The Automated Compilation of Comprehensive Hardware Design Search Spaces of Algorithmic-Based Implementations for FPGA Design Exploration.

A Thesis

Submitted to the Faculty

of

Drexel University

by

Michael A. Balog, III

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

June 2007

© Copyright 2007
Michael A. Balog, III. All Rights Reserved.

Dedications

To my parents and loving fiancée...

Acknowledgements

I would first like to thank my advisor Warren Rosen for keeping me focused, helping me convey my ideas, and pushing me to write better "introductions".

I would like to thank everyone at Rydal Research for their overall support during this entire process, especially Francisco Quirós for his expert knowledge of Prolog, Jon Adams for helping me with the many iterations and edits of my thesis, and Diane Hecht for helping me with my thesis content and final proof reading. I would also like to thank everyone at BuLogics that helped me, especially Ryan Buchert for taking time to discuss my research, Bryce Nichols for help with Linux, and Jeff Picciotti for help with my presentation.

Finally I would like to thank Chelsea's and my parents. Peggy and Kevin were very encouraging and helped me to relax and stay focused on my writing. My mother and father have been the most supportive parents anyone could ever ask for, without their love and assistance I would have never made it as far as I have today.

Lastly Chelsea, I can't say enough about her ongoing support, encouragement, and help over the past five years. Without her by my side I would never have finished.

Table of Contents

List of Tables vii

List of Figures viii

Abstract xi

1. Introduction 1

    1.1 The Manual Approach to Application Implementation 3

        1.1.1 Choosing an Architecture 4

        1.1.2 FPGA Implementation Process 7

        1.1.3 Hardware Descriptive Languages 9

2. Background of Previous Work 13

    2.1 Introduction 13

    2.2 Research in Hardware Descriptive Languages 15

    2.3 High Level Language to Hardware Design Compilers 17

        2.3.1 SUIF based HLL Compilers 19

        2.3.2 Non-SUIF HLL Compilers 26

    2.4 Estimation Tools 32

    2.5 Summary 35

3. Creating a Comprehensive Hardware Design Search Space 37

    3.1 Introduction 37

    3.2 Programming Paradigms 39

        3.2.1 Imperative and Declarative FIR design example 40

    3.3 A Declarative Programming Language 45

        3.3.1 Language Syntax 45

3.3.2 Data Flow Analysis ... 47

3.3.3 Equation Rewriting ... 50

3.4 Hardware Description Language Coding Style ... 54

3.5 Summary ... 57

4. Demonstration Compiler System ... 59

4.1 Introduction ... 59

4.2 Expression Generator ... 61

4.3 Hardware Compiler ... 67

4.4 Design Generator ... 78

4.5 Summary ... 81

5. Results ... 83

5.1 Introduction ... 83

5.2 Hardware Design Search Space ... 84

5.2.1 Expanding the Metrics of the Hardware Designs Search Space 86

5.3 Design Space Exploration ... 90

5.3.1 Parallel Designs ... 90

5.3.2 Serial Designs ... 93

5.3.3 Hybrid Designs ... 97

5.4 Accuracy of the Hardware Design Search Space ... 101

5.5 Summary ... 107

6. Conclusion ... 109

6.1 Summary of Contributions ... 109

6.2 Recommendations for Future Work ... 111

Bibliography ... 114

A. List of Acronyms 122

B. Example of VHDL design styles 125

B.1 Behavioral Design 125

B.2 RTL Design 126

C. VHDL files for examples in Chapter 4. 135

C.1 Structural RTL VHDL for a 16-Bit unsigned adder. 135

C.2 VHDL design for example Z_{54} 138

Vita 148

List of Tables

3.1 Syntax for a functional declarative programming language. 46

3.2 Synthesis LUT estimates and PAR LUT usages for a behavioral and RTL design. 56

4.1 A condensed table of a 16-bit data width building block library. 71

4.2 Data flow table for `Z54`. 72

4.3 Hardware Data flow table for `Z54`. 75

List of Figures

1.1 FPGA Design Process ... 8

3.1 Operational flow of a HLL application to hardware compiler. ... 38

3.2 ANSI C description of a variable tap FIR filter. ... 42

3.3 ANSI C description of an eight tap FIR filter. ... 43

3.4 Functional declarative description of a variable tap FIR filter. ... 43

3.5 Functional declarative description of an eight tap FIR filter. ... 44

3.6 Data flow graph for the serial eight tap FIR filter example. ... 49

3.7 Data flow graph for the parallel eight tap FIR filter example. ... 49

4.1 Operational flow of the prototype system for the creation and exploration of a comprehensive hardware design search space. ... 60

4.2 Block Diagram of the Expression Generator. ... 62

4.3 Data flow graphs for the expressions y_1 to y_5 ... 63

4.4 Rearranged inputs of the data flow graphs for the expressions y_1 to y_5 ... 64

4.5 Sample of a generated equation space for eight summed operations of multiplication. ... 66

4.6 Block Diagram of the Hardware Compiler. ... 67

4.7 Building Block Library Entry for a 16-bit Adder. ... 69

4.8 Data flow graph for `Z54`. ... 72

4.9 Examples of multiplexer trees for $A_{i=0\sim6}$ and $B_{i=0\sim6}$. ... 73

4.10 Hardware data flow graph for Z54. 76

4.11 Block diagram of the Design Generator. 79

4.12 Hardware Design Search for eight summed operations of *multiplication*. . . 80

5.1 Area vs. Delay and Area vs. Latency plots of the hardware design search space for the eight-Tap FIR filter example. 85

5.2 Three-dimensional plot of the hardware design search space. 89

5.3 Snapshot of the parallel hardware design search space. 91

5.4 Hardware dataflow graphs for combinatorial parallel designs Z_4 and Z_{18}. . . 92

5.5 Hardware dataflow graphs for combinatorial parallel designs Z_6 and Z_{14}. . . 93

5.6 Hardware dataflow graphs for clocked parallel designs Z_4 and Z_{18}. 94

5.7 Snapshot of the serial hardware design search space. 95

5.8 Hardware dataflow graphs for combinatorial serial designs Z_{30} and Z_{31}. . . . 96

5.9 Hardware dataflow graphs for clocked serial designs Z_{30} and Z_{31}. 97

5.10 Snapshot of the hybrid hardware design search space. 98

5.11 Hardware dataflow graphs for combinatorial hybrid designs Z_{34} and Z_{49}. . 99

5.12 Hardware dataflow graphs for clocked hybrid designs Z_{34} and Z_{49}. 100

5.13 Wrapper for the hardware designs created by the prototype hardware compiler. 102

5.14 Comparison of estimated and actual area & delay performance characteristics for the parallel form designs. 105

5.15 Comparison of estimated and actual area & delay performance characteristics for the serial form designs. 106

5.16 Comparison of estimated and actual area & delay performance characteristics for the hybrid form designs. 106

Abstract

The Automated Compilation of Comprehensive Hardware Design Search Spaces of Algorithmic-Based Implementations for FPGA Design Exploration.

Michael A. Balog, III
Advisor: Dr. Warren Rosen

Over the past few years FPGA hardware has become a logical choice for implementing cutting-edge signal processing applications. While there have been advances in FPGA technology, the common process of creating specialized hardware implementations for them is a manual one involving extensive design exploration. Design exploration is a process that requires a designer to look for designs that fit a set of performance characteristics such as size, throughput, or power depending on the application and it can be the most time consuming step when creating FPGA hardware. This process is a nontrivial task that requires extensive background knowledge of both FPGA hardware and the application being implemented. While advances have been made in automating the process of design, there is still a gap between the application writers and hardware engineers that can be filled.

This thesis presents a novel approach for automating the generation of hardware design search spaces that contain a comprehensive set of ways to implement signal processing algorithms with FPGAs. To accomplish this we generate a set of equivalent mathematical representations for an input equation via a novel declarative programming language that avoids a number of difficulties associated with the imperative languages used by previous approaches. We show that this equation space is bounded in terms of bracketing and ordering of mathematical operations, and

that by changing the way an equation is written we can generate unique hardware instantiations (designs). The generated instantiations are mapped to heterogeneous computing architectures and written in a structural hardware descriptive language style to ensure that the intended instantiation will behave as predicted in hardware. A software system was created based on this approach that generates an equation space for varying numbers of *summed multiplications* and converts each representation into a comprehensive hardware design search space that can be analyzed for performance characteristics such as size, throughput, latency, and power.

1. Introduction

Field Programmable Gate Arrays (FPGAs) are becoming ubiquitous in high-performance military and commercial signal processing systems where the relatively small numbers of systems produced preclude the use of expensive specially designed and fabricated integrated circuits. In the past the most common way to implement signal processing applications was specialized software running on general purpose or specialized Digital Signal Processing (DSP) processors. Newer, more advanced signal processing applications now require more processing power than these common processors can deliver, and to implement these complicated applications industry is choosing FPGAs.

An FPGA is an integrated circuit that contains reprogrammable logic components and interconnects that can be programmed to duplicate the functionality of custom-designed integrated circuits. The added advantage that FPGAs may be reprogrammed to match the requirements of ever-changing signal processing algorithms provides flexibility unmatched by integrated circuits.

The biggest problem limiting the usefulness of FPGAs is the long and costly process of converting a signal processing application into a Hardware Descriptive Language (HDL) that can be used to program the device. Typically, an application is developed by an expert in signal analysis who writes and evaluates the application in a high-level language such as C or Matlab. The algorithm must be converted into HDL by a hardware engineer. The hardware engineer performs this conversion through a time-consuming process of trial-and-error, *guessing* at the best

way to partition the application and distribute the various components among the resources available in the FPGA. Each guess must be implemented and synthesized into a low-level bit-stream that programs the target FPGA device. The bit-stream must be analyzed to determine if the design is optimal in terms of speed, size and throughput; if it isn't the process must be iterated. This iterative process is called design exploration, and is complicated by the fact that neither the application developer nor the hardware engineer has a detailed understanding of the other's function, creating the potential for miscommunication which frequently increases the cost and time of development.

This thesis presents a novel way of automatically generating guaranteed optimized and correct HDL designs for complex applications. By automatically generating these HDL designs, we accelerate the process of implementing an HLL application in FPGA hardware and ensure that the design represents the HLL application with the least amount of excess overhead while meeting performance criteria. We accomplishes this through the generation of equivalent mathematical representations of an input equation via a novel declarative programming language that avoids a number of difficulties associated with the imperative languages used by previous approaches. We generate only heterogeneous computing architecture designs so as to represent the maximum possible performance for each design. This entire process alleviates the time spent rewriting different designs due to poor performance or repartitioning of the algorithm. Overall this approach reduces the process of design exploration to one of merely choosing the best design(s) that fits an application's requirements.

The remainder of this chapter discusses in greater detail the manual iterative

approach to implementing an application in FPGA hardware. Section 1.1 reviews how an application is broken into sections for implementation, briefly reviews the importance of internal architectures, and how they are currently implemented in FPGA hardware.

1.1 The Manual Approach to Application Implementation

When implementing an application in FPGA hardware there are many different parameters that need to be defined, and this process can be arduous and time consuming if a clear and concise implementation is not identified immediately. When an application is implemented in an FPGA, the designer(s) must develop the internal hardware architecture within the device. The conventional approach to creating the internal architecture of an FPGA implementation of a signal processing application starts with the design and analysis of a signal processing algorithm. The algorithm is usually a software-based floating point behavioral model that can be tested extensively over a short period of time. It is designed and tested by a signal analyst and is verified for overall correctness. Once testing and verification is complete, a Fixed-Point Approximation Model (FPAM) is created from the algorithm by calculating performance and estimating fixed-point ranges to get a numeric approximation with an acceptable percent of error. The FPAM is simulated and tested for correctness, but the models are complicated and can take much longer to verify. When complete, hardware engineer(s) *blocks-out* sections of the FPAM to code into a HDL representation that will exactly mimic the algorithm.

The current way that a FPAM is *blocked-out* is to examine the steps of the algorithm for hardware exploits and estimate performance of each potentially *blocked-out*

section. Hardware exploits are mathematical operations that can be either combined, removed, or optimized for common logical operations. For example any required trigonometric, exponential, or logarithmic function values are approximated and stored in registers. *Best Guess* estimates of FPGA area usage, circuit delay, and design throughput are made to determine how each *blocked-out* section of the algorithm will perform. These estimations are made by engineers who have years of experience in interpreting applications for FPGA hardware, however the *Best Guess* methodology can result in quite a few design iterations of application if estimates were made incorrectly.

1.1.1 Choosing an Architecture

With the FPAM *blocked-out*, the hardware engineers must select how to implement the sections while attempting to maintain desired area and delay requirements. To the hardware designer there are a number of architectures available, and they must choose the best one based on an application's performance requirements such as area or throughput. An FPGA is that of a flexible device and can either mimic a general purpose processor or be configured as a specialized hardware circuit that mimics the application. FPGAs provide this flexibility through the use of a two-dimensional array of Configuration Logic Blocks (CLBs) with integrated block memories that are connected together with a dynamic routing structure. This requires the hardware engineer to depict a specific internal architecture when implementing the *blocked-out* FPAM.

Common computer architectures fall into two categories-sequential or parallel processing. The most common approach to sequential processing is to use the von

Neumann architecture [75]. This architecture contains a memory device to store data and instructions, a computational/functional unit to perform operations, and a control unit to handle the movement and scheduling of data/operations. As computers have increased in complexity, other architectures such as the superscalar architecture [67] have emerged to increase the computational performance of the von Neumann architecture. Superscalar architectures [57] are a parallel processing approach that uses more than one function unit to execute multiple instructions during a single cycle. To perform multiple instructions per cycle, these architectures use Instruction Level Parallelism (ILP), a technique that identifies independent instructions that can be dispatched simultaneously. ILP requires the controller to pre-fetch a set of instructions and decide which can be executed independently. Even though ILP allows for multiple instructions to be performed at once, operations are still transferred to and from memory sequentially. Both of these architectures suffer from having only one data path to move data to and from the processing elements. In order to maximize performance the control logic must keep the computational units busy, requiring extra scheduling of the hardware resources (which requires more dedicated hardware).

When von Neumann style architectures like these are implemented inside an FPGA all of the typical architectural components must be created and a set of instructions must be defined. The functional units are tailored for the application and can be either simple arithmetic or complicated algorithmic functions. A configurable data path that connects the functional units to memory device(s) is implemented and operated by a controller. Since the controller dispatches the data and instructions, the schedule of operations needs to be modeled and instructions are either

placed into a memory device or incorporated into the design. These implementations have been shown to have a greater performance than general purpose or even digital signal processors. At an instruction level they have potential for being reused in other applications, but require extensive testing and modeling to be performed prior to implementation.

An alternative approach to the common computer architecture is the heterogeneous computing architecture in which the implementation resembles the data path (data flow) of a specific application. This style, sometimes called a flow-through architecture, is parallel in data flow and operation execution. This architecture removes the need for a central controller and is more conducive to implementing mathematics due to the intrinsic nature of how algorithm data flows through the functional units that depict the operations of the application. Since the data is flowing through the functional units, small distributed memories along the data path fair better than a single large memory used by the von Neumann style approach [24]. The drawback of this approach is that implementing this type of designs is a not a simple task, and each design is tightly coupled to a particular application, making it unsuitable for another application.

FPGA devices are excellent for implementing heterogeneous computing architectures due to their configurable nature. As seen in the von Neumann style architectures, memory connectivity also plays a large role in how data can be processed. Since FPGAs have distributed blocks of memory throughout their two-dimensional array of CLBs, they are a logical choice for implementing heterogeneous architectures that require memory integrated into the design. The number and size of these memories are defined by the algorithm and the capabilities of the FPGA being used.

This flow-through architectural style has more advantages over a von Neumann architecture, since it can be designed to use the highest level of parallelism for the application given the resources of the FPGA.

1.1.2 FPGA Implementation Process

The flow chart in Figure 1.1 shows the process of manually implementing an application in FPGA hardware. At each decision point in the process, the design must meet different performance criteria determined by the hardware designer, such as area, throughput, and power usage. If the design does not meet these requirements the hardware designer must start over. This process of design and analysis, known as design exploration, is not limited to just design simulation, but is repeated throughout the entire design and implementation process.

The first step is the decision of which style of architecture to use. This is usually performed by an experienced hardware designer reviewing the particular requirements of the application. Once selected, the architecture is described using HDL and careful attention must be given to how it is written, as the style of how the HDL is written will greatly influences the rest of the process. When completed the HDL code is simulated to test for logical correctness. If the simulation tests fail, the design must be debugged and possibly rewritten. The next step is to create a list of FPGA-specific components and to describe how they are connected for the intended circuit. Synthesis tools automate this process by converting HDL-coded designs into FPGA netlists. Once the design is synthesized, the tool(s) gives estimates of area usage and timing delay for the circuit. If these estimates do not meet the expectations of the design, the process must be iterated from the beginning.

With the FPGA netlist complete, the design can be placed and routed to an FPGA using vendor-specific place-and-route (PAR) tools and a binary program file (bit-stream) is generated. Final area and delay values are given by the PAR tool for the generated bit-stream file. If they do not meet the design expectations the process must be iterated again.

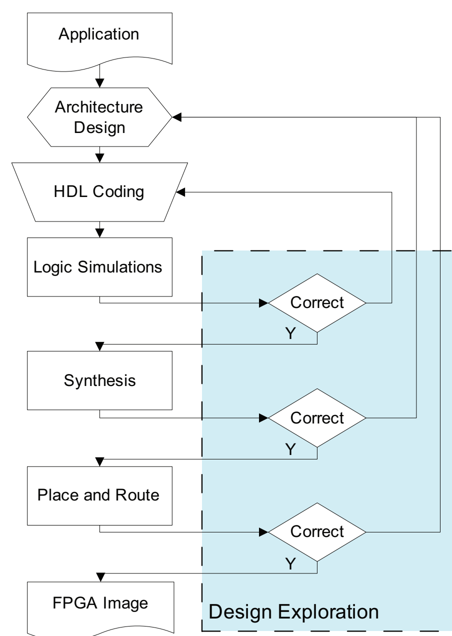

Figure 1.1: FPGA Design Process

There are many different ways to implement the same algorithm, each having different tradeoffs in area, timing, and/or throughput. There is always a chance that even if implemented properly, a circuit will not meet necessary performance criteria at a synthesis or PAR level. FPGA resource usage and performance values must

be tracked as different designs are created until a design that meets the criteria is selected. This whole process of creating and evaluating different designs can take months to complete, since it can take weeks to produce just one design. Pre-modeling and estimation in the early stages of the design process can help to identify possible designs that will meet the performance criteria and thus reduce the time to design.

1.1.3 Hardware Descriptive Languages

In an effort to accelerate the FPGA design process, synthesis tools were developed with the ability to interpret high-level HDL designs descriptions. The style of how a HDL design is encoded has a major impact on the post-design tool's ability to extract an intended implementation. This is especially true in large designs and has lead to a style of coding called "designed for synthesis". This style defines a set of terms that, when used to generate HDL, forces synthesis tools to implement the intended architecture.

A popular HDL language used to describe FPGA hardware is VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language). VHDL was developed by the U.S. Department of Defense as an initiative to create a language for specifying the behavior of ASICs. It was extended to support simulations and was also made into an IEEE Standard (1076). Due to VHDL's original intended use as a simulation language, some limitations must be taken into account when using it to describe hardware designs for implementation in FPGAs. As a result, there is a specified subset of each language that is used to describe synthesizable circuits.

Over time three coding styles have been developed that help a user create a

"design for synthesis." The first style is called Register Transfer Level (RTL) coding for synthesis (sometime called structural coding for synthesis). This style of coding provides information about all the operations, communication, and connectivity for a given circuit. It includes any specific information about which FPGA resources have been instantiated in the design and is meant to leave little interpretation to the synthesis tool. The RTL style will produce designs that are as close to the input code as intended by the designer, but requires a considerable amount of effort and knowledge on the part of the designer. Since synthesis merely translates this code to an FPGA netlist, little to no optimizations are performed by synthesis to remove redundant or irrelevant logic.

The second style is called coding for behavioral synthesis (sometimes called high-level synthesis). This style of coding provides a representation of the algorithm in sequential execution steps, similar to software [9]. It describes the input/output, procedures, and high level intentions of the designer. Behavioral coding provides the designer with a higher level of abstraction and leaves the details of hardware instantiation to the synthesis tools. Most common synthesis tools will use this behavioral code and specify any specific timing, components, and low level functionality to be performed. This coding style can sometimes be prone to errors in the estimation of area and delay for complicated designs because synthesis performs optimizations to remove or re-pack logic where it deems necessary to meet a certain performance metric.

The third style is called coding for algorithmic synthesis [13], it attempts to decouple input/output timing and architectural dependencies from the source code while speeding up the time to design hardware. This style allows the designer to

focus on the algorithm, which can be specified in an ANSI C++ language. Compile-time directives are used to help the synthesis tool interpret an architecture from the ANSI C++ code. The output of this tool is either a timed RTL HDL design or an FPGA netlist ready for PAR. The designer is still required to depict the architecture, thus the tool is merely assisting in the translation from one high-level language to another.

The iterative manual approach to implementing an application in FPGA hardware requires hand writing components specifically for the particular design. The amount of time required to create such a custom solution often makes FPGAs less popular for small applications. Design exploration is also necessary in order to find an implementation that meets the requirements of an application. This process can take weeks to accomplish, since implementing one design can take days.

In an effort to accelerate this process, research has been performed to achieve automating the translation of an application's High-Level Language (HLL) description written in ANSI C or Matlab to a specific hardware architecture implementation. The focus has been on creating a HDL design from a HLL input, but not exploring different designs. While this assists hardware engineers in creating the FPGA implementation, it still requires them to break apart the application into sections for implementation and decide what architecture to implement; therefore the designer is left with performing any design exploration iteratively.

In Chapter 2 we discuss the current state of the research in this area and review the available commercial tools that translate HLL descriptions of an application into hardware designs. Next, in Chapter 3, we present the approach of this research, to create a comprehensive hardware design search space for the exploration of an opti-

mal design. Chapter 4 reviews how we applied this approach to a prototype system created to demonstrate the advantages of the approach, and Chapter 5 discusses the results generated by that system for an example application of a FIR filter. Finally, in Chapter 6, we conclude the current work and describe future directions.

2. Background of Previous Work

In this chapter we discuss the current research and commercial systems that attempt to automate the manual process of implementing a signal processing application in FPGA hardware. These vary from new languages used to describe an application; to compilers that create optimized HDL designs of an application.

2.1 Introduction

Most of the early research in the area of design automation created new methods to describe signal processing applications for FPGA implementation. These methods required designers to convey the exact implementation architecture, and as a result showed that their techniques, for using C-like descriptions to describe hardware, did not accomplish the goal of automating the process of hardware design. The reason for these shortcomings is a result of the many intricacies of signal processing applications, and the expectations put upon the application writers. The application writers were required to have a high level of intuition when writing the design with regards as how to break apart the application, describe it so that the compiler would interpret the design correctly, and basically know how the target FPGA device operates.

Since these languages were originally developed to provide new ways for describing an application in a higher abstraction level, another approach attempted to make the hardware design process more *transparent* to the application writer. But just like the previous research, it used high level C-like descriptions that could not

produce optimal designs without some knowledge of the overall hardware design being implemented. While these systems translated the designs to VHDL that could be synthesized, the synthesis tools were required to recognize all forms of components from simple counters, registers, and multipliers to complex mathematics. This became a tough task for synthesis tools to accomplish optimally, as they were not intended to consider any high-level performance requirements of an application but simply to translate VHDL into FPGA netlists.

Another group of systems automats the implementation of applications written in C-like languages and translates them into hardware. The strategy of this new research is to approach the design process in way that leads to the most parallel hardware representation from a data flow description. Their focus is on loop exploration and other parallel processing techniques to identify and optimize pre-defined sections of the input code to facilitate the most parallel design that could fit in the FPGA. Though it was shown that the time to design was reduced when using these systems; the designs produced by these systems were larger and slower than those written by hand.

All of these systems require the hardware engineers to make *guesses* about what sections could benefit from different types of optimizations, either by writing compiler directive statements or organizing the algorithm in such a way that the compiler will recognize those sections as different blocks. In most cases when these systems would perform estimates, synthesis tools were used for area calculation. These systems were shown to be inaccurate when estimating area and delay, since most of them use a behavior style of HDL coding. Overall these systems require that the input HLL application be written in a way that still conveys an architecture and

they produce HDL designs that are interpreted by the synthesis tools for a final implementation.

The remaining chapter is organized into 4 sections. Section 2.2 discuss the early languages that were developed to raise the level of abstraction from the hardware designer when coding designs. In Section 2.3 we discuss the automated hardware design compilers, were the bulk of the research has been done in this area. This section includes a review of both research and commercial systems. In Section 2.4 we discuss estimation tools that were developed over time to assist in early design space exploration when determining the performance of a design. Section 2.5 summarizes the content present in this chapter.

2.2 Research in Hardware Descriptive Languages

Some early attempts at automating the HDL design process were languages that attempted to raise the level of abstraction from the hardware designs and allowed for the creation of designs without the knowledge of the low-level details of FPGA implementation. HardwareC [45] is a C-like language that provides a hardware description for a target FPGA. It utilizes a C-style syntax structure to convey the description, but requires timing and resource constraints like that of synthesizable RTL coding. Transmogrifier C [28] uses a subset of the C syntax and extends its capabilities with a set of compiler directives that indicate a specific hardware circuit. It was tailored for a specific series of FPGAs (TM-1 Xilinx, TM-2 Altera) and only supports simple integer data, loops, and if-then statements. Complicated arithmetic operations like division, multiplication, and array manipulations are not supported. SystemC [56], a more popular language, can be used to describe hardware at a RTL

level like VHDL or Verilog. It was develop by Synopsys and has become its own language standard. The SystemC community has created a library of components that can plug into an application for accelerated design. It requires the designer to have a high level of knowledge of the target hardware. SystemC also requires the designer to learn a new style of conceptualizing hardware through their C-style language.

Handel-C [22] is a language designed by Celoxica for creating FPGA and ASIC images through a small subset of ANSI-C expressions. It requires that the use of memories, signals, and parallel operations to be explicitly declared. It employs a CSP (Communicating Sequential Processes) style of modeling with a very strict constraint set. Like VHDL and Verilog, it is possible to create designs that cannot be synthesized.

Esterel-C Language (ECL) [47] developed by Cadence is a HDL and compilation suite based on ANSI-C. Esterel-C is a synchronous/reactive (SR) style language intended for software designers to model computational architectures such as real time systems. SR models excel at concurrent and complex control logic applications but leave little for interpretation or design space exploration since they must be specified at such an intricate low level.

