# ABSTRACT

HOURANI, RAMSEY S. A Performance Analysis Framework for the Design of DSP Systems. (Under the direction of Professors Winser E. Alexander and W. Rhett Davis).

The competitive market for generating smaller and faster portable devices compels designers to meet stringent design requirements while keeping time-to-market as low as possible. Application specific integrated circuits and field programmable devices are popular choices for implementing computationally intensive signal processing systems on hardware platforms due to their high speed and low power characteristics. However, the gap between the complex digital signal processing (DSP) algorithm and the efficient hardware implementation continues to grow with the advances in technologies and stringent design constraints. This work presents a performance analysis framework as a method for bridging the gap between the DSP algorithm and the efficient hardware implementation.

The work presented in this dissertation focuses on refining a basic DSP algorithm, such as an FIR filter, to several hardware implementations that are closely matched in performance while meeting design constraints. The design methodology in this work uses CAD and EDA tools accepted by both industrial and academic venues. The framework refinement process invokes C++ scripts that efficiently generate and model the DSP algorithms and hardware designs at multiple levels of abstraction. Additionally, scripts within the framework invoke a Synopsys Design Compiler that accurately and efficiently estimates circuit-level performance metrics including area, throughput and power dissipation.

This work demonstrated the performance analysis framework using three computationally intensive DSP algorithms as case studies. The first DSP case study focuses the essential FIR filter. Cost functions provided by the framework reduce the design space to a set of efficient hardware designs that meet specific performance metrics. The application of the FIR filter is extended to a more complex DSP algorithm such as an adaptive equalizer where the framework methodology is applied to search the design space for efficient equalizer architectures. A more specific DSP algorithm is presented as a third case study where the framework design options allow a designer to select multiple types of optimizations suitable for the hardware implementation of a discrete wavelet transform system. The results of this work illustrate the efficiency in refining a computationally intensive algorithm to synthesizable hardware implementations with minimal effort from the designer.

**A Performance Analysis Framework for the Design of DSP Systems**

by

**Ramsey S. Hourani**

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

**Electrical Engineering**

Raleigh, North Carolina

2007

**APPROVED BY:**

| | |
|---|---|
| Dr. William Edmonson | Dr. Zhilin Li |
| Dr. Xun Liu | |
| Dr. Winser E. Alexander<br>Chair of Advisory Committee | Dr. W. Rhett Davis<br>Co-chair of Advisory Committee |

## Dedication

To my parents,
Salim and Buran Hourani

my brother,
Sammy

and my sisters,
Eman, May and Leena

## Biography

Ramsey Hourani was born to Salim and Buran Hourani on September 23, 1976 in Ahmadi, Kuwait. He grew up in Kuwait till the age of 14 and then moved to Jordan where he attended The New English School in Amman, Jordan. Upon completion of his secondary education, he moved to the U.S.A. in 1994 to pursue his college and graduate education. In May 1998, he obtained his Bachelors of Science degree in Electrical Engineering from Iowa State University (ISU) in Ames, IA. He worked for two years with the Cellular Subscriber Sector for Motorola as a hardware designer for CDMA wireless systems. He began his Master of Science program in Electrical Engineering at North Carolina State University (NCSU) in August of 2000 which he successfully completed in August 2001. Afterward, he joined the University of North Carolina at Charlotte faculty as a lecturer where he taught undergraduate Electrical and Computer Engineering courses. In January 2003, he entered the Ph.D. program where his research interests included developing efficient hardware architectures that map digital signal and image processing applications onto hardware platforms.

## Acknowledgements

I would like to thank my advisor and mentor, Dr. Winser E. Alexander for his patience and perseverance, for his encouraging words, his invaluable knowledge, and priceless advice throughout my research. I would also like to thank Dr. W. Rhett Davis for taking the time to listen to my ideas and guide me in fine tuning my work. I also thank Dr. William Edmonson, Dr. Zhilin Li, and Dr. Xu Liu for their assistance throughout my dissertation research. I would also like to thank all my instructors who helped me in all aspects of my education.

