minus	229	-65	-41	98	4		14	-2	-1	0	159	166	1	4	1
huge_mult	225	-66	-39	124			11	-1		0	158	178	1	4	1
huge_plus_minus	243	-83	-48	104	-4	-2	14	-1		0	160	169	1	5	1
huge_mixed	117	-47	-24	66	2		29	-4		0	70	90	1	4	1
huge_long	92	-45	-23	56	-2	-4	25	-3	3	0	47	66	1	7	1
huge_wide	333	-91	-63	162	-14	-14	14	-2	-2	0	228	235	1	4	1

Table 3.6: Effect of register sharing on component requirements

turned off: first with unlimited register sharing, and then with a mux input limit of 2. A report of the resulting logical structure is made in Table 3.6, while Table 3.7 accounts for the area and performance consequences of the final gate level designs. Columns headed with a 'B', which stands for 'Baseline', give the data from the previous partitioned test cases. ∞ indicates unlimited sharing, and a column header of '2' signifies the case with a register mux input limit of 2. To facilitate comparison, Table 3.6 shows only the difference in the number of units needed with respect to the baseline. If there is no change, the cell is left empty. Likewise, Table 3.6 reports the percentage difference as compared to the baseline case. Unlimited sharing results in an approximately 33% reduction in registers and the limited sharing about 20%. The *_long cases benefit the most from unlimited sharing at 50%, which suggests RIft has more difficulty packing examples with a high amount of internal dependencies. The effects on the number of FU input muxes is somewhat minimal and mostly due to a variation in the number of FUs instantiated. The maximum size of the FU muxes is marginally reduced, most likely due to the reduced number of distinct input sources. The greatest change is that, by definition, most registers now have muxes, and the maximum number of inputs will have increased. Even with unlimited sharing, RIft

	Area(FPGA LEs)			Performance (MHz)		
Test Cases	Base	∞(%)	2(%)	Base	∞(%)	2(%)
Sym16_FIR	2,537	-2.2	-2.6	33.11	-5.2	-0.5
large_plus	3,942	19.7	-4.9	25.36	-19.6	-6.2
large_minus	4,203	17.6	-4.0	23.50	-17.1	-7.2
large_mult	6,593	8.9	-3.4	47.72	-7.8	-5.7
large_plus_minus	4,487	14.9	-7.9	22.74	-9.6	-1.0
large_mixed	4,730	-1.8	-11.8	24.46	-6.7	-0.6
large_long	4,341	-6.6	-3.9	28.25	-13.7	-3.6
large_wide	6,738	2.7	-3.8	25.58	-10.4	-9.8
huge_plus	8,200	13.0	-1.0	22.04	-16.7	-10.2
huge_minus	9,013	11.7	-6.8	19.97	-11.4	-8.5
large_mult	12,910	10.6	-2.6	40.13	-12.6	2.2
huge_plus_minus	9,542	12.0	-6.2	20.23	-16.8	-8.2
huge_mixed	10,232	3.5	-2.5	22.46	-10.5	-10.4
huge_long	9,301	4.1	-4.1	23.64	-17.0	-14.2
huge_wide	13,239	8.7	-5.3	20.94	-23.0	-11.1
Average (of large_*, huge_*)		8.5	-4.9		-13.8	-6.7

Table 3.7: Effect of register sharing on area & performance

still reuses existing wires whenever possible, which tends to limit the mux size to four or five inputs, and at most eight for the *_long cases.

While the original intent of register sharing had been to save area and possibly remove complexity from the FU level muxes, the area and performance results suggest this is not a very good idea. At best, using two input register muxes is supposed to be a free improvement on an FPGA platform since this mux is already incorporated in the same LE as the register. In fact, actual area is slightly reduced by 1-10% and 2.9% on average with two-input muxes, while the cases with mixed FU types seemed to fare the best. However, performance is reduced by a larger 6.9% on average. The cause of this performance hit most likely results from the fact that almost every remaining register now has a mux and thus the FSM must manage that many more control signals to steer them. The case of unlimited sharing is much worse, where area rose on average 11.5% and as high as 20% in one case. At the same time, performance dropped 13% on average and 21% at the extreme. From these comparisons regarding mux sharing, two important conclusions can be made, at least for HLS on FPGAs. The first is that routing has a much larger role than registers in the final area and performance of a design. The second somewhat surprising conclusion is that multiplexing, or the sharing of registers is actually found to be detrimental to large scale HLS design! This conclusion is in direct contrast to conventional thinking and methodology, and, to the author's knowledge, has not previously been published.

If the target technology was ASIC instead of FPGA, the results could be expected to be less dramatic. As mentioned earlier, the ratio of mux to FU area and delay costs will likely be less, given the distributed nature of large muxes in FPGAs. Thus the area and performance improvements of managing interconnects may also be less. Nevertheless, the principles of managed interconnects still hold true. This would suggest that improvements may not be as great on a proportional basis, or may not materialize until relatively larger muxes are required. Given that ASIC implemented designs often are performance oriented, these smaller improvements may yet be significant. Thus, though these conclusions are more immediately applicable to FPGA implementations, they may be useful in ASIC platforms as well.

Though RIft performance depends to a certain extent on the design it is being used on, overall RIft has demonstrated the significant real performance gains that can be made to HLS derived ICs by considering the important effects of interconnects, especially for large, realistic applications.

Chapter 4

Fault Tolerance

The new fault tolerance system introduced in the following sections was developed in effort to determine if a more efficient alternate approach for large parallel fault tolerant systems could be devised. A goal is to report what the costs might be in terms of area and actual performance. The final design was motivated by the following criteria and rationale:

- Repair Intermittent & Permanent Faults

  Most non-critical applications such as consumer, telecommunication, etc. can or already do tolerate infrequent single event faults in the data path. Yet these are the most costly to remedy. Alternatively, repair of intermittent and permanent faults would greatly increase reliability and lifespan. This level of tolerance would also suggest that repair need not be immediate.

- Minimization of Interconnects

  Reuse existing infrastructure as much as possible. Minimize interconnect area and balance mux sizes. This is theorized to minimize the period and reduce area, factors that are especially costly with the inclusion of FT.

- FSM simplification

  It would be better not to change both the timing and positioning for FT from a complexity standpoint. Redundancy implies LOs *must* be moved to alternate modules, thus timing should be constrained. If possible, limit each LO to allocation on one normal and one redundant FU. Also, the execution FSM and FT FSM should ideally be separate, which should reduce the possibility that the combined FSM will contribute to the critical path.

- Fault Tolerance FSM Scalability

  This is required for arbitrary amounts of parallelism.

- Internally Initiated Compensation

  For the great majority of applications, human initiated repair can be as or more costly than complete product replacement. Thus compensation must be self initiated.

- Online Fault Detection & Isolation

  This is required in order to achieve the prior requirement of uninitiated self repair.

- Platform Agnostic

  An HLS solution should not rely on unique features of a particular target technology such as ASIC or FPGAs. Parallel FT designs would be useful with both ASICs and FPGAs, and loss of generality would severely limit the usefulness or adoption of any such CAD tool.

- Module Agnostic

  This means that the design should accommodate arbitrary module types. This would preclude the use of mathematical properties and transformations for advancement of HLS objectives.

The result is a vision of an architectural approach that adds robustness to everyday, non-life critical applications in a way that is transparent to the end user. It is possible that an architecture based on such a design philosophy could also indicate to the end user, once FT has been activated, that reliability has been compromised and

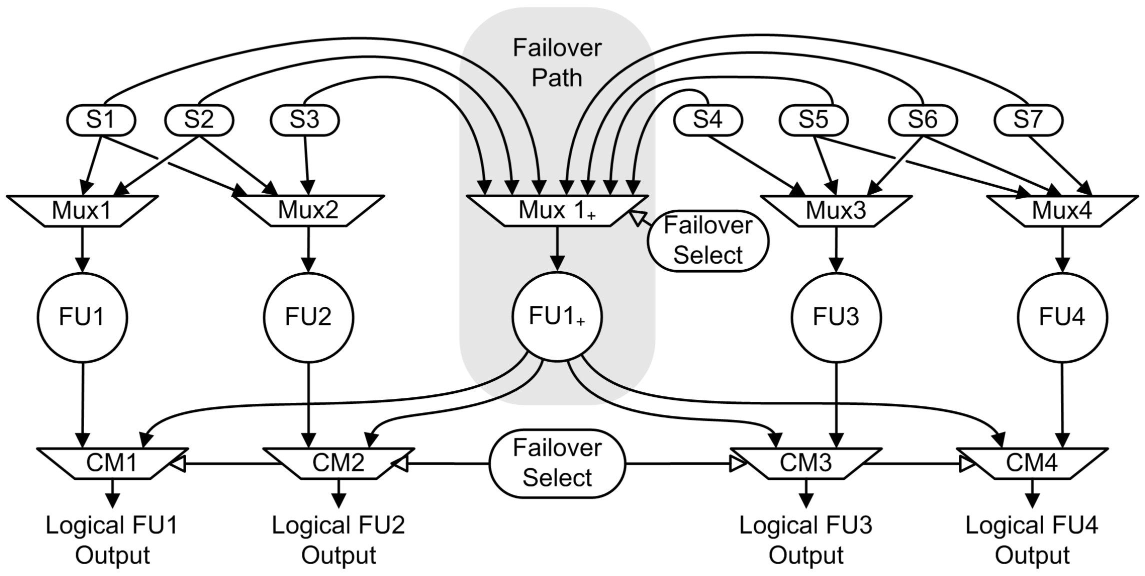


Figure 4.1: Common $N + 1$ redundancy

that preemptive replacement when convenient (perhaps under warranty) might be considered.

In the next section, the strategy behind RIFT will be presented in detail. Section 4.1.3 details different implementations of distinct Failover FSMs (FFSM) to control RIFT. Detailed experimental results are presented in Section 4.3. Finally a critique of RIFT and its FT capabilities is made in Section 4.4.

4.1 RIFT Implementation Strategy

4.1.1 RIFT Topology

The requirements in the previous section are fairly constrictive. Separation of fault and control FSMs suggests that redundancy should be managed separately from the HLS aspects of the design in much the same manner that [24] proposes FPGA redundancy in a way that is transparent to the compiler. To achieve this, our starting point is the classic $N + 1$ fail over redundancy approach [38].