The goal for all of these languages was to create a more general syntax when describing an application compared to conventional RTL Verilog or VHDL. They require the designer to direct the compiling tools in creating the style of hardware architecture by depicting the I/O ports, bit widths, and explicit parallelism needed. This requirement places more effort on designers to learn the syntax techniques of another language, with only a benefit of programming FPGA hardware in a

version of C-code that looks more like traditional HDL. While this might be slightly more accessible to a C programmer, it still requires the designer to understand and describe an implementation at a hardware level.

2.3 High Level Language to Hardware Design Compilers

Over the past several years a number of systems have been developed to automate the process of converting directly from a HLL description to FPGA or other configurable hardware. Typically these systems are based on an existing software compiler known as SUIF (Stanford University Intermediate Format) [65] and most use a different unique C-style language as their front-end input. Others contributed to this development by creating a specialized C code called SA-C (Single Assignment C-code) [42] while some use Matlab M-Files [71] for an input language. Current commercial systems [50] can take ANSI C inputs and translate them to an RTL representation through compiler directives. As was shown with the SA-C compiler, a decrease in the time-to-design circuits was accomplished, but resulted in circuits that were twice as slow as those designed manually [36].

Hyper [61] was an early system for designing ASIC structures that showed parallel processing exploits and data flow analysis could be used to created computational engines and control flow. Hyper noted that a for a von Neumann style of architecture, the implementation of the controller and data path largely determine the capability of the circuit, since the controller is responsible for managing the flow of data and institutions between memory and multiple function units. It was also discussed that early behavioral specification of an algorithm is crucial in exposing temporal and operational information so that a performance and area tradeoff

analysis can be conducted.

Many techniques of parallel processing are utilized to accomplish this discovery of available performance tradeoffs. Some of the techniques used are strip-mining, splitting loops to form a multiple concurrent loops. Loop fusion, the process of fusing multiple loops together into one loop. Loop tilling, the breaking of a large loop into smaller tiles. Loop Unrolling is expansion of nested loop so that outer loops may exploit inner loop parallelism. Other techniques of parallel processing require data organization techniques. CSE (Constant Subexpression Elimination) is the process of eliminating constants from a data set. Data reuse is the concept of storing already fetched data for use in future operations. These techniques in form or another are employed in most of the current research.

To use these techniques, most systems rely on the SUIF (Stanford University Intermediate Format) [65] compiler. SUIF was designed as a development format for research in parallel processing to identify parallelism in sections of an application. It uses front-end systems to transform different formats of input code to a SUIF IR (Intermediate Representation), a dataflow representation. SUIF then performs loop exploits and memory access analysis, and facilitates custom optimizations on the SUIF IR. These optimizations allow a user to develop new ”passes” in the environment and implement different data flows.

Most research systems use customized front-ends to parse C, FORTRAN, or Matlab into the SUIF IR and then used SUIF to automate the processes of translating a HLL application to a hardware implementation. The new dataflow representations are then scheduled onto a von Neumann structure or in a few cases a data path structure inside the FPGA. The terminology may differ, but each is consistent in

performing a form of loop optimization, memory analysis, and in some cases estimation. The most popular method for design space exploration is manually creating different front-end input hardware renditions that have variation of the structure through compiler directives. Unfortunately most designs can not be expanded to their most parallel form while others are made to fit into a sequential processing architecture. Since most use a front-end source code that resembles a sequential description of an application, a clean extraction of dataflow operations is hindered. This issue leads most of the systems to spend great deal of effort in memory access strategies and hardware scheduling analyses.

2.3.1 SUIF based HLL Compilers

The BRASS (Berkeley Reconfigurable Architectures, Systems, and Software) Research Group created a SoC (System on a Chip) called Garp [15], that combines a MIPS-II processor with a reconfigurable coprocessor much like a FPGA on the same chip. Their compiler creates an effective hardware/software partition for the Garp chip. The compiler uses SUIF to break the input code into hyperblocks [19], which are hierarchically arranged nodes that contain a set of basic components derived from a section of loop code that is frequently executed. These hyperblocks are translated to a DFG representation with respective control dependencies and the Garp compiler maps these nodes to modules inside the Garp chip.

The Garp research inspired the Napa-C compiler [31]. The Napa-C compiler explores the problem of automatically mapping array variables to memory banks [32] to their SoC similar to Garp's. The programmer tells the compiler (via compiler directives) which sections of the code are executed by the processor, which sections

should become reconfigurable hardware, and what sections are implemented via software. Their focus is on RISC-based processor architecture and they later developed support for controlling the complex interface between a FPGA and a host processor.

The Nimble system compiles C-style applications to a hardware/software platform consisting of a CPU and FPGA [48]. This research focuses on a partitioning algorithm that would perform fine-grained examination of ILP. When examining the algorithm the system attempts to reduce the number of loops by examining only the "interesting" structures. These structures are interpreted in hardware such that the loops are executed purely sequentially. It uses Garp's compiler to extract hardware kernels from the application and uses an Architecture Description Language (ADL) to make it retargetable.

Streams-C [27] is a stream oriented compiler, based on the Napa-C and MARGE (Malleable Architecture Generator) [29], that creates hardware circuits for use on the WildForce Board from Annapolis Micro Systems [3]. MARGE maps parallel data operations to a specific chip's architecture at gate or component RTL, by creating structures from basic blocks of C code in the form of a functional unit, data path, and control unit. The Streams-C compiler takes a C-style input which conveys the operations of "streams" through compiler directives (open, close, read, etc.) and uses MARGE to create FPGA hardware processes. These streams are routed through the FPGA to the different processes using a CSP (Communication Sequential Process) programming model [30]. A comparison of the implemented designs was performed between the compiler and a manual approach. The compiler's design was about three times larger in area, twice as slow, and a factor of ten times faster to implement. This shows that an automated compiler system can have a

major impact on the time-to-design, but requires a better strategy to implement the design.

The PipeRench [33] project focuses on virtualizing hardware by using a high-speed reconfiguration chip to implement a design of any size. They developed an architecture which uses a virtual set of pipeline stages call stripes and a compiler that focuses on the configuration time, compatibility, and logic configuration. Their compiler can compile an application written in a specific dataflow language and generate a configuration that is loaded to their specific device.

The Phoenix project [72] uses CASH (Compiler for ASH (Application Specific Hardware)) [16] to transform programs written in ANSI C to RTL Verilog hardware. Their system introduces Spatial Computation (SC) [17] which is a hardware architecture that is distributed with no centralized control. CASH does this by using SUIF to partition the input C code and translate it into the Pegasus dataflow IR. Pegasus uses a Static Single Assignment(SSA) model which is an IR for programs in which each variable is assigned only once. Like the Garp C compiler, CASH creates a DFG with hyperblocks so that the Pegasus IR can be used to perform loop exploits while examining memory dependences without excessively inhibiting parallelism [74]. The CASH Asynchronous Back-end (CAB) synthesizes each node in the DFG as unique pipeline stages and creates structural Verilog code. The circuits created use an asynchronous communication scheme with a 4-phase handshake to pass data between each block and pipelining is only performed by a matching the delay at each stage.

Weinhardt and Luk [76] showed that their compiler could examine applications written in a C-style description and create designs that maximized parallelism within

the application to a specific FPGA netlist. This compiler, known as SPC (SUIF Pipeline Compiler), uses SUIF to perform custom passes to examine each loop section and perform software based parallel processing techniques. Their approach is based on the assumption that the most parallel structure for a particular application will produce the greatest possible speed-up that can be obtained. Their benchmark results showed how designs that went through the SPC performed better then executed on a general purpose processor. When using this prototype compiler, scheduling techniques and resource management are performed by the user and design space exploration was not carried out.

The XPP-VC (XPP-Vectorizing C) compiler employs SUIF and the techniques utilized in the SPC [20]. It specifically targets a reconfigurable computing architecture called XPP whose structure is that of a two-dimensional array of processing elements. XPP is configurable for different arithmetic and logic operations through use of a distributed memory architecture, interconnect, and configuration manager to control the processing element array.

The Raw (Raw Architecture Workstation) [55] compiler takes C or Fortran and creates Verilog HDL code using SUIF to perform parallelizing compiler techniques [1]. The Raw compiler focuses on memory disambiguation, scheduling, and data partitioning to create designs specifically targeted for the Raw prototype chip. The authors show that using multiple small memories increases the performance of their resulting hardware structures and that they can create finite state machines that control the process of data over an interconnect [4] automatically. They developed a technique called "virtual wires" that maximizes the usage of computational units on the Raw chip in localized areas in order to limit the wire length between

components.

DEFACTO (Design Environment for Adaptive Computing Technology) [12] is a system that takes a high level C language description and creates behavioral VHDL for the WildStar PCI board from Annapolis Micro Systems [3]. It utilizes SUIF while performing a design space exploration through use of Monet behavioral synthesis tool to obtain area and delay estimates [64].

The DEFACTO system requires the input C code to be partitioned into loops that later become stages in the hardware design. Coarse-grain pipelining is then implemented through SUIF passes to extract the data flow and perform loop unrolling, tiling, and data permutations to create the different architectures [83]. Each loop becomes a pipe stage between which a FIFO and communication block is inserted that handles queuing data. It tries to "balance" each stage by adjusting loop parameters for a particular stage and comparing the ratio of F (Fetch) rate / C (Consumption) rate. The area and delay values are estimated by taking the SUIF IR and transforming it into behavioral VHDL code. That code is then handed off to a behavioral synthesis tool that compiles the design and passes back the information for use in their design space exploration. DEFACTO's design space exploration is done by adjusting the unroll factor of each loop to create different area/delay usages [84]. Once an acceptable balance is met through the use of a greedy algorithm, the compilation is complete and the design is ready to be fully synthesized by other EDA tools.

DEFACTO noted that larger designs had a better performance then smaller design according to the behavioral synthesis estimates. When implemented these larger design exhibited a 20% degradation in clock speed and required a greater

area than estimated [63]. This is most likely due to behavioral synthesis' inability to interpret the design as intended. DEFACTO's search space is limited for two reasons. First, they only focus on memory accesses and loop complexity to enable possible speedups of the application. Second, the performance estimates are based on behavioral synthesis, which is prone to having inaccuracies in estimating performance.

ROCCC (Riverside Optimizing Configurable Computing Compiler) [73] is a compiler that takes C code and generates RTL VHDL from sections that are the most frequently used. The ROCCC system uses the "90-10 rule" and targets CSoC (Configurable System-on-a-Chip) hardware solutions [68]. Their system is base on the SUIF2 and MachSUIF (Machine-SUIF) [43] compilers. MachSUIF was developed at Harvard University as a backend for SUIF to perform machine level optimization and analysis passes. ROCCC uses it to perform SSA (Static Single Assignment) modeling, control flow graph, and dataflow graph analysis. It utilizes parallel processing techniques like loop level exploits, storage optimizations, and pipelining to create designs that are about two to three times larger in area but operate close to the same clock rate as compared to manual designs [38].

ROCCC has written their own customized IR called CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics) [35]. There are two parts to CIRRF, Hi-CIRRF and Lo-CIRRF. Hi-CIRRF is comprised of C-code implemented with macros and/or compiler directives to dictate how the design is to be interpreted. Lo-CIRRF is the previous code converted (from Hi-CIRRF) to a machine language through MachSUIF. The Lo-CIRRF is used to describe the dataflow for all the nodes and how they interact together.

When creating a design the ROCCC system analyzes the dataflow and categorizes the nodes into two groups, hard nodes that support the hardware implementation and soft nodes that represent the operations from the input C code [38]. A repeated structure is applied as the compiler creates VHDL from CIRFF. This structure contains a ”smart buffer” that is intelligent enough to manage a block memory while keeping current data and clearing out unused data [37]. This ”smart buffer” feeds the data to the next component called the dataflow block. This block performs the actual operation(s) that the compiler interpreted from the C-code and translated to VHDL code. Finally the dataflow block outputs its results to a write buffer which is connected to another block memory. Even though the nodes are transferred to VHDL as a structure, they ROCCC system relies on the IEEE 1076.3 VHDL libraries for implementing all the soft nodes [18]. This implementation requires synthesis tools to interpret the algorithmic parts of their design, and furthermore; it does not support division and needs an extraneous FSM (Finite State Machines) to manage flow control.

MATCH (MATlab Compiler for Heterogeneous) [7] computing system uses SUIF and MATLB M-File with compiler directives to translate an architecture into RTL VHDL for FPGA implementation. This process requires the hardware designer to interpreter an application’s architecture. The M-File is translated into SUIF IR and a MATLAB AST (Abstract Syntax Tree) is created. Parallel processing techniques are then performed on this AST to analyze dependences, loops exploits, and memory usage. Once complete, a synthesizable VHDL hardware circuit is produced that utilizes finite state machine controllers and algorithmic operators to describe the functions from the Matlab code [40]. The VHDL produced uses ”process statements”

and assumes that all variables are stored in localized memory. No automated design space exploration is performed.

Nayak et al. [54] developed an estimator for the MATCH system that could predict area usage within 16% and circuit delay within 13% for a Xilinx XC4010 FPGA. These numbers were verified though Synplify [69] and XACT [81], logic synthesis tools from Synplicity and Xilinx. The estimator uses a "precision and error analysis algorithm" to determine the number of bits required to represent either integer or floating point variables. This requires them to estimate all the IP cores used and expand those cores based upon the interpretation of the input Matlab code. These estimates, while close to synthesis's predicted values, are off when compared to the final place and routed values. Using these estimations, they performed and automated design space exploration that utilizes the loop unrolling compiler directive for the MATCH system to create designs.

Using the MATCH system, AccelChip created a product called AccelChip DSP design environment, an automated platform for taking Matlab M-Files to hardware synthesis [6]. It was capable of using AccelChip's own hardware components or simple Matlab arithmetic to create designs. These designs were then implemented on FPGA hardware through the MATCH system. In early 2006, AccelChip was bought by Xilinx Corp. who now provides a tool called AccelDSP™ [79].

2.3.2 Non-SUIF HLL Compilers

ASC (A Stream Compiler) is a FPGA hardware compiler that takes conventional C++ programs, written with compiler directives, and creates FPGA netlists. The architecture style and timing information is conveyed at the time-of-design so

that the compiler can explore different levels of design space optimizations. These optimizations can affect the implementation style, memory use, and data path at a structural, arithmetic, and gate level. ASC uses design tradeoffs like throughput, latency, or area to create stream based architectures. To do this it utilizes PAM-Blox II which generates modules for the arithmetic specified by the input C++ code and can handle integer and floating point operations. PAM-Blox II is tightly coupled to a specific FPGA hardware since it creates FPGA netlists. ASC allows the user to perform an iterative design space exploration through re-implementing and directing the compiler to optimize for a different design strategy.

SPARK [39] is a C to synthesizable RTL VHDL compiler that takes behavioral ANSI-C code and utilizes parallel processing techniques to schedule design graphs using speculative code motions and loop transformations. SPARK uses heuristics from scheduling loop transforming and common sub expression elimination (CSE) to generate an optimized dataflow graph. It then has an interconnect-minimizing resource binder and outputs the controller architecture that can then be placed into an FPGA or mapped to ASIC technology. The system creates a hardware architecture that is common to a von Neumann style that has an optimized interconnect for a given application. Any design space exploration requires the input code to be manually rewritten and interpreted by the compiler iteratively.

The Cameron Project [23] developed SA-C (Single Assignment C) [42] as a HLL variant of the C programming language to help exploit the ability to program hardware circuits from a high-level software language. This created a higher level of abstraction from an application's actual hardware implementation. The focus of SA-C is to create circuits on FPGAs for image processing applications. It restricts

operators, removes pointers, and bands recursive function calling in order to prevent programmers from applying a von Neumann architecture style that often does not map well on FPGAs. The variable types must be declared, dynamic arrays can be used, and at program time compiler directives are set to dictate how to organize the hardware structure [53].

A compiler [25] was developed to translate Cameron's SA-C to behavioral VHDL through the use of dataflow graphs [62]. This compiler uses a series of parallel processing techniques like loop exploits, data analysis, and pipelining to create the dataflow graph representation as it parses the SA-C input code. These techniques are similar to those used in the SUIF systems but this compiler accomplishes them without the use of SUIF. Each node in the dataflow graph can be classified as either a generator reduction, data transfer mechanism, arithmetic logic operator, or data buffer. The arithmetic logic operator nodes are represented as combinatorial logic circuits. The generator reduction nodes are implementations of loop operators. These loop operators require a sequential multi-clock controller with one or more finite state machines to coordinate their operation. The data transfer mechanism nodes move data into or out of the data buffers. All these nodes operate on a data ready style where they "fire" when all data tokens have arrived. The generated output VHDL code then implements a heterogeneous computing architecture that represents the interpreted dataflow graph dictated by the input SA-C code.

An estimation technique [46] was also developed to use dataflow graphs created by the Cameron compiler to predict FPGA area usage within a 5% margin. This technique did not take into account any memory usage or controller overhead. Its approach uses the internal dataflow representation of the compiler and employs a

curve fitting technique to approximate the area for each node. A general area formula for each node type was generated from synthesis data and a regression analysis was performed to obtain a coefficient values. These values are used at compile-time to approximate the area usage. This estimation technique does not have the ability to estimate low-level structure timing form the dataflow representation, only area is estimated.

In the Cameron system (SA-C, compiler, and estimation technique) no automated design space exploration is performed, furthermore when writing the SA-C code the designer is required to interpret the degree of parallelism to be used for the architecture. Only simple automated pipelining is performed, this is accomplished by placing registers in long combinatorial paths to reduce clock delay.

It has been shown [36] that translating the SA-C code to VHDL produces designs that are twice as slow then those created manually, but decreases the time-of-design by factor of ten. Even though the number of generated lines of VHDL code is about two to ten times greater then a manual approach it is thought that this approach easies the burden of "code maintenance" as the process of updating a design only requires the automated compiler to regenerate the VHDL. In a manual approach the entire design process of VHDL and architecture design needs to be repeated. SA-C succeeded in reducing the amount of hardware knowledge required by a designer, but requires them to be experts in interpretation styles that depict architectures through loop manipulations, data types, and high level tradeoffs.

Mentor and Celoxia both have compilers that will take an ANSI C/C++, Handel-C, or SystemC application and compile then directly to hardware. Catapult C [50] from Mentor Graphics is a compiler that offers a path from an abstract C-style

specification to a RTL hardware implementation. Catapult C synthesis uses ANSI C++ written for the compiler and creates a synthesis level netlist for a design. Celoxica compiler, called DK design suite, translates Handel-C, SystemC, and/or ANSIC C to a synthesized netlist while performing similar tasks compared to the Catapult C system. DK Design Suite uses C-based modules to develop different systems and can accelerate simulations of hardware/software co-verification.

In a review of the Catapult C tools [8], it was stated that architectural C synthesis designers still must specify how the data is transferred to and from a design. The main focus for the tool is accelerating a translation of C through synthesis directives while interpreting a structure. To do this it is employs loop transformations, data scheduling, and variable/array mapping. The compiler produces usable RTL hardware that is acceptable in terms of area and overall performance. Since the process is automated, the time it takes to compile from algorithm to gates is greatly reduced and prone to less error when compared to a manual process. Unfortunately the compiler still requires a designer to have a good understanding of the algorithm and architecture since they are dictating the structure to the compiler. While the compiler assists in automating the implementation process, the requirement of hardware and application knowledge prevents it from automating a design space exploration. The system is aimed at dataflow applications with minimal control issues and the designer needs thorough training to lean the optimization techniques for Catapult C style coding.

Matlab offers a hardware creation tool called HDL Coder [70] that works with their Simulink® platform. Simulink® [71] allows a user to graphically simulate DSP applications. HDL Coder generates VHDL code from Simulink® designs with

some intervention from the user. It uses pre-made blocks connected together as they correspond to components in a FPGA implementation. As the designer connects the Simulink® blocks, a data path is created and information like bit width and storage must be defined. HDL Coder allows certain IP cores to be ported from VHDL to allow the simulation of these core through ModelSim (a simulation environment for VHDL and Verilog designs) [34]. HDL Coder is an attempt to take a high level program like Simulink® and accelerate the prototyping phase, and is not intended to be a design space exploration tool.

Many different systems are available that produce core components that accelerate the process of implementing signal processing designs in FPGAs. Altera® has developed MegaCore® [2] a set of intellectual property (IP) cores to assist in the development of high-performance algorithmic functions optimized for Altera® devices. Xilinx also offers similar functionality in their Core Generator [81] product. In addition, the SPIRAL project [21] provides behavioral Verilog IP generators for the Discrete Fourier Transform and Discrete Cosine Transform, as well as multiplierless filters. To generate these cores [59], SPIRAL uses their own proprietary language known as signal processing language (SPL), which was originally intended for creating software-based implementations for different general purpose processors like Intel, AMD, and SUN.

SPIRAL's research is focused on creating implementations of transform algorithms for general purpose computer hardware architectures [60]. They accomplished this by generating different interpretations at an algorithmic level in their SPL and then translate an optimal interoperation into a software implementation that is intended to be executed on von Neumann style processors. While SPIRAL

has expanded to generating FPGA hardware implementations, they only use their technique for generating different versions of an algorithm in order to find an implementation that is best suited for a particular hardware architecture.

2.4 Estimation Tools

The systems described in Section 2.3 are compilers that focus on translating a HLL description to hardware. Some performed estimations of their designs to assist in a primitive form of design space exploration, while others concentrated on creating specialized hardware architectures. The systems and research in this section focus only on estimating and analyzing possible hardware implementations.

Ptolemy [44] is a system that is targeted towards modeling, simulation, and prototyping of heterogeneous signal processing systems. It assists in the prototyping of DSP systems by synthesizing assembly code for programmable DSP cores and simulating hardware through computational models. Ptolemy implements a coordination framework that is not restricted to any particular architecture, this frame work consist of basic objects called blocks that communicate through interfaces called "portholes". These blocks can then be assembled to form objects such as stars, galaxies (which contains blocks and/or stars), the objects when assembled together form a Universe which represents the actual application. Ptolemy uses SDF (synchronous dataflow), CT (continuous time), DT (discrete time), DE (discrete-events), CI (component interaction), CSP (communicating sequential processes), FSM (finite state machine), and others. to simulate the different models of computation. This coordination framework can be used to perform manual design space exploration on concurrent architectures with hardware/software partitioning

analysis capabilities.

Design Trotter [52] is a tool that aides a designer in converging on a hardware architecture and application mapping by estimating area and delay tradeoffs for applications written in a C-style code with restrictions. The tool performs hardware estimations [10] using structural exploration and then physical mapping estimation. The architectures created during the structural exploration revolve around a von Neumann style and expose the tradeoffs by varying the number of computation and memory units. The design space exploration is performed by exploiting a degree of parallelism in the number of control steps, execution units, and memory accesses and then schedules an H/CDFG (Hierarchical Control and Data Flow Graph) to a particular architecture. Estimations are then performed to accurately gauge the performance of a possible design through use of pre-estimated performance tables that were created manually through synthesis tools. The tool is able to achieve a 10% accuracy for delay estimations and 18% accuracy for area estimations.

Brandolese et al. [14] developed an estimator for analyzing SystemC hardware designs to predict area usage to within 25%. The estimator uses a three step approach; the first is to identify the form of the estimation process as algebraic and/or algorithmic. This is usually done manually by the designer. The second is to identify any variables in the design; this is done by both the system and the user. Then the third is to define the estimation parameters for each block. The blocks are broken down into five categories: finite state machines, operations, glue logic, multiplexers, and registers. Each category received an associated weight that was used to sum the number of flip-flops and look up tables used. The research identified that critical attention needs to be given to the glue logic between SystemC modules, where

accuracy errors in predicting this logic could be as high as 59%.

Yan et al. [82] quantifies function blocks in terms of software execution time and hardware area/delay to explore better hardware/software partitioning schemes within the Lycos [49] environment. Their method is to count all the functional units, multiplexers, registers, and control logic and then apply an estimated value of area and delay for each. Their system was able to predict area within 13% and delay within 8% for VLIW (Very Long Instruction Word) and CGRA (Coarse-Grained Reconfigurable Array) architectures.

Hamed et al. [41] perform area estimates on structural RTL VHDL for FPGAs. Their system uses a Boolean Minimizer called Espresso which is commonly used in VLSI design systems. Espresso can create a standard output netlist that is then used to estimate LUT usage. Their system is able to generate estimations on area of structures within 60%. They attributed this large percent variance to the methods of how logical synthesis tools (like Leonardo) share resources across multiple clock cycles.

Enzler et al. [26] use dataflow graphs to map a representation of a particular FPGA, and then characterize all the operations to get estimated values of area and performance (delay, throughput, and latency). Only a specific set of operators are supported and design space exploration is limited to three different styles (pipelined, replicated, and decomposed). The tool examines loop exploits to determine the decomposition or replication values and places the results into an equation for area, performance, and I/O usage. Routing effects and control overhead are not taken into account and memory usage is ignored. Furthermore it assumes a uniform partition for all pipeline stages and calculates the latency based on the number predicted pipe

stages.

Bjureus et al. [11] examine the execution of Matlab M-files at an instruction level to perform estimates and design space exploration. Their tool allows a user to trace through the input code in a hierarchal manor by generating a dataflow graph, annotating it with resource data, and creating an area/delay grid. The dataflow graph is then scheduled and the resources are allocated to the operations. By varying the number of input channels, bit widths, and the area/delay of the device, the tool produces estimates and performs design space exploration. The estimates produced are based on a component library that is within 10% of the actual area. This process was shown to take several hours, and only presents a style of architecture to be used as the starting point for a manually hardware implementation.

2.5 Summary

While most of the systems presented raised the level of hardware abstraction, they still require some knowledge of how to break apart the application. Most of them use SUIF to represent and perform parallel processing techniques while creating hardware implementations of HLL applications. SUIF incorporates techniques found in software compilers that include; loop exploitation, data manipulation, and the scheduling of application operations. These techniques assist in the translation of high level input code, written in a sequential manor, for use with structures that resemble von Neumann architectures.

Since most of these systems apply a von Neumann style of architecture in their translation from HLL to hardware implementation, they are required to focus on optimally scheduling the application instructions for von Neumann architectures

implemented within the FPGA. These systems hinge on their ability to implement efficient controllers, which has been shown to be crucial in creating effective von Neumann implementations. Research [24] has shown that a multiple memory system with a heterogeneous computing architecture produces better results then a von Neumann architecture for algorithmic base applications, still only a few of the systems focus on creating heterogeneous computing architectures from a HLL input.

Another issue identified is that most systems also use a behavioral coding style when generating the HDL of their designs. This style of coding forces synthesis tools to make decisions about the architecture that may or may not have been implied. This can result in designs that will perform worse then those created manually and is prone to errors when estimating performance and area. It is crucial to the design space exploration process that accurate estimates of performance are provided.