I cannot neglect the brotherhood I developed among my fellow colleagues through the course of my research. I would like to thank Lalit Ponnala for reviewing my papers, Cranos Williams for his challenging games of racket ball, YoungSoo Kim for our countless discussions, Ishita Dalal for implementing a number of my ideas, Ravi Jenkal for his constructive feedback regarding my work, and my close friend Senanu Ocloo for encouraging me through those days of despair. I would also like to thank Lisa McGee, Sandeep Hattangady, Maitrik Diwan, ChuanHua Xing, Julian Taylor, Treshauna Wright, Musaab Mohammedali and Gary Charles.

My life-long friends whom I have grown to respect more and more with every passing day always encouraged me to pursue this degree from the start. I would like to thank my close friend Clint Stanerson and his wife and children. I would also like to thank Mezyad Ammoura, Omar Masri, Mohammed Al-Najjar and their families. I also thank my dear friend Ayman Zohbi who has looked up to me and listened to my words of advice when he decided to pursue his PhD. All my friends have contributed in some part to the success of my work.

I would like to acknowledge and thank my parents Salim and Buran for their constant prayers for me. I would also like to thank my brother Sammy for his encouragements and my sister Eman (Mimi) for listening to me vent during my days of hardship. I would also like to thank my sister Leena, her husband Mathew, and her two sons Ramsey and Sammy for their kind words of encouragement. I also thank my sister May, her husband Samir and her precious son Ramsey for shining light into my days as a student. Last but not least, I thank all my extended family for wishing me luck throughout my studies.

Above all, I thank God who provided me with the guidance to successfully pursue this degree. I could not have completed this work without His companionship.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **ASIC** | Application Specific Integrated Circuit |
| **BER** | Bit Error Rate |
| **CDF** | Control Data Flowgraph |
| **CLB** | Configurable Logic Block |
| **DCT** | Discrete Cosine Transform |
| **DF** | Direct Form |
| **DFF** | D-type Flip Flop |
| **DSP** | Digital Signal Processing |
| **DWT** | Discrete Wavelet Transform |
| **EDA** | Electronic Design Automation |
| **FDA** | Filter Design and Analysis |
| **FIR** | Finite Impulse Response |
| **FFT** | Fast Fourier Transform |
| **FPGA** | Field Programmable Gate Array |
| **FSM** | Finite State Machine |
| **GDS** | Graphic Data Systems |
| **GUI** | Graphical User Interface |
| **HLS** | High Level Synthesis |
| **IC** | Integrated Circuit |
| **IIR** | Infinite Impulse Response |
| **IP** | Intellectual Property |
| **ISI** | Inter-Symbol Interference |
| **ITRS** | International Technology Roadmap for Semiconductors |

| | |
|---|---|
| **LMS** | Least Mean Square |
| **MAC** | Multiply/Accumulate |
| **MSB** | Most Significant Bit |
| **MSE** | Mean Square Error |
| **OSCI** | Open SystemC Initiative |
| **PAF** | Performance Analysis Framework |
| **PE** | Processing Element |
| **QMF** | Quadrature Mirror Filter |
| **RTL** | Register Transfer Level |
| **SFG** | Signal Flow Graph |
| **SIR** | Signal-to-Interference Ratio |
| **SNR** | Signal-to-Noise Ratio |
| **SoC** | System-on-Chip |
| **SOS** | Second Order Section |
| **TF** | Transpose Form |
| **TLM** | Transaction Level Modeling |

# List of Symbols

| | |
|---|---|
| $\alpha$ | Switching Activity Factor |
| $\lambda$ | Technology Feature Size |
| $A$ | Silicon Area |
| $AT$ | Area Efficiency (Area-Delay Product) |
| $AT^2$ | Area-Delay Squared Product |
| $C_1$ | Filter Input Scale Factor |
| $C_2$ | Filter Coefficient Scale Factor |
| $C_3$ | Filter Output Scale Factor |
| $C_{switch}$ | Switching Capacitance |
| $D$ | Filter Down-sample Rate |
| $f_{op}$ | Design Operating Frequency |
| $f$ | Fixed-point fractional size |
| $J$ | Linear-phase FIR Filter Length |
| $K$ | Filter Signal Length |
| $L$ | Hardware Latency |
| $M$ | FIR filter order |
| $N$ | IIR filter order |
| $P_{dynamic}$ | Dynamic Power Dissipation |
| $P/A$ | Power Density |
| $PT$ | Power Efficiency (Energy) |
| $PT^2$ | Energy-Delay Product |
| $Qx.y$ | Fixed-point representation ($x$-integer, $y$-fraction) |