The most common $N + 1$ configuration mentioned in HLS FT is the "common

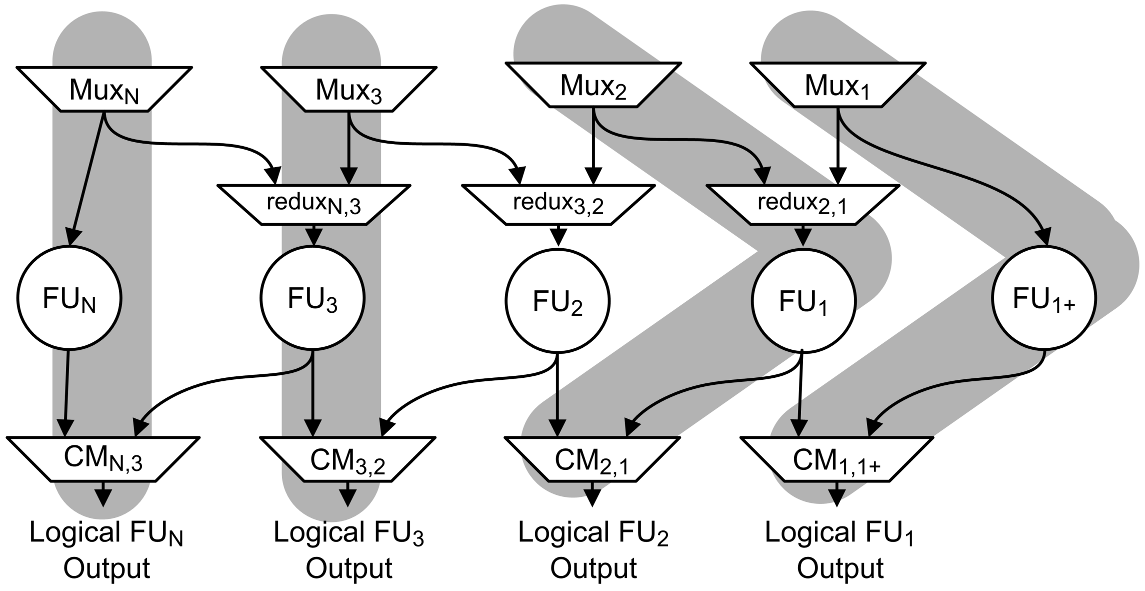


Figure 4.2: Balanced $N + 1$ redundancy

redundant unit" model, as seen in Figure 4.1, of which the FUs have only a single operand and input mux for clarity. It is obvious that this configuration is not scalable, as the redundant mux will disproportionately require more inputs as more FUs are added, significantly increasing the critical path. An alternative Balanced $N+1$ system has thus been selected. Figure 4.2 demonstrates this balanced "buddy" arrangement. FU_{1+} is the redundant FU and the lower "buddy" to FU_1. FU_1 is a standard functional unit that is the "upper buddy" to FU_{1+}. All other standard functional units are arranged in this pattern, where each has an upper and lower adjacent buddy. The exception is that the uppermost unit has only a lower buddy, while the redundant unit only has an upper buddy. The upper and lower terminology is used because "left" and "right" can be misleading, especially when correlating each side with bit indexing for the FT FSM (Section 4.1.3). If a particular FU fails, the logical operations to be processed by that FU will transparently be transferred to the failed FU's lower buddy. The lower buddy will offload its work to its own lower buddy, and so forth, until the redundant FU is pressed into service. The reason for this configuration is that there is no need for one redundant unit to incorporate all the inputs used by all N standard functional units. Instead, each standard unit must accommodate only its own inputs and those strictly used by its upper buddy. The inputs of the upper buddy that are only used for failover are not included in the lower buddy's input set. Although each standard functional unit's input mux size is increased, it will be an increase that can be balanced among the whole group of functional units. Thus a large increase in the critical path in the common configuration is exchanged for a smaller distributed path delay in the balanced topology. Another advantage of this arrangement is that every non-redundant unit has exactly two configurations: a standard mode and a redundant mode. If, for instance, FU_2 is found to be faulty, FU_2 and all its lower buddies can switch to their redundant modes, thus allowing the real FU_2 to be bypassed, as indicated by the shaded paths in Figure 4.2. This feature allows fail over to be orchestrated independently of the control FSM, a fact which is used in Section 4.1.3 to separate the control and fault FSMs.

4.1.2 Performance & Area Optimization

Because the goal is to avoid assignments of an LO to multiple FUs as much as possible, in contrast to most HLS derived FT approaches, HLS techniques are not used to determine alternate configurations for FT. HLS, however, does yet have a critical role, which will be elucidated shortly. An example of more detailed depiction of the "Redundant Muxes" in Figure 4.2, which are affectionately called "reduxes" , is shown in Figure 4.3. Figure 4.4 demonstrates an aggregated mux structure that is logically equivalent to the configuration shown in Figure 4.3. Table 4.1 lists how the inputs feeding each collapsed mux might be categorized with respect to that mux. If an input is used in standard mode by both a mux and its upper buddy, it is part of the "shared upper input" set, and similarly so for the lower set. If, however, an input is only used in standard mode on a particular mux, but not its buddies, it will be part of the core set. The converse is that the lower buddy will have said input as a member of the redundant input set. This type of input is represented by S2 in Figure 4.3 and Table 4.1. If a mux has only a core input, the cost to add redundancy will be one new input on the lower buddy, or a 100% increase. If N adjacent muxes have the same connection in their shared lower input set, only one redundant input *still* need be added, implying a relative increase of only N^{-1}. Thus it is beneficial to map operations to functional units in such a way that LOs with common inputs be bound on adjacent FUs, especially if they cannot be bound on the same FU. The equivalent formal optimization goal would be to minimize the redundant and core input sets of every mux. Though increasing N to include the whole set of muxes would certainly reduce redundancy costs, minimization of each mux's complete set of inputs should obviously still be of higher priority. Area and performance of the mux network should be improved if LOs can be mapped in such a way so as to reuse wiring as much as possible, and thus use fewer distinct connections. If, through the above goals, LOs can be mapped such that each mux has only a small subset of the complete set of input sources, taking into account both standard and redundant paths, the critical path and area might both be improved.

RIFT, described in the previous chapter in the context of reducing interconnects, was developed for the purpose of FT oriented HLS optimization as just described. It works by incrementally mapping complete operations, including register assignment and mux wiring requirements, to the balanced $N+1$ fail over topology. FUs are added incrementally in step with the mapping routine, but only as needed. Some of the incremental, path based scheduling concepts used in RIFT were first presented in [12]. However, RIFT has been designed to accommodate the balanced $N+1$ fault tolerant topology and to more aggressively minimize interconnects.

Apart from what has already been described in Chapter 3 regarding RIFT, only a few additions are needed to enable FT support. The first step is to start out by instantiating one FU for each class of operation and also a second redundant FU which becomes the first's lower buddy. As the incremental mapping of LOs to FUs proceeds, any connection made to a FU must also be made to that FU's lower buddy, if not already present. This represents an extra cost, and so must be included in the cost calculations performed for every possible allocation under consideration. Equation

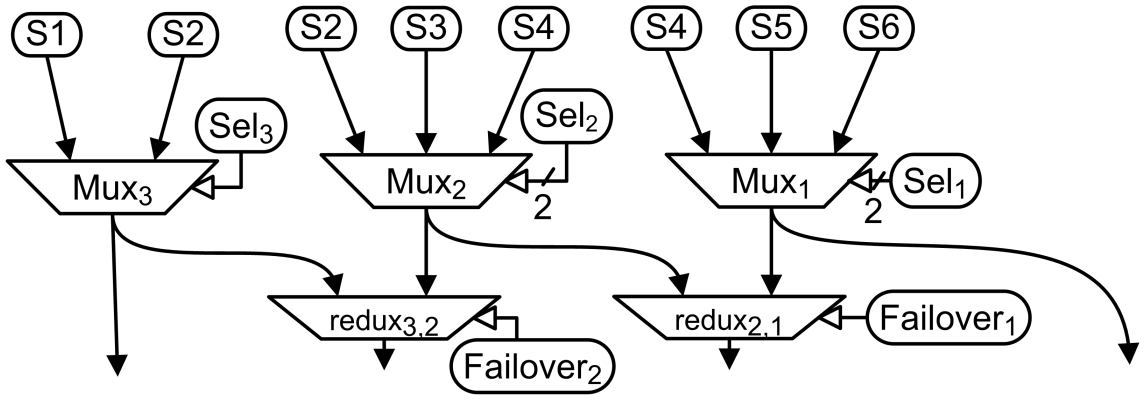


Figure 4.3: Redundant multiplexor (”redux”) FU input network

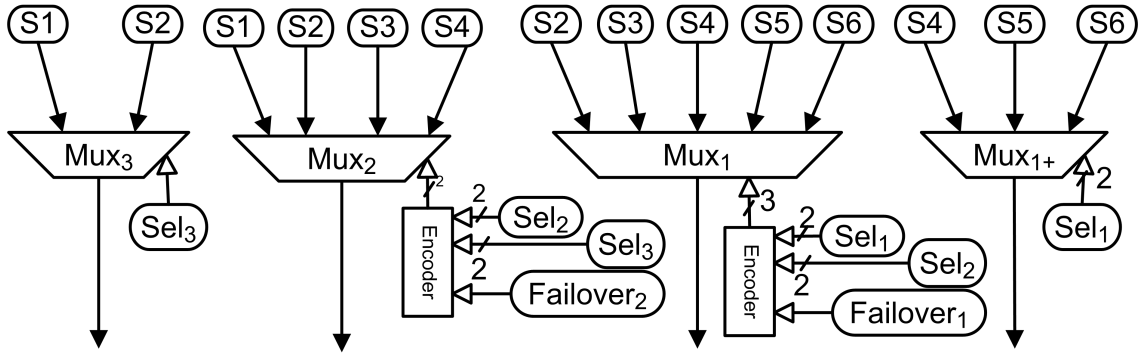


Figure 4.4: Equivalent single level representation

Input Set	W.R.T MUX_2	W.R.T MUX_1
Redundant	S1	S2, S3
Shared Upper	S2	S4
Core	S3	S5, S6
Shared Lower	S4	

Table 4.1: FU mux input set classifications

3.1 must be adjusted to include the extra FT costs.

$$\begin{aligned} Cost_{i \to j} = C_{input} \cdot \sum_{k=1}^{|operands|} (e_j^k + e_{j-1}^k) \\ + C_{reg} \cdot f_j \\ + \overline{f_j} \cdot C_{reg\_wire} \cdot g_j \\ + C_{marginal} \cdot T_j \end{aligned} \tag{4.1}$$

This is achieved simply with the addition of the term e_{j-1}^k in the first term, which represents the additional cost of the redundant wire to the lower buddy if it is not already present.

RIFT specifies FU level muxes as condensed units, instead of the separate mux and redux configuration originally presented. It is likely that the ordering of inputs will be different for the lower buddy mux that is required to take over for the upper mux. This would require the FSM to change the signal to be sent to select the same input based upon the failover state. Because one objective was not to complicate the main FSM by adding FT control, an intermediate block must be built that passes the normal select signals in standard mode, but which accepts the upper buddy select signals and converts them in failover mode.

Conversion blocks are instantiated to feed the select of each mux belonging to a non-redundant FU. Algorithm 2 describes the logic employed to convert Mux_1's select signal for the example in Figures 4.3 & 4.4. It is assumed that the index values for each input starts at '0' on the left and is incremented for each input to the right. Notice the algorithm does not specify a conversion for the case when Mux_2_sel equals '0'. This is because '0' corresponds to input S1 of Mux_2, which is part of its "Redundant" set. Thus, '0' does not need to be converted because it is not used by Mux_2 in standard mode.

While there is some cost associated with the conversion module, there is, however, a second advantage. Without them, Verilog compilers would have a much easier time identifying the presence of redundant input subsets amongst buddy muxes. This setup should serve to keep the FU muxes more distinct and thus less adverse to a datapath fault that might affect both the standard and failover input connections. Another

Algorithm 2: Mux select failover conversion

```
Input: Mux_1_select, Mux_2_select, Fail_over$_2$
Output: Mux_1_select_converted
if $Fail\_over_2$ = $Off$ then
1   Mux_1_select_converted ← Mux_1_select
else
    switch Mux_2_sel do
        case '1'
2           Mux_1_sel_converted ← '0'
        case '2'
3           Mux_1_sel_converted ← '1'
        case '3'
4           Mux_1_sel_converted ← '2'
        otherwise
5           Mux_1_sel_converted ← Don't Care
```

advantage is that this setup helps to obscure, as will be discussed in SubSection 4.1.4, the fault detection strategy to the RTL compiler.

4.1.3 Fault Detection & Isolation

In order to use the FT capabilities discussed so far, a system needs to be in place that will both detect the presence of faults and determine which FU is faulty. It is important to define what is meant by the terms "error", "fault", and "defect". An error has occurred when a logical inconsistency is found in a computation. Its origins may be due to the environment or an internal problem. A fault is an error that is more precisely defined because its origin has been isolated to a particular hardware component. A fault may be caused by an external factor such as an energized particle hitting and altering a stored charge, which would be an SEU. In this case, it could be expected that the fault, and thus the logical error produced by it, will be unlikely to recur. Thus this is referred to as a transient fault. A fault may also be the result of a defect in that particular hardware component. A permanent defect causes