A better strategy is required to search for the different ways an algorithm can be broken up and determine which implementation is best for a particular application. Only a few these systems described in this chapter performed automated design space exploration, and when it was performed they either created a small set of designs based on loop exploits or, in some cases, just a pipelined version of the previous design.

3. Creating a Comprehensive Hardware Design Search Space

In this chapter we describe our approach to automatically generating optimized and correct HDL designs for complex applications. This approach involves generating a comprehensive set of equivalent mathematical representations of the application via a novel declarative programming language that avoids a number of difficulties associated with the imperative languages used by previous approaches. By using the theory of Catalan numbers and compositions of n we show that the number of possible equivalent expressions is bounded. We also give examples of how the approach works and contrast the approach with previous attempts to automate this process.

3.1 Introduction

The automated systems presented previously in Chapter 2 use imperative programming languages to describe an application and require the designer to define a hardware structure for each implementation. This hinders the ability of those systems to perform a comprehensive design space exploration. The approach taken in this research is to generate an exhaustive space of equivalent mathematical expressions representing an application using a declarative programming language, and then to translate those expressions into unique heterogeneous computing architecture designs. These designs are expressed in a structural RTL HDL format guarantying a high level of accuracy when estimating performance criteria (area, delay, & latency, etc.). Once these hardware designs are created and performance

estimations generated, a comprehensive hardware design search space is compiled. This hardware design search space can then be used to perform design space exploration, the process of examining different designs and evaluating their performance criteria to find an optimal hardware implementation for a particular application.

Figure 3.1 represents the three-stage process of this approach. The first is to represent a high-level language (HLL) application as an exhaustive set of equivalent mathematical expressions called the *equation space*. The second stage is to map each of the expressions in the equation space to heterogeneous hardware designs that represent the dataflow structure of each expression. Performance criteria are estimated and placed into a table comprising the hardware design search space. The third translates each design into a VHDL file written in a structural RTL style of coding to accurately represent the intended hardware implementation of the HLL application.

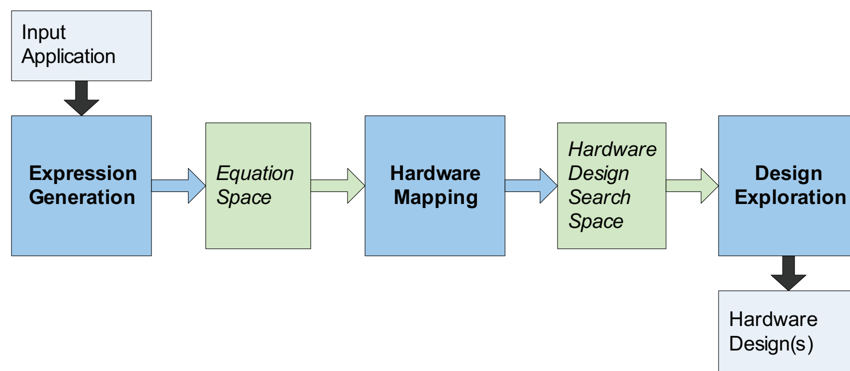

Figure 3.1: Operational flow of a HLL application to hardware compiler.

Section 3.2 briefly reviews imperative and declarative programming languages and describes how the design process of the previous systems was hindered by using imperative languages to describe their input HLL application when creating hardware implementations. Section 3.3 describes a declarative programming language syntax and the process of mapping the language to hardware designs, and gives two examples depicting different structural hardware designs using a FIR filter as a representative application. Section 3.4 discusses the benefits of structural RTL coding and a brief example of how area estimates vary between synthesis estimates and actually placed designs for a behavioral VHDL example.

3.2 Programming Paradigms

Imperative programming languages, sometimes called procedural programming languages, are used by all of the previous HLL-to-HDL systems to describe an input application. Imperative programming describes "how" an application is to be executed through a set of commands or procedures on a von Neumann style processor. As these programs execute they change the state of the processor through the use of an implied control. This control is accomplished by using structural programming elements that have entry and exit points through which state control is passed. These elements are arranged in a hierarchal structure that describes both the control and data flow for an application.

Examples of implied control elements are concatenation and flow control statements. Concatenation statements are either arithmetic operations or assignments that place results from operations into local registers and/or system wide memory. The flow control statements determine what a processor will execute next in a pro-

gram; these are selection statements such as conditional branching (`If-Then-Else`), repetition statements found in loops (`For-Do-While`), or subroutine statements performed in unconditional branching (function calls). All of these elements (concatenation and flow control) assist the programmer in writing applications that are characteristic of von Neumann style processors.

The present research takes a different approach by using a declarative programming language to describe "what" the application is rather then "how" the application should be implemented. Declarative programming languages accomplish this by describing an application as either a logical or functional relationship of programming elements. Backus [5] discussed how most imperative programming languages are bound to von Neumann architectures and called these von Neumann languages. He created his own declarative language that was not based on the von Neumann architecture, called "function-level functional programming language," which supports the description of an application in an algebraic style without using variables. This research applies this same concept, but uses a set of basic mathematical operators to describe the tasks of an application in an equation format.

3.2.1 Imperative and Declarative FIR design example

An example of how a declarative programming language can be used to describe an application is shown here by comparing two different imperative ANSI C descriptions and their corresponding declarative descriptions for a Finite Impulse Response (FIR) [58] filter. The FIR filter is described as

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k), \tag{3.1}$$

where

$$h(n) = \begin{cases} b_n, & 0 \leq k \leq M-1 \\ 0, & \text{otherwise} \end{cases} \quad (3.2)$$

where M is the length (number of taps) of the filter. For this example we are considering an eight tap filter where $M = 8$.

Figures 3.2 and 3.3 show two different ANSI C descriptions for Equation 3.1, called `Filter_1` and `Filter_2`. Each requires an operator to pass the correct filter coefficients and data values required by each step of the filter operation. `Filter_1` uses a loop statement to perform eight sequential multiplications and additions. The variable *ArrayLength* is used as a designator to stop and then exit the iterative loop function. As a von Neumann language this implementation is useful since it maximizes code reuse by providing other functions with the capability to use this code to perform filters of different lengths. In contrast to the first example, `Filter_2` uses eight sequential concatenation elements of multiply and store, then sums the results together with seven calls of addition to produce the final answer.

An equivalent declarative description of `Filter_1` is shown in Figure 3.4. It represents the same interpretation of the FIR Filter, dictating an iterative number of *ArrayLength* additions and multiplications, but uses a predefined declarative function called `$` which specifies a summation over the range of 0 to *ArrayLength* while performing the operation $h_k * x_k$. An equivalent declarative description of `Filter_2` is shown in Figure 3.5, which uses the same number of operations as the ANSI C example, but specifies them in one expression.

Both imperative ANSI C examples accomplish the additions and multiplications

```
//FIR filter
//Usage: FIR_Filter1(h, x, ArrayLength)
//Input:    h and x are vectors of length ArrayLength.
//  h is the coefficients of the filter
//  x is the data
//Output: y = summation of (h(k)*x(k)) for k = 0 to ArrayLength
int FIR_Filter1(int h[ArrayLength], int x[ArrayLength],
                int ArrayLength){
  //----Initialization
   int ptr      = 0;
   int y          = 0;
  //----Filtering
   while(ptr != Array_Len)
   {
      y = y + w[ptr]*x[ptr];
  //----Y will contain the filtering result
      ptr = ptr + 1;
   }
   return y; }
```

Figure 3.2: ANSI C description of a variable tap FIR filter.

necessary for a FIR filter, but each has a distinctly different style in accomplishing the tasks. While one has slightly more flexibility than the other they both will execute similarly on von Neumann architectures. When translating these imperative descriptions into hardware designs, the previous systems took an approach that was based on either von Neumann style architectures or heterogeneous computing architectures. In these examples, it is not clear how either translates to hardware since no architectural information has been conveyed.

The previous systems that translated imperative HLL applications to von Neumann architectures used an application's implied control flow and compiler direc-

```
//FIR filter
//Usage: FIR_Filter2(h, x)
//Input:    h and x are vectors of length 8.
//  h is the coefficients of the filter
//  x is the data
//Output: y = summation of (h(k)*x(k)) for k = 0 to 8
int FIR_Filter2(int h[8], int x[8]){
  //----Initialization
   int t0, t1, t2, t3, t4, t5, t6, t7 = 0;
   int y          = 0;
  //----Filtering
      t0 = h[0]*x[0];
      t1 = h[1]*x[1];
      t2 = h[2]*x[2];
      t3 = h[3]*x[3];
      t4 = h[4]*x[4];
      t5 = h[5]*x[5];
      t6 = h[6]*x[6];
      t7 = h[7]*x[7];
  //----Y will contain the filtering result
   y=t0+t1+2+t3+t4+t5+t6+t7;
   return y;}
```

Figure 3.3: ANSI C description of an eight tap FIR filter.

```
'Input: h and x are vectors of length ArrayLength.
'   h is the coefficients of the filter
'   x is the data
'Output: y = summation of (h(k)*x(k)) for k = 0 to ArrayLength
y = $ [ k=0~ArrayLength, h_k * x_k ]
```

Figure 3.4: Functional declarative description of a variable tap FIR filter.

```
'Input: h and x are vectors of length 8.
'   h is the coefficients of the filter
'   x is the data
'Output: y = summation of (h(k)*x(k)) for k = 0 to 8
  y = h0 * x0 + h1 * x1 + h2 * x2 + h3 * x3 +
        h4 * x4 + h5 * x5 + h6 * x6 + h7 * x7
```

Figure 3.5: Functional declarative description of an eight tap FIR filter.

tives. For the examples of `Filter_1` and `Filter_2`, a set of functional units would be implemented that vary in the number of multipliers, adders, and memory elements. The number of elements depends on how the input code is written, how the system interprets this code, and what hardware architecture is depicted by the designer.

For those systems that generate heterogeneous computing architectures from an imperative HLL application, a similar approach is used with the addition of a strict hardware interpretation. While compiler directives assist in specifying particular components, the application data flow is also used to interpret a heterogeneous computing architecture. For the hardware design of `Filter_1`, a loop is used to designate a sequential process. Only one multiplier and adder would be generated with a controller to operate them. Compiler directives could be used to dictate a different number of multipliers or adders but this would also require rewriting the loop function to depict a new controller. The hardware design of `Filter_2` would strictly follow the data flow of operations, connecting together a set of eight multipliers and seven adders. These two designs have drastically different hardware area usages and associated delays, and are just the beginning of a design space exploration.

In order to generate these and other different designs, the previous systems used parallel processing techniques to extract the control and data flow from the different versions of imperative HLL code. Since an architecture is not easily interpreted from imperative code, each system developed different techniques and compiler directives to convey a hardware architecture. To perform design space exploration, each system requires different descriptions of the input application to be created manually. New compiler directives, structural coding styles, and hardware knowledge are all used to depict the different designs. This leaves the number and style of different designs up to the creativity of the designers.

3.3 A Declarative Programming Language

This research uses its own declarative programming language to express an application in a way that is easily mapped to hardware structures and can be rewritten to represent different structures. Basic mathematical operators were chosen since they support a wide variety of processing tasks. These operators have a one-to-one mapping between themselves and hardware implementations, allowing the compiler to extract a structural heterogeneous computing architecture directly from the language without using compiler directives. By using mathematical operators, the application can also be rewritten as different mathematically equivalent expressions, each representing a unique heterogeneous computing architecture.

3.3.1 Language Syntax

The syntax for this language mimics mathematical equation descriptions by stating on the left hand side a static name used to specify the expression, and on the

right hand side the description of operations to be performed. For this language, a required (” ”) character is used to separate the syntax of commands which is shown in Table 3.1. An application being described by this language requires a specific name for each equation, and a description of the necessary operations. Examples of this language were shown in Section 3.2.1 Figures 3.4 and 3.5 in the declarative FIR design example.

Table 3.1: Syntax for a functional declarative programming language.

Command	Usage	Description
$name = f$	$y = f$	*name* of function f
( )	(f)	parentheses denote procedural order of the binary operation f
$*$	$a * b$	multiplication of a and b
%	a % b	a divided by b
$+$	$a + b$	addition of b to a
$-$	$a - b$	subtraction of b from a
$\$[name{=}range,\ f_{name}]$	$\$[\mathrm{i}{=}\ m{\sim}n,\ f_i]$	$\sum_{i=m}^{n} f_i$ where f_i is a binary operation iterated by *name*
$\#[name{=}range,\ f_{name}]$	$\#[\mathrm{i}{=}\ m{\sim}n,\ f_i]$	$\prod_{i=m}^{n} f_i$ where f_i is a binary operation iterated by *name*

Since one of the goals of this language is to be easily translated into hardware devices, it is important to conceptualize how these operations can be implemented inside FPGAs. There are many different implementation strategies, but all require the use of FPGA elements called functional logic blocks. In the effort to accelerate mathematical operations, FPGA vendors have designed specialized elements that

assist certain parts of addition and multiplication and placed them in the FPGA. These elements operate on two operands at a time, and for this reason we define all the calculations in this language to be binary operations.

We define binary operations as calculations that use up to two operands and generate a single result. Examples of binary operators are addition, subtraction, multiplication, and division. When declaring an equation that uses multiple binary operators to perform a calculation, a structure is implied that conveys the connectivity of these operations in hardware. Functions like $\sum$ and $\prod$ are considered iterated binary operators since they define a series of iterated calculations to be performed. When using these iterated binary operators in this language, a sequential behavior is implied that mimics a set of serial operations to be done in hardware. To determine how these operators are connected, data flow graphs are created that represent each expression for a given equation.

3.3.2 Data Flow Analysis

Data flow is defined as the movement of data as it passes through a computing architecture. Data flow graphs, for this system, comprise connected nodes that have either one or two operands as input and only one output. A data flow graph comprising such elements is called a binary tree graph, and represents the flow of data from the input of an expression to its calculated output.

To extract the data flow from this language, this research follows a mathematical order of operations specified by the input expression. The order of operations defines the sequence in which the calculations are evaluated. The order in which they are evaluated starts from left to right with:

1. Parenthesizes,
2. Functions,
3. Multiplications and Divisions,
4. Additions and Subtractions.

Using this order ensures that the data flow graphs that are created accurately represent the expression dictated by our declarative language.

The structure of each data flow graph generated is used to create the hardware structure representing the expression. Each node is linked to a hardware component used to depict the operation of the node in the FPGA. The components are wired together to represent the flow of data into and out of each node. This wiring together becomes the structure inside the FPGA that represents a heterogeneous computing architecture. Data flows from input to output of a node and on to the next node. This cascade of data flows down through all the nodes, updating each as new data is presented. Once all the nodes are stable, the calculation is complete. Parallel operations occur naturally since the nature of the process of constructing the hardware allows for maximum parallelism of the data flow.

Two examples of data flow graphs were generated from the declarative descriptions in Section 3.2.1. Each graph in Figures 3.6 and 3.7 represents the operations and their order for the descriptions in Figures 3.4 and 3.5, respectively. These graphs represent the data flow starting at the top with the input data and moving down to the calculated result at the bottom of the graph. The first expression in Figure 3.6 uses the function `$` to specify a sequence, and in this case *ArrayLength*

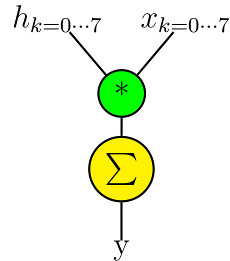

Figure 3.6: Data flow graph for the serial eight tap FIR filter example.

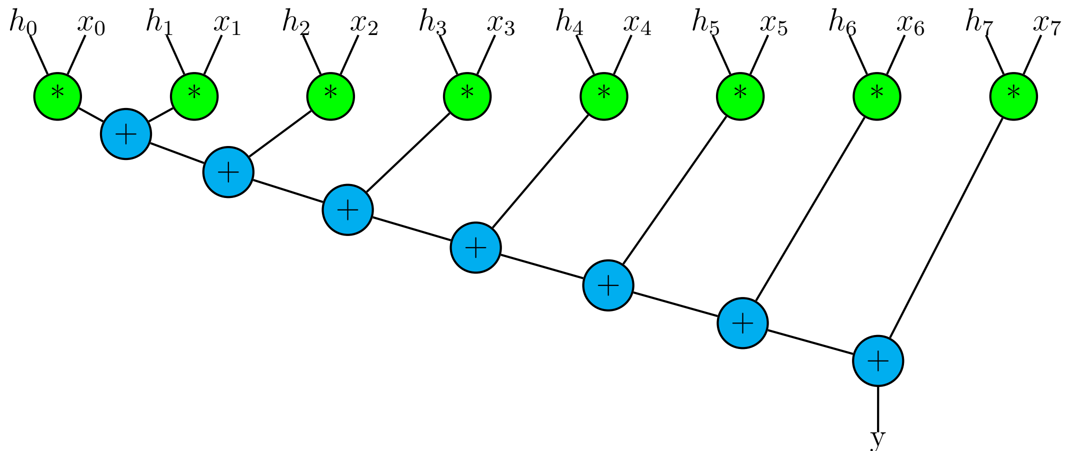

Figure 3.7: Data flow graph for the parallel eight tap FIR filter example.

is set to the value 8. When this design is mapped to hardware, it will use a single multiplier and a summation component to implement this expression. In the second design in Figure 3.7 the eight *multiplies* are independent of one another, and thus can be calculated in parallel. In hardware there would be eight *multipliers* wired to a set of seven *adders* that calculate a result in series. Both of these designs are equivalent representations of the FIR filter but each requires a different

amount of FPGA resources to implement the structure, and each will have different performance characteristics.

3.3.3 Equation Rewriting

By automating the processes of creating different equivalent equations, this approach can avoid the necessity of requiring the designer to depict different hardware structures. In order to create these and other designs, our approach uses mathematical equation rewriting techniques to generate equivalent expressions for the same equation. By adjusting the number or order of operations, different hardware structures are interpreted that become part of the hardware design search space. Since there are an infinite number of ways an equation can be re-written, we apply simple mathematical properties as restrictions or rules to create a bound on the number of possible expressions generated. To do this we apply four rules that assist in limiting the number of generated equations, creating different orders of operations, and varying the number of required operations for an expression.

The first rule is to disallow the use of the properties of identity and inverse. Additive and multiplicative identities are defined as $a + 0 = a$ and $a * 1 = a$, and the additive and multiplicative inverse are defined as $a + (-a) = 0$ and $a * \frac{1}{a} = 1$. If these identities were allowed, operations that only add to the number of required calculations would occur. Since the data flow graph of each expression is directly mapped to hardware, adding these extra calculations would requires more components to be used in the design while returning no operational benefit, and therefore their use will not be allowed in rewriting an equation.

The second rule is to allow the use of the algebraic properties of commutative

and associative laws in rewriting equations. To do this, the binary operations of subtraction and division must be defined as $a - b = a + (-b)$ and $\frac{a}{b} = a\left(\frac{1}{b}\right)$.

Commutative law : $a + b = b + a$ for addition and $a * b = b * a$ for multiplication.

Associative law : $a + (b + c) = (a + b) + c$ for addition and $a * (b * c) = (a * b) * c$ for multiplication.

These laws enable the use of bracketing-like operations and thus assist in creating different data flows that modify the order of operations for an expression.

The third rule is to imply operator precedence, which specifies a particular sequence for operations to be performed. In this case, we use parentheses to explicitly dictate precedence for like operators. All operations are binary in execution, thus we group like operations into binary bracketing. Binary bracketing [77] is the bracketing of at most a single binary operation. The number of unique bracketed forms that can be generated from n binary operations is quantified by the theory of Catalan numbers [66] C_n.

Theorem 1 (Catalan Numbers)

The formula for Catalan numbers is given as

$$C_n \equiv \frac{1}{n+1}\binom{2n}{n} \tag{3.3}$$

$$= \frac{(2n)!}{(n+1)!n!}, \tag{3.4}$$

where $\binom{2n}{n}$ *is the binomial coefficient.*

Example: The expression $y = a + b + c + d$ uses 3 binary operations of addition. Calculating the Catalan number of C_3 with equation 3.4 we find that there can be

only 5 unique binary bracketed expressions. These are

$$\begin{aligned} y &= ((a+b)+c)+d, \\ &= (a+(b+c))+d, \\ &= a+((b+c)+d), \\ &= a+(b+(c+d))), \\ &= (a+b)+(c+d). \end{aligned}$$

The fourth rule is to adjust the limits of a sum or product in order to create different sets of sequential structures. Adjusting a summation limit can be defined as $\sum_{i=a}^{c} f(i) = \sum_{i=a}^{b} f(i) + \sum_{i=b+1}^{c} f(i)$, where $a \leq b$ and $b \leq c$. The number of different combinations into which a summation or product limit can be broken is quantified by a combinatorial composition [78] of n, where n is the range of the summation or product limit.

Theorem 2 (Combinatorial Composition)
The composition of a positive integer n is the number of ways in which the number n can be summed using positive numbers. There are

$$2^{n-1} \tag{3.5}$$

compositions of n, where $n \geq 1$. To determine the composition of n into k parts, the equation

$$C_k(n) = \binom{n-1}{k-1} \tag{3.6}$$

$$= \frac{(n-1)!}{(k-1)!(n-k)!.} \tag{3.7}$$

can be used.

Example: The expression $z = \sum_{i=0}^{3}$ has a range of n to be equal to 4. Using Equation 3.5 for Combinatorial Composition we find that there are 8 different compositions for the number 4. These are

$$\begin{aligned} 4 &= 4, \\ &= 3+1, \\ &= 2+2, \\ &= 1+3, \\ &= 2+1+1, \\ &= 1+2+1, \\ &= 1+1+2, \\ &= 1+1+1+1. \end{aligned}$$

The summation can therefore be broken down into 8 partial sums by adjusting the summand limits as shown here

$$\begin{aligned} z &= \sum_{i=0}^{3}, \\ &= \sum_{i=0}^{2} + \sum_{i=3}^{3}, \\ &= \sum_{i=0}^{1} + \sum_{i=2}^{3}, \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{0} + \sum_{i=1}^{3}, \\
&= \sum_{i=0}^{1} + \sum_{i=2}^{2} + \sum_{i=3}^{3}, \\
&= \sum_{i=0}^{0} + \sum_{i=1}^{2} + \sum_{i=3}^{3}, \\
&= \sum_{i=0}^{0} + \sum_{i=1}^{1} + \sum_{i=2}^{3}, \\
&= \sum_{i=0}^{0} + \sum_{i=1}^{1} + \sum_{i=2}^{2} + \sum_{i=3}^{3}.
\end{aligned}
$$

The language described in this section enables a simple one-to-one mapping from equation space to hardware design search space by interpreting a hardware design from a mathematical equation description. Using these rules an equation space of equivalent expressions can be created such that each has a unique set of binary operators and/or bracketing that exhaustively explores all possible structures at a binary operator level. By using Catalan numbers and compositions of n we will show that the number of possible equivalent expressions is bounded. As the equation space is translated into hardware, a design space is created that can be explored to find an implementation that best meets the set of performance criteria.

3.4 Hardware Description Language Coding Style

Previous systems that perform design space exploration used a behavioral style of HDL coding to describe their components for arithmetic operations, control logic elements, and structure connectivity. This behavioral coding enables these systems to quickly produce designs without creating RTL components, but performance

estimates were prone to errors as high as 18% for area and as high as 20% for delay. Most of these inaccuracies occurred during the process of interpreting behavioral code into RTL FPGA representations during synthesis.

In an effort to generate accurate estimations, a structural description of various FPGA elements (structural RTL) connected together to implement a components should be used in the design. By using structural RTL, synthesis has less to interpret when translating the HDL to a hardware list and thus will lead to more accurate estimations for performance of a pre-synthesized design. To accomplish this, libraries of basic building block components must be created that perform each operation supported by the language. These components must be written as structural RTL, specifying the connectivity of FPGA elements and associated performance values. These libraries can then be used to map the operators in our declarative language syntax from the equation space to the design search space.

Using libraries of components also enables us to specify different implementations for each of the supported operations. Complicated structures can be specially built to accomplish functional blocks such as summation, product, or other high-level expressions. The only requirements is that these special components are described at a structural RTL and their area usage, delay, and latency be detailed.

To demonstrate the accuracy of structural RTL HDL coding, an example implementing the equation $Z = A * B + C$ in two different design styles was performed. The first style used was a behavioral coding, shown in Appendix B.1. This design relies on the IEEE 1076 libraries to specify hardware descriptions for the operations. The design allows the synthesis tool to implement the multiplier and adder in any style defined by the vendor. The second implementation was a structural RTL cod-

ing shown in Appendix B.2. It uses a uses a specific style of multiplier and adder that was implemented in specific FPGA elements. Both designs were targeted for a Xilinx VirtexII Pro 50, and synthesized with Precision RTL® [51] synthesis, then placed and routed with ISE Foundation™ [81].

Table 3.2 shows the estimated number of functional blocks from synthesis and the actual number of functional blocks used from place-and-route. The RTL design is larger due to the style of multiplier that was implemented, although the more important fact is that synthesis estimates showed a 6.7% variance compared to the actual usage for the behavioral design. The estimates for the structural RTL design were basically identical to the actual usage.

Table 3.2: Synthesis LUT estimates and PAR LUT usages for a behavioral and RTL design.

Designs	Synthesis (LUT estimate)	Place and Route (LUT usage)	% Error $(1 - \frac{estimated}{used})$
Behavioral	1067	1144	6.73%
RTL	2080	2079	-0.05%

Using a building block library provides the ability to create a larger hardware design search space by using different hardware implementations for the same basic operations. When performing design space exploration, using a structural RTL coding style leaves little to interpretation during synthesis, as was shown in the HDL coding example.

3.5 Summary

This chapter described our method for automating the process of creating a comprehensive hardware design search space in order to perform design space exploration to find an optimal hardware implementation defined by a set of performance criteria. We defined the syntax of our declarative language and discussed how it can be used to functionally describe HLL applications at a mathematical equation level intended for FPGA devices. This language provides the ability to perform an early discovery of possible design architectures through an automated equivalent equation rewriting process. Each equivalent equation can be mapped to a hardware design that is evaluated for area usage, delay, and latency, and the compilation of these designs becomes a hardware design search space that demonstrates performance tradeoffs for the different designs.

The contrast between imperative languages, which describe "how" to implement an application, and declarative languages, which describe "what" operations an application requires was discussed. Previous HLL to HDL systems require compiler directives to translate imperative descriptions of an application into a single FPGA hardware design, and any automated design space exploration was minimal if performed at all. This approach enables designers to perform a comprehensive design space exploration of mathematically based applications written in the proposed declarative language to find an implementation(s) that meets particular performance criteria. In Section 3.2 an example of how a declarative language can be used to describe the same style of execution as an imperative language for a mathematical equation was shown.