| | |
|---|---|
| $R$ | Design Throughput |
| $T$ | Unit-Sample Response |
| $T_d$ | Design Delay |
| $T_{d,max}$ | Critical Path Delay |
| $U$ | Filter Up-sample Rate |
| $V_{th}$ | Threshold Voltage |
| $V_{DD}$ | Supply Voltage |
| $\omega$ | Angular Frequency ($rad/s$) |
| $w$ | Fixed-point word size |
| **S** | Symbol description |

# Chapter 1

# Introduction

Silicon process technology has greatly advanced since the discovery of the transistor in 1948 by John Bardeen and Walter Brattain at the Bell Telephone Laboratories [1]. In 1965, Gordon Moore predicted that the number of transistors per integrated circuit (IC) would continue to double every eighteen months or so [2]. Until recently, a system required multiple IC's on a single board, each IC performing a certain application. Technology feature sizes have continued to decrease to the stage where what was once designed using several IC's can now be designed on a single IC consisting of billions of transistors. The design of a system on a single chip is referred to as a System-on-Chip (SoC). The efforts of designing such complex digital circuits has also increased over the years, which paved the way for developing computer-aided design (CAD) and electronic design automation (EDA) tools and methodologies. These methodologies provide system designers the means for continuously refining abstract design specifications to realizable hardware and software blocks. The methodologies continue to increase in complexity as the functionality of the systems grow. Modifying the design methodologies and design tools may become impractical or inefficient to meet the implementation gap. Therefore, designers are forced to develop new and improved CAD tools to cater to complex designs.

The successful development of many systems requires a high design productivity to deal with the huge complexities of the system while using limited design resources and tools. $C/C^{++}$ is a tool of choice among many system designers, which is the building block for a system at the functional or algorithmic level. Designers are capable of representing

the system at multiple levels of design abstraction using extensions of $C^{++}$, which is an important strategy for the successful design of a system [3]. System designs can be refined through different levels of abstraction. The design details become more visible as the design progresses from a higher level of abstraction to a lower level. This allows the system designers to either refine the design to meet the system specifications or to modify the specifications to meet realizable design performance constraints at any level of abstraction.

Digital signal processing (DSP) algorithms are examples of computationally intensive systems that can perform better when mapped onto a hardware platform. A computationally intensive algorithm requires several iterative stages of coding, designing and simulation in order to refine it to a synthesizable hardware block as Figure 1.1 illustrates [4]. MATLAB, $C/C^{++}$ or a combination of the two are commonly used as a high-level language to describe a DSP algorithm. Designers use performance criteria set by the system specifications to determine the partitioning of a DSP algorithm into software and hardware components. Parts of the algorithm that are refined to hardware components undergo rigorous design and analysis procedures in order to ensure that the hardware performance metrics meet the system specifications. Designers model the hardware algorithm at different levels of abstraction until the design is described at the synthesizable register transfer level (RTL). One of the bottlenecks in the hardware refinement process is manually designing the DSP hardware model while maintaining a high level of performance. Hardware designers manually and iteratively add design details to the point that commercial Synthesis tools, such as Synopsys Design Compiler [5], can further refine the design to a level suitable for IC fabrication. Refining all or parts of an algorithm to an efficient hardware implementation requires the efforts of several groups of designers. Each group tends to be proficient at describing the design at a particular level of abstraction. Designers that adopt the refinement process of Figure 1.1 are faced with the challenge of designing an efficient hardware implementation in a reasonable time frame. The expertise gap between the different groups of designers hinders the task of designing an efficient hardware architecture that meets both time-to-market constraints and system performance specifications.