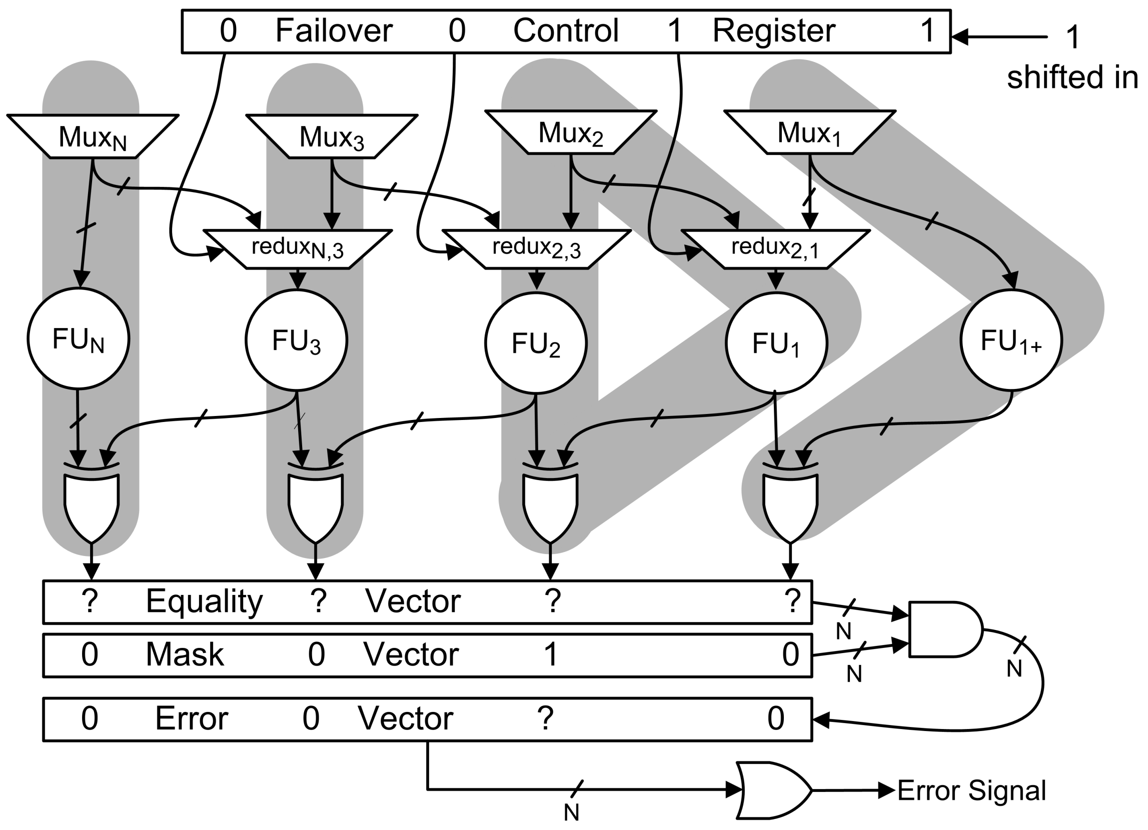


Figure 4.5: $N+1$ Topology used for error detection

a permanent fault, however this may not always be expressed as a logical error. An intermittent defect is one that results in sporadic but recurring faults and thus potentially errors as well. Often an intermittent defect will develop into a permanent defect.

The recovery methods discussed in Section 2.3 deal with how to catch and correct every *error* that is expressed, often at considerable cost. The premise behind RIFT is instead to offer the detection and repair of defects, while possibly missing some errors. If this proposition is acceptable for the application under consideration, RIFT offers an elegant solution to FT.

All error detection strategies operate on the premise of duplicating units of work, either temporally or spatially, and comparing the results for verification of correctness. In the RIFT $N+1$ failover topology, this can be done, when in standard mode where all units are still assumed to be without fault, by putting an FU in failover mode. The result will be that both the FU in failover mode and its lower buddy will be operating with the same input data. This is made possible because of the addition of redundant inputs to the lower buddies' input mux and the conversion block driving that mux's select signal. If the failover FU has already been found to be defective, the circuit would already be ignoring its output regardless. Otherwise, the result can be compared with that of the lower buddy for an equivalence check. Thus, at any given time, a single FU in a buddy chain can be tested by putting it into failover mode and using its lower buddy for verification. Of course, as in a normal failover, all other lower FUs must also be in failover mode. The shaded paths in Figure 4.5 represent the active data channels in use. In this particular case, FU_2 is in failover mode and its lower buddy, FU_1, is duplicating FU_2's calculations to produce a result that can be used to verify FU_2's correct operation. Note that, for clarity, the collector muxes are not shown and single operand FUs are used. The comparison operation is completed by applying a bitwise XOR to the outputs of each FU buddy pair. Because the switch to failover mode can be done transparently for any FU, each FU can be sequentially tested during normal operation. Thus RIFT forms the basis for what is essentially a transparent, online, roving FU verification ability.

4.1.4 Masking Redundancy from the Compiler

As alluded to previously in Section 4.1.2, at a fundamental level, the FT infrastructure should never work. Modern compilers go to great lengths to identify and remove "redundant" logic, in part because this is by definition untestable. Theoretically, the values produced by FUs are being compared in such a way that, unless there is a fault, a difference will never be detected. Of course, the compiler cannot anticipate that differences are expected due solely to a fault, something that is outside the compiler's consideration. Thus there is the situation that the entire FT infrastructure is activated by a condition which should logically never occur. Normally, when such situations are discovered, the compiler will try to remove or "optimize" the redundant logic out of the design. Obviously if this were to happen it would be detrimental to the usefulness of the redundancy intentionally added to compensate for faults. Fortunately the compiler can be deceived. In this case, the use of a shifting error detector, FU muxes that have largely indistinguishable redundant input sets, and the novel FU mux select conversion blocks all serve to greatly obfuscate the existence of a very large redundant logic path. Verification that the redundant systems remain intact has been obtained by checking whether the redundant FUs, as well as other components, have actually been instantiated. It should be noted that conventional HLS FT systems that rearrange LOs for FT will in general not experience this problem.

4.2 FSMs for Failover

The previous paragraph describes a method to detect errors, however, more must be done to isolate a defective FU based on the errors that are reported. If an FU and its lower buddy are in failover mode and report an error, it is uncertain which of the two FUs is actually defective and thus which one should be compensated for. A separate Failover FSM (FFSM) is added to the RIFT architecture to coordinate defect testing to facilitate the identification of defects from reported errors. A separate FFSM must be added for each class of FU present. Several approaches with varying costs and quality of coverage can be taken.

4.2.1 Common FFSM Infrastructure

Three different examples of possible FFSMs are presented, yet they share some common infrastructure as presented in Figure 4.5. The "Failover Control Register"(FCR) consists of N bits, one for each non-redundant FU. Its purpose is to enable the respective FU's failover mode by controlling the select conversion units and also the collector muxes that dictate the flow of data through the datapath. Since, when the N^{th} FU is required to be in failover mode all the lower FUs must also be failed over, it is convenient to instantiate the FCR as a shift register that shifts in a '1' from the lower side. Determining the correct state of the FCR is the principle task of each version of the FFSM. The second common part of the FFSM infrastructure is the generation of the Error signal. The "Equality Vector" also consists of a single bit for each FU of the class, excluding the redundant FU. Each Equality Vector bit is driven by a non-zero detector coupled with a comparator. The comparator is a multi-bit exclusive OR (XOR) gate that detects if there is an inconsistency between the monitored FU and its lower buddy. The Equality Vector's purpose is ostensibly to report an error, but at any given point, most of the FU pairs are expected to be different as they are not in test mode and, consequently, most of this vector will always be reporting "erroneous" values. Thus the Mask Vector is needed, which is the same width as the Equality Vector and consists of all zeros except a single '1'. The '1' is shifted to match the FU pair that is under test. By bitwise ANDing the Mask Vector with the Equality Vector, only the result from the comparator corresponding to the FU under test will be passed, which is called the Error Vector. An Error Signal is then generated by ORing all the bits of the Error Vector together. The least scalable part of the whole FFSM system is this final '1' detector, as it must accept an input from every FU. Because large OR gates have a relatively short path and the result is immediately registered, it is very unlikely the OR will ever affect the critical path. In the next several subsections, this infrastructure is the basis of the different FFSM variants.

4.2.2 The Simple FFSM

The first FFSM to be presented is the "simple" FFSM. It is outlined in detail in Algorithm 3. The problem that must be solved by each FFSM is determining, when an error is detected, which of the failover buddies is correct and which is at fault. The general idea of the simple approach is to loop through each FU and test them sequentially. If an error is detected when testing, for instance, FU_2, either FU_1 or FU_2 is responsible for that error. The simple FFSM will take note of that error by shifting from stage 0 to stage 1 (Step 19) and proceeding as normal (Step 20) to begin testing FU_3. Assuming the faulty FU continues to produce errors, the FFSM can deduce whether FU_2 or FU_1 is faulty based on whether a second error is detected while testing FU_3. If one is, the simple FFSM determines that FU_2 is faulty, shifts

Algorithm 3: Simple Failover FSM

```
Parameter: N, THRESHOLD
Input: $I_{ERROR}$, $I_{RESET}$, $COUNTER$
State: $V_{FCR}[N-1:0]$, $S_{STAGE}$, $S_{ERR\_LATCH}$
Output: $O_{FAIL}$
if ( $I_{RESET}$ ) then
    /*Reset All State Variables */
    $V_{FCR} \leftarrow [0...01]_2$   /*The Failover Control Register (binary vector) */
    $S_{ERR\_LATCH} \leftarrow 0$   /*Latches an Error for each COUNTER loop */
    $S_{STAGE} \leftarrow 0$   /*Distinguishes the 1$^{st}$ and 2$^{nd}$ stage of fault */
    $O_{FAIL} \leftarrow 0$   /*Set in Failover mode: Disables testing */
else if ( $O_{FAIL}$ ) then
    /*Failure mode: Determine if $V_{FCR}$ needs shifting down */
    if ( $S_{STAGE} = 1$ ) then
        $V_{FCR} \leftarrow 0 \gg V_{FCR}$
        $S_{STAGE} \leftarrow 0$
else if ( $COUNTER = THRESHOLD$ ) then
    /*Check if Error occurred this loop: act appropriately */
    if ( $S_{STAGE} = 1$ ) then
        $O_{FAIL} \leftarrow 1$
        $V_{FCR} \leftarrow 0 \gg V_{FCR}$
        if ( $S_{ERR\_LATCH}$ ) then $S_{STAGE} \leftarrow 0$
    else if ( $V_{FCR}[N-1]$ ) then
        if ( $S_{ERR\_LATCH}$ ) then $O_{FAIL} \leftarrow 1$
        else $V_{FCR} \leftarrow [0...01]_2$
    else
        if ( $S_{ERR\_LATCH}$ ) then $S_{STAGE} \leftarrow 1$
        $V_{FCR} \leftarrow V_{FCR} \ll 1$
    $S_{ERR\_LATCH} \leftarrow 0$
else if ( $I_{ERROR}$ ) then
    /*Record occurrence of error during this COUNTER loop */
    $S_{ERR\_LATCH} \leftarrow 1$
```

the FCR down one bit, transfers to the fault mode, and resets the stage to 0 (Steps 11-14). If an error is not detected, the simple FFSM deduces that FU_1 must have been the faulty FU. In this case the FFSM also goes to the Fault mode and shifts the FCR down one bit (Steps 12-13), however the stage is left at 1. This way, when the fault mode is entered in the next clock cycle, the FFSM knows it must shift the FCR down one more time to compensate for faulty FU_1 (Step 8). Because of this extra step, the FCR need only support a single bit down shift, as opposed to a two bit down shift.

The isolation procedure just presented must deal with the special case of diagnosing the N^{th} FU, not counting the lowest redundant FU. Since there is no $(N+1)^{th}$ FU, there is no way to look for the presence of a second error. Thus when the FFSM starts testing the N^{th} FU, it prepares to roll over to the beginning of the buddy chain (Step 17). If the $(N-1)^{th}$ FU does not report an error, but the N^{th} FU does, the FFSM assumes that the N^{th} FU is faulty and, without the second verification test of stage 1, goes directly to the fault mode (Step 16).

Although this FFSM is fairly simple, it possesses at least one major inadequacy. In the event of the detection of an SEU unrelated to an actual defect, the simple FFSM *will* go into fault mode whether a second error is detected or not. This is the reason that testing of each FU is carried out until the $COUNTER$ reaches the $THRESHOLD$ and only then is the decision on how to proceed made. The $THRESHOLD$ is, in fact, when $COUNTER$ becomes all ones in the actual RIFT implementation, and the designer can only specify the width of $COUNTER$, which is shared by the FFSM for all FU classes. During each clock where $COUNTER$ does not equal $THRESHOLD$, the Error Signal S_{ERROR} is checked (Step 22), and if an error is detected, flagged in the S_{ERR_LATCH} state bit (Step 23). This reduces the likelihood that errors originating from the upper unit will be missed in the second stage after the first error has been detected. Though the designer can adjust the susceptibility to intermittent faults by adjusting the $COUNTER$ width, exposure to SEU is in no way reduced. If this should be an issue for the application in question, other FFSM strategies may be more suitable.