In Section 3.3, a declarative language syntax was proposed that easily maps from an equation description to a heterogeneous computing architecture without compiler directives thereby enabling the automated creation of an equation space. It accomplishes the translation process by depicting an application as a series of mathematical operations and extracting a hardware structure from the operational dataflow. The example in Section 3.2 showed two declarative descriptions for the same application, and in Section 3.3 we showed how the dataflow graphs for these declarative descriptions can be represented as unique heterogeneous computing hardware structures. To constrain and automate the process of generating the equation space, a set of mathematical equation rewriting techniques were proposed in Section 3.3 that place a boundary on the number of equivalent mathematical expressions that could be generated. The compilation of these equivalent expressions becomes the equation space that is mapped to unique heterogeneous computing designs.

When mapping each expression in the equation space to the hardware design search space accurate estimates of the performance characteristics must be generated. These estimates represent each design in the search space and require a high level of accuracy if precise design space exploration is to be performed. To ensure that the estimates are still valid after synthesis and place-and-route design processes, we proposed the use of a structural RTL coding style. The example in Section 3.4 showed how values estimated from synthesis were almost identical to actual usage values for a structural RTL coding example but varied for the popular behavioral style coding use be the previous research systems.

4. Demonstration Compiler System

In this chapter we describe the prototype system created to demonstrate the approach described in the previous chapter. The system automates the creation of a comprehensive hardware design search space that can be used to perform design space exploration for an optimal FPGA hardware implementation of an application with respect to a given set of performance criteria. It generates the hardware design search space by creating a comprehensive set of different equivalent mathematical representations of an application and then mapping them to unique heterogeneous hardware structures. During this process it estimates each design's performance characteristics and generates the corresponding structural RTL VHDL description.

4.1 Introduction

The prototype demonstration system we created uses Prolog, Visual Basic, and Excel to create a comprehensive hardware design search space. Figure 4.1 represents the operational flow of this system. Its key component is a hardware compiler written in Visual Basic and is pictured in the center of the figure. This hardware compiler maps expressions written in the declarative language described in Chapter 3 to unique designs using components provided in a building block library. It accomplishes this by translating the expressions into entries in a Hardware Data Flow Table (HDFT), which describes what components are used and how they are connected together. The resulting designs mimic heterogeneous hardware architecture designs and directly represent the implied data and control flow structure of

the input expression. The HDFT is also used by the performance estimator to characterize each design. The design generator which is pictured on the right side of the figure translates the HDFT into files that represent the application in hardware. As each expression is mapped and generated into a hardware design, they are placed into a spreadsheet that constitutes the hardware design search space.

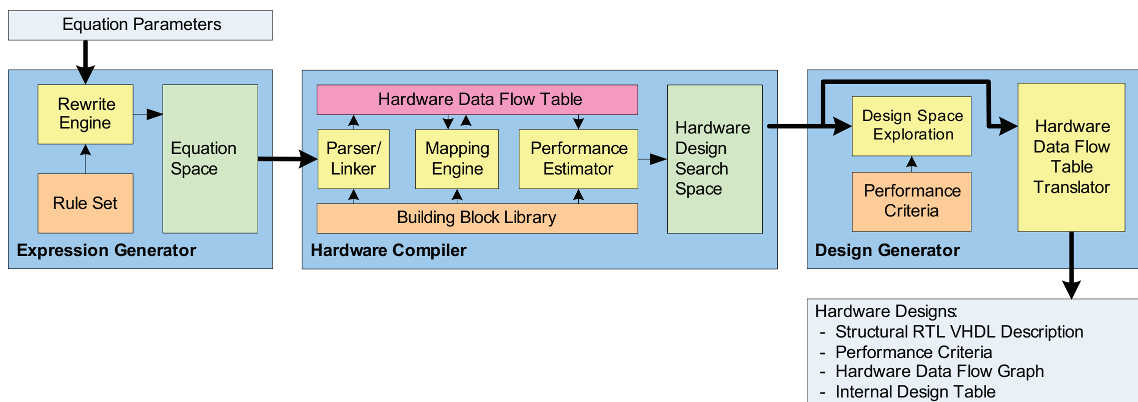

Figure 4.1: Operational flow of the prototype system for the creation and exploration of a comprehensive hardware design search space.

In order to automate the creation of the hardware design search space, an equation generator (pictured on the left side of the Figure 4.1) was written in Prolog. It represents the first stage of the approach and creates unique forms of N summed *multiplies*. It generates a text output file representing an equation space that contains all of the possible unique expressions using different binary bracketing and summation functions to imply the various hardware structures. This equation space is used by the hardware compiler to generate the hardware design search space.

The last component of the system performs design space exploration to select a hardware implementation that best matches the performance criteria specified by a particular application. This is performed by using Excel to manually examine and manipulate the spreadsheet containing the designs. Once a design is chosen, the output VHDL files are generated in a way that accurately represents the application in FPGA hardware. To generate these output files, the HDFT is used to directly translate a design to a structural RTL-style VHDL representation or hardware data flow diagram.

In the following Sections a detailed description of the key components of this demonstration system is given and the creation of a hardware design search space for the eight-tap FIR filter described in previous chapter by Equation 3.1 is described. In Section 4.2 we discuss how the expression generator creates the equation space and an example of the equation space for eight summed *multiplications* is shown. In Section 4.3 the process through which an equation space is mapped into the hardware design search space is described and a brief description of the current building block library is presented. Data flow diagrams and tables that depict how one expression from the example equation space is mapped to an estimated hardware design are shown. Section 4.4 discusses this final component of the approach, the design generator. This section also reviews a brief sample of the hardware design search space for this example and describes how the output files are generated.

4.2 Expression Generator

The purpose of the expression generator is to create an equation space of equivalent expressions for a given input equation. The equation space is set of expressions

that vary in their implied order and number of operations that calculates the same input equation. The flow of this block is shown in Figure 4.2, where an input equation length and a set of rules enables a rewrite engine to generate an equation space.

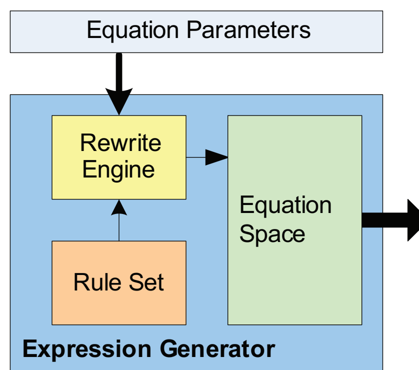

Figure 4.2: Block Diagram of the Expression Generator.

This implementation of this part of the approach uses Prolog, a logical declarative programming language, to generate the equation space with a set of rules that describe logic functions. The functions generate the equations in the style of N summed *multiplies* of two-input elements. It generates three different forms using binary bracketing. The first is a parallel hardware structure ($A_0 * B_0 + A_1 * B_1 + \cdots + A_{N-1} * B_{N-1}$), the second is summation to imply serial hardware structure ($\sum_{i=0}^{N-1} A_i * B_i$), and the third is a hybrid of both forms, ($\sum_{i=0}^{M-1} A_i * B_i + A_M * B_M + \cdots + A_{N-1} * B_{N-1}$, where $0 < M < N$).

The parallel forms of the expressions imply separate hardware components for

each operation to be performed in terms of `*` and `+`. By using the `*` and `+` operators with different bracketing forms, unique hardware structures are interpreted from the order of implied operations. This order is consider the temporal order for the execution of data for the expression. It was discovered by an examination of the equation space that occasionally several unique bracketing forms in the equation space would result in equivalent structures when mapped to the hardware design search space. The different implied hardware structures for the equation $Y = A + B + C + D$ from the example in Chapter 3 can be seen in Figure 4.3. The binary bracketed equivalent expressions are also repeated here for convenience:

$$\begin{aligned} Y_1 &= ((A+B)+C)+D, \\ Y_2 &= A+(B+(C+D))), \\ Y_3 &= (A+(B+C))+D, \\ Y_4 &= A+((B+C)+D), \\ Y_5 &= (A+B)+(C+D). \end{aligned}$$

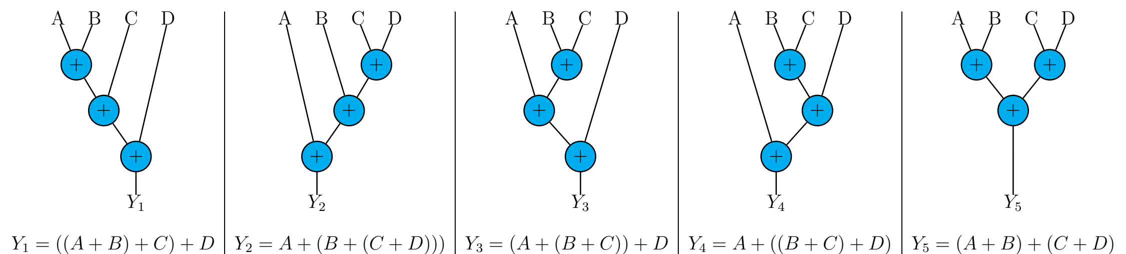

Figure 4.3: Data flow graphs for the expressions y_1 to y_5

By keeping the implied temporal order and rearranging the input names (A, B, C, and D) in each of the data flow graphs, two *unique* hardware data flow structures were discovered for the four element binary bracketed expression. These two structures are shown in Figure 4.4. The first structure is represented by the expressions Y_1 through Y_4 and the second structure is represented by expression Y_5.

The number of unique binary bracketed expressions is quantified by the theory of Catalan Numbers, which shows that it grows significantly as the number of like operations increases. To limit the duplicated hardware structures that are implied by different expression, we added another rule to the existing set described in Chapter 3. During generation, a process was inserted in the Prolog program that does not allow the creation of more then one unique hardware structure for each parallel form of the expressions of N binary bracketed like operations. This decreases the number of identical hardware structures in the hardware design search space.

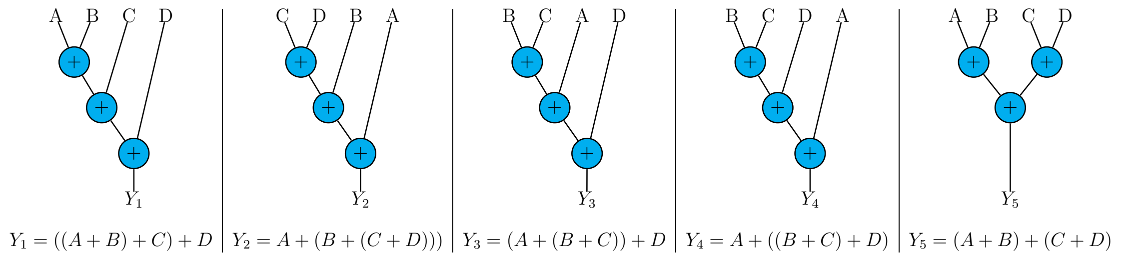

Figure 4.4: Rearranged inputs of the data flow graphs for the expressions y_1 to y_5

The summation function, denoted by the use of `$`, is used to generate the second form of the expressions called the serial form. These expressions imply an iterative

sequence of summed calculations of the operation inside the brackets ([ ]) of the function. In this case, the * operator is used to convey multiplication of two operands (A_i&B_i) inside the sum function seen here as $ [i=0∼8 $A_i * B_i$].

Using the $ function requires more than just an adder to sum over a range; it also implies the use of a comparator and counter to track the number of iterations. When comparing the + operation to add two operands together with the $ function to do the same, the $ requires a greater number of FPGA elements. This implied a set of control components in the $ function that adds to the number of required FPGA elements not implied in the + operation. To maximize performance gain with respect to area, a requirement was added to the Prolog program to only generate $ functions that perform summations with a range of two or more.

The last form of the expressions generated is a hybrid that combines both the serial and parallel forms into one expression. The hybrid form expressions are created by varying the limits of summation functions and adding the remaining operands in a parallel form. The creation process for these forms starts by using one $ function with a range of two and then adding to it the remaining number of like operations in the different parallel form expressions. Structurally different hybrid expressions are crated by increasing the range of the $ function until the number of like operands has been exhausted. Another $ function is added and the process is repeated until all possible ranges of $ functions and remaining parallel form expressions are exhausted. To reduce the number of duplicated hardware designs we apply the same rule above from the parallel form when generating the hybrid form expressions.

Once all forms of the expressions are generated, the program places them into a file representing the equation space which is used by the hardware compiler to

create hardware designs. Figure 4.5 shows a few samples of the different expression for the equation space generated by the Prolog program for the eight-tap FIR filter example. It contains 54 unique expressions of the $N = 8$ summed operations of multiplication grouped into the parallel, serial, and hybrid forms.

```
% Parallel forms
Z1 = ( ( ( ( A0 * B0 + A1 * B1 ) + A2 * B2 ) + A3 * B3 ) +
     ( ( ( A4 * B4 + A5 * B5 ) + A6 * B6 ) + A7 * B7 ) )
Z2 = ( ( ( ( A0 * B0 + A1 * B1 ) + A2 * B2 ) + A3 * B3 ) +
     ( ( A4 * B4 + A5 * B5 ) + ( A6 * B6 + A7 * B7 ) ) )
Z3 = ( ( ( A0 * B0 + A1 * B1 ) + ( A2 * B2 + A3 * B3 ) ) +
     ( ( ( A4 * B4 + A5 * B5 ) + A6 * B6 ) + A7 * B7 ) )
Z4 = ( ( ( A0 * B0 + A1 * B1 ) + ( A2 * B2 + A3 * B3 ) ) +
     ( ( A4 * B4 + A5 * B5 ) + ( A6 * B6 + A7 * B7 ) ) )
...
% Serial forms
Z27 = ( $ [ i=0~3 Ai * Bi ] + $ [ i=4~7 Ai * Bi ] )
Z28 = ( $ [ i=0~3 Ai * Bi ] + ( $ [ i=4~5 Ai * Bi ] +
     $ [ i=6~7 Ai * Bi ] ) )
Z29 = ( $ [ i=0~2 Ai * Bi ] + ( $ [ i=3~5 Ai * Bi ] +
     $ [ i=6~7 Ai * Bi ] ) )
Z30 = ( $ [ i=0~1 Ai * Bi ] + ( $ [ i=2~3 Ai * Bi ] +
     ( $ [ i=4~5 Ai * Bi ] + $ [ i=6~7 Ai * Bi ] ) ) )
Z31 = $ [ i=0~7 Ai * Bi ]
...
% Hybrid (Combination of both)
Z51 = ( ( $ [ i=0~4 Ai * Bi ] + $ [ i=5~6 Ai * Bi ] ) + A7 * B7 )
Z52 = ( ( $ [ i=0~3 Ai * Bi ] + $ [ i=4~6 Ai * Bi ] ) + A7 * B7 )
Z53 = ( ( $ [ i=0~2 Ai * Bi ] + ( $ [ i=3~4 Ai * Bi ] +
     $ [ i=5~6 Ai * Bi ] ) ) + A7 * B7 )
Z54 = ( $ [ i=0~6 Ai * Bi ] + A7 * B7 )
%\end{Verbatim}
```

Figure 4.5: Sample of a generated equation space for eight summed operations of multiplication.

4.3 Hardware Compiler

The hardware compiler performs the task of translating a mathematical expression into a Hardware Data Flow Table (HDFT) that represents the expression as hardware components. The flow of this block is shown in Figure 4.6, where a building block library conveys the information required to translate the declarative syntax expressions into the hardware operators, functions, and implied flow control components of a design. The mapping engine uses the implied temporal order of the expression to create a hardware design flow and then the performance estimator generates the performance criteria of the design. The compilation of these estimates becomes the hardware design search space used later during design exploration.

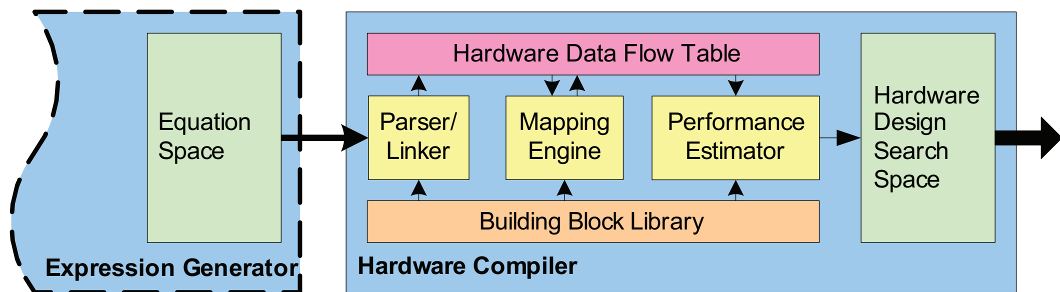

Figure 4.6: Block Diagram of the Hardware Compiler.

To map between the equation space and hardware design search space, a compiler was created in Visual Basic that iteratively translates each expression by performing three steps of parsing/linking, mapping, and estimating. First it parses each expression into an array of elements that are linked to the components of a building

block library. This parsed array is then converted into an intermediate data flow table retaining all specified temporal order for the expression. The table is then used by the second step to create both combinatorial and clocked designs for each expression. The intermediate data flow table is mapped into a final HDFT which contains nodes linked by specific hardware components that comprise the design. This table is used to generate output design files and to calculate the values placed into a spreadsheet for estimated area in terms of lookup tables (LUTS), delay in terms of nanoseconds (ns), and latency in terms of clock cycles (cc).

To perform the linking, mapping, and estimating steps of the compiler, a set of components were created to form the building block library. They were written using the structural RTL style of VHDL coding and their performance characteristics were estimated. The current library contains mathematical operations for unsigned addition, multiplication, and summation, as well as the hardware flow components for a two-input multiplexer, register, and comparator. Each component inside the library was built with Xilinx Virtex-II FPGA family elements. The library entry for each component specifies its declaration and area usage, delay, and latency. The library structure is read at the start of the compiler and placed into a memory structure to be used by both the parser/linker and mapping engine in the Visual Basic program.

A library entry for a 16-bit unsigned adder written as an array of smaller elements that use FPGA LUTs, XOR gates, and fast carry logic is shown in Figure 4.7. The VHDL code for this adder is also included in Appendix C.1. The entry contains a `Library Name` field used when generating a unique name for each data flow node. The `Library Symbol` field is used during parsing to specify what operators are

supported and what components can be linked to them. The `Delay` and `Area` fields are used when calculating the FPGA usage values. These values vary depending on the data width and complexity of the components.

To map to a component, the compiler requires each port to be categorized as either `Data` or `Control`. The direction of each port is specified by the column and has an associated width delimited by the "`|`" character which is easily seen in this example. The `Generic` column is used to specify what parameters are configurable, and in this library component the use of `N` determines width for the ports `A, B, & C`. By using this generic value for width, the same component can be reused in other designs without extensive rewriting.

<table>
<tr><th rowspan="2"></th><th rowspan="2"></th><th rowspan="2">Gen.</th><th colspan="2">Data</th><th colspan="2">Control</th><th rowspan="2">Component Declaration</th></tr>
<tr><th>In</th><th>Out</th><th>In</th><th>Out</th></tr>
<tr><td>Library Name:</td><td>Add</td><td>N</td><td>A | N</td><td>C | N</td><td></td><td></td><td>COMPONENT U_Add</td></tr>
<tr><td>Comp. Name:</td><td>U_Add</td><td></td><td>B | N</td><td></td><td></td><td></td><td>GENERIC (</td></tr>
<tr><td>Comp. Symbol:</td><td>+</td><td></td><td></td><td></td><td></td><td></td><td>N : natural := 16);</td></tr>
<tr><td>Delay (ns):</td><td>4.5</td><td></td><td></td><td></td><td></td><td></td><td>PORT (</td></tr>
<tr><td>Area (LUTS):</td><td>16</td><td></td><td></td><td></td><td></td><td></td><td>A : IN std_logic_vector (N-1 DOWNTO 0);</td></tr>
<tr><td>Init. Clock Cycle:</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td>B : IN std_logic_vector (N-1 DOWNTO 0);</td></tr>
<tr><td>Run Clock Cycle:</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td>C : OUT std_logic_vector (N-1 DOWNTO 0));</td></tr>
<tr><td>Color:</td><td></td><td></td><td></td><td></td><td></td><td></td><td>END COMPONENT;</td></tr>
</table>

Figure 4.7: Building Block Library Entry for a 16-bit Adder.

A condensed form of a building block library with a data width of 16 bits (`N`= 16) is shown in Table 4.1. The `Component Type` field classifies each symbol as operation (OPP), function (FUN), constant (CST), input (INP) or output (OTP). The types classified as OPP are directly mapped to hardware without requiring control flow

or special handling and is seen in the cases of `mult` and `add`. Components classified as FUN require special treatment when mapping since they require specific control flow sequences like `Mux` and `AdAc`. Inputs (INP) and outputs (OTP) are designators for the external ports of a design and are used when translating the hardware data flow table to the VHDL syntax.

Any components that have zero initial and running latency also do not contain a register and thus do not require a clock or reset control signals. If a component has a latency of one or more then a registered output is implied. The compiler takes this into account by attaching an appropriate clock and reset control signals as well as any other control flow components as necessary. Examples of this are the OPP `Reg` type that actually represents a hardware register requiring a clock signal and the FUN `AdAc` that requires both a clock signal and specific control flow components.

With the building block library loaded into memory, the Visual Basic program executes the parser/linker to translate an input expression and link it to building block library components. Once the parser is complete, the expression is converted from the input declarative expression to an intermediate data flow table representing the temporal order of the operational data flow. If there are any syntax errors or undefined operators, a function is executed that displays the point of failure in the input expression.

Table 4.2 displays the the intermediate data flow table for the expression `Z54 = ( $ [ i=0 6 Ai * Bi ] + A7 * B7 )` from the example in Figure 4.5. The Figure 4.8 is the graphical representation of the intermediate data flow table. It is easily seen from the graphical representation that the inputs $A_{i=0\sim6}$ and $B_{i=0\sim6}$ represent a range of iterated inputs to be multiplied by the node (`*`) and summed by the node

Table 4.1: A condensed table of a 16-bit data width building block library.

Lib. Name	Lib. Idx.	Comp. Symbol	Comp. Type	Delay (ns)	Area (LUTS)	Init. Latency (cc)	Run Latency (cc)
Add	1	+	OPP	4.5	16	0	0
Mult	2	*	OPP	9.00	240	0	0
Reg	3	Reg	OPP	1	0	1	1
Mux	4	Mux	FUN	2.5	16	0	0
AdAc	5	$	FUN	5.5	20	1	1
Const	6	Const	CST	0	16	0	0
Data In	7	UsrIn	INP	0	0	0	0
Data Out	8	UsrOut	OTP	0	0	0	0
Ctrl. In	9	CtrIn	INP	0	0	0	0
Ctrl. Out	10	CtrOut	OTP	0	0	0	0

(`$`). These are considered compressed nodes and will be expanded by the mapping engine. Looking at the intermediate data flow table, the `control` and `data` fields represent their respective flows into and out of each node and are used to create the structural wiring when translating the HDFT into the VHDL syntax. The `Type` and `Component Symbol` field information is provided by the building block library and the remaining information of `Node Name`, `Library Index`, and `Range` is used by the mapping engine later.

The mapping engine creates the implied control structure of the input expression by placing flow control nodes into the intermediate data flow table while creating the HDFT. To do this, the mapping engine first expands any compressed input nodes and then connects them by creating a tree of multiplexers. It generates two different

Table 4.2: Data flow table for Z54.

Table Idx.	Control In	Control Out	Data In	Data Out	Type	Comp. Sym.	Node Name	Lib. Idx.	Range
0	–	–	8	–	OTP	UsrOut	Z_{54}	8	–
1	–	–	–	5	INP	UsrIn	A_i	7	i=0∼6
2	–	–	–	5	INP	UsrIn	B_i	7	i=0∼6
3	–	–	–	7	INP	UsrIn	A_7	7	–
4	–	–	–	7	INP	UsrIn	B_7	7	–
5	–	–	1,2	6	OPP	$*$	–	2	–
6	–	–	5	8	FUN	$	–	5	i=0∼6
7	–	–	3,4	8	OPP	$*$	–	2	–
8	–	–	6,7	0	OPP	+	–	1	–

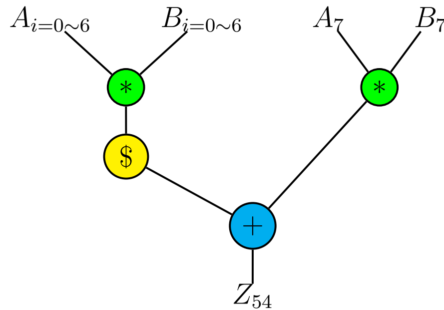

Figure 4.8: Data flow graph for Z54.

versions (combinatorial and clocked) of hardware designs during the creation of the HDFT. The first is the combinatorial design that only uses registers inferred by components and is not intended to be internally pipelined. The second is a clocked design that places registers in critical paths to decrease the combinatorial delay; this becomes an internally pipelined design.

When expanding the compressed input nodes, a flow control structure, a set of components, and a $ function in the branch are required. The compiler described here creates a multiplexer (mux) tree as the flow control structure, but any style of multiple-data-input/single-output structure could be used. For example a queue, a block memory, or a register set could be used as the control flow structure, or the entire structure could be left to a designer to implement. The mux tree generated here uses two-to-one mux components to enable the selection of any number of input nodes into a single output node. The output is connected to the operation(s) implied by the $ function. An example of this can be seen in Figure 4.9, which shows two mux trees for the expanded inputs of $A_{i=0\sim6}$ and $B_{i=0\sim6}$.

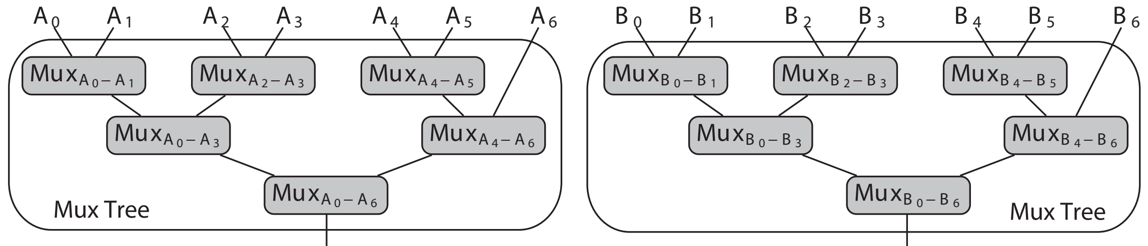

Figure 4.9: Examples of multiplexer trees for $A_{i=0\sim6}$ and $B_{i=0\sim6}$.