Algorithm developers and hardware designers are therefore faced with an important design concern: how to get from a DSP algorithm to a hardware implementation quickly, and how to determine if that hardware implementation performs well. Currently, there is an abundance of design methodologies and tools, some of which have made it into commercial hardware synthesis companies, that refine a high-level DSP routine to a synthe-

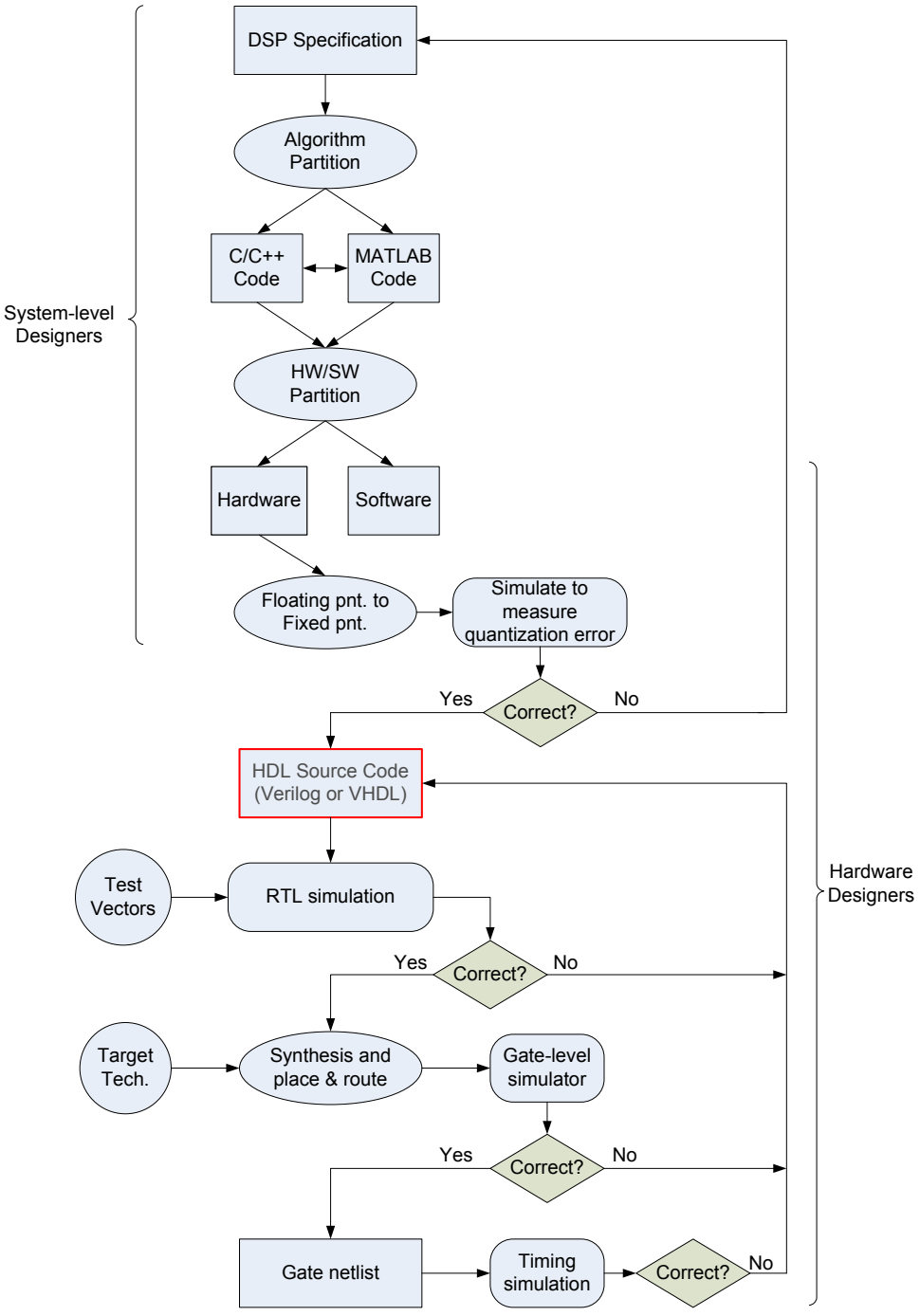



Figure 1.1: Hardware refinement process: from algorithm to netlist.

sizable hardware design. However, most of these tools overlook the importance of assessing the quality of the hardware design to determine it's efficiency in performance. Designer either have to contend with the hardware designs generated by these tools, or take it upon themselves to seek that golden hardware design that stands out among the rest in the design space.