4.2.3 The Counting FFSM

The "Counting" FFSM is an attempt to mitigate vulnerability to SEUs seen in the simple FFSM and is presented in Algorithm 4. With this FFSM, there is no $COUNTER$ to wait a certain amount of clock cycles for every FU test. Instead a counter of designer specified width is included with each FU pair comparator. This is used to count the number of errors caught while that particular FU is in test mode. Once the counter has reached the top of its range, it is prevented from restarting from 0 and also forces that particular Equality Vector bit to 1, which indicates that a number of errors have occurred. This allows the FFSM to be fairly confident that any Error Signal received is not SEU induced. This FFSM continuously tests each FU in a sequential manner until it sees an Error Signal for two FU tests in sequence (Steps 9-14). The end FUs are still the weak points of the counting FFSM setup. Each one lacks a second pair with which a second test may be used to corroborate a failure detected by the first reported error. As with the simple FFSM, the failure of an end FU must be assumed if an error signal reported in absence of an error from the next innermost FU. For example, if while testing FU_1 an error signal is detected, FU_1 has already detected $2^W - 1$ errors, where W is the designer designated width of each failover pair counter. If FU_2 does not also report an error, the FFSM assumes that FU_{1+} is the source of the errors and acts appropriately (Steps 13-14). The danger is that FU_1 is the true source of the errors, and due to the randomness of the errors caught, the FU_1 counter fills up before the FU_2 counter, in which case the fault would be erroneously attributed to FU_{1+}. In an effort to reduce this possibility, RIFT increases the width of the lower and upper error counters by one bit. Thus, for an end failover pair, an error condition will not be reported until twice, or $2 * (2^W - 1)$ errors are reported, which is much less likely to occur before the FU_2 counter saturates if FU_1 is the actual faulty unit. If the FU_2 counter saturates first, the FFSM will simply ignore it until a second unit reports an error (Step 15). Although this version of FFSM will be more reliable in the case of SEUs, it comes at the cost of more storage bits and counter logic which will grow proportionally with the number of FUs required.

Algorithm 4: Counting Failover FSM

```
Parameter: N, THRESHOLD
Input: $I_{ERROR}$, $I_{RESET}$
State: $V_{FCR}[N-1:0]$, $S_{STAGE}$
Output: $O_{FAIL}$
if ( $I_{RESET}$ ) then
    /*Reset All State Variables                                        */
    $V_{FCR} \leftarrow [0...01]_2$   /*The Failover Control Register (binary vector)  */
    $S_{STAGE} \leftarrow 0$          /*Distinguishes the $1^{st}$ and $2^{nd}$ stage of fault  */
    $O_{FAIL} \leftarrow 0$           /*Set in Failover mode: Disables testing  */
else if ( $O_{FAIL}$ ) then
    /*Failure mode: Determine if $V_{FCR}$ needs shifting down          */
    if ( $S_{STAGE} = 1$ ) then
        $V_{FCR} \leftarrow 0 \gg V_{FCR}$
        $S_{STAGE} \leftarrow 0$
else if ( $S_{STAGE} = 1$ ) then
    if ( $I_{ERROR} = 1$ ) then
        $O_{FAIL} \leftarrow 1$
        $V_{FCR} \leftarrow 0 \gg V_{FCR}$
        if ( $V_{FCR}[2] = 1$ ) then
            $S_{STAGE} \leftarrow 0$   /*The Redundant Unit is defective   */
    else $S_{STAGE} \leftarrow 0$   /*Error not verified: back to Stage 0   */
else
    /*Stage = 0                                                        */
    if ( $I_{ERROR} = 1$ ) then
        if ( $V_{FCR}[N-1] = 1$ ) then
            $S_{STAGE} \leftarrow 0$    /*This error detected on last FU    */
            $O_{FAIL} \leftarrow 1$     /*Shift back twice                  */
        else
            $S_{STAGE} \leftarrow 1$    /*Goto Stage 2                      */
            $V_{FCR} \leftarrow V_{FCR} \ll 1$
    else
        if ( $V_{FCR}[N-1] = 1$ ) then $V_{FCR} \leftarrow [0...01]_2$   /*Last FU: Restart */
        else $V_{FCR} \leftarrow V_{FCR} \ll 1$   /*Test next FU              */
```

4.2.4 The Circular FFSM

The most critical weakness of the previous two FFSMs is the method by which they extrapolate the existence of a recurring fault in the FUs on the either end of the buddy failover chain. Not only is the method subject to error, but the special cases in the FFSM control logic for dealing with end FUs complicates the FFSMs. It can be noted in Figure 4.5 that FU_{1+} lacks its own input and collector mux because it does not have its own logical identity. It would be possible to give FU_{1+} its own input mux and connect FU_N such that FU_{1+} can mirror FU_N's calculations as well as FU_1. Because FU_{1+} would not be an actual failover upper buddy to FU_N, only a comparator need be attached between them. By an extension of the FCR and other FFSM infrastructure vectors, the testing of FU_{1+} against FU_N would provide a second reference point for distinguishing a defect in either of the end FUs in an elegant manner. If the extra input mux was orchestrated at the beginning of the RIFT mapping process, it would also be subject to RIFT's normal interconnect optimization. Unfortunately, although this solution is technically feasible, due to the non-trivial development needed to implement it, this FFSM has not yet been incorporated into the RIFT CAD tool and thus comparative experimental results are not presented. Should this method be combined with the simple FFSM method, it will still be subject to SEU induced faults. However, a combination of the counting and circular FFSM methodologies would result in a fairly compact and robust FFSM architecture.

4.3 Experimental FT Results

The experimental procedure for FT testing is largely the same as that described in Section 3.8. The test cases used are all the same. Again, there is little or no previous work that explores FT HLS for realistically sized examples as is presented here. FT of a fashion similar to that implemented in RIFT has been added to List for the sake of comparison to an interconnect unaware approach. The CAD tool basically adds an extra redundant FU for each type and arbitrarily connects them together in an $N + 1$ buddy failover system similar to RIFT. Redundant wiring is added as needed.

Test Cases	Clocks	# Instances		FU Muxes				Reg Muxes			
		List	RIFT	List		RIFT		List		RIFT	
		$+,-,*,R$	$+,-,*,R$	#	>	#	>	#	>	#	>
Sym16_FIR	10	35,0,0,85	35,0,0,129	70	26	70	15	85	8	0	0
large_plus	13	35,0,0,85	35,0,0,129	70	26	70	15	85	8	0	0
large_minus	13	0,35,0,85	0,35,0,126	70	26	70	19	85	8	0	0
large_mult	26	0,0,35,85	0,0,36,130	70	26	72	15	85	8	0	0
large_plus_minus	13	19,19,0,86	18,20,0,134	76	25	76	19	82	8	0	0
large_mixed	31	8,7,7,50	8,9,8,72	44	40	50	26	43	11	0	0
large_long	58	6,5,8,25	7,7,8,52	36	26	44	25	24	12	0	0
large_wide	9	23,18,21,163	23,18,21,213	124	18	124	16	134	4	0	0
huge_plus	20	50,0,0,161	54,0,0,214	100	37	108	20	155	10	0	0
huge_minus	20	0,50,0,161	0,50,0,226	100	38	100	26	155	10	0	0
huge_mult	32	0,0,63,155	0,0,63,228	126	31	126	22	142	10	0	0
huge_plus_minus	20	26,26,0,164	27,26,0,239	104	36	106	28	158	11	0	0
huge_mixed	56	9,9,16,74	11,11,17,117	68	59	78	44	72	16	0	0
huge_long	79	8,7,12,49	9,10,13,95	54	48	64	40	48	16	0	0
huge_wide	16	22,22,40,230	22,22,40,334	168	32	168	24	221	7	0	0

Table 4.2: List vs. RIFT: Clock steps and required components

It would be possible to improve the List FT further by pairing FUs that have the most similar input requirements together, however, this has not been done. It could also be argued that List FT is an overly fair representation of other interconnect unaware systems because the $N + 1$ topology is inherently more scalable than those that randomly reschedule redundant LOs. As such, the following results should be interpreted in the context of the relative effects of managing or ignoring interconnects for realistically sized HLS FT designs.

The first set of results, Table 4.2, lists the attributes of both the List and RIFT FT solutions. Table 4.3 lists, for convenience, the same information in terms of the difference between the List FT and RIFT FT structures. It can be seen that, once again, RIFT tends to use more FUs than List, and perhaps does so a little more erratically than RIFT without FT. RIFT uses between 30 and 50% more registers. Because of the extra units, RIFT tends to have several more FU muxes, however, the maximum size of these FU muxes is significantly smaller than for List. The sheer size of List's FU muxes is evidence of the significance of ignoring routing in HLS.

One area that RIFT is less competitive in is the number of FUs that it instantiates. RIFT guarantees that it can make the same latency as List, however, the way in which

Test Case	FUs Instantiated (+,-,*)	# Registers	FU muxes #	FU muxes >	Reg muxes #	Reg muxes >
Sym16_FIR	0,0,0	6	-1	-2	-6	-3
large_plus	0,0,0	44		-11	-85	-7
large_minus	0,0,0	41		-7	-85	-7
large_mult	0,0,1	45	2	-11	-85	-7
large_plus_minus	-1,1,0	48		-6	-82	-7
large_mixed	0,2,1	22	6	-14	-43	-10
large_long	1,2,0	27	8	-1	-24	-11
large_wide	0,0,0	50		-2	-134	-3
huge_plus	4,0,0	53	8	-17	-155	-9
huge_minus	0,0,0	65		-12	-155	-9
huge_mult	0,0,0	73		-9	-142	-9
huge_plus_minus	1,0,0	75	2	-8	-158	-10
huge_mixed	2,2,1	43	10	-15	-72	-15
huge_long	1,3,1	46	10	-8	-48	-15
huge_wide	0,0,0	104		-8	-221	-6

Table 4.3: The difference in component requirements (RIFT - List)

LOs are mapped using connection costs often results in a different amount of FUs being instantiated. For instance, large_plus_minus uses 19 adders and 19 subtraction units, whereas the RIFT version uses 18 and 20. In half the cases, however, the RIFT approach requires more than one extra FU instance per class to add FT, while List will always require only one extra FU. This is a consequence of RIFT's incorporation of the extra redundant FU per class into the decision tree from the beginning of the mapping process. The costs associated with connecting both the standard and the redundant unit are considered from the beginning, and thus can lead to a much different design than the non-FT design which does not have the extra redundancy costs to consider.

Gate level results are presented in Table 4.4 and also in Figures 4.6 and 4.7. The results overall are fairly favourable for RIFT with improvements over List scheduling of on average 21.6% and 34.4% for area and frequency improvements, respectively. The trend that the larger examples experience a larger gain tends to hold with FT as well. The most startling gains are made by *_mult. Though huge_mult does not improve area over large_mult, performance improvements are much greater at 76.4%, perhaps possibly due to its use of multiple clock cycles. RIFT gives its worst

	Area(FPGA LEs)			Performance (MHz)		
Test Cases	List	RIFT	%Less	List	RIFT	%Gain
Sym16_FIR	3,897	3,789	2.8	24.60	25.25	2.6
large_plus	12,161	8,352	31.3	12.97	14.54	12.1
large_minus	15,696	12,101	22.9	27.27	37.75	38.4
large_mult	11,728	6,581	43.9	58.23	69.35	19.1
large_plus_minus	12,389	9,692	21.8	12.82	14.96	16.7
large_hetero	10,987	9,150	16.7	11.23	16.58	47.6
large_long	8,158	8,439	-3.4	16.63	18.16	9.2
large_wide	14,417	12,346	14.4	13.72	15.26	11.2
huge_plus	29,683	18,644	37.2	8.54	12.14	42.2
huge_minus	30,949	19,660	36.5	8.11	12.03	48.3
huge_mult	34,242	24,411	28.7	17.61	31.06	76.4
huge_plus_minus	29,586	20,667	30.1	7.4	11.73	58.5
huge_hetero	26,975	22,191	17.7	6.82	9.75	43.0
huge_long	20,505	19,501	4.9	8.14	10.99	35.0
huge_wide	30,701	25,512	16.9	10.7	13.22	23.6
Average (of large_*, huge_*)			21.6			34.4

Table 4.4: List vs. RIFT: Area and performance

improvements in the *_long cases. This is due to a large amount of operations being run on a smaller amount of FUs, which tends to reduce the amount FU muxes can be reduced by. Consider large_long's maximum FU mux size to register ratio of 25:26 for List and 52:25 for RIFT. This suggests that for List, at least the largest mux has inputs from almost every register plus some inputs. RIFT uses twice as many registers and the largest mux uses only half of those, but this still means 25 inputs for the largest FU mux. Clearly, RIFT's decreasing returns for cases of high internal dependency are showing for this test case. The 9% timing improvement likely is due to the gate level critical path reduction of four in register muxes.