Since each node in the HDFT is bound to a hardware component in the building block library, the only procedures left to perform are to connect the implied control flow components, generate unique names for each node, and estimate design performance. These are all performed by the performance estimator. The unique names are generated from a combination of the library name and temporal order of each

operator. After the names are generated each entry in the HDFT is estimated for area, delay, and latency. For the combinational designs, the values are placed into a hardware design search space to be used later and the HDFT is handed off to generate the combinatorial VHDL design file. In the case of the clocked designs, the smallest clocked delay between nodes is used to insert registers in appropriate places in the longest combinatorial delay path. Registers are then placed in the remaining paths, creating a pipelined design that is re-estimated and handed off to generate the clocked VHDL design file.

When inserting the registers in clocked designs, pipeline balancing is performed by placing a register between nodes that are not in the same clock boundary. This is accomplished by traversing the HDFT and examining the inputs at each node. If they are all at the same clock cycle count in the running latency calculation, nothing is done. If they are not, a register is placed between the output of the source node and the input of destination node. This is repeated until all paths of the data flow table have been exhausted. Once complete, the HDFT now represents a pipelined design for the expression that can be re-estimated for area, delay, and latency and placed into the hardware design search space.

For the example of `Z54`, the hardware mapping engine has created two different designs, a combinatorial and clocked version. The graphical representation of the clocked version is shown in Figure 4.10 and the HDFT is shown in Table 4.3. The compiler has automatically expanded the compressed inputs A_i and B_i, generated the unique node names, and created two mux trees which are easily seen in Figure 4.10. A register was placed after $\texttt{Mult}_1$ to balance the clocked data boundary already implied in the $\texttt{AdAc}_0$ node, and a register was also placed after $\texttt{Add}_0$ to hold

the final calculated output of the design.

Table 4.3: Hardware Data flow table for Z54.

Table Idx.	Control In	Control Out	Data In	Data Out	Type	Comp. Sym.	Node Name	Lib. Idx.	Range
0	–	–	36	–	OTP	UsrOut	Z_{54}	8	–
1	Sel(2)	–	26,27	5	Auto	Mux	A_0A_6	4	–
2	Sel(2)	–	31,32	5	Auto	Mux	B_0B_6	4	–
3	–	–	–	7	INP	UsrIn	A_7	7	–
4	–	–	–	7	INP	UsrIn	B_7	7	–
5	–	–	1,2	6	OPP	$*$	Mult_0	2	–
6	Clk, Rst, Const1	Sel, EnOut	5	8	FUN	\$	AdAc_0	5	i=0∼6
7	–	–	3,4	37	OPP	$*$	Mult_1	2	–
8	–	–	6,37	0	OPP	$+$	Add_0	1	–
9	–	–	–	23	Auto	UsrIn	A_0	7	–
10	–	–	–	23	Auto	UsrIn	A_1	7	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
21	–	–	–	30	Auto	UsrIn	B_5	7	–
22	–	–	–	32	Auto	UsrIn	B_6	7	–
23	Sel(0)	–	9,10	26	Auto	Mux	A_0A_1	4	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
32	Sel(1)	–	30,22	2	Auto	Mux	B_4B_6	4	–
33	–	–	–	–	Auto	Const	$Const1$	6	–
34	–	–	–	–	Auto	CtrIn	Clk	9	–
35	–	–	–	–	Auto	CtrIn	Rst	9	–
36	Clk, Rst	–	8	0	Auto	Reg	Reg_0	1	–
37	Clk, Rst	–	7	8	Auto	Reg	Reg_1	1	–

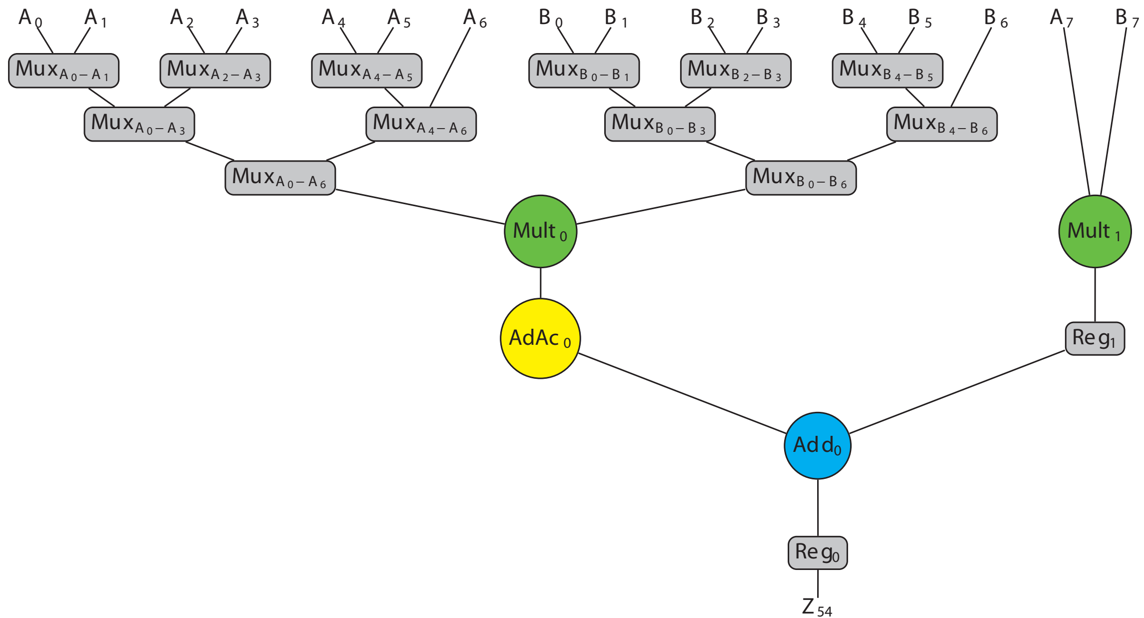

Figure 4.10: Hardware data flow graph for `Z54`.

When calculating the performance estimations, the values are broken into the three main categories of *area*, *latency*, and *delay* for a design. *Area* is calculated by multiplying the number of components by the amount of LUTs required by each component. *Latency* is defined as the number of clock cycles a design requires to calculate the output. Latency is estimated in two different ways, one for combinatorial designs and another for clocked designs. There are two subcategories for latency, the first is *initial* latency defined as the number of clock cycles it takes for the first valid output value to be calculated. The second subcategory is *steady-state* latency defied as the number of clock cycles a design requires to sustain consecutive valid output values. *Delay* is defined as the time in nanoseconds for a single clock

period. The manor in which *Delay* is determined also varies for either combinatorial or clocked designs. Delay has three subcategories, the first is *clock* delay defined as the time required for one clock cycle. The second subcategory is *initial* delay defined as the time required for the first valid output values to be calculated. The third subcategory is *steady-state* delay defined as the time a design requires to sustain consecutive valid output values.

To calculate *latency* and *delay* for the combinatorial designs we must take into account the fact that that these designs are not intended to be pipelined. This means that the *initial* and *steady-state* latencies and delays are equal and represent the number of clock cycles in the longest path. The *clock* delay is estimated as the longest time a registered set of operations requires in the design. The *clock* delay is propagated to all the nodes that require a clock period. These values take into account nodes that have latencies (clock cycles) greater then zero, and the clock period and number of clock cycles is used to calculate the delay of these nodes. The performance estimations for combinatorial designs are therefore area, latency represented by the longest path's clock cycle count, and delay values for the *clock*, *initial*, and *steady-state* delay categories of the design.

For clocked designs the *initial* and *steady-state* latencies can vary depending on the number of registers and stalled nodes, since pipelined design stages can requires more the one clock cycle to calculated a value. The *clock* delay is estimated as the longest delay between two registers and is calculated in the same manner as the combinatorial design. It is passed to all clocked components throughout the design as the common design clock period. By using both *clock delay* and *latency* (clock cycle count), the delay for the entire design in nanoseconds can be estimated

for both the *initial* and *steady-state* values. The performance estimations for the clocked designs are therefore area, *latency* represented by the number of clock cycles for both the *initial* and *steady-state* categories, as well as the value for the *clock*, *initial*, and *steady-state* delay.

In the final step a hardware design search space is generated that contains each declarative input expression mapped to both combinatorial and clocked hardware designs. The estimated performance values are the only criteria that represent each hardware design in the search space. These estimated values are critical to analyzing different design tradeoffs.

4.4 Design Generator

The design generator enables the manual exploration of the hardware design search space for a design(s) that best meets the performance criteria for an application. The flow of this component is shown in Figure 4.11. Each design in the hardware design space is classified by estimates that accurately portray a target FPGA area usage, design latency, and clock/design delay. As each design is placed into the hardware design search space, the design generator also translates the HDFT into a set of output files that can be used to examine the design or implement it in an FPGA.

After each design is mapped, this system places the estimated performance values in an Excel spreadsheet that was created at the start of the hardware compiler. This sheet is used to perform design space exploration and examine different performance tradeoffs. Figure 4.12 shows a sample of a hardware design search space for the FIR example used in this chapter. The green highlighted entries represent clocked designs

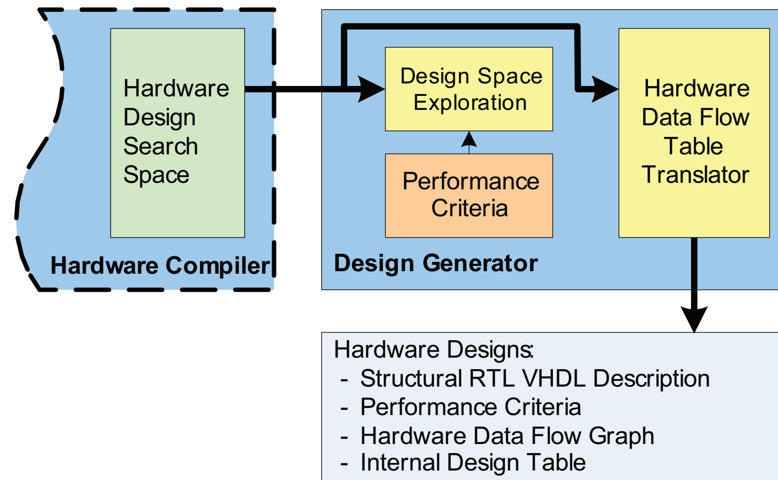

Figure 4.11: Block diagram of the Design Generator.

and the non-highlighted entries are combinatorial designs. The estimated values in this sheet are critical, since they are the only information representing the design's performance after post design tools have transformed the generated VHDL file into a FPGA image. Once a design is chosen, a resulting VHDL file can be used to finalize the FPGA implementation design process.

As the estimations are placed into the spreadsheet, the HDFT is used to generate output files. The design output generator creates data flow graph representations written in both XML and PostScript formats with delay and latency information for each node. The design output generator also exports the hardware data flow table as an Excel spreadsheet that can be used for future work. Finally the output VHDL design file is written using a structural RTL style of coding to ensure that synthesis tools will interpret the design as intended. Each node is defined by a separate building block library component and is "wired" together structurally. Data and

Designs	Area											Latency		Delay			Throughput		Power
	Add	Mult	Reg	Mux	AdAc	Const.	Data Input	Data Output	Control Input	Control Output	Total (LUT)	Init. (cc)	Run (cc)	Clock (ns)	Init. (ns)	Run (ns)	Data Rate In (MB/s)	Data Rate Out (MB/s)	Est. Power (mW)
Z1_PL0	7	8	0	0	0	0	16	1	0	0	2032	0	0	27.00	27.00	27.00	1185.19	74.07	172.12
Z1_PL1	7	8	13	0	0	0	16	1	2	0	2032	3	1	9.00	27.00	9.00	3555.56	222.22	516.35
Z2_PL0	7	8	0	0	0	0	16	1	0	0	2032	0	0	27.00	27.00	27.00	1185.19	74.07	172.12
Z2_PL1	7	8	12	0	0	0	16	1	2	0	2032	3	1	9.00	27.00	9.00	3555.56	222.22	516.35
Z3_PL0	7	8	0	0	0	0	16	1	0	0	2032	0	0	27.00	27.00	27.00	1185.19	74.07	172.12
Z3_PL1	7	8	12	0	0	0	16	1	2	0	2032	3	1	9.00	27.00	9.00	3555.56	222.22	516.35
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Z52_PL0	2	3	0	10	2	1	16	1	2	0	960	4	4	19.50	87.00	87.00	367.82	22.99	94.92
Z52_PL1	2	3	2	10	2	1	16	1	2	0	960	5	4	19.50	97.50	78.00	410.26	25.64	112.59
Z53_PL0	3	4	0	8	3	1	16	1	2	0	1204	3	3	19.50	72.00	72.00	444.44	27.78	124.01
Z53_PL1	3	4	2	8	3	1	16	1	2	0	1204	4	3	19.50	78.00	58.50	547.01	34.19	141.21
Z54_PL0	1	2	0	12	1	1	16	1	2	0	716	7	7	22.00	158.50	158.50	201.89	12.62	58.68
Z54_PL1	1	2	2	12	1	1	16	1	2	0	716	8	7	22.00	176.00	154.00	207.79	12.99	74.43

Figure 4.12: Hardware Design Search for eight summed operations of *multiplication*.

control signals (wires) are created in the VHDL syntax and are connected to the output of each node and to their respective inputs as dictated by the `data` and `control` fields of the HDFT.

`Z54_PL1` is the clocked design for the expression `Z54`. It was used as an example in Section 4.3 and the output VHDL file is included in Appendix C.2. A breakdown of the number of building block library components used is shown under the area column in Figure 4.12. The calculated estimation for area usage of this design is 716 LUTs. The clock delay for this design is 22 ns. This time represents the clock period, and it can be used to evaluate the initial latency of 8 clock cycles representing a delay of 176 ns for the first valid calculated output. After that time, every 7 clock cycles, or 154 ns, another calculated value is valid. This assumes that the design is given input data when required at the start of each calculation. This spreadsheet representation of the hardware design search space can also be used to calculate values for the design's required throughput categorizing both the required input

and expected output rates in terms of MB/s. Since this design uses a 16-bit data width the output throughput can be calculated as almost 13 MB/s and requires an input rate of almost 208 MB/s to sustain that output rate. These and other estimated performance characteristics will be discussed later in Chapter 5.

4.5 Summary

This Chapter described the demonstration prototype system that comprised Prolog, Visual Basic, and Excel components that creates a comprehensive hardware design search space. A Prolog front-end assists in automating the process by generating an equation space of unique expressions for N number of summed *multiplies* in three forms; parallel, serial, and hybrid. A configurable building block library definition was described and a method for configuring that library for different data widths and expanding it to include new components was shown using brief examples.

The Visual Basic hardware compiler that uses the equation space and building block libraries to parse/link expressions, map them into designs that are combinatorial or clocked, and to estimate the performance of those designs was discussed. The hardware compiler uses a parser to extract the temporal order of operations from each input expression and then link them to the components in the building block library. It maps them into a HDFT that represents the expression as either a combinatorial or clocked hardware design. The compiler estimates the performance values of area, clock delay, and design latency and places them into an Excel spreadsheet that represents the hardware design search space. This spreadsheet can be manipulated by a designer to search for an implementation of particular performance criteria for an application.

The importance of the HDFT as the intermediate representation used throughout the process was shown. The HDFT is crucial when estimating the performance characteristics as well as generating the corresponding output files. An HDFT-to-structural RTL VHDL translator was developed that creates the output design files in a manner that leaves little for interpretation by synthesis tools. Both graphical and table representations of the HDFT are also exported by the translator and can be used for future translation into other formats for functionality testing and algorithm verification.

An example of an eight-tap FIR filter equation 3.1 was used throughout the discussion to demonstrate the approach. A sample of the equation space generated by the Prolog program, and hardware data flow tables were shown to demonstrate how the compiler maps each design from the equation to the hardware design search space. Finally a sample of the hardware design search space was given for the declarative expressions of the eight summed *multiplies* that represent the different ways to implement an eight-tap FIR filter in FPGA hardware. The building block library components used by this compiler and the final VHDL design for `Z54_PL1` are included in Appendix C for reference.

5. Results

In this chapter we demonstrate the effectiveness of the approach by examining in detail the hardware design search space generated for the example of the FIR filter previously described. We show how the design search space is used to select the optimal design in terms of characteristics such as throughput and latency and also describe how more complex metrics such as throughput in terms of input and output data rates and power requirements can be derived in terms of these parameters to produce a more comprehensive search space. In addition, the accuracy of the entire hardware design search space is quantified.

5.1 Introduction

The prototype demonstration system previously described generates a comprehensive hardware design search space by using a declarative language to specify different mathematical expression of *summed multiplications*. It generated a hardware design search space for the eight-Tap FIR filter example we have discussed in the previous chapters. We review this space by examining first the overall area, delay, and latency for each design, and then expand the space by applying an additional set of metrics that quantify throughput and power for each design.

An examination of each design form shows how their performances are characterized throughout the space. While some designs provide the largest throughput they also require the greatest area. Compromises in throughput can be made to find designs that have a lower area requirement, but require greater power in order to

meet that throughput. This examination of design tradeoffs exemplifies the design space exploration process.

To verify the values provided by the hardware design search space we synthesized and placed-and-routed each design to obtain the actual area and delay values. By comparing the estimated to the actual values we show that this system is capable of producing estimates that are within 2.1% of actual area and within 4.4% of actual clock delay.

Section 5.2 of this chapter discusses how the estimated performance characteristics of area, latency, and delay can be used to calculate power usage and throughput of the designs in terms of input and output data rates in the hardware design search space. In Section 5.3 we review the capabilities and requirements of the different design forms and show in-detail what information can be extracted from the design space. Finally, in Section 5.4 we present the process used to verify the hardware design search space and the accuracy of our estimates.

5.2 Hardware Design Search Space

As discussed in Chapter 4, the hardware design search space contains multiple designs, each representing one of three different structural forms of heterogeneous computing architectures-parallel, serial, and hybrid. These designs have distinctly different performance characteristics that can be evaluated against the requirements of the application.

Figure 5.1 represents the eight-Tap FIR filter's hardware design search, for which the prototype compiler generated 54 equivalent expressions and then mapped them to 108 FPGA implementations. This figure presents two plots of these 108 FPGA

implementations; the first represents the area vs. the clock delay and the second represents the area vs. latency.

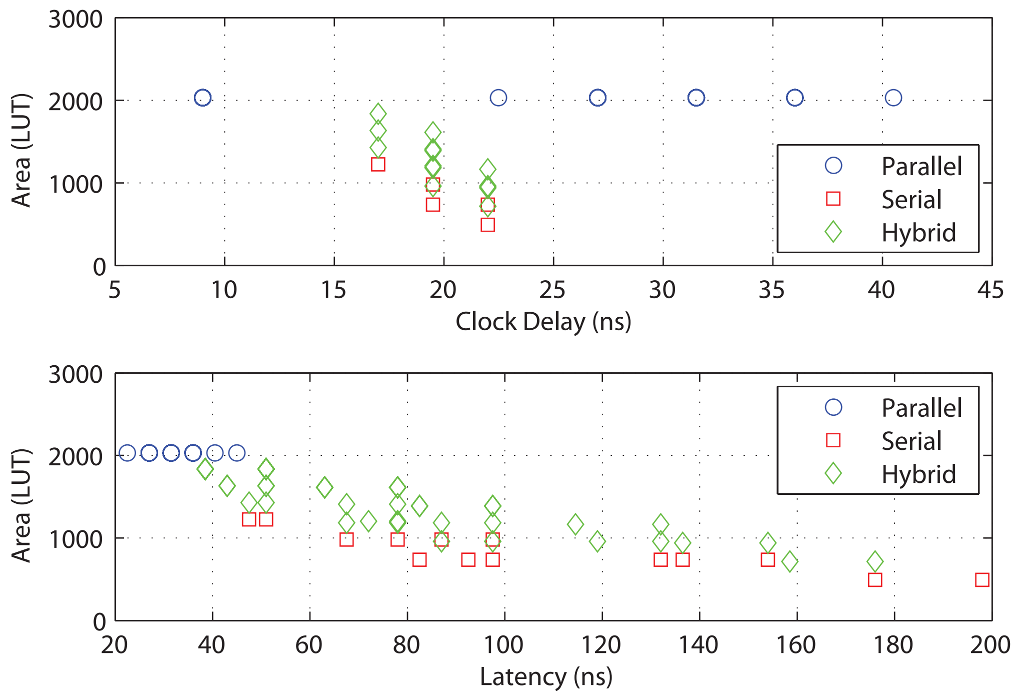

Figure 5.1: Area vs. Delay and Area vs. Latency plots of the hardware design search space for the eight-Tap FIR filter example.

The parallel form designs have some of the lowest latency values and require the greatest area. In both plots the combinatorial parallel form designs have a clock delay that is the same as their latency, as they do not require a clock. The clock/latency delay for those designs ranges from 22.5 ns to 45 ns. The clocked versions of the

parallel from designs have a clock delay that differs from their latency and in some cases overlap each other. All of the clocked versions of the parallel designs have a clock delay of 9 ns and require an area of 2032 LUTS. Where these designs vary is in their latencies which range from 27 ns to 45 ns due to the mathematical order of operation represented by the designs.

The serial and hybrid forms of designs exhibit the largest range of performance characteristics. They have varying clock and latencies delays, because both clocked and combinatorial versions require registers and clocked signals. The required area for the designs varies between 492 LUTs and 1836 LUTs. The designs operate between 17 ns and 22 ns clock periods and have latencies that range from 38.5 ns to 198 ns.

5.2.1 Expanding the Metrics of the Hardware Designs Search Space

This hardware design search space can be used to evaluate more than just area, delay, and latency of a design; it can also be used to evaluate throughput in terms of input and output data rate and power.

To understand how these are generated, we first define the terms used in the search space. The clock cycle delay, $CC - Delay$, is estimated by determining the longest delay path between two registers. By counting the number of clock cycles required for data to flow from the input to the output of a design, a representation of the initial latency ($Init - Latency$) in number of clock cycles can be evaluated, and is some multiple of the value of $CC - Delay$. If a design does not require a clock the $Init - Latency$ is equal to the $CC - Delay$.

Steady state latency, $SS - Latency$, refers to the time a design requires for

consecutive calculated values to arrive at the output, given the design is consistently supplied with input values. It is determined in clock cycles and evaluated using $CC - Delay$. Again if a design does not require a clock, then the $SS - Latency$ is equal to the $CC - Delay$.

Throughput, Tp, is the number of calculated output values per second a designs is capable of producing and is defined as

$$Tp = \frac{1}{SS - Latency}. \tag{5.1}$$

Input and Output data rates, (DR_{IN} & DR_{OUT}) respectively, are represented as throughput in terms of MB/s and require BW, the bit-width of the data path in bytes. The output data rate is defined as

$$DR_{OUT} = BW * frac1SS - Latency, \tag{5.2}$$

The input data rate is defined as

$$DR_{IN} = (N * BW) * frac1SS - Latency \tag{5.3}$$

and requires N number of input data paths to be defined.

Power usage, Pw, is estimated in terms of mW from the area and delay estimates

and a coefficient for power. [1]

$$Pw = NumLUT * \frac{1}{CC - Delay} * Pc, \quad (5.4)$$

where $NumLUT$ is the number of LUTS operating at the particular clock frequency $\frac{1}{CC-Delay}$. Pc is the power coefficient for a singe LUT operating at a particular frequency with an "average" amount of FPGA routing and a 50% toggle rate.

With these equations the hardware design search space presents a more thorough representation of the design tradeoffs for examination. A three-dimensional plot in Figure 5.2 Plot(A) represents the generated hardware design search space for the eight-tap FIR filter. It plots the associated area (LUT), output data (MB/s), initial latency (ns), and power (mW) represented by color, for each design. In addition there are 3 two-dimensional side view plots of the three-dimensional plot for easier analysis of the design search space. Plot (B) shows the area vs. latency and power for each design. Plot (C) shows latency vs. throughput with associated power, and plot (C) shows area vs. throughput with associated power.

In order to evaluate the multifaceted hardware designs and their tradeoffs efficiently, the clocked parallel designs were not plotted with the rest of the design space for this figure as they overshadowed the rest of the design space. These and the rest of the designs will be discussed in greater detail later in the next section.

[1] The coefficient for power usage was calculated using Xilinx's Web based power estimation tool [80].

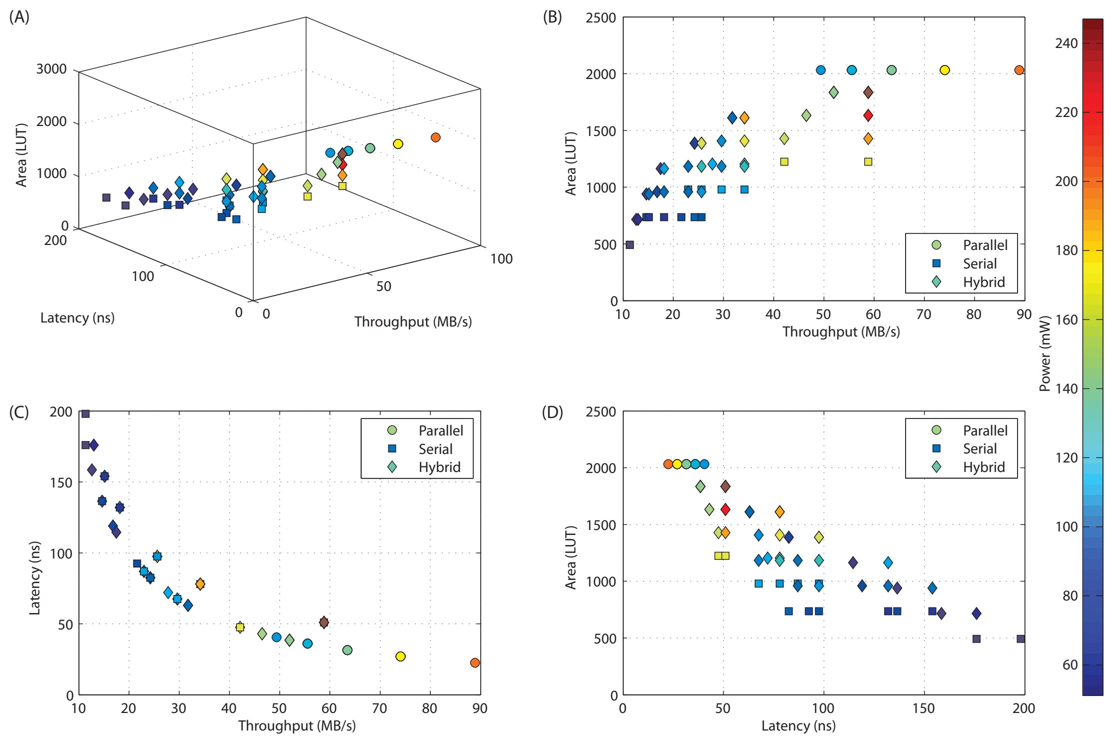

Figure 5.2: Three-dimensional plot of the hardware design search space.