## 1.1 Overview

DSP algorithms can be refined to run on general purpose processors, digital signal processors, field programmable gate arrays (FPGA), or application specific integrated circuits (ASIC). The choice of platform depends on cost and performance constraints set by the complexity of the algorithms and by the system designers who refine the algorithms. Figure 1.2 illustrates an overview for comparing the cost and performance for different platforms [6], where cost refers to the time and effort required to develop the final product and performance refers to throughput, area, and power dissipation of the design. An example of some performance numbers that support the trend in Figure 1.2 can be obtained from [7].

General purpose processors are highly flexible, allowing designers to conveniently implement all or parts of a system in a relatively short time frame. However, the efficiency at which these processors can implement the algorithms is generally low, accounting for low throughput and high energy dissipation. A DSP algorithm can be refined such that it is mapped onto on FPGA or fabricated into an ASIC. For such platforms, the hardware implementation generally exhibits higher performance than the general purpose processors, but at a cost of limited flexibility. Additionally, mapping a DSP algorithm to an application specific hardware block requires the efforts of several designers, which increases design cost and prolongs design times. Therefore, designers would like to achieve a compromise between design flexibility, refinement effort and hardware performance.

One approach to improve the efficiency and performance of DSP hardware blocks is to employ a design methodology that combines several programming languages and EDA tools capable of efficiently generating hardware blocks and assessing the performance. The main advantage of this approach is to minimize the gap between the high and low levels of abstraction. MATLAB and $C/C^{++}$ seem to be the programming languages of choice among system designers for digital signal and image processing applications. Therefore,

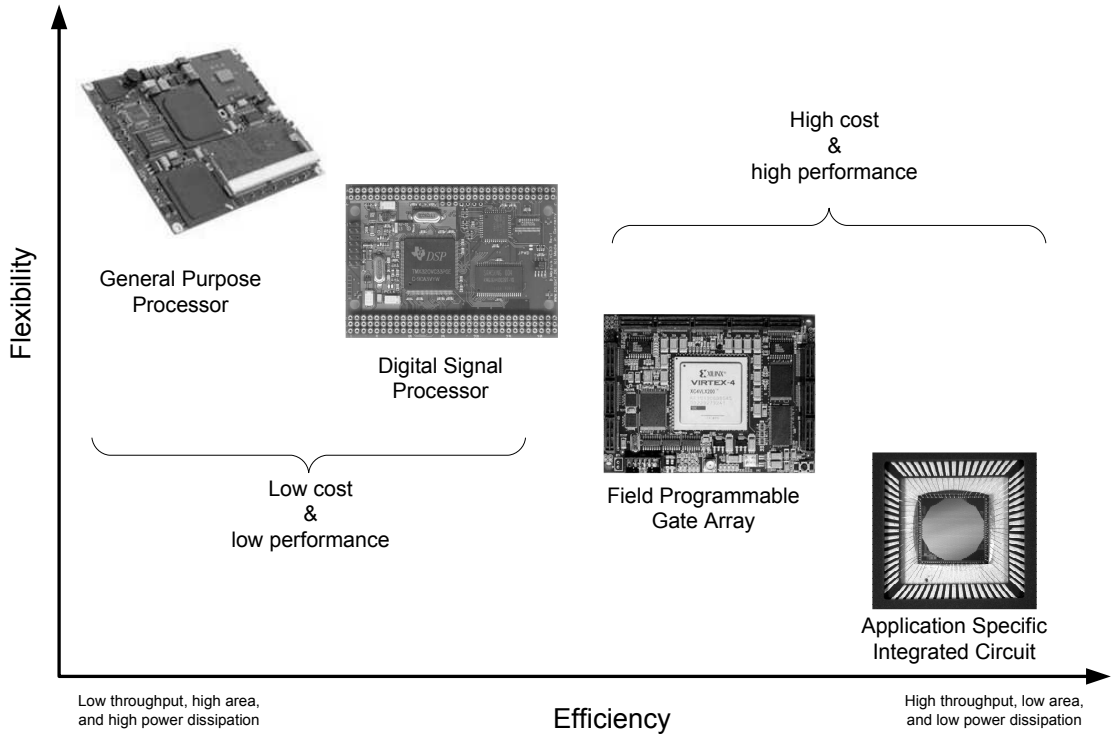



Figure 1.2: Flexibility vs. efficiency , *source* [7]

an extension to $C^{++}$ can be established to allow the system designers to use it at multiple levels of abstraction. SystemC has been developed and is supported by an industry consortium known as the Open SystemC Initiative (OSCI) [3]. It consists of a number of major EDA companies making SystemC attractive for system designers to progressively refine algorithms and systems to lower levels of design detail within a single environment.