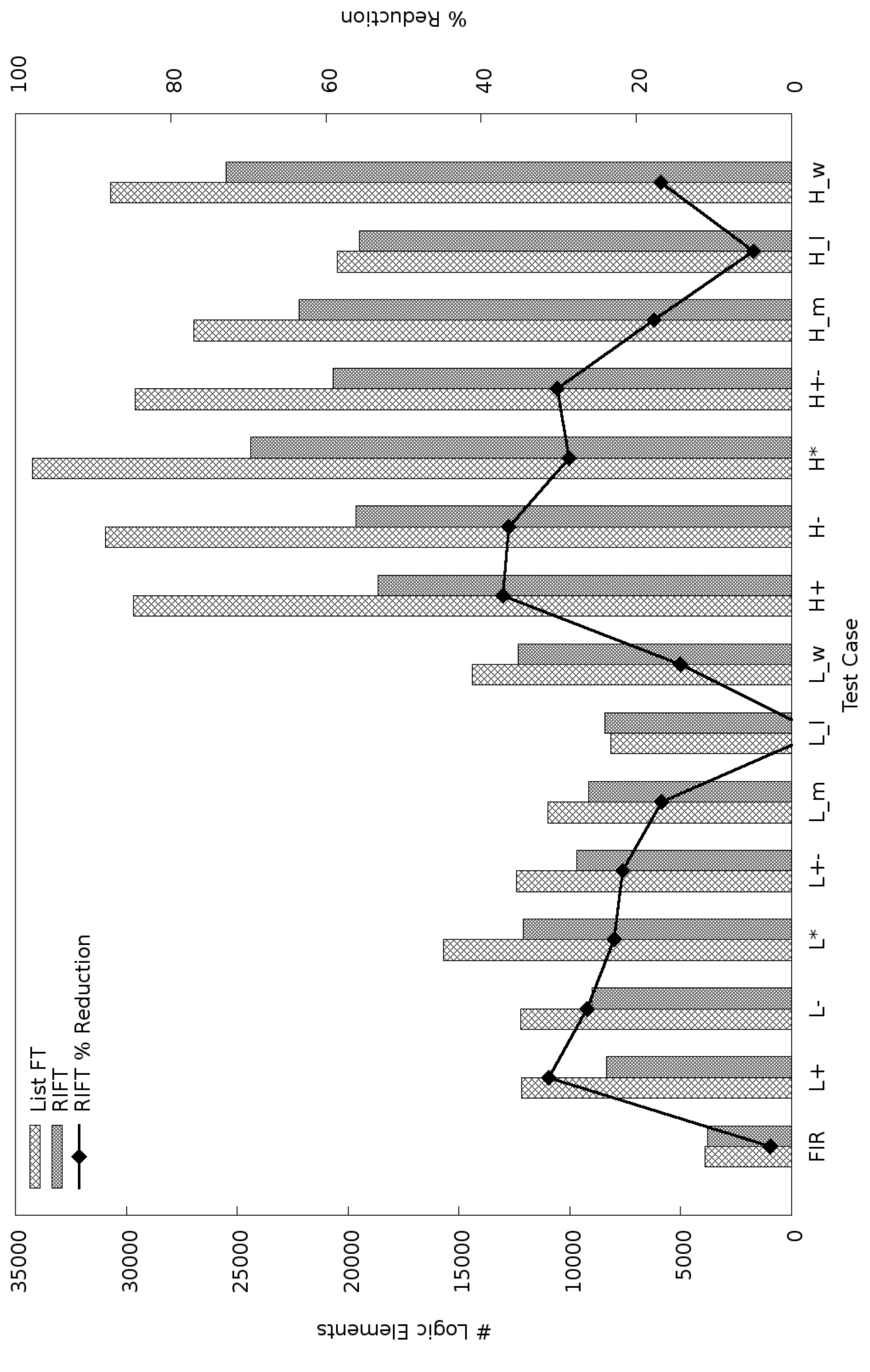


Figure 4.6: Area comparison of List and RIFT (See Table 4.4)

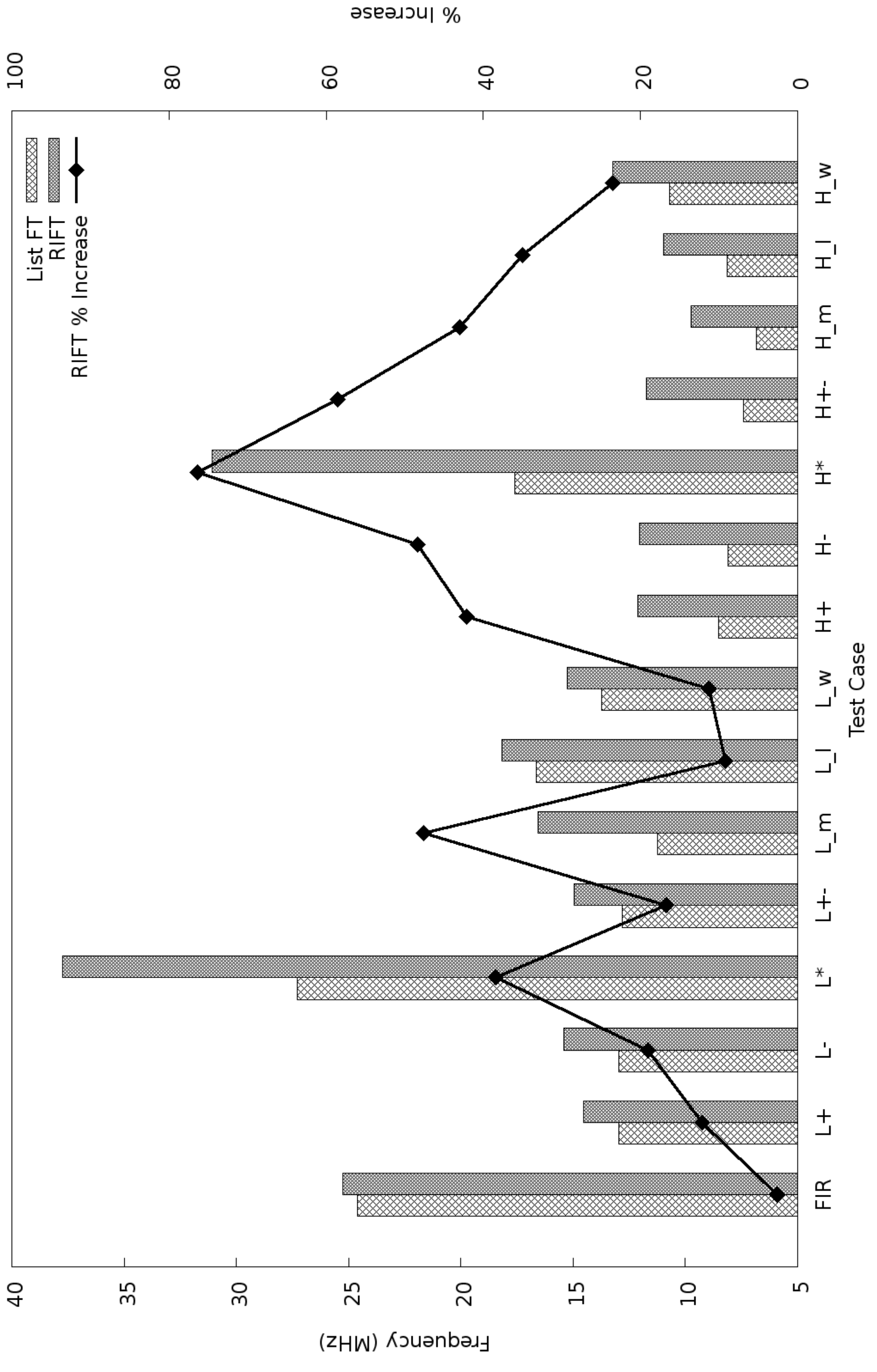


Figure 4.7: Performance comparison of List and RIFT (See Table 4.4)

				FU muxes						Reg muxes					
	# Registers			#			>			#			>		
Test Case	B	∞	2	B	∞	2	B	∞	2	B	∞	2	B	∞	2
Sym16_FIR	29	-5	-5	17	1	1	9			0	6	7	1	2	1
large_plus	129	-37	-29	70		2	15		-1	0	91	89	1	3	1
large_minus	126	-37	-25	70	2	4	19	-3	-4	0	85	95	1	4	1
large_mult	130	-39	-28	72	4	-2	15	-1	-1	0	91	93	1	3	1
large_plus_minus	134	-39	-25	76		4	19	-3	-2	0	93	101	1	4	1
large_mixed	72	-26	-17	50	2	-2	26	-3	-2	0	46	50	1	4	1
large_long	52	- 27	-15	44		-6	25	-5		0	25	32	1	6	1
large_wide	213	-43	-34	124			16	-2	-2	0	131	121	1	3	1
huge_plus	214	-58	-38	108	-8	-8	20	1		0	151	166	1	4	1
huge_minus	226	-73	-45	100		2	26	-3	-3	0	152	170	1	4	1
huge_mult	228	-82	-41	126	4	2	22	-5	-4	0	146	178	1	5	1
huge_plus_minus	239	-71	-53	106		-2	28	-2	-3	0	164	176	1	4	1
huge_mixed	117	-45	-23	78		-4	44	-2	2	0	69	90	1	6	1
huge_long	95	-49	-28	64	4	-2	40	-12		0	46	66	1	7	1
huge_wide	334	-98	-67	168	-14	-14	24	-1	-1	0	227	239	1	4	1

Table 4.5: Effects of register sharing on the logical structure

The effects of allowing unlimited register sharing and register sharing with a limit of two inputs is presented in Tables 4.5 and 4.6. As in Table 3.6 and 3.7, the results are recorded as the change against the baseline test case, which uses register partitioning. Changes in the number of FU muxes is mainly due to a change in the amount of FUs instantiated. The maximum size of the FUs are reduced by several inputs, often by more than the maximum register size is increased. Yet these changes mostly result in both area increases and performance decreases. The exception seems to be huge_long again. Because for this case the largest mux has inputs from about half of the registers, a nearly 50% reduction in registers seems to have a highly leveraged affect on performance.

The overall negative effects of trying to "balance" routing complexity between the FU and register routing levels, though perhaps not intuitive, should be expected. Indeed, the purpose of using wiring cost coefficients was originally intended to allow complexity to be moved between the two levels. The reasons are best given algebraically, but first the context must be explained. When registers are partitioned, there are no muxes at the register level (excluding control statements). As the limit

	Area(FPGA LEs)			Performance (MHz)		
Test Cases	Base	∞(%)	2(%)	Base	∞(%)	2(%)
As_16_FIR	3,789	3.7	0.2	25.25	-2.3	-8.0
large_plus	8,352	9.8	14.2	14.54	3.23	-0.9
large_minus	8,982	7.9	12.7	15.43	-10.95	-6.0
large_mult	12,101	6.8	6.8	37.75	-22.0	-7.5
large_plus_minus	9,692	3.7	10.4	14.96	-8.76	-5.9
large_mixed	9,150	4.0	4.7	16.58	-5.97	-3.4
large_long	8,439	-5.7	-3.1	18.16	-2.2	-0.8
large_wide	12,346	6.2	15.0	15.26	-9.63	-1.0
huge_plus	18,644	4.2	7.0	12.14	-9.64	1.6
huge_minus	19,660	6.3	11.3	12.03	-47.71	-0.6
large_mult	24,411	8.8	10.5	31.06	-17.3	-20.8
huge_plus_minus	20,667	6.8	11.1	11.73	-22.59	6.9
huge_mixed	22,191	-2.5	1.7	9.75	-3.38	1.9
huge_long	19,501	-9.6	-2.6	10.99	27.12	-2.9
huge_wide	25,512	6.3	10.6	13.22	-40.32	-5.9
Average (of large_*, huge_*)		3.8	7.9		-12.1	-3.2

Table 4.6: Effects of sharing registers on area and performance

on the number of register inputs is raised, muxes will be inserted in front of the registers and they will grow in size. The register muxes grow in size because the registers are now having values packed to them more aggressively and from more sources. As such, the number of registers needed will be reduced, as evidenced by Table 4.5. This should mean there are less distinct sources and thus the FU muxes should in general become smaller. Although this is not a direct causal relationship, this can be thought of as a shifting of complexity from the FU level to the register level. To understand the ramifications of this "relationship" the following model is used. The model is a function of N, which is the number of inputs allowed to each register. Because it is being proposed that the interconnect " complexity" is inversely proportional between the register and FU levels, the model will define FU input size to be $X - N$, where X is a threshold representing maximum "complexity", or number of inputs. Consider, using two-input muxes, the relationships between the number of inputs, N, a mux has and its size and critical path delay.

$$MUX_{Area}(N) \propto N - 1$$
$$MUX_{Delay}(N) \propto \lceil log_2(N) \rceil$$

The total area of this hypothetical system can now be constructed by adding $MUX_{Area}(N)$ and $MUX_{Area}(X - N)$ and likewise for delay. The resulting relationships follow:

$$TOTAL_{Area}(N) \propto (N - 1) + (X - (N - 1)) = X \tag{4.2}$$
$$TOTAL_{Delay}(N) \propto \lceil log_2(N) \rceil + \lceil log_2(X - N) \rceil \tag{4.3}$$

The resulting relationships are plotted in Figures 4.8 and 4.9, where X has been set to 64. The value N is the bottom x-axis, and for convenience, the inverse amount of FU inputs are numbered on the upper x-axis. Area, as is obvious from Equation 4.2, should be fairly constant, which is reflected by the fairly consistent area loss due to register sharing in Table 4.6. Despite the idea that area should remain constant, the fact is that for most of the test cases there are more register muxes to be added then there are FU muxes to remove. This would justify the overly negative effect on area as complexity is moved to the register level. The delay curves in Figure 4.9 show how delay due to the mux size increases in a step-wise fashion with the $\lceil log_2 \rceil$ of the number of inputs, or "complexity". The addition of the register and FU mux delay results in a stepwise negative parabola. Thus delay is maximized, when the measure of "complexity", N, is evenly spread between the two levels. This suggests that moving some inputs from a large mux to a small or nonexistent mux is, unfortunately, a very effective way to introduce extra levels of gate delay to the system. The optimal tradeoff is actually to put all interconnect complexity at a single level. This would seem to be the reason register partitioning is so effective and why, as concluded in the last section, register sharing is detrimental for large HLS systems.