5.3 Design Space Exploration

Variations in area, clock delay, and latency represent tradeoffs in performance and are important in determining what designs are optimal for a particular application. In this section we compare each form of the designs and review how they excel in different ways.

5.3.1 Parallel Designs

Parallel designs exhibit the greatest potential for high throughput since they represent the most concurrent execution for a given application. While these designs require the greatest area, their latency is among the smallest in the design space. By introducing registers into the parallel designs, the required clock period decreases significantly and these designs become streamlined in the moving of data through their architecture. This is seen when comparing their throughput rates in Figure 5.3, a snapshot of the parallel form hardware design search space.

This figure is broken into two columns, on the left are the combinatorial designs and on the right the clocked designs. The four main performance characteristics are presented for the design space, the first being delay, which includes both clock delay and initial latency. For the combinational designs these values are the same since these designs do not require a clock, but for the clocked versions it can be seen that the latency varies for each design. Area usage, the next characteristic, is constant for all designs, with the calculated throughput (Tp) and power usage shown to be dependent on the design's clock rate and latency.

In both combinatorial and clocked versions of the expressions, the order of op-

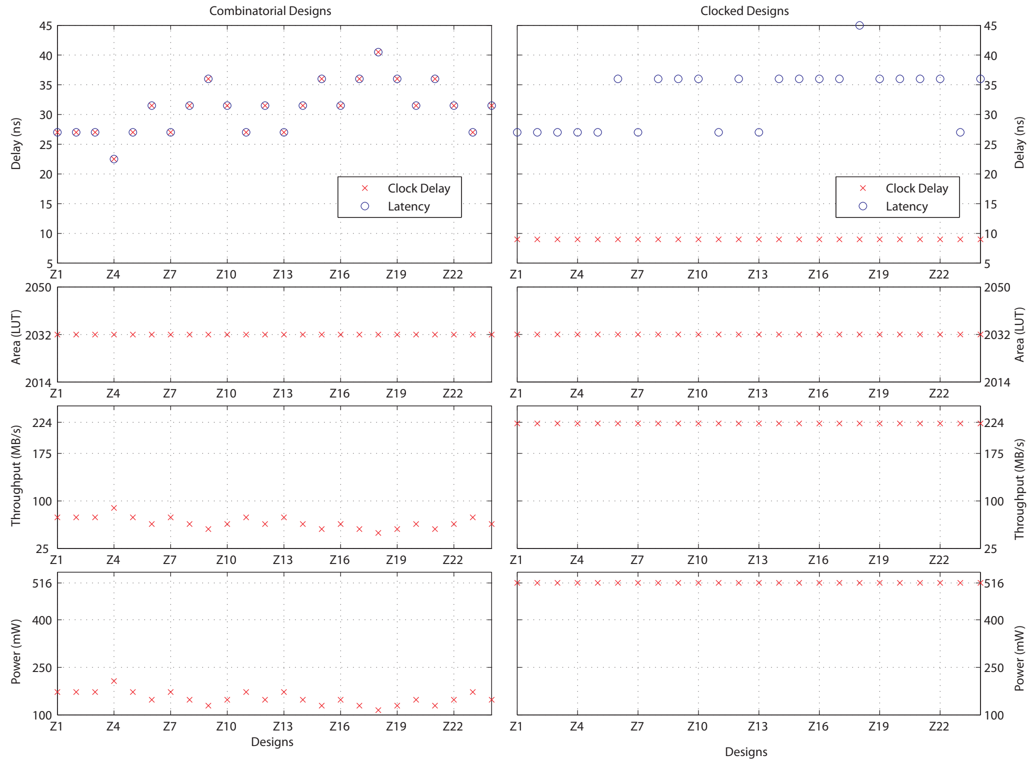

Figure 5.3: Snapshot of the parallel hardware design search space.

erations plays a significant role in determining the latency of the design. This is shown in Figure 5.4, which contains the hardware dataflow graphs of designs Z_4 and Z_{18}. Both designs have a LUT count of 2032, but Z_4 has a latency of 22.5 ns and Z_{18} a latency of 40.5 ns. By adding registers to these two designs they become clocked and pipelined. The clocked versions have a clock delay of 9 ns, but their initial latencies are 27 ns and 45 ns respectively. These designs require the same area but their mathematical order of operations directly lead to different designs and thus different latencies.

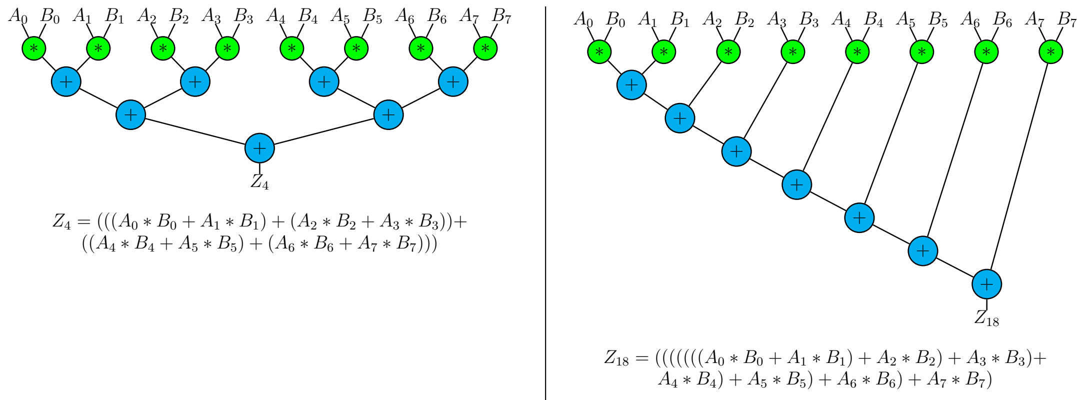

Figure 5.4: Hardware dataflow graphs for combinatorial parallel designs $\texttt{Z}_4$ and $\texttt{Z}_{18}$.

The ordering of the operators for equations $\texttt{Z}_4$ and $\texttt{Z}_{18}$ is determined by the bracketing of the generated expression shown in Figure 5.4. The number of different possible ways a design can be binary bracketed is determined by the theory of Catalan Numbers, as discussed earlier in Chapter 3. As the number of operations grows the corresponding Catalan Number grows faster. To reduce the number of redundant designs, the compiler developed for this research eliminated identical structures during generation of the equation space. This was discussed at a greater detail in Chapter 4. Despite the resulting reduction there are designs such as $\texttt{Z}_6$ and $\texttt{Z}_{14}$ shown in Figure 5.5 that demonstrate how two unique parallel forms can still have the same area, delay, and clock delay but vary structurally.

Figure 5.6 contains the hardware dataflow graphs of the clocked designs for equations $\texttt{Z}_4$ and $\texttt{Z}_{18}$. $\texttt{Z}_4$ has three pipeline stages where $\texttt{Z}_{18}$ has five pipeline stages. To create these pipeline designs, the compiler determines a global clock delay for the design by examining all the nodes and identifying the longest *grouped* node

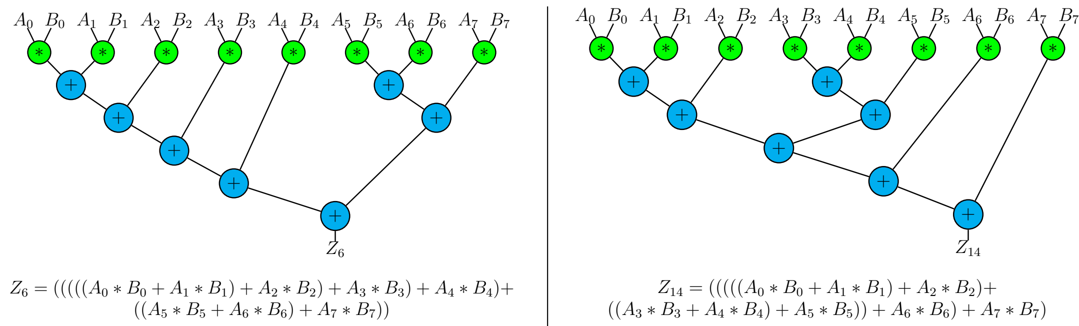

Figure 5.5: Hardware dataflow graphs for combinatorial parallel designs Z_6 and Z_{14}.

delay as the clock delay of the design. It then goes back through the design and inserts registers to create a set of pipeline stages so the data will arrive at the proper nodes in the correct order. When completed both of these designs have the same throughput but their initial latencies vary, Z_4 requires 3 clock cycles (27 ns) whereas Z_{18} requires 5 clock cycles (45 ns) to produce the first valid output.

5.3.2 Serial Designs

Serial form designs exhibit the smallest area for any particular latency, as they can represent either a single operator or group of operators working in parallel to calculate an entire set of inputs sequentially. This concept is similar to the parallel processing approach described in Chapter 1. By varying the number of processing elements in the design, different design tradeoffs are exposed.

The serial forms within the design space for the eight-Tap FIR filter can have up to four concurrent summation operations. This is due to the definition of how a summation operator can be used, as described in Chapter 4. Both the combinatorial

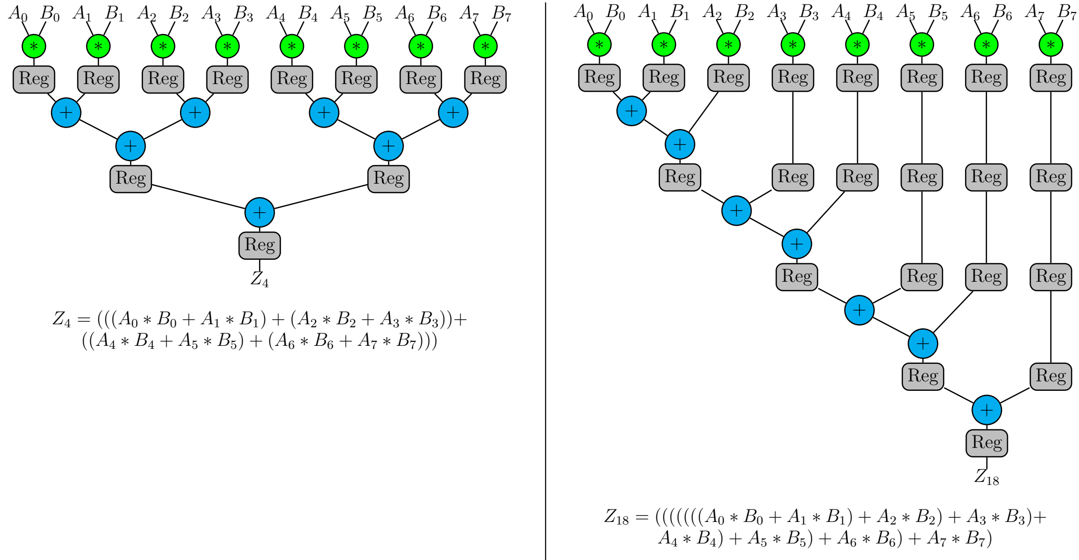

Figure 5.6: Hardware dataflow graphs for clocked parallel designs $\mathtt{Z_4}$ and $\mathtt{Z_{18}}$.

designs and clocked designs within the serial design space perform the same with respect to area and clock delay. This is a result of the summation operator used in all serial designs, which requires a clock and imposes a set amount of clock stalls to generate a result. This relationship between both combinatorial and clocked versions of the serial designs can be seen in Figure 5.7, the snapshot of the serial form hardware design search space.

The main difference between these architectures and a parallel processing architecture is that the final result of each ($\sum$) node is *summed* by separate adder operators forming a single output data path. This can be seen in Figure 5.8 It involves splitting the input data into sections that being processed concurrently, but inside each section they are calculated sequentially. Processing elements operate concurrently, and in this case they are the *grouped* nodes of both the ($*$) and ($\sum$)

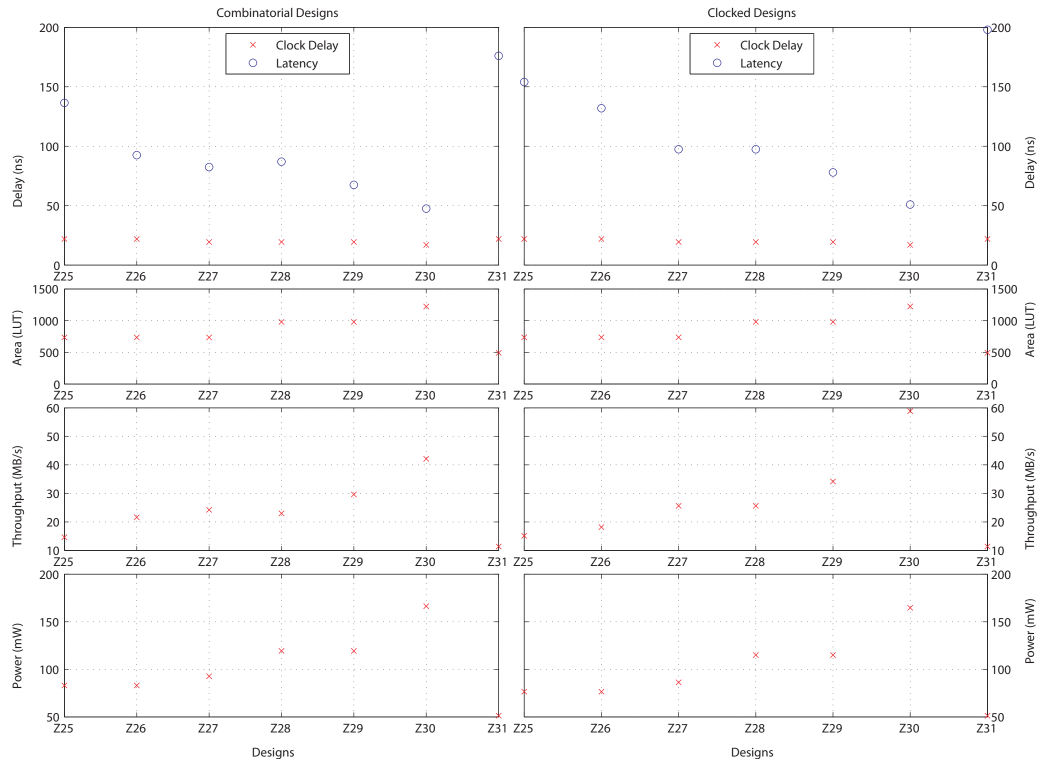

Figure 5.7: Snapshot of the serial hardware design search space.

nodes. Data paths into the *grouped* nodes comprise mux trees that direct the input data into the multiplier operators ($*$) node. A simple controller imbedded inside the ($\sum$) node changes the input data path by controlling the mux trees while *summing* the calculations from the ($*$) nodes (multiplier operators). Distributing the input data this way is a standard technique of parallel processing.

The relationship of mux tree size, number of summation operators, and size of input data path is shown in Figure 5.8. The figure contains the hardware dataflow graphs for Z_{30} and Z_{31}. Z_{30} has a clock delay of 17 ns whereas Z_{31} has a clock delay of

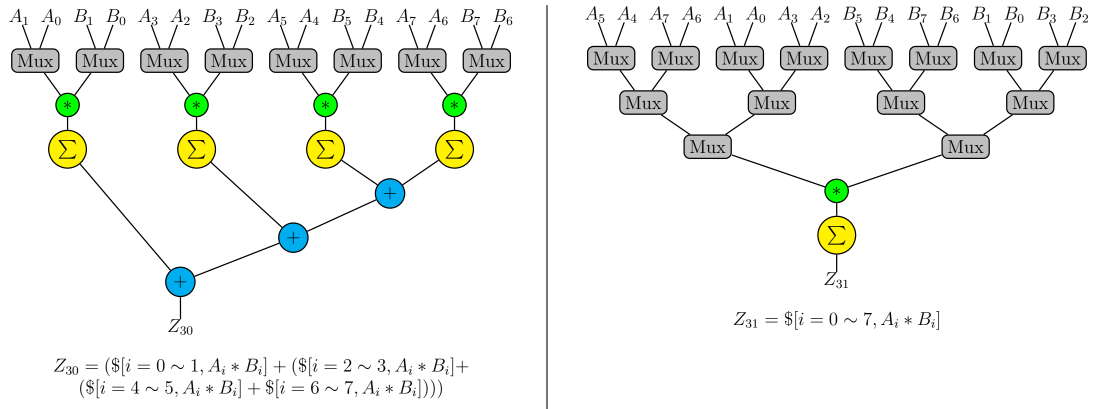

Figure 5.8: Hardware dataflow graphs for combinatorial serial designs Z_{30} and Z_{31}.

22 ns. This is due to the larger mux trees for design Z_{31}. The remaining operations of adding the ($\sum$) nodes together becomes the only section of the designs that could be a possible candidate for registers in the clocked versions. The same process used in the parallel form designs to create the clocked versions is used here. When determining the clock delay the *grouped* nodes of the summation, multiplication, and mux trees are calculated as the "clocked" path.

The clocked versions of Z_{30} and Z_{31} are shown in Figure 5.9. Since there are not enough parallel addition operations to require the placement of a register between the remaining (+) nodes (addition operators), only a single register is placed at then end of the designs. This is due to the clock delay for each design. In this case the longest path of the *grouped* nodes is from the input (A_1) through the mux tree, the ($*$) node, and to the ($\sum$) node. For Z_{30} the delay is 17 ns and for Z_{31} it is 22 ns. In both cases the clock delay is greater then the delay of the remaining *additions*, and therefore there is no need to place a register until the end.

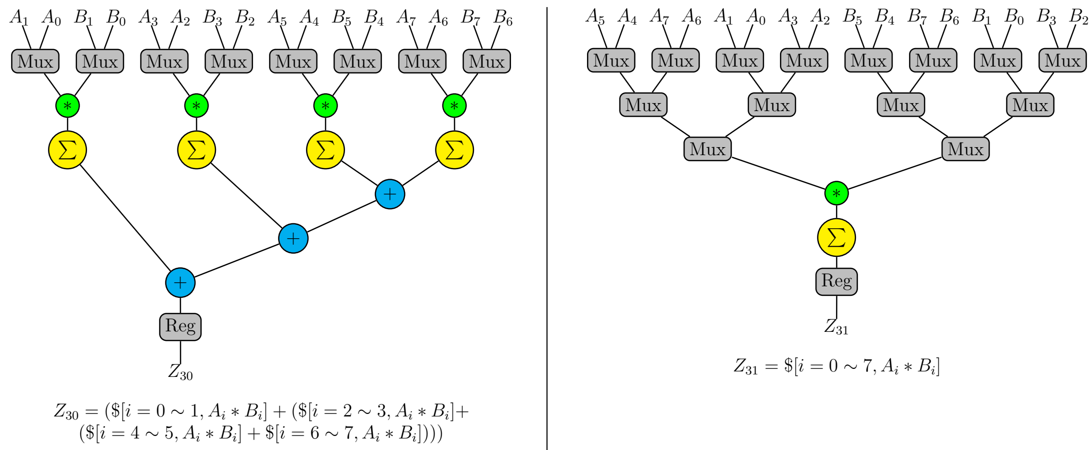

Figure 5.9: Hardware dataflow graphs for clocked serial designs $\mathtt{Z}_{30}$ and $\mathtt{Z}_{31}$.

The clocked versions of the expressions $\mathtt{Z}_{30}$ and $\mathtt{Z}_{31}$ behave similarly to their combinatorial counterparts. The main difference is that the clocked designs use their last register to hold an output while the adders are calculating another answer. This increases the design throughput for the clocked designs, as shown by comparing the performance characteristics for combinatorial and clocked versions of $\mathtt{Z}_{30}$ in Figure 5.7. The throughput for the combinatorial version of $\mathtt{Z}_{30}$ is 42.1 MB/s whereas the throughput of the clocked version is 58.8 MB/s.

5.3.3 Hybrid Designs

Hybrid designs are a combination of both the parallel and serial designs, where portions of the serial designs are converted to parallel representations. In some cases, these designs exhibit a larger area while accomplishing the same throughput and while requiring less power to operate. This is shown in the snapshot of the

hybrid hardware design search space in Figure 5.10.

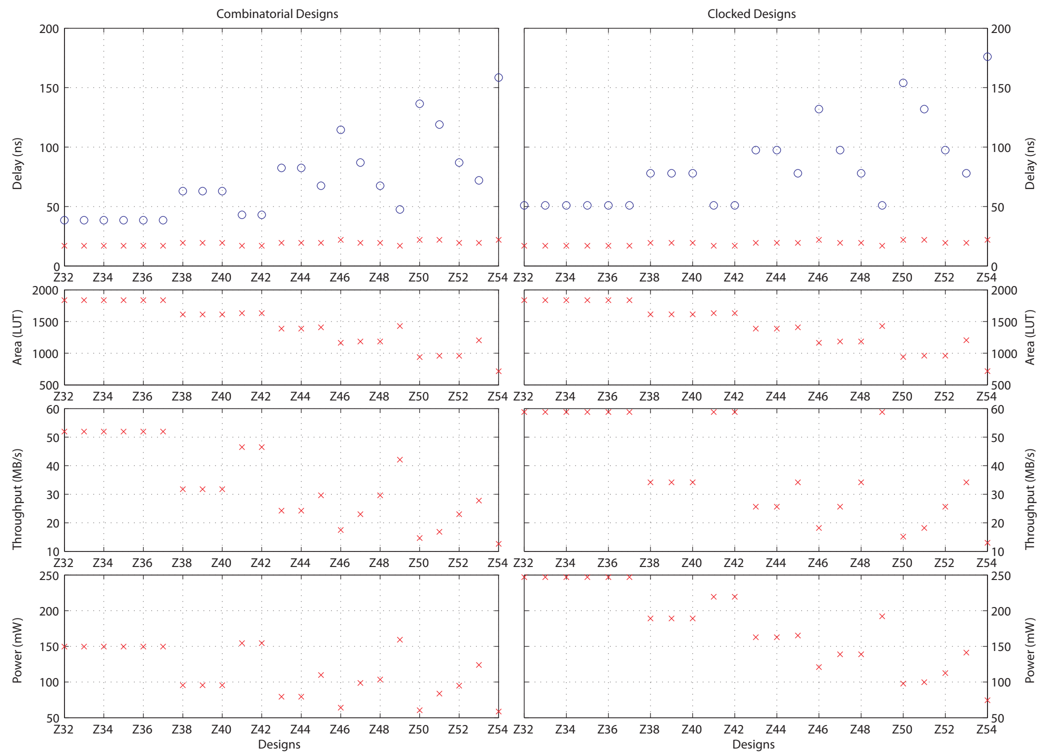

Figure 5.10: Snapshot of the hybrid hardware design search space.

An example of how sections of a serial design can be replaced with parallel forms is shown in Figure 5.11. The figure contains the hardware dataflow graphs for designs Z_{34} and Z_{49}. These designs are two of the hybrid combinatorial versions for the previous serial design Z_{30} in Figure 5.8 which use parallel components in place of one or more of the *grouped* summation and/or multiplication operators.

When comparing hybrid to serial designs, the power differences between the

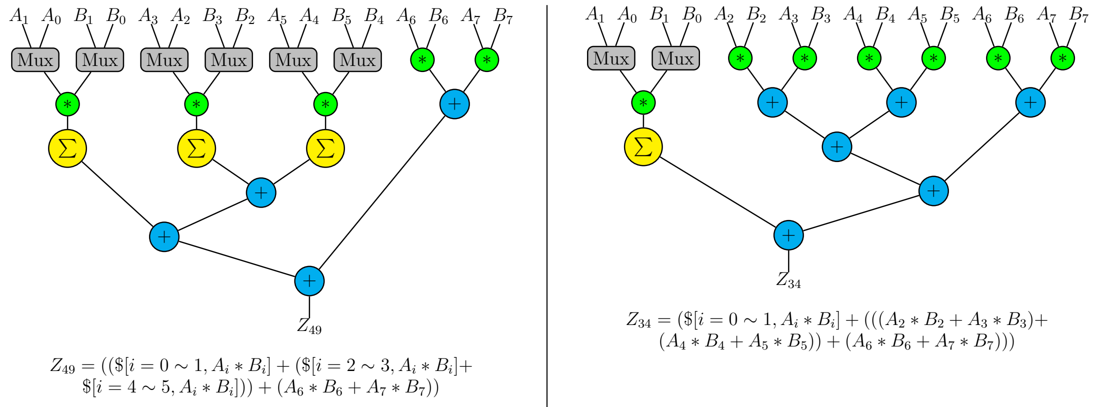

Figure 5.11: Hardware dataflow graphs for combinatorial hybrid designs Z_{34} and Z_{49}.

designs vary in relation to the number of LUTs consumed and their clock delay. The serial design Z_{30} uses four multiplier and summation operators to calculate its output, and has an estimated power usage of 166.3 mW. The hybrid forms of Z_{34} and Z_{49} use 149.7 mW and 159.3 mW respectively. While the power of these designs is lower, their throughput is greater. The serial design Z_{30} has an estimated throughput of 42.1 MB/s, the same as for hybrid design Z_{49}, but uses about 7 mW less power. It is interesting to note that the throughput for hybrid design Z_{34} is 51.95 MB/s and uses about 16.6 mW less power.

When comparing the hybrid combinatorial versions to their clocked counterparts we see the expected increase in throughput found in both of the other design forms. Figure 5.12 shows how the hardware dataflow graphs for the clocked versions of expressions Z_{34} and Z_{49} have only two pipeline stages. These two designs require two clock cycles to calculate the *summed multiplies*, and it is this stall that is taken into

account when determining the latency of a design. Figure 5.12 shows that by placing registers in strategic places, the designs become pipelined, increasing throughput, but also increasing power usage as shown in the hybrid hardware designs space snapshot in Figure 5.10.

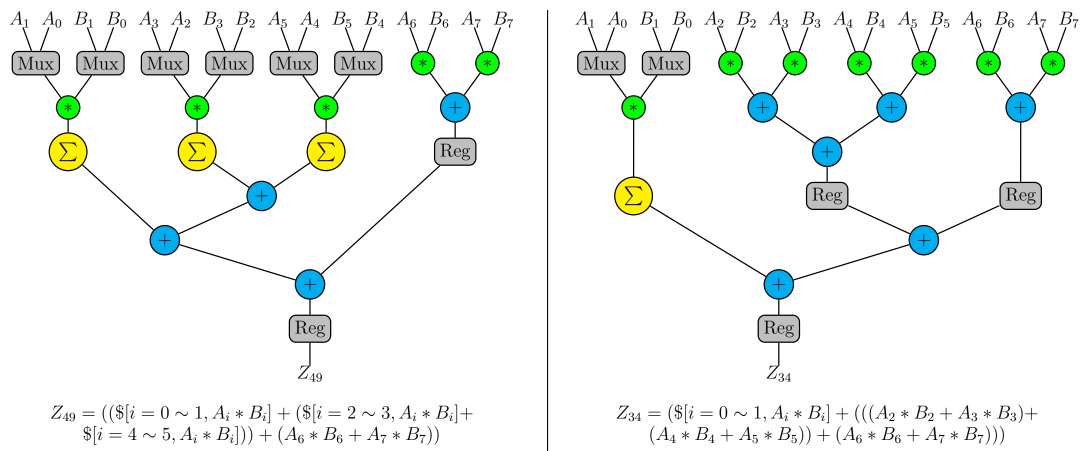

Figure 5.12: Hardware dataflow graphs for clocked hybrid designs Z_{34} and Z_{49}.