## 1.2 Motivation

The possibilities of modeling a functional algorithm at the hardware level are so great that the efforts spent in designing and analyzing the hardware architectures significantly reduces designer productivity and increases time-to-market. One of the main obstacles designers face when creating and analyzing the DSP hardware designs at the RTL is an increase in refinement efforts and an increase in synthesis times. The International Technology Roadmap for Semiconductors (ITRS) calls for design tools and methodologies

to support approaches for designing at the system-level [8]. Two important levels of this approach are system-level synthesis and Intellectual Property (IP) reuse. The cost and effort for generating such tools and IPs are so great that they are not available to the general user. As a result, designers continue to create more tools that are specialized for a small user base. The resulting confusion in using these tools increases the design complexity and degrades the quality of the hardware implementations.

DSP algorithm developers, as well as hardware designers, concentrate their efforts on establishing methodologies that efficiently refine a DSP algorithm to a synthesizable hardware design. Architectural diversities expand the design space for a signal processing algorithm to tens of realizable hardware designs. Varying technology size and supply voltages further extends the design space to hundreds of possible design permutations, each with a unique set of area, throughput and power metrics. This leads to performance tradeoffs in the design space with no accepted way of quantifying performance efficiency using a single cost function that combines area, throughput and power. Designers are therefore forced to exhaustively search the design space looking for hardware designs that perform well and meet final design specifications, a process which also prolongs design time and degrades productivity.

The goal is therefore to develop a framework that efficiently generates hardware designs for DSP applications while simultaneously minimizing area, delay and power for a given set of design constraints. An important consideration in the development of such a framework is to include both DSP algorithm developers and hardware designers as the primary users. This is accomplished through the use of EDA design tools and programming languages commonly utilized by both groups of designers. Selecting the appropriate tools is important in reducing the design gap between the different groups of designers.

## 1.3 Contributions of This Work

This work focused on developing a design flow that generates hardware blocks for common DSP functions and guides the designer through selecting an efficient hardware implementation. The flow starts at the specification level where limited hardware parameters are available to the designer. The goal was to reduce the tasks of manually and iteratively designing hardware architectures for DSP functions in order to meet system specifications.

This work was developed into a design and performance analysis framework (PAF) that facilitated the refinement process of an algorithm to a set of synthesizable hardware blocks. The resulting framework can be used to guide the designer through selecting a hardware implementation that performs well for a specific application. The hardware designs generated by the framework are suitable for mapping DSP algorithms onto ASICs and FPGAs.

This framework was established with the high-level signal processing designer in mind, who we refer to as the system designer. The framework allows the system designer to select the type of signal processing function he or she wishes to refine and analyze. Distinctive from other frameworks and hardware IP generators, our framework generates tens of different implementations for a single DSP function along with the performance metrics. This process tends to take several weeks or months if the tasks of design and analyses were performed manually. In such cases, the conventional design methodologies would require the DSP algorithm to be handed over to hardware designers who then describe the hardware blocks based on the high-level descriptions. The hardware designers possess the necessary skills to include optimizations, such as pipelining and parallelism, that can be used to decrease design cost, reduce power dissipation, and increase throughput. However, any deviation from the initial hardware design would need to be approved by the system designer for functional and algorithmic verification. For example, a Direct Form (DF) finite impulse response (FIR) filter is a direct hardware implementation for the filter used in a least mean square (LMS) adaptive channel equalizer [9]. However, it was shown in [10] that a Transpose Form (TF) FIR filter exhibits better throughput than the DF FIR filter. A hardware designer wanting to use a TF FIR filter in place of a DF filter would unknowingly alter the algorithm of the adaptive equalizer. Haykin showed that, depending on the statistics of the signals in the adaptive filter system, a TF FIR filter may be more desirable than a DF FIR filter when considering algorithmic performances such as convergence speed [11].