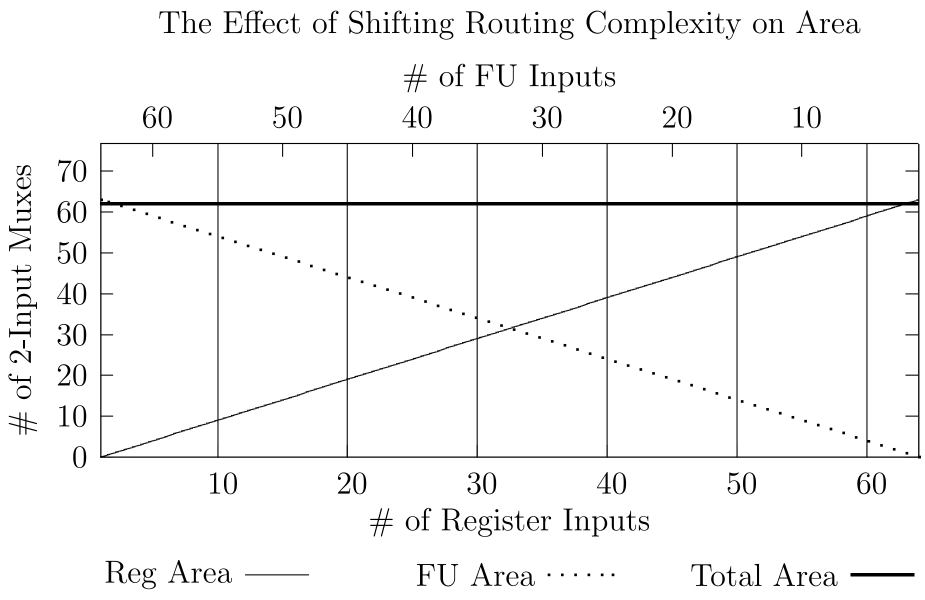


Figure 4.8: Area as complexity is moved between FU & register muxes

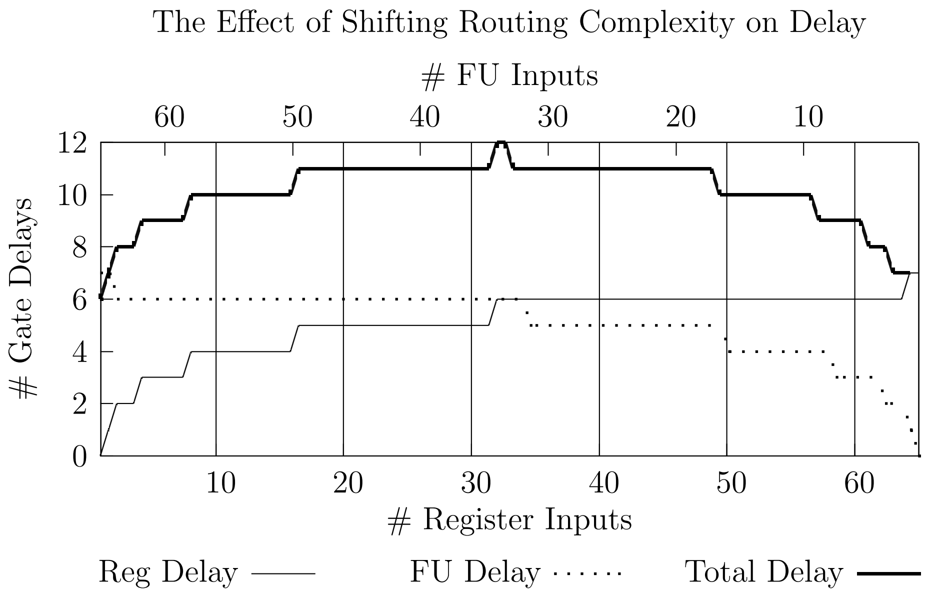


Figure 4.9: Delay as complexity is moved between FU & register muxes

	# Components Added				FU muxes		Reg Muxes	
Test Case	+	-	*	Reg	#	>	#	>
Sym16_FIR	1		1	6	5	2	-6	-3
large_plus	1			44	2	2	-85	-7
large_minus		1		41	2	6	-85	-7
large_mult			2	45	4	2	-85	-7
large_plus_minus		2		48	4	6	-82	-7
large_mixed	1	3	2	22	12	1	-43	-10
large_long	2	3	1	27	14	2	-24	-11
large_wide	1	1	1	50	6	7	-134	-3
huge_plus	5			53	10		-155	-9
huge_minus		1		65	2	6	-155	-9
large_mult			1	73	2	6	-142	-9
huge_plus_minus	2	1		75	6	8	-158	-10
huge_mixed	3	3	2	43	16		-72	-15
huge_long	2	4	2	46	16	-1	-48	-15
huge_wide	1	1	1	104	6	8	-221	-6

Table 4.7: Structure of RIFT FT vs. List without FT

	Area (FPGA LEs)				
Test Cases	List	List FT	Cost(%)	RIFT FT	Cost(%)
Sym16_FIR	2,550	3,897	52.8	3,789	48.6
large_plus	6,594	12,161	84.4	8,352	26.7
large_minus	6,865	12,202	77.7	8,982	30.8
large_mult	9,178	15,696	71.0	12,101	31.8
large_plus_minus	6,662	12,389	86.0	9,692	45.5
large_mixed	5,744	10,987	91.3	9,150	59.3
large_long	4,868	8,158	67.6	8,439	73.4
large_wide	8,094	14,417	78.1	12,346	52.5
huge_plus	13,925	29,683	113.2	18,644	33.9
huge_minus	14,381	30,949	115.2	19,660	36.7
large_mult	18,920	34,242	81.0	24,411	29.0
huge_plus_minus	14,102	29,586	109.8	20,667	46.6
huge_mixed	14,625	26,975	84.4	22,191	51.7
huge_long	11,963	20,505	71.4	19,501	63.0
huge_wide	17,200	30,701	78.5	25,512	48.3
Average (of large_*, huge_*)			86.4		44.9

Table 4.8: Area costs of adding List and RIFT based FT

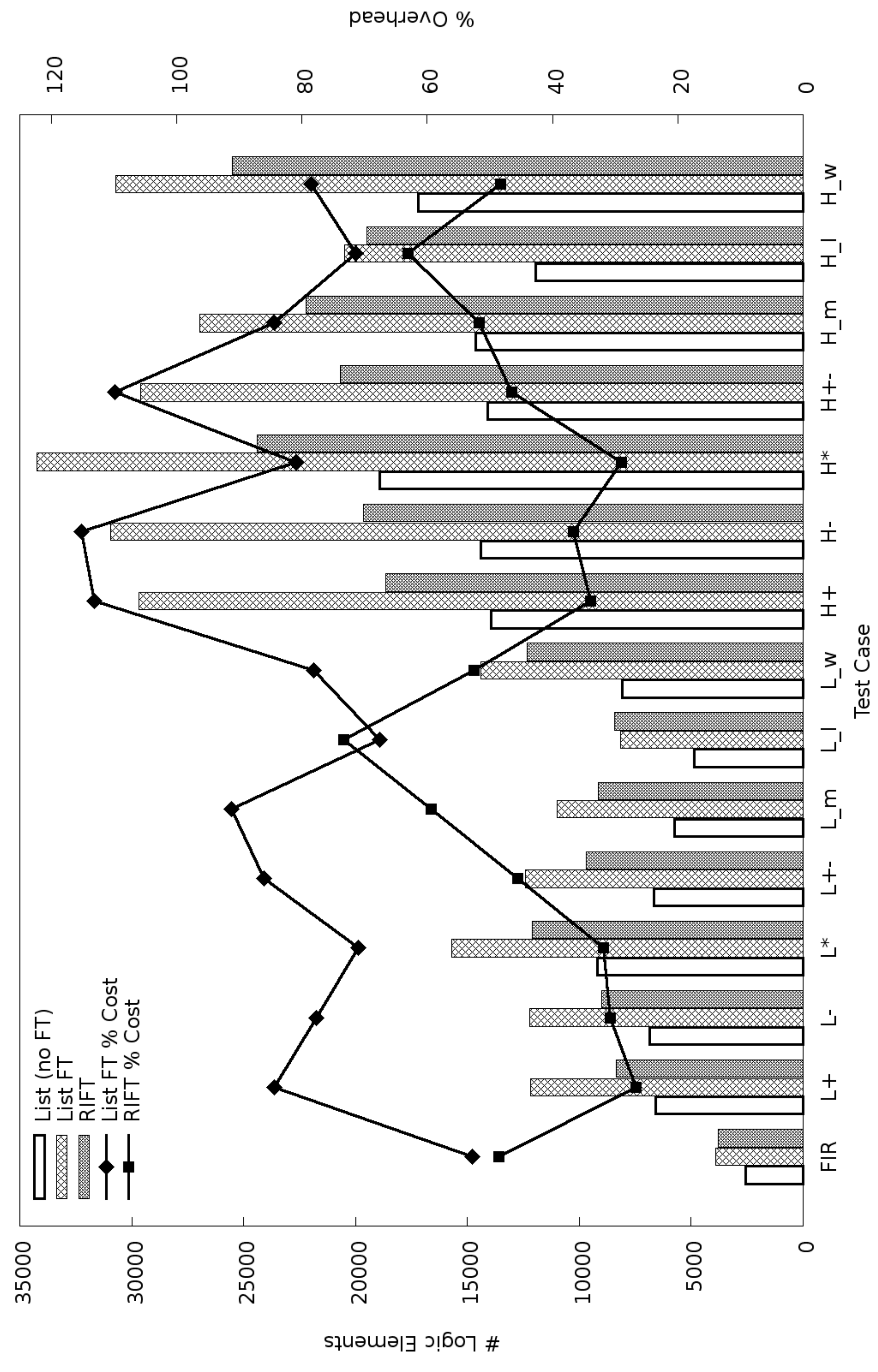


Figure 4.10: Area cost of adding List and RIFT based FT (See Table 4.8)