5.4 Accuracy of the Hardware Design Search Space

The accuracy of the hardware design search space is critical when exploring it to find an optimal hardware implementation. The previous systems described in Chapter **??** that perform estimations have been able to predict area with an accuracy of 5 to 18% and delay with an accuracy of 8 to 20 %. Some of these systems only performed estimations, while others generated the designs with performance estimations. By using the approach proposed in this research, this prototype compiler automatically generated and created a search space of 108 designs and was able to predict the area to within 2.1% and the clock delay to within 4.4 %.

These estimates were verified by implementing each design as an actual FPGA image and recording the LUT counts and clock delays. To implement the designs in the hardware design search space, a combination of an automated and manual design process was employed. This required each design generated by the prototype compiler to be compiled, synthesized, and placed and routed by hand. The target FPGA used for this was a Xilinx Virtex-II Pro 50 (XC2VP50-7) FPGA.

To perform the manual verification process, a wrapper was created that allowed each design to be plugged in and operated through a set of registers. This was considered a wrapper since it *wraps* around each design, allowing them to each use the same FPGA port configuration. This was important as it minimized timing variations associated with the different routing efforts by enabling the same consistent register-to-register transfer implementation within the FPGA for each design.

The wrapper that was used to implement each design is shown in Figure 5.13; it has two sets of registers for the inputs A_1 thru A_7 and B_1 thru B_7. It also has a

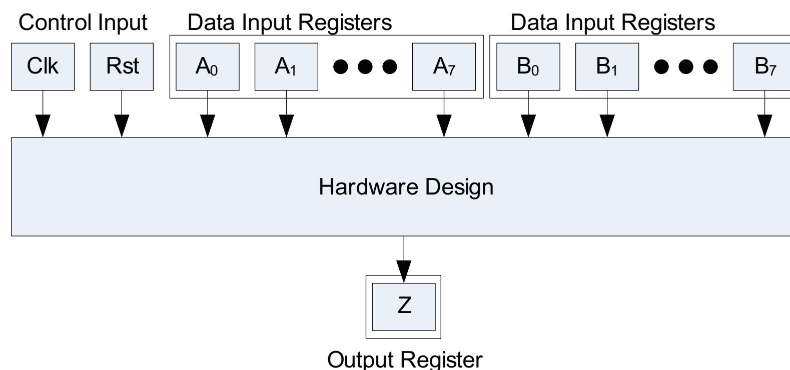

Figure 5.13: Wrapper for the hardware designs created by the prototype hardware compiler.

register to hold the final output value that is generated by the design. Associated clock and reset signals were connected to all of the registers in the wrapper as well as to the device if required. An instantiation of the wrapper was created for each design in the hardware design space, and then compiled to verify that each design's syntax was correct before proceeding with synthesis.

After each design wrapper was compiled, it was synthesized and then placed and routed. Synthesis used the VHDL library files described in Chapter 4 for the building block components, the design file generated by the compiler, and the wrapper to create an FPGA netlist. The netlist consists of the actual FPGA elements and their connectivity for the design. The *effort* (a configurable parameter in the toolset) that synthesis uses when trying to meet a given timing for a design greatly affects the translation process from HDL to netlist. Since synthesis generates the netlist that is used by place-and-route, its interpretations can help or hinder the remainder of the design process. Since the building block libraries are written in RTL VHDL,

synthesis was not required to interpret the design. Synthesis was required to add any port I/O buffers, clock buffers, and global clock routing associated with the wrapper. Even with the design written in RTL VHDL we noticed that if the timing *effort* was set too high the synthesis tool would manipulate the RTL specified designs by repacking LUT logic. This *effort* would produce unacceptable random results and this was especially true in the case of the combinatorial designs.

Once each design was converted into a synthesized netlist it was mapped to FPGA-specific elements and then placed and routed by FPGA vendor-specific tools. In this case we used Xlinx's Foundation Tool Set targeted for the Virtex-II Pro 50. During this step, the design is mapped to actual FPGA elements that the synthesis tool has previously selected. In this case we have specified which elements are to be used in the RTL VHDL building block library. As a result, the tool is merely verifying that all of the components in the netlist exist in the Virtex-II Pro 50 FPGA.

Once mapping is complete, the tool performs an unconstrained placement. This allows the tool to have the maximum potential to find an optimal layout for the mapped components. After all the components have been placed to physical locations inside the FPGA, the router determines the best path for each signal wire through the device. There are different levels at which the router can be asked to perform. To verify that we were not asking for too much, the tool was configured to use a default setting of medium *effort*. This leaves an an additional level of effort to be used later if a design was persistently not meeting the required timing, though this was not necessary for the design space generated by the compiler for this example.

Figure 5.14 shows the estimated and actual area and delay for each of the combinatorial and clocked designs in the parallel form hardware design search space. The area estimates for the combinatorial designs were slightly greater then the actual values. This can be attributed to the synthesis tool repacking LUT logic when interpreting the VHDL deigns. Even though the designs were written using a structural RTL style VHDL, the synthesis tools still performed optimizations when generating the netlists. This was not the case when synthesizing the clocked VHDL designs. By placing registers between the LUT logic inside the nodes, the synthesis tools were not able to replace or repack logic, and thus the estimated area for the designs was close if not exact. This was also the case with estimated timing, and can be seen by comparing the estimated to actual values between the combinatorial and clocked designs in Figure 5.14.

Figure 5.15 shows the estimated and actual area and delay performance characteristics for the designs of the serial form. The estimates for combinatorial and clocked versions were identical due to the fact that the serial designs are already clocked. All but one of the designs exhibited a 0.4% or less difference in estimated area. Design Z_{30} required almost 10% more area, and this is attributed to the larger number of *summing* addition operators used after the summation operators. Designs Z_{28} and Z_{29} showed the greatest variation of estimated timing. This could be attributed to the additions after the summation operators as well, where Z_{31} was estimated to be slower than it actually was able to perform. This is attributed to the placement and routing effort of the compiler, since its longest delay was the input paths through the mux trees.

Figure 5.16 shows the estimated and actual area and delay performance charac-

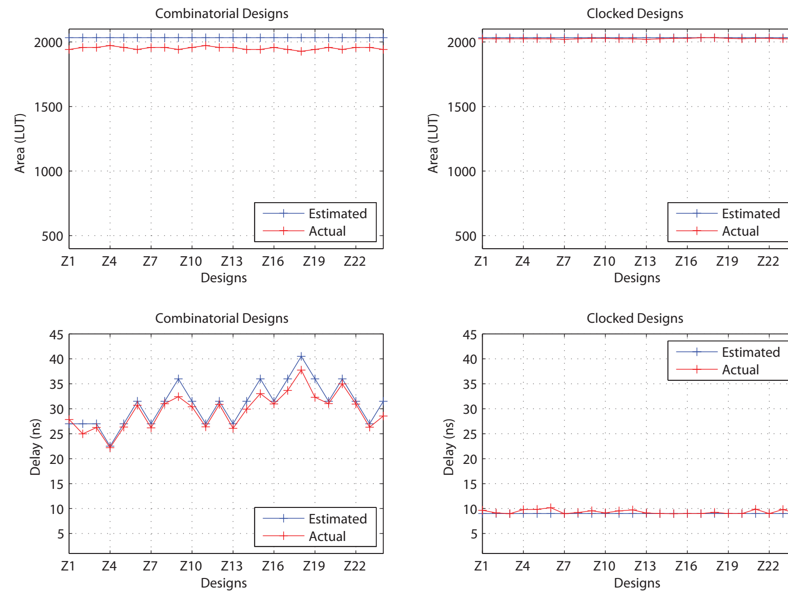

Figure 5.14: Comparison of estimated and actual area & delay performance characteristics for the parallel form designs.

teristics for the hybrid form designs. These designs required the most effort from the tools since we were not able to specify that we wanted the combinatorial path to operate at a different frequency then the clocked path. This meant that the synthesis tool was required to interpret the design such that the entire design would operate at the clock delay required by the summation operators. Even though synthesis used this higher *effort* to generate the netlist, the designs performed close to the predicted estimations. The clocked versions exhibited more LUT logic packing and were usually smaller in size then estimated, while the clocked version varied in either direction. The delay for the combinatorial versions was usually slightly greater then estimated, but almost the same for the clocked versions.

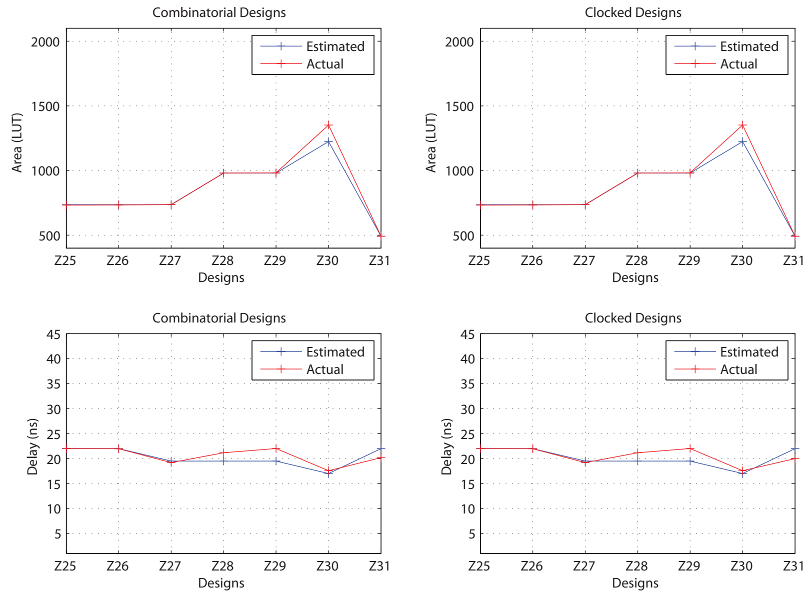

Figure 5.15: Comparison of estimated and actual area & delay performance characteristics for the serial form designs.

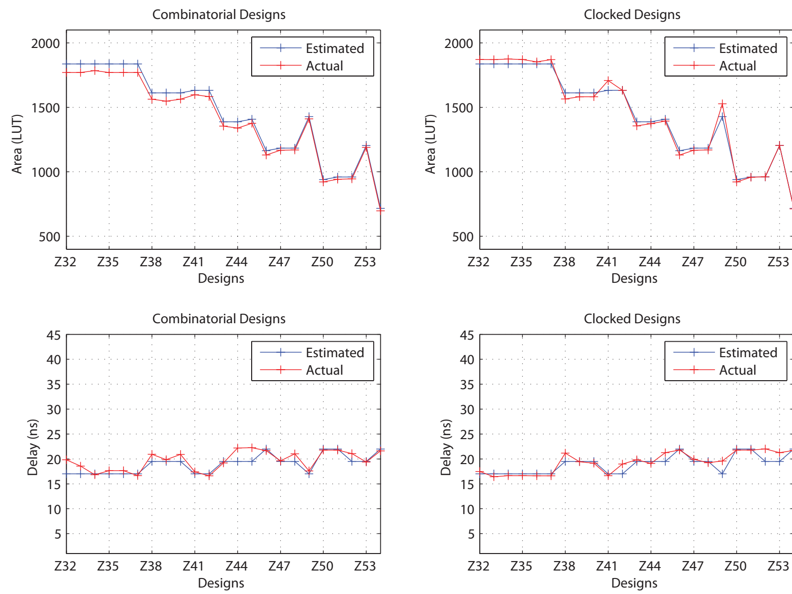

Figure 5.16: Comparison of estimated and actual area & delay performance characteristics for the hybrid form designs.

5.5 Summary

Our prototype demonstration system created a hardware design search space for an eight-Tap FIR filter. We showed how this hardware design search space could be expanded to include throughput and power for each of the designs. This information was used to explore the tradeoffs of the different designs without having to spend the time implementing them as actual FPGA images (bit-streams).

The overall accuracy of the hardware design search space was greater than any of the previous systems described in Chapter 2. Only a few of the previous research systems produced estimates of their designs, and when they did it was only in terms area and delay.

Most notable of the designs produced by the previous systems are the serial designs, which are implied by loop declarations in the previous compiler systems and are implied through use of the summation function in this research. Varying the types of serial designs was a major focus for most of the previous systems, since they were bound to loop style optimizations. The serial designs produced here are similar to the processing element approach of the previous systems, but resemble heterogeneous computing architectures rather then a standard parallel processing architecture. Furthermore this research produced three different architectural forms for use in the hardware design search space.

The fully parallel forms of designs were not considered by most of the previous systems. The few that did only mapped instructions to pre-defined von Neumann style processing elements. If the architecture was not fixed, it was created from the instructions given by the application writer or hardware designer. This research

performs more than this by discovering an optimal ordering of operations through the generation of all the equivalent expression for a given application.

The final forms of designs produced by and unique to this research, are the hybrid designs. These designs share properties of both the serial and parallel forms and produce unique tradeoffs that were discussed earlier. Power and throughput are only two of the performance criteria used to describe how these designs are unique, and while sometimes larger then their all-serial counterparts, the hybrid designs exhibit a lower power usage estimate and, at times, a greater throughput for the same required area.

6. Conclusion

In this chapter we review the contributions of this thesis and present the future directions for potential research in the area of automated FPGA design.

6.1 Summary of Contributions

In this thesis we presented a brief introduction to the manual process of implementing an application in FPGA hardware. We discussed how previous research attempted to automate this process by translating an imperative HLL application written in a sequential C-like language to a HDL design. We discussed how design exploration is a key step in the manual design process that was not addressed by these systems. We then presented our approach of automatically generating guaranteed optimized and correct HDL designs for applications at an algorithmic level in order to enable an inclusive design exploration. We demonstrated how our prototype system could create these useful hardware designs and quantified their accuracy.

In order to call a design optimized it must meet an application's performance requirements. Through the generation of a comprehensive representation of possible hardware designs, we accelerated the manual process of design space exploration by designing multiple versions of the same application for hardware and producing estimates of performance. This process alleviates the necessary time spent when designing multiple versions of the same application implementation.

With this process a designer can now start with an algorithmic description of an application and then generate a comprehensive representation of all possible hard-

ware designs, which was not possible with any of the previous systems described in Chapter 2. This significant increase in the amount of designs generated is primarily due to the previous systems reliance on imperative languages and SUIF to interpret, optimize, and create their hardware implementations. SUIF was intended to optimize software using a set of parallel processing techniques intended for von Neumann style architectures. This hindered the ability of the previous research to extract different hardware implementation architectures and create a comprehensive hardware design search space. By generating equivalent mathematical representations of an input equation via a novel declarative programming language this thesis avoids a number of difficulties associated with imperative languages. Through the use of the theory of Catalan numbers and compositions of n we show that the number of possible equivalent expressions for an expression is bounded, and thus show that the hardware design space for these expressions is also bounded. Furthermore, this thesis translates the designs to heterogeneous computing architectures, as these architectures have been found to perform better then von Neumann architecture for algorithmic based applications.

The Prototype system, that we created to demonstrate the approach, automates the creation of a comprehensive hardware design search space that can be used to perform design space exploration for an optimal FPGA hardware implementation of an application with respect to a given set of performance criteria.

The prototype system generates the hardware design search space by first creating a comprehensive set of different equivalent mathematical representations for *summed multiply*-based algorithms. It does this by applying a simple Prolog program that generates three different unique forms of expressions. The system, created

in Visual Basic, maps the generated expressions to unique heterogeneous hardware structures, extracts the data flow, and maps the operations to a building block library. This library contains structural RTL descriptions of components that perform the different operations supported by the system. Estimates of performance characteristics are generated for each design, including size, delay, throughput, and power. These designs are then coded using the structural RTL components, assuring that the designs meet the performance characteristics estimated when implemented in the target hardware environment. Finally the entire hardware design search space is translated into VHDL files, hardware data flow models, and an Excel spreadsheet containing performance estimations. This excel spreadsheet can then be explored for a design(s) that best meets the application's requirements as opposed to spending months implementing different designs by hand to find one that may or may not be the best choice

6.2 Recommendations for Future Work

We envision this system becoming a fully developed automation tool that fills the gap between the signal analyst and hardware engineer. This tool will use a signal analyst's depiction of an HLL application broken into sections of algorithm level descriptions to generate hardware design search spaces for each section. The search spaces will then be used in conjunction with a hardware engineers' understanding of the overall applications purpose and performance requirements to determine a final overall implementation that best suites the needs of the application and target environment.

Summed inner product operations are found in most mathematical transforms, filters operations, and matrix calculations. The current front-end equation generator creates an equation space for these (*summed multiplies*), and while this demonstrates the usability of this system, it could be extended to more complex types of applications or algorithms. Possible integration with tools that perform analytical equation solving like Maple™or Mathematica® would extend the usefulness of our research by providing the current system with the capability of handling larger signal processing algorithms. These tools could be used to expand into greater variety of applications or even extend the current equation space. This enables a signal analyst to work in an environment with which they are familiar when translating algorithms into hardware implementations.

The building block libraries could also be expanded to include floating-point mathematical operations, as well as more specialized high-level functions. New specialized cores could be used to increase the usability of the hardware design search space by incorporating FPGA-accelerated elements. For example, some future building block components could incorporate fast multipliers blocks that are currently in Xilinx Virtex II, IV, and V family FPGAs. Some future specialized cores could implement mathematical transform *blocks* that are configurable for a set dimensional size and bit-width in order to take advantage of known data manipulations.

This prototype system is not limited by the approach of this thesis; the front-end equation generator could be used for many types of applications. Earlier pruning of the generated equation space could also accelerate this system. The goal of this thesis is not to remove the hardware designer from the design process it is meant to assist them in an application's mapping from a high level description to a hardware

implementation by automating the process of generating a comprehensive and highly accurate hardware design search space for design exploration.

Bibliography

[1] Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, and Michael Taylor. The raw compiler project. In *SUIF Compiler Workshop*, August 1997.

[2] Altera Corp. FPGA, CPLD, and Structured ASIC: ALtera, the Leader in Programmable Logic. `http://www.altera.com/`.

[3] Annapolise Micro Systems Inc. The FPGA performance company. `http://www.annapmicro.com/`.

[4] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 70–80, Los Alamitos, CA, 1999. IEEE Computer Society Press.

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

[6] P. Banerjee, M. Haldar, A. Nayak, V. Kim, R. Anderson, and R. Uribe. Accelfpga: A dsp design tool for making area-delay tradeoffs while mapping matlab programs onto fpgas. Proc. International Signal Processing Conference, April 2003.

[7] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. *fccm*, 00:39, 2000.

[8] Fabrice Baray, Henri Michel, Pascal Urard, and Andres Takach. C synthesis methodology for implementing dsp algorithms. GSPx(Global Signal Processing Conferences & Expos for the Idustry), 2004.

[9] J. Bhasker. *A VHDL Primer*. Prentice Hall, Upper Saddle River, NJ, third edition, 1999.

[10] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet. Design space pruning through early estimations of area/delay tradeoffs for fpga implementations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1950–1968, 2006.

[11] Per Bjuréus, Mikael Millberg, and Axel Jantsch. Fpga resource and timing estimation from matlab execution traces. In *CODES*, pages 31–36, 2002.

[12] Kiran Bondalapati, Pedro C. Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: A design environment for adaptive computing technology. In *IPPS/SPDP Workshops*, pages 570–578, 1999.

[13] Bryan Bowyer. Just what is algorithmic synthesis. FPGA and Structured ASIC Journal, December 2005.

[14] C. Brandolese, W. Fornaciari, and F. Salice. An area estimation methodology for fpga based designs at systemc-level. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 129–132, 2004.

[15] BRASS Research Group (Berkely Reconfigurable Architecture Systems and Software). Garp: Combining a processor with a reconfigurable computing array. `http://brass.cs.berkeley.edu/garp.html`.

[16] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2–4 2002.

[17] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, Boston, MA, October 2004.

[18] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in roccc: C to vhdl compiler for fpgas. In *Int. Workshop On Applied Reconfigurable Computing (ARC 2006)*, Delft, The Netherlands, March 1-3 2006.

[19] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, 2000.

[20] João M. P. Cardoso and Markus Weinhardt. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. *Lecture Notes in Computer Science*, 2438:864–??, 2002.

[21] Carnegie Mellon University. SPIRAL Software/Hardware Generation for DSP Algorithms. urlhttp://www.spiral.net/hardware.html.

[22] Celoxica. DK Design Suite: Complete design environment for C-based algorithmic design entry, simulation and synthesis. `http://www.celoxica.com/products/dk/default.asp`.

[23] Colorado State University. Cameron project. `http://www.cs.colostate.edu/cameron/`.

[24] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[25] Bruce A. Draper, A. P. Willem Böhm, Jeff Hammes, Walid Najjar, J. Ross Beveridge, Charlie Ross, Monica Chawathe, Mitesh Desai, and José Bins. Compiling SA–C programs to FPGAs: Performance results. *Lecture Notes in Computer Science*, 2095:220, 2001.

[26] Rolf Enzler, Tobias Jeger, Didier Cottet, and Gerhard Troster. High-level area and performance estimation of hardware building blocks on FPGAs. In *Field-Programmable Logic and Applications (Proc. FPL'00)*, pages 525–534, 2000.

[27] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[28] David Galloway. The transmogrifier C hardware description language and compiler for fpgas. In Peter Athanas and Kenneth L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 136–144, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[29] M. Gokhale, J. Kaba, A. Marks, and J. Kim. Malleable architecture generator for FPGA computing. In John Schewel, editor, *High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic, Proc. SPIE 2914*, pages 208–217, Bellingham, WA, 1996. SPIE – The International Society for Optical Engineering.

[30] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56, 2000.

[31] Maya B. Gokhale and Janice M. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, Washington, DC, USA, 1998. IEEE Computer Society.

[32] Maya B. Gokhale and Janice M. Stone. Automatic allocation of arrays to memories in fpga processors with multiple memory banks. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 63, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *ISCA*, pages 28–39, 1999.

[34] Mentor Graphics. Modelsim - a comprehensive simulation and debug environment for complex asic and fpga designs. `http://www.model.com/default.asp`.

[35] Z. Guo and W. Najjar. A compiler intermediate representation for reconfigurable fabrics. In *16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, Madrid, Spain, August 2006.

[36] Z. Guo, D. C. Suresh, and W. A. Najjar. Programmability and efficiency in reconfigurable computer systems. Workshop on Software Support for Reconfigurable Systems, February 2003. held in conjunction with the Int. Conf. Of High-Performance Computer Architecture, Anaheim, CA.

[37] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *SIGPLAN Not.*, 39(7):249–256, 2004.

[38] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from C codes for FPGAs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. *vlsid*, 00:461, 2003.

[40] Malay Haldar, Anshuman Nayak, Alok Choudhary, Prith Banerjee, and Nagraj Shenoy. Fpga hardware synthesis from matlab. In *VLSID '01: Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*, page 299, Washington, DC, USA, 2001. IEEE Computer Society.

[41] B.A. Hamed, A. Salem, and G.M. Aly. Area estimation of lut based designs. In *Electrical, Electronic and Computer Engineering, 2004. ICEEC '04. 2004 International Conference on*, pages 39–42, 2004.

[42] Jeffrey Hammes, Bruce A. Draper, and A. P. Wim Böhm. Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. In *ICVS*, pages 83–97, 1999.

[43] Harvard University. Machine SUIF. `http://www.eecs.harvard.edu/hube/software/software.html`.

[44] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[45] David Ku and Giovanni DeMicheli. HardwareC – a language for hardware design (version 2.0). Technical report, Stanford University, Stanford, CA, USA, 1990.

[46] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for lut-based fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):104–122, 2006.

[47] Luciano Lavagno and Ellen Sentovich. ECL: A specification environment for system-level design. In *Design Automation Conference*, pages 511–516, 1999.

[48] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 507–512, New York, NY, USA, 2000. ACM Press.

[49] J. Madsen, J. Grode, P. Knudsen, M. Petersen, and A. Haxthausen. Lycos: The lyngby cosynthesis system, 1997.

[50] Mentor Graphics Corp. Catapult C synthesis. `http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm`.

[51] Mentor Graphics Corp. Precision RTL®. http://www.mentor.com/products/fpga_pld/synthesis/precision_rtl/.

[52] Y.Le Moullec, J-Ph.Diguet, T.Gourdeaux, and J-L.Philippe. Design trotter : System-level dynamic estimation task a 1st step towards platform architecture selection. Cambridge Int. Science Pub, December 2005. Issue 4.

[53] W. Najjar, W. Bohm, B. Draper, J. Hammes, R. Rinker, J. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing, 2003.

[54] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Accurate area and delay estimators for fpgas. In *DATE*, volume 00, page 0862, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[55] Massachusetts Institue of Technology Laboratory for Computer Science. Raw architecture workstation. http://www.cag.csail.mit.edu/raw/.

[56] Open SystemC Initiative. SystemC community. http://www.systemc.org/.

[57] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, May 2002.

[58] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Prentice Hall, Upper Saddle River, NJ, third edition, 1996.

[59] M. Puschel, A. C. Zelinski, and J. C. Hoe. Custom-optimized multiplierless implementations of dsp algorithms. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 175–182, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[61] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Des. Test*, 8(2):40–51, 1991.

[62] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and W. Bohm. An automated process for compiling data flow graphs into reconfigurable hardware. *IEEE Transactions on VLSI Systems*, 9(1):130–139, February 2001.

[63] Byoungro So, Pedro C. Diniz, and Mary W. Hall. Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 514–519, New York, NY, USA, 2003. ACM Press.

[64] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, 2002. ACM Press.

[65] Stanford University. The stanford suif compiler group. `http://suif.stanford.edu/`.

[66] Richard Stanley and Eric W. Weisstein. Catalan number. MathWorld–A Wolfram Web Resource `http://mathworld.wolfram.com/CatalanNumber.html`.

[67] Jon M. Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. Oreilly and Associates Inc, 2006.

[68] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 189–198, New York, NY, USA, 2003. ACM Press.

[69] Synplicity, Inc. Synplify: Fpga synthesis software. `http://www.synplicity.com/products/synplifypro/index.html`.