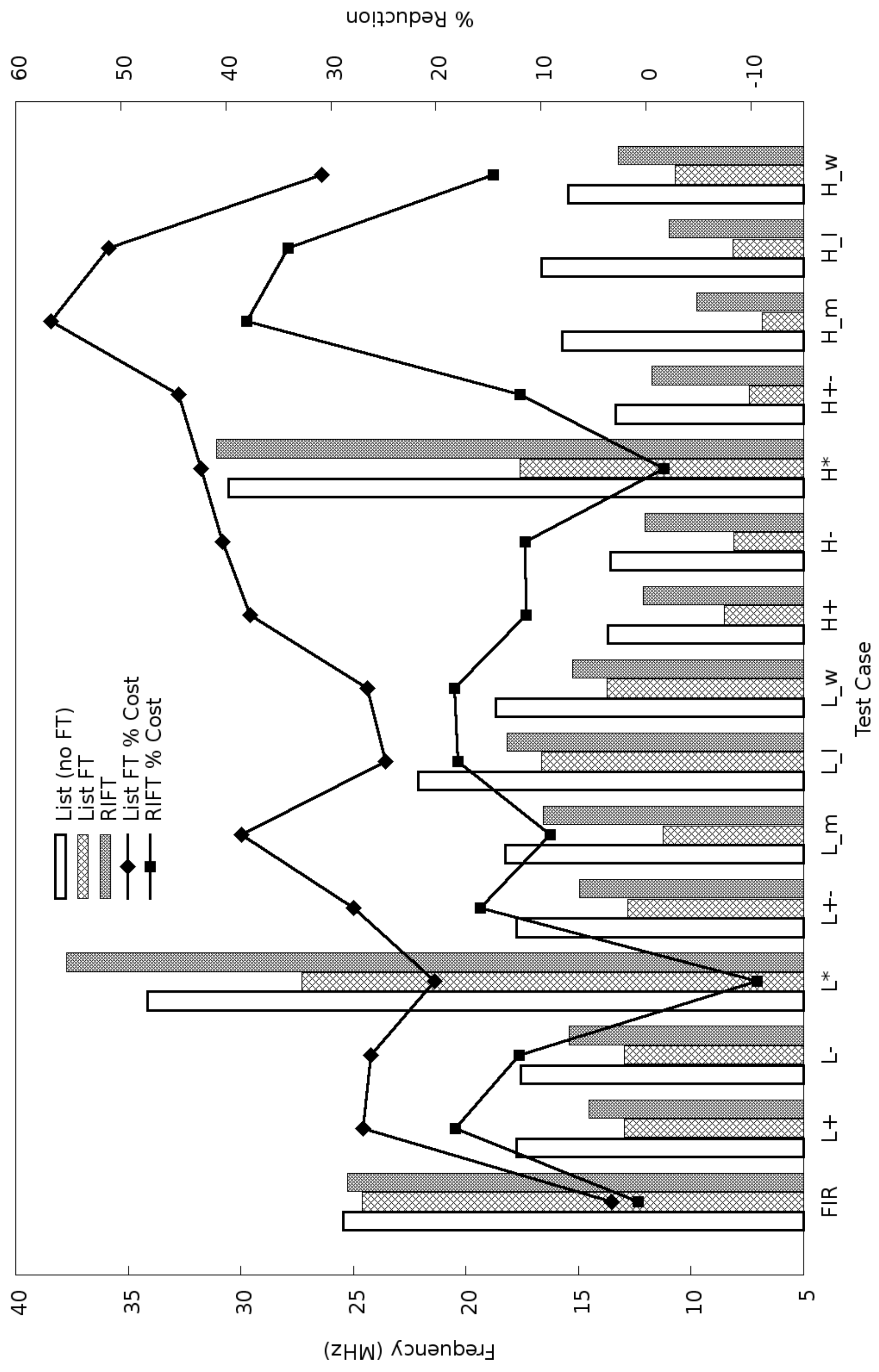


Figure 4.11: Performance cost of adding List and RIFT based FT (See Table 4.9)

	Performance (MHz)				
Test Cases	List	List FT	Cost(%)	RIFT FT	Cost(%)
Sym16_FIR	25.44	24.60	3.3	25.25	0.7
large_plus	17.76	12.97	27.0	14.54	18.1
large_minus	17.55	12.96	26.2	15.43	12.1
large_mult	34.13	27.27	20.1	37.75	-10.6
large_plus_minus	17.76	12.82	27.8	14.96	15.8
large_mixed	18.25	11.23	38.5	16.58	9.2
large_long	22.12	16.63	24.8	18.16	17.9
large_wide	18.67	13.72	26.5	15.26	18.3
huge_plus	13.71	8.54	37.7	12.14	11.5
huge_minus	13.59	8.11	40.3	12.03	11.5
large_mult	30.54	17.61	42.3	31.06	-1.7
huge_plus_minus	13.33	7.40	44.5	11.73	12.0
huge_mixed	15.72	6.82	56.6	9.75	38.0
huge_long	16.66	8.14	51.1	10.99	34.0
huge_wide	15.47	10.70	30.8	13.22	14.5
Average (of large_*, huge_*)			35.3		14.3

Table 4.9: Performance costs of adding List and RIFT based FT

In addition to determining the relative costs of RIFT versus List based FT, one of the goals of this research was to examine the actual costs of adding FT to large designs. Table 4.7, compares the number of logical units RIFT requires to the base case of List without FT insertion. Tables 4.8 and 4.9 give the area and performance costs of both List and RIFT based FT compared to a baseline case of List without FT. Figures 4.10 and 4.11 give the same data in chart form. . Logically RIFT requires marginally more FU instances. Register usage is increased by 30 to 50%, which is what allows complexity to be offset to the FU level of muxes. This offset is reflected in a small increase in the maximum size of FU muxes, while register mux maximum size is reduced by the same amount or more. This would translate into, at most, an introduction of a one gate delay to the FU muxes while the register level is reduced by up to four gate delays.

In terms of area costs, List based FT is shown to cost an average of 85%, and as much as 115% in one case. Compared to this RIFT does the same job at an average

	Area(FPGA LEs)			Performance (MHz)		
Test Cases	RIft	RIFT	%Cost	RIft	RIFT	%Cost
Sym16_FIR	2,537	3,789	49.3	33.11	25.25	-23.7
large_plus	3,942	8,352	111.9	25.36	14.54	-42.7
large_minus	4,203	8,982	113.7	23.50	15.43	-34.3
large_mult	6,593	12,101	83.5	47.72	37.75	-20.9
large_plus_minus	4,487	9,692	116.0	22.74	14.96	-34.2
large_hetero	4,730	9,150	93.4	24.46	16.58	-32.2
large_long	4,341	8,439	94.4	28.25	18.16	-35.7
large_wide	6,738	12,346	83.2	25.58	15.26	-40.3
huge_plus	8,200	18,644	127.4	22.04	12.14	-44.9
huge_minus	9,013	19,660	118.1	19.97	12.03	-39.8
huge_mult	12,910	24,411	89.1	40.13	31.06	-22.6
huge_plus_minus	9,542	20,667	116.6	20.23	11.73	-42.0
huge_hetero	10,232	22,191	116.9	22.46	9.75	-56.6
huge_long	9,301	19,501	109.7	23.64	10.99	-53.5
huge_wide	13,239	25,512	92.7	20.94	13.22	-36.9
Average (of large_*, huge_*)			104.8			-38.3

Table 4.10: Cost of the FT in RIFT

cost of 45% and up to a maximum of 63% of area. To rephrase this result, RIFT can be used to add FT to a List designed non-FT circuit for 52% of the cost that a List based FT approach would use, on average. The performance cost is a 35% and 14.3% average reduction in operating frequency for List based FT and RIFT based FT designs, respectively. Thus the cost of using RIFT FT is 41% of the cost of using List for FT insertion. It is interesting to note, that for the *_mult cases, RIFT with FT actually outperforms List without it. These results suggest that directly managing routing considerations in HLS design can effectively reduce the cost of adding FT by half.

4.4 FT Analysis

With the discussion on implementation of FT now complete, all the necessary elements required for an efficient, fully functional FT system have been presented. A critique is presented of the overall effectiveness of the system in suppressing faults.

Most prior work is carried out under the "single fault" disclaimer, which states that only one fault at a time can be accommodated. This not only alleviates resources needed for redundancy, it also eliminates the possibility that two faults in concert might foil diagnosis. RIFT has a similar warranty, though with some caveats. For any grouping of FUs chained together in a "buddy-buddy" failover chain, one fault in that set of FUs can be accommodated. Thus, in the general case, RIFT may actually be able to deal with as many faults as there are FU classes. However, this assumes that the faults cooperate and evenly distribute themselves across the FU classes, a decidedly unlikely proposition. It would be possible to extend the warranty to X number of faults by adding X redundant units and arranging the extra failover wiring the same way RIFT currently does for X=1. Based on the results in the previous section, RIFT would seem the best way to do this. However, the extra routing requirements would be extremely cost prohibitive. Although RIFT has shown itself to be a far superior method of adding FT than an interconnect unaware approach like List, the actual cost of adding FT to a non-FT RIFT design is still quite high at an average cost of 105% and 38% for area and performance, respectively. Consequently, a stong economic argument could be made that complete system duplication or triplication may be more efficient than increasing X past 1. Another way to increase robustness would be to segment failover chains so that each consists of only a fraction of the number of FUs belonging to that class. This would not increase the guaranteed fault coverage, but it would make it more likely that a subsequent fault could also be handled. There are two types of costs associated with using smaller chains. The first is, as an $N+1$ architecture, the extra redundant unit for each chain. The second comes in the form, again, of increased routing requirements because of reduced partitioning possible within each subchain, which will require input muxes to have larger sets of inputs. Neither of these methods for increasing the number of tolerable faults was compelling enough to incorporate into RIFT.

Another caveat of RIFT, which should not be an issue for large systems, is that the approach requires at least two FUs plus an extra redundant FU per class to properly isolate and compensate for faults. RIFT thus automatically instantiates at least three FUs when directed to add FT to a design. This requirement would obviously

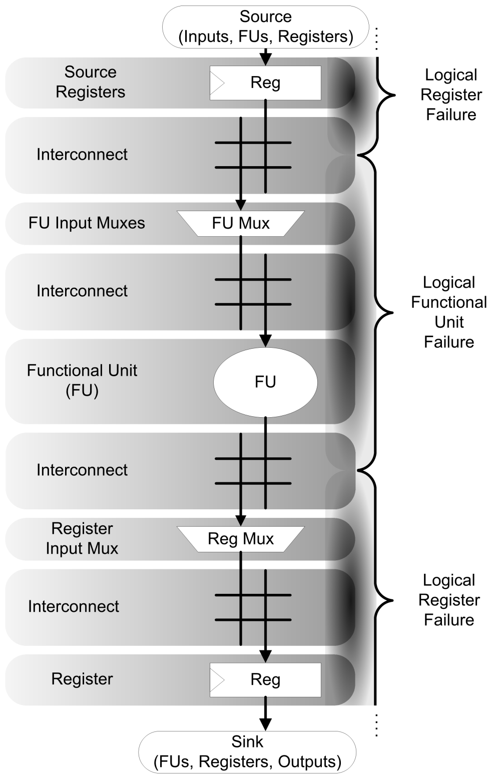


Figure 4.12: Logical fault coverage

put RIFT at a competitive disadvantage for trivial or highly serial benchmarks. For the large designs this research is concerned with, however, this is not an issue.

Figure 4.12 is a representation of a generic data path. The suggestion is that any given fault will appear to damage either a logical FU or a register entity. Each entity consists of itself and its input mux(es). Also included is the input wiring up to the divergence points where that wiring feeds more than just that entity and also output wiring down to the divergence point. Unfortunately, the transition point between an FU and a register is poorly defined at the gate level, especially in the case of FPGAs, which use statically configured switches for routing. Though it is seldom mentioned, there can be points between the output divergence point of one entity and the input divergence point of the next. For instance, from an FU to multiple registers, a register feeder wire might split off from the main at different points. A fault between entities might then affect some but not all receiving registers. In this case, it would appear as though several of the registers were faulty as a result of a single real fault. The same would be true for the case of a single register feeding multiple FUs. RIFT will not be able to properly recover from this class of fault, even though it does concern the datapath.

There are several other faults that RIFT cannot correct. Under the single fault assumption, some faults in the RIFT infrastructure itself are covered. Failures in the redundant inputs, comparators, or collector muxes will all be observed to be an error in the logical FU and thus garner the same result. An unfortunate result, if faults incurred are proportional to area usage, is the RIFT infrastructure will actually increase the likelihood of experiencing a fault. Further research into by how much the benefits of FT are reduced by this factor is a generic FT issue and outside the current scope. Nonetheless, it does exist.

Finally, there are components of the final circuit that are completely unprotected by RIFT. These include the voting circuitry, the actual FSM, RIFT's own FFSMs, and the actual inputs to and from the design. Almost all research in the field of HLS FT assumes these components to be immune to faults or to have its own built in FT system, which is rarely implemented in experimental setups. An example of a controller with FT built in is [21]. As such, though outside the scope of this research, an actual FT design should undergo a formal characterization of reliability. This topic is thoroughly addressed by Lala in [38], among others, and includes the effects of voter reliability on the overall reliability of $N + M$ redundant systems, where M is the number of redundant units.