[70] The MathWorks, Inc. The mathworks - Simulink® hdl coder. `http://www.mathworks.com/products/slhdlcoder/`.

[71] The MathWorks, Inc. MATLAB® - The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`.

[72] Carnegie Mellon University. Phoenix project. `http://www.cs.cmu.edu/~phoenix/`.

[73] University of California, Riverside. Riverside optimizing compiler for configurable computing (roccc). `http://www.cs.ucr.edu/~roccc/`.

[74] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic synthesis (IWLS)*, pages 501–508, Temecula, CA, June 2004.

[75] J. von Neumann. First draft of a report on the edvac. Technical report, University of Pennsylvania, 1945.

[76] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, February 2001.

[77] Eric W. Weisstein. Binary bracketing. MathWorld–A Wolfram Web Resource url http://mathworld.wolfram.com/Bracketing.html.

[78] Eric W. Weisstein. Composition. MathWorld–A Wolfram Web Resource `http://mathworld.wolfram.com/Composition.html`.

[79] Xilinx Inc. MATLAB language-based design tool for implementing high performance DSP systems. `http://www.xilinx.com/ise/dsp_design_prod/acceldsp/index.htm`.

[80] Xilinx Inc. Xilinx: Power Solutions. `http://www.xilinx.com/products/design_resources/power_central/`.

[81] Xilinx Inc. Xilinx Programmable Logic Company. `http://www.xilinx.com/`.

[82] Leipo Yan, Thambipillai Srikanthan, and Niu Gang. Area and delay estimation for fpga implementation of coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 41(7):182–188, 2006.

[83] Heidi Ziegler, Byoungro So, Mary Hall, and Pedro C. Diniz. Coarse-grain pipelining on multiple fpga architectures. *fccm*, 00:77, 2002.

[84] Heidi E. Ziegler, Mary W. Hall, and Byoungro So. Search space properties for mapping coarse-grain pipelined fpga applications. In *LCPC*, pages 1–16, 2003.

Appendix A. List of Acronyms

ADL Architecture Descriptive Language

ANSI American National Standards Institute

ASIC Application Specific Integrated Circuit

AST Abstract Syntax Tree

CGRA Coarse Grained Reconfigurable Array

CI Component Interaction

CLB Configuration Logic Block

CPU Central Processing Unit

CSE Constant Subexpression Elimination

CSoC Configurable System on a Chip

CSP Communication Sequential Processes

CST Constant

CT Continuous Time

DE Discrete Events

DSP Digital Signal Processing

DT Discrete Time

EDA Electronic Design Automation

FIFO First In First Out

FIR Finite Impulse Response

FPAM Fixed Point Approximation Model

FPGA Field Programmable Gate Array

FSM Finite State Machine

FUN Function

HDFT Hardware Dataflow Table

HDL Hardware Descriptive Language

HLL High Level Language

ILP Instruction Level Parallelism

INP Input

IR Intermediate Representation

LUT Lookup Table

MUX Multiplexor

OPP Operation

OTP Output

PAR Place-and-Route

REG Register

RTL Register Transfer Level

SA-C Single Assignment C-Code

SDF Synchronous Dataflow

SC Spacial Computing

SoC System on a Chip

SR Synchronous/Reactive

SSA Static Single Assignment

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

VLIW Very Long Instruction Word

XML Extensible Markup Language

Appendix B. Example of VHDL design styles

In this appendix are the two design examples used in section 3.4. Both are VHDL implementations of the equation $Z = A * B + C$. The first design is a behavioral implementation that uses the IEEE 1076 libraries to describe the multiplication and addition. The second is a structural RTL coded implementation that uses an array multiplier and fast carry adder. These designs were synthesized and then placed-and-routed to produce estimates and verify the actual function block usage discussed in Chapter 3.

B.1 Behavioral Design

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY AddMult_Behav IS
   GENERIC(
      N : integer := 32
   );
   PORT(
      A   : IN     Std_Logic_Vector (N-1 DOWNTO 0);
      B   : IN     std_logic_vector (N-1 DOWNTO 0);
      C   : IN     std_logic_vector (2*N-1 DOWNTO 0);
      Z   : OUT    std_logic_vector (2*N-1 DOWNTO 0)
   );
END AddMult_Behav ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE Behav OF AddMult_Behav IS
-- Tell Synthesis to NOT use the dedicated multiply block
    attribute dedicated_mult: string;
    attribute dedicated_mult of Output:signal is "OFF";
BEGIN
```

```
   process (A,B,C)
   variable temp : std_logic_vector(2*N -1 downto 0);
   begin  -- process
       temp := unsigned(A) * unsigned(B);
       Z    <= unsigned(temp) + unsigned(C);
   end process;

END Behav;
```

B.2 RTL Design

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY AddMult_RTL IS
   GENERIC(
      N : integer := 32
   );
   PORT(
      A   : IN     std_logic_vector (N-1 DOWNTO 0);
      B   : IN     std_logic_vector (N-1 DOWNTO 0);
      C   : IN     std_logic_vector (2*N-1 DOWNTO 0);
      Z   : OUT    std_logic_vector (2*N-1 DOWNTO 0)
   );
END AddMult_RTL ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- pragma synthesis_off
LIBRARY ArrayAdd;
LIBRARY ArrayMult;
-- pragma synthesis_on
ARCHITECTURE struct OF AddMult_RTL IS
    SIGNAL temp : std_logic_vector(2*N-1 DOWNTO 0);
   -- Component Declarations
   COMPONENT ArrayAdd_Struc_RTL
   GENERIC (
      N : natural
```

```
   );
   PORT (
      a : IN     std_logic_vector (N-1 DOWNTO 0);
      b : IN     std_logic_vector (N-1 DOWNTO 0);
      z : OUT    std_logic_vector (N-1 DOWNTO 0)
   );
   END COMPONENT;
   COMPONENT ArrayMult_Struc_RTL
   GENERIC (
      N : natural
   );
   PORT (
      a : IN     std_logic_vector (N-1 DOWNTO 0);
      b : IN     std_logic_vector (N-1 DOWNTO 0);
      z : OUT    std_logic_vector (2*N-1 DOWNTO 0)
   );
   END COMPONENT;
BEGIN
   U_0 : ArrayAdd_Struc_RTL
      GENERIC MAP (
         N => 2*N
      )
      PORT MAP (
         a => z,
         b => sig2,
         z => OUtput
      );
   U_1 : ArrayMult_Struc_RTL
      GENERIC MAP (
         N => N
      )
      PORT MAP (
         a => sig0,
         b => sig1,
         z => z
      );
END struct;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY ArrayMult_Struc_RTL IS
  GENERIC(
      N : natural := 8
   );
   PORT(
      a : IN     std_logic_vector (N-1 downto 0) ;
      b : IN     std_logic_vector (N-1 downto 0) ;
      z : OUT    std_logic_vector (2*N-1 downto 0)
   );
END ENTITY ArrayMult_Struc_RTL;

ARCHITECTURE Struc OF ArrayMult_Struc_RTL IS
  type two_d_array is array
    (natural range <>, natural range <>) of std_logic;
  signal p, x, y, c : two_d_array (N downto 0, N downto 0);
  component PE_MultComp
      PORT(
      A_in  : IN     std_logic  ;
      B_in  : IN     std_logic  ;
      C_in  : IN     std_logic  ;
      P_in  : IN     std_logic  ;
      A_out : OUT    std_logic  ;
      B_out : OUT    std_logic  ;
      C_out : OUT    std_logic  ;
      P_out : OUT    std_logic
   );
  end component;

BEGIN
 ix: for j in 0 to N-1 generate
   x(0,j) <= a(j);
   p(0,j) <= '0';
 end generate ix;
 iy: for i in 0 to N-1 generate
   y(i,0) <= b(i);
   c(i,0) <= '0';
 end generate iy;
```

```
 GI: for i in 0 to N-1 generate
   GJ: for j in 0 to N-1 generate
     G1: if (i< N-1)and(j < N-1) and (J>0) generate
       cell : PE_MultComp port map (X(i,j), y(i,j), c(i,j),
            p(i,j), x(i+1,j), y(i,j+1), c(i,j+1), p(i+1,j-1));
       end generate G1;
       G2: if (j = N-1) and (i /= N-1) generate
         cell : PE_MultComp port map(X(i,j), y(i,j), c(i,j),
            p(i,j), x(i+1,j), open , p(i+1,j), p(i+1,j-1));
       end generate G2;
       G3: if (j = 0) generate
         cell : PE_MultComp port map(X(i,j), y(i,j), c(i,j),
            p(i,j), x(i+1,j), y(i,j+1), c(i,j+1), Z(i));
       end generate G3;
       G4: if (i=n-1)and (j > 0)and (j<n-1) generate
          cell : PE_MultComp port map(X(i,j), y(i,j), c(i,j),
            p(i,j), x(i+1,j), y(i,j+1), c(i,j+1), Z(n-1+j));
       end generate G4;
       G5: if (i=n-1) and (j=N-1) generate
          cell : PE_MultComp port map(X(i,j), y(i,j), c(i,j),
            p(i,j), open, open, z(n+j), Z(n-1+j));
       end generate G5;
     end generate GJ;
 end generate GI;
END ARCHITECTURE Struc;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY PE_MultComp IS
   PORT(
      A_in  : IN     std_logic;
      B_in  : IN     std_logic;
      C_in  : IN     std_logic;
      P_in  : IN     std_logic;
      A_out : OUT    std_logic;
      B_out : OUT    std_logic;
      C_out : OUT    std_logic;
      P_out : OUT    std_logic
```

```
   );
END PE_MultComp ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- pragma synthesis_off
LIBRARY Unisim;
-- pragma synthesis_on
ARCHITECTURE struct OF PE_MultComp IS
   COMPONENT LUT4
   GENERIC (
      INIT : bit_vector
   );
   PORT (
      I0 : IN     std_ulogic;
      I1 : IN     std_ulogic;
      I2 : IN     std_ulogic;
      I3 : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   -- pragma synthesis_off
   FOR ALL : LUT4 USE ENTITY Unisim.LUT4;
   -- pragma synthesis_on
BEGIN
   A_out <= A_in;
   B_out <= B_in;
   I0 : LUT4
      GENERIC MAP (
         INIT => X"E888"
      )
      PORT MAP (
         O  => C_out,
         I0 => C_in,
         I1 => P_in,
         I2 => B_in,
         I3 => A_in
      );
```

```
   I1 : LUT4
      GENERIC MAP (
         INIT => X"9666"
      )
      PORT MAP (
         O  => P_out,
         I0 => C_in,
         I1 => P_in,
         I2 => B_in,
         I3 => A_in
      );
END struct;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY ArrayAdd_Struc_RTL IS
 GENERIC(
    N : natural := 4
    );
  PORT(
    a : IN     std_logic_vector (N-1 downto 0) ;
    b : IN     std_logic_vector (N-1 downto 0) ;
    z : OUT    std_logic_vector (N-1 downto 0)
    );
END ENTITY ArrayAdd_Struc_RTL;

ARCHITECTURE struct OF ArrayAdd_Struc_RTL is
  component PE_AddComp
    PORT(
      A_in  : IN     std_logic  ;
      B_in  : IN     std_logic  ;
      C_in  : IN     std_logic  ;
      C_out : OUT    std_logic  ;
      P_out : OUT    std_logic
      );
  end component;
  signal c : std_logic_vector(N downto 0);
BEGIN
```

```
  GI: for i in 0 to N-1 generate
    G0: if (i = 0) generate
      cell : PE_AddComp port map
        (A(i), B(i), '0', c(i+1), z(i));
    end generate G0;
    G1: if (i>0) and (i<N-1) generate
      cell : PE_AddComp port map
        (A(i), B(i), c(i), c(i+1), z(i));
    end generate G1;
    G2: if (i=N-1) generate
      cell : PE_AddComp port map
        (A(i), B(i), c(i), open, z(i));
    end generate G2;
  end generate GI;
END struct;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY PE_AddComp IS
   PORT(
      A_in  : IN     std_logic;
      B_in  : IN     std_logic;
      C_in  : IN     std_logic;
      C_out : OUT    std_logic;
      P_out : OUT    std_logic
   );
END PE_AddComp ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- pragma synthesis_off
LIBRARY Unisim;
-- pragma synthesis_on
ARCHITECTURE struct OF PE_AddComp IS
  SIGNAL Lut2_out : std_ulogic;
  COMPONENT LUT2
   GENERIC (
```

```
      INIT : bit_vector
   );
   PORT (
      I0 : IN     std_ulogic;
      I1 : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   COMPONENT MUXCY
   PORT (
      CI : IN     std_ulogic;
      DI : IN     std_ulogic;
      S  : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   COMPONENT XORCY
   PORT (
      CI : IN     std_ulogic;
      LI : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   -- pragma synthesis_off
   FOR ALL : LUT2 USE ENTITY Unisim.LUT2;
   FOR ALL : MUXCY USE ENTITY Unisim.MUXCY;
   FOR ALL : XORCY USE ENTITY Unisim.XORCY;
   -- pragma synthesis_on
BEGIN
  I0 : LUT2
     GENERIC MAP (
        INIT => X"6"
      )
      PORT MAP (
         O  => Lut2_out,
         I0 => A_in,
         I1 => B_in
      );
   I1 : MUXCY
```

```
      PORT MAP (
         O  => C_out,
         CI => C_in,
         DI => A_in,
         S  => Lut2_out
      );
   I2 : XORCY
      PORT MAP (
         O  => P_out,
         CI => C_in,
         LI => Lut2_out
      );
END struct;
```

Appendix C. VHDL files for examples in Chapter 4.

C.1 Structural RTL VHDL for a 16-Bit unsigned adder.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY ArrayAdd_Struc_RTL IS
 GENERIC(
    N : natural := 16
    );
  PORT(
    a : IN     std_logic_vector (N-1 downto 0) ;
    b : IN     std_logic_vector (N-1 downto 0) ;
    z : OUT    std_logic_vector (N-1 downto 0)
    );
END ENTITY ArrayAdd_Struc_RTL;

ARCHITECTURE struct OF ArrayAdd_Struc_RTL is
  component PE_AddComp
    PORT(
      A_in  : IN     std_logic  ;
      B_in  : IN     std_logic  ;
      C_in  : IN     std_logic  ;
      C_out : OUT    std_logic  ;
      P_out : OUT    std_logic
      );
  end component;
  signal c : std_logic_vector(N downto 0);

BEGIN
  GI: for i in 0 to N-1 generate
    G0: if (i = 0) generate
      cell : PE_AddComp port map
        (A(i), B(i), '0', c(i+1), z(i));
    end generate G0;
    G1: if (i>0) and (i<N-1) generate
```

```
      cell : PE_AddComp port map
        (A(i), B(i), c(i), c(i+1), z(i));
    end generate G1;
    G2: if (i=N-1) generate
      cell : PE_AddComp port map
        (A(i), B(i), c(i), open, z(i));
    end generate G2;
  end generate GI;
END struct;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY PE_AddComp IS
   PORT(
      A_in  : IN     std_logic;
      B_in  : IN     std_logic;
      C_in  : IN     std_logic;
      C_out : OUT    std_logic;
      P_out : OUT    std_logic
   );
END PE_AddComp ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- pragma synthesis_off
LIBRARY Unisim;
-- pragma synthesis_on

ARCHITECTURE struct OF PE_AddComp IS
   SIGNAL Lut2_out : std_ulogic;
   COMPONENT LUT2
   GENERIC (
      INIT : bit_vector
   );
   PORT (
      I0 : IN     std_ulogic;
      I1 : IN     std_ulogic;
```

```
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   COMPONENT MUXCY
   PORT (
      CI : IN     std_ulogic;
      DI : IN     std_ulogic;
      S  : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   COMPONENT XORCY
   PORT (
      CI : IN     std_ulogic;
      LI : IN     std_ulogic;
      O  : OUT    std_ulogic
   );
   END COMPONENT;
   -- pragma synthesis_off
   FOR ALL : LUT2 USE ENTITY Unisim.LUT2;
   FOR ALL : MUXCY USE ENTITY Unisim.MUXCY;
   FOR ALL : XORCY USE ENTITY Unisim.XORCY;
   -- pragma synthesis_on

BEGIN
   I0 : LUT2
     GENERIC MAP (
        INIT => X"6"
     )
      PORT MAP (
         O  => Lut2_out,
         I0 => A_in,
         I1 => B_in
      );
   I1 : MUXCY
      PORT MAP (
         O  => C_out,
         CI => C_in,
         DI => A_in,
```

```
         S  => Lut2_out
      );
   I2 : XORCY
      PORT MAP (
         O  => P_out,
         CI => C_in,
         LI => Lut2_out
      );
END struct;
```

C.2 VHDL design for example Z_{54}

```
--Built With Library Version: v 1.0.0
library ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Z54_Z54_PL1 IS
  GENERIC(
    Ninp : natural := 16;
    Nint : natural := 16;
    Nout : natural := 16;
    SelSize_i0_6 : natural := 3;
    ItrSize_i0_6 : natural := 7
  );
  PORT (
    A7 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B7 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A0 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A1 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A2 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A3 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A4 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A5 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    A6 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B0 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B1 : IN std_logic_vector(Ninp-1 DOWNTO 0);
```

```
    B2 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B3 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B4 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B5 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    B6 : IN std_logic_vector(Ninp-1 DOWNTO 0);
    Clk : IN std_logic;
    Rst : IN std_logic;
    Z54 : OUT std_logic_vector(Nout-1 DOWNTO 0)
  );
END Z54_Z54_PL1;

library ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
LIBRARY BuildBlock;

ARCHITECTURE struct OF Z54_Z54_PL1 IS

-- Component Declarations

 COMPONENT U_Add
   GENERIC (
      N : natural := 16
   );
   PORT (
      A : IN     std_logic_vector (N-1 DOWNTO 0);
      B : IN     std_logic_vector (N-1 DOWNTO 0);
      C : OUT    std_logic_vector (N-1 DOWNTO 0)
   );
   END COMPONENT;


  COMPONENT U_Mult
   GENERIC (
      N : natural := 16
   );
   PORT (
      A : IN     std_logic_vector (N-1 DOWNTO 0);
      B : IN     std_logic_vector (N-1 DOWNTO 0);
```

```
   C : OUT    std_logic_vector (N-1 DOWNTO 0)
 );
 END COMPONENT;


COMPONENT RegC
 GENERIC (
   N : natural := 16
 );
 PORT (
   RegI : IN     std_logic_vector (N-1 DOWNTO 0);
   RegO : OUT    std_logic_vector (N-1 DOWNTO 0);
   Clk  : IN     std_logic ;
   Rst  : IN     std_logic
 );
 END COMPONENT;


COMPONENT Mux_2to1
 GENERIC (
   N : natural := 16
 );
 PORT (
   MuxIn1 : IN     std_logic_vector (N-1 DOWNTO 0);
   MuxIn2 : IN     std_logic_vector (N-1 DOWNTO 0);
   MuxOut : OUT    std_logic_vector (N-1 DOWNTO 0);
   S  : IN     std_logic
 );
 END COMPONENT;


 COMPONENT U_AddAcc
 GENERIC (
   N        : natural := 16;
   SelSize  : natural := 1;
   IterSize : natural := 2
 );
 PORT (
 A     : IN     std_logic_vector (N-1 DOWNTO 0);
```

```
      Clk   : IN     std_logic;
      EnIn  : IN     std_logic;
      Rst   : IN     std_logic;
      C     : OUT    std_logic_vector (N-1 DOWNTO 0);
      EnOut : OUT    std_logic;
      S     : OUT    std_logic_vector (SelSize-1 DOWNTO 0)
   );
   END COMPONENT;


--  Optional embedded configurations

-- pragma synthesis_oFnum
FOR ALL : U_Add USE ENTITY BuildBlock.U_Add;
FOR ALL : U_Mult USE ENTITY BuildBlock.U_Mult;
FOR ALL : RegC USE ENTITY BuildBlock.RegC;
FOR ALL : Mux_2to1 USE ENTITY BuildBlock.Mux_2to1;
FOR ALL : U_AddAcc USE ENTITY BuildBlock.U_AddAcc;
-- pragma synthesis_on
-- Internal signal declarations

  SIGNAL Mux_A0A2_A6_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B0B2_B6_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mult_0_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL AddAcc_0_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Sel_i0_6 : std_logic_vector(SelSize_i0_6-1 DOWNTO 0);
  SIGNAL EN_AddAcc_0 : std_logic;
  SIGNAL Mult_1_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Add_0_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_A5Mux_A6_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_A3Mux_A4_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_A1Mux_A2_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_A0A2_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_A3Mux_A6_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B5Mux_B6_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B3Mux_B4_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B1Mux_B2_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B0B2_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Mux_B3Mux_B6_o : std_logic_vector(Nint-1 DOWNTO 0);
```

```
  SIGNAL Const1 : std_logic_vector(1-1 DOWNTO 0);
  SIGNAL Reg_0_o : std_logic_vector(Nint-1 DOWNTO 0);
  SIGNAL Reg_1_o : std_logic_vector(Nint-1 DOWNTO 0);
BEGIN

  Mux_A0A2_A6: Mux_2to1
    GENERIC MAP(
      N => Nint
    )
    PORT MAP(
      MuxIn1 => Mux_A3Mux_A6_o,
      MuxIn2 => Mux_A0A2_o,
      MuxOut => Mux_A0A2_A6_o,
      S => Sel_i0_6(2)
    );

    Mux_B0B2_B6: Mux_2to1
    GENERIC MAP(
      N => Nint
    )
    PORT MAP(
      MuxIn1 => Mux_B3Mux_B6_o,
      MuxIn2 => Mux_B0B2_o,
      MuxOut => Mux_B0B2_B6_o,
      S => Sel_i0_6(2)
    );

    Mult_0: U_Mult
    GENERIC MAP(
      N => Nint
    )
    PORT MAP(
      A => Mux_A0A2_A6_o,
      B => Mux_B0B2_B6_o,
      C => Mult_0_o
    );

    AddAcc_0: U_AddAcc
    GENERIC MAP(
```

```
  N => Nint,
  SelSize => SelSize_i0_6,
  IterSize => ItrSize_i0_6
)
PORT MAP(
  A => Mult_0_o,
  C => AddAcc_0_o,
  Clk => Clk,
  Rst => Rst,
  EnIn => Const1(0),
  S => Sel_i0_6,
  EnOut => EN_AddAcc_0
);

Mult_1: U_Mult
GENERIC MAP(
  N => Nint
)
PORT MAP(
  A => A7,
  B => B7,
  C => Mult_1_o
);

Add_0: U_Add
GENERIC MAP(
  N => Nint
)
PORT MAP(
  A => AddAcc_0_o,
  B => Reg_1_o,
  C => Add_0_o
);

Mux_A5Mux_A6: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
```

```
  MuxIn1 => A6,
  MuxIn2 => A5,
  MuxOut => Mux_A5Mux_A6_o,
  S => Sel_i0_6(0)
);

Mux_A3Mux_A4: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => A4,
  MuxIn2 => A3,
  MuxOut => Mux_A3Mux_A4_o,
  S => Sel_i0_6(0)
);

Mux_A1Mux_A2: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => A2,
  MuxIn2 => A1,
  MuxOut => Mux_A1Mux_A2_o,
  S => Sel_i0_6(0)
);

Mux_A0A2: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => A0,
  MuxIn2 => Mux_A1Mux_A2_o,
  MuxOut => Mux_A0A2_o,
  S => Sel_i0_6(1)
);
```

```
Mux_A3Mux_A6: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => Mux_A3Mux_A4_o,
  MuxIn2 => Mux_A5Mux_A6_o,
  MuxOut => Mux_A3Mux_A6_o,
  S => Sel_i0_6(1)
);

Mux_B5Mux_B6: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => B6,
  MuxIn2 => B5,
  MuxOut => Mux_B5Mux_B6_o,
  S => Sel_i0_6(0)
);

Mux_B3Mux_B4: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => B4,
  MuxIn2 => B3,
  MuxOut => Mux_B3Mux_B4_o,
  S => Sel_i0_6(0)
);

Mux_B1Mux_B2: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => B2,
```

```
  MuxIn2 => B1,
  MuxOut => Mux_B1Mux_B2_o,
  S => Sel_i0_6(0)
);

Mux_B0B2: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => B0,
  MuxIn2 => Mux_B1Mux_B2_o,
  MuxOut => Mux_B0B2_o,
  S => Sel_i0_6(1)
);

Mux_B3Mux_B6: Mux_2to1
GENERIC MAP(
  N => Nint
)
PORT MAP(
  MuxIn1 => Mux_B3Mux_B4_o,
  MuxIn2 => Mux_B5Mux_B6_o,
  MuxOut => Mux_B3Mux_B6_o,
  S => Sel_i0_6(1)
);

Reg_0: RegC
GENERIC MAP(
  N => Nint
)
PORT MAP(
  RegI => Add_0_o,
  RegO => Reg_0_o,
  Clk => Clk,
  Rst => Rst
);

Reg_1: RegC
```

```
    GENERIC MAP(
      N => Nint
    )
    PORT MAP(
      RegI => Mult_1_o,
      RegO => Reg_1_o,
      Clk => Clk,
      Rst => Rst
    );

    Z54 <= Reg_0_o;
  Const1 <= conv_std_logic_vector(1,1);
END struct;
```

Vita

Mr. Michael Balog was graduated from Drexel University with a B.S.E.E in 1999 and an M.S.E.E in 2002. He started his career as a co-op with Robotic Systems Technology where he worked on the Mobile Detection Assessment and Response System (MDARS) project for the Department of Defense. MDARS was an autonomous ground vehicle used to patrol warehouses. Mike participated in the design of a new charging system, gyro, and accelerometer for the Global Positioning System on the vehicle. He later worked at Boeing Helicopters in the vertical airspace division where he assisted in designing the avionics system for the Bell-Boeing 609 civil Tilt Rotor aircraft. For his third co-op he worked for Intel Corporation where he provided technical support for Intel's line of embedded microprocessors. While pursuing his Masters Degree in Electrical Engineering at Drexel he worked on the DARPA SPIRAL project, which is aimed at developing automated techniques for optimizing DSP algorithms for particular hardware architectures. During this time Mike wrote his Master Thesis on "A Flexible Framework for FFT Processors." As part of this work he was part of the team that developed one of the first reported multidimensional million-point FFT implemented in an FPGA. In 2002 Mike joined the staff at Rydal Research, where he developed Rydal's PMC Reconfigurable Computing Module. In 2003 Mike became a partner and Chief Technical Officer at BuLogics Inc., a start-up company that is developing Home Automation products. At BuLogics Mike has been responsible for researching and developing new technologies for use in their Home Automation products and has played a key role in obtaining substantial funding for the company.