RIFT currently only has the capability to design for faults in a logical FU. A next logical step for future work would be to incorporate FT for registers in a way that embodies the ideals of simplicity and scalability that are the basis of RIFT. From the results already presented, it is very critical that a solution's highest priority should be to minimize the amount of additional routing needed. A promising approach may be to add an extra register per FU grouping for an $N+1$ arrangement and then somehow pipeline the verification process to avoid impacting the design's delay. Regardless, it will be a difficult challenge to solve well.

Chapter 5

Conclusion

By building and demonstrating RIFT, three contributions are made. The first is to give credence to the possibility that, though proposed HLS approaches to FT are comprehensive, feasible, and conceptually interesting, they often ignore the ramifications of implementation in realistic designs of substantial size. It is generally accepted that adding FT to a design is costly, especially as the size of the design under consideration escalates. The fact that the body of work considering the area and delay costs of FT, particularly with respect to interconnects, is relatively small, suggests this is a difficult problem to treat well. It may be that slow rate of commercial acceptance for HLS in general and HLS FT in particular has meant that issues arising with large designs have not yet become prominent. Or perhaps the converse is true and HLS FT has not gained popularity commercially because HLS FT is not yet able to address usefully sized designs in an economically justifiable way. Results presented here for a small common benchmark indicate an area cost of approximately 50% as well a performance reduction of about 3% when List's particular mix of FT is added. However, when sample designs are enlarged to nearly a thousand operations, these costs jump to as much as 100% and 50% respectively. Thus this research suggests that scalability, especially with consideration to FT, is an important but somewhat dormant issue. The first contribution is not to solve this issue, but merely to bring it to light.

The second contribution is an attempt to demonstrate how the cost of realistically sized HLS FT systems can be reduced, particularly as pertains to the interconnect issue. By managing interconnect issues from the beginning, even at the cost of possibly suboptimal mapping, RIFT manages to reduce, in some instances, area and performance penalties by a factor of three. This is fairly substantial improvement. However, once HLS scalability is better understood, it should possible to improve upon these results and introduce interconnect oriented methods for other configurations as well.

The final contribution is made partly in response to recent interest in bringing FT features from the exclusive domain of critical spacecraft, military, and medical systems to the more mundane IC applications of everyday life [6]. The vision behind the criteria listed in the beginning of Chapter 4 is for a unique and relatively inexpensive FT system that places greater emphasis on robustness instead of the data integrity assurances required of critical systems. Though unsuitable for critical applications, this FT model could be used to enhance the reliability of the vast majority of computing infrastructure that encompasses and facilitates modern society. Because of its overriding emphasis on interconnect reduction, RIFT could be regarded as one of the first FPGA " specific" approaches to FT that allows fully transparent compensation. For large production runs of FPGAs using the same design, RIFT could be used to improve manufacturing yield by circumventing faults without the need for design recompilation or even FPGA reconfiguration. However, the larger potential, considering how FPGAs are increasingly dominating "common" applications, may be the use of HLS FT methods to extend these product's lifespans. FT may be costly, but if there is already extra space reserved on the FPGA, perhaps for "future proofing", it may be possible to add FT essentially for free. In any regard, as IC fabrication capabilities move to 0.65 microns and beyond, architectural FT may well become an essential tool for long term reliability.

Bibliography

[1] *Stratix II Device Handbook, Volume 2.*

[2] M. Abramovici, C. Stroud, and J. Emmert. Online BIST and BIST-based diagnosis of FPGA logic blocks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(12):1284–1294, December 2004.

[3] AMD. $www.amd.com/us-en/0,,3715\_11787,00.html$. 2004.

[4] A. Antola, V. Piuri, and M. Sami. On-line diagnosis and reconfiguration of FPGA systems. In *Electronic Design, Test and Applications, IEEE International Symposium on*, pages 291–296, 2002.

[5] J. Armstrong. Chip-level modeling with HDLs. *Design & Test of Computers, IEEE*, 5(1):8–18, Febuary 1988.

[6] A. Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):227–237, June 1997.

[7] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, January 2002.

[8] S. Bhattacharya, S. Dey, and F. Brglez. Effects of resource sharing on circuit delay: An assignment algorithm for clock period optimization. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):285–307, April 1998.

[9] D. Blough, F. Kurdahi, and S. Y. Ohm. High-level synthesis of recoverable VLSI microarchitectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(4):401–410, December 1999.

[10] R. Camposano. From behavior to structure: High-level synthesis. *IEEE Design & Test of Computers*, 7(5):8–19, 1990.

[11] W. Chan and A. Orailoğlu. High-level synthesis of gracefully degradable ASICs. In *European Design and Test Conference, Proceedings*, pages 50–54, 1996.

[12] R. J. Cloutier and D. E. Thomas. The combination of scheduling, allocation, and mapping in a single algorithm. In *IEEE Design Automation Conferance*, pages 71–76, 1990.

[13] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[14] J. A. Fisher. CHECK trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computer*, C-32(7):478–490, July 1981.

[15] D. Frank, R. Dennard, E. Nowak, P. Solomon, Y. Taur, and H.-S. P. Wong. Device scaling limits of Si MOSFETs and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[16] L. Guerra, M. Potkonjak, and J. Rabaey. Behavioral-level synthesis of heterogeneous BISR reconfigurable ASIC's. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions On*, 6(1):158–167, March 1998.

[17] R. Gupta, N. J. Claudionor, and G. D. Micheli. Program implementation schemes for hardware-software systems. *Computer*, 27(1):48–55, January 1994.

[18] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):441–470, 2004.

[19] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2):302–312, Febuary 2004.

[20] L. J. Hafer and A. C. Parker. Automated synthesis of digital hardware. *IEEE Trans. Computers*, 31(2):93–109, 1982.

[21] S. Hamilton, A. Hertwig, and A. Orailoğlu. Self recovering controller and datapath codesign. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 596–601, 1999.

[22] S. Hamilton and A. Orailoğlu. On-line test for fault-secure fault identification. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(4):446–452, August 2000.

[23] S. N. Hamilton and A. Orailoğlu. Behavioral synthesis for easy testability in data path scheduling. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 252–260, 1998.

[24] F. Hanchek and S. Dutt. Methodologies for tolerating cell and interconnect faults in FPGAs. *Computers, IEEE Transactions on*, 47(1):15–32, January 1998.

[25] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(4):464–475, April 1991.

[26] INTEL. *intel.com/pressroom/archive/releases/*20040907*corp_a.htm*. 2004.

[27] J. A. G. Jess. Designing electronic engines with electronic engines: 40 years of bootstrapping of a technology upon itself. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19(12):1404–1427, December 2000.

[28] R. Karri, K. Hogstedt, and A. Orailoğlu. Computer-aided design of fault-tolerant VLSI systems. *Design and Test of Computers, IEEE*, 13(3):404–412, Fall 1996.

[29] R. Karri, B. Iyer, and I. Koren. Phantom redundancy: A register transfer level technique for gracefully degradable data path synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(8):877 – 888, August 2002.

[30] R. Karri, K. Kim, and M. Potkonjak. Computer aided design of fault-tolerant application specific programmable processors. *Computers, IEEE Transactions on*, 49(11):1272–1284, November 2000.

[31] R. Karri and A. Orailoğlu. Transformation-based high-level synthesis of fault-tolerant ASICs. In *IEEE Design Automation Conference*, pages 662–665, 1992.

[32] R. Karri and A. Orailoğlu. Time-constrained scheduling during high-level synthesis of fault-secure VLSI digital signal processors. *Reliability, IEEE Transactions on*, 45(3):404–412, September 1996.

[33] P. Kollig and B. Al-Hashimi. Simultaneous scheduling, allocation and binding in high level synthesis. *Electronics Letters*, 33(18):1516–1518, August 1997.

[34] V. V. Kumar and J. Lach. Heterogeneous redundancy for fault and defect tolerance with complexity independant area overhead. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 571–578, 2003.

[35] F. J. Kurdahi and A. C. Parker. REAL: a program for REgister ALlocation. In *DAC*, pages 210–215, 1987.

[36] G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Behavioral synthesis of fault secure controller/datapaths based on aliasing probability analysis. *IEEE Transactions on Computers*, 49(9):865–885, 2000.

[37] P. Lala and A. Burress. Self-checking logic design for FPGA implementation. *Instrumentation and Measurement, IEEE Transactions on*, 52(5):1391–1398, October 2003.

[38] P. K. Lala. *Fault Tolerant and Fault Testable Hardware Design.* Prentice-Hall International, Englewood Cliffs, N.J, 1985.

[39] S. Liao. Towards a new standard for system-level design. In *Hardware/Software Codesign, Eighth International Workshop on*, pages 2–6, 2000.

[40] T. A. Ly and J. T. Mowchenko. Applying simulated evolution to high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):389–409, March 1993.

[41] D. MacMillen, M. Butts, R. Camposano, D. Hill, and T. W. Williams. An industrial view of electronic design automation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19(12):1428–1448, December 2000.

[42] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Inc., Burlington, MA, 1994.

[43] G. D. Michell and R. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.

[44] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[45] A. Orailoğlu. Graceful degradation in synthesis of VLSI ICs. In *Defect and Fault Tolerance in VLSI Systems, IEEE International Symposium on*, pages 301–311, 1998.

[46] A. Orailoğlu and R. Karri. Coactive scheduling and checkpoint determination during high-level synthesis of self-recovering microarchitectures. *IEEE Transactions on Very Large Scale Systems (VLSI) Systems*, 2(3):304–311, September 1994.

[47] A. Orailoğlu and R. Karri. Automatic synthesis of self-recovering vlsi systems. *Computers, IEEE Transactions on*, 45(2):131–142, Febuary 1996.

[48] J. Patel and L. Fung. Concurrent error detection in alus by recomputing with shifted operands. *Computer, IEEE Transactions on*, C.37(7):589–595, July 1982.

[49] J. Patel and L. Fung. Concurrent error detection in multiply and divide arrays. *Computer, IEEE Transactions on*, C.32(4):417–422, April 1983.

[50] P. Paulin and J. Knight. Force-directed scheduing for the behavioral synthesis of ASIC's. *CAD/ICAS, IEEE Trnasactions on*, CAD-8(6):661–679, July 1989.

[51] M. Renovell, J. Portal, J. Figueras, and Y. Zorian. Testing the interconnect of RAM-based FPGAs. *Design & Test of Computers, IEEE*, 15(1):45–50, January-March 1998.

[52] M. Rim, A. Mujumdar, R. Jain, and R. de Leone. Optimal and heuristic algorithms for solving the binding problem. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(2):211–225, June 1994.

[53] A. Rincon, G. Cherichetti, J. Monzel, D. Stauffer, and M. Trick. Core design and system-on-a-chip integration. *Design & Test of Computers, IEEE*, 14(4):26–35, October-December 1997.

[54] P. M. Russo. Vlsi impact on microprocessor evolution, usage, and system design. *IEEE Journal of Solid-State Circuits*, SC-15(4):397–406, August 1980.

[55] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation, 2nd Ed.* Digital Press, Burlington, MA, 1992.

[56] A. H. S.N. Hamilton and A. Orailoğlu. Efficent self-recovering ASIC design. *Design & Test of Computers, IEEE*, 15(14):25–35, October-December 1998.

[57] SUN. $sun.com/processors/ultrasparc-iv$. 2004.

[58] M. Tahoori, E. McCluskey, M. Renovell, and P. Faure. A multi-configuration strategy for an application dependent testing of FPGAs. In *Proceedings 22nd VLSI Test Symposium*, pages 25–29, 2004.

[59] C.-J. Tseng and D. Siewiorek. Automated synthesis of data paths in digital systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 5(3):379–395, July 1986.

[60] K. Wu and R. Karri. Algorithm level re-computing- a register transfer level concurrent error detection technique. In *Computer Aided Design, IEEE/ACM International Conference on*, pages 537–543, 2001.

[61] K. Wu and R. Karri. Algorithm level recomputing using allocation diversity: A register transfer level approach to time redundancy-based concurrent error detection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(9):1077–1087, September 2002.

[62] K. Wu and R. Karri. Selectively breaking data dependences to improve the utilization of idle cycles in algorithm level re-computing data paths. *Reliability, IEEE Transactions on*, 52(4):501–511, December 2003.

Index