A secondary goal of this framework was to introduce concepts inherent to both signal processing system designers and hardware architects while utilizing a single design environment. Algorithm developers and system designers may not acquire the necessary hardware expertise to realize hardware efficiencies from a higher level of design detail. That task is typically left to the hardware designer to optimize the architectures. However, hardware optimizations may go unnoticed if there is limited communication between the system designers and the hardware architects. Therefore, this work was developed to accommodate designers with different levels of hardware expertise. DSP algorithm developers with lim-

ited hardware experience can use the framework to analyze low-level hardware optimizations earlier in the design process. This allows them to provide the hardware designers sufficient details for the appropriate low-level optimizations. Alternatively, hardware designers can use this framework for generating IP cores that are optimized for specific DSP applications while maintaining correct input/output behavior of the DSP algorithm.

This work was initially developed for designing and analyzing the performance of hardware structures for FIR filters. We then considered DSP applications that required FIR filters as a major component of the overall system such as adaptive equalizers and quadrature mirror filter (QMF) banks. The framework currently generates performance reports for metrics such as design area, critical-path timing, throughput, hardware latency, computational precision, and power dissipation. Additionally, the framework provides the user cost functions that evaluate and assess the performance of the hardware structures within the design space. The performance metrics and cost functions can be used to assist the designer in making architectural selections for hardware blocks when designing larger DSP systems.

## 1.4 Organization

Chapter 2 presents an overview for a subset of design flows, CAD tools, EDA tools, and frameworks used to support designers in constructing hardware architectures for DSP algorithms. It categorizes the different flows and tools, highlighting the merits of each and how our framework addresses some of their deficiencies.

Chapter 3 introduces our design and performance analysis framework, along with the design methodology we developed for generating synthesizable DSP hardware IP blocks. The performance metrics and cost functions we use as part of our framework are discussed in greater detail. The chapter also describes the process we adopted for efficiently estimating the performance metrics and generating the pertinent cost functions for the hardware designs.

Chapter 4 demonstrates the merits of our framework using a simple DSP function such as a digital FIR filter. It begins by presenting basic hardware structures for FIR filters and later describes the different types of low-level optimizations that improve the performance of the filter design. Next, the chapter describes the process of using the

performance metrics and cost functions to guide a system designer in selecting a hardware implementation that performs well in the design space for specific performance constraints.

Chapters 5 and 6 illustrate how we use our framework to generate and analyze hardware designs for other, more complex, DSP algorithms that require FIR filters as a major component of the design. Chapter 5 uses the case study of adaptive channel equalizers whereas Chapter 6 uses the case study of discrete wavelet transforms. Both chapters highlight the appropriate types of optimizations, such as pipelining, parallelism, and/or voltage scaling, that improve the performance of the hardware designs in terms of area, throughput and power dissipation.

Chapter 7 evaluates the effectiveness of utilizing our performance analysis framework. The chapter illustrates the pertinence of the performance metrics and the usefulness of the cost functions for analyzing the hardware design space. The chapter also analyzes the accuracy and efficiency for refining a DSP algorithm to a hardware block that performs well.

Chapter 8 highlights how well our framework addressed two main design goals that algorithm developers and hardware designers are concerned with: 1) how to efficiently refine a high-level DSP design specification to a synthesizable hardware block and 2) what's an effective method for assessing the quality of the hardware architectures in the design space. The chapter concludes by examining how this work could be extended for generating hardware designs for other computationally intensive DSP functions. Finally, the chapter introduces the preliminary steps for pruning our framework to refine basic DSP algorithms into efficient hardware designs and mapping the structures onto FPGAs.

# Chapter 2

# Related Work: Refining DSP Algorithms to Hardware

This chapter presents an overview of relevant work in refining high-level algorithms to low-level hardware implementations. The chapter begins by mentioning the important levels of design abstraction suitable for refining a computationally intensive DSP algorithm to a synthesizable RTL model. Next, the chapter discusses the importance of each level of abstraction and highlights the appropriate EDA tools for designing at that level. We present a survey of tools described in the literature for algorithmic refinement, and mention several commercial-based high-level synthesis tools that are used to generate hardware designs from high-level code descriptions.

## 2.1 Levels of Design Abstraction

The process of refining a DSP algorithm to a synthesizable hardware model is iterative, wherein the system level designer initially models the algorithm using a high-level language such as $C^{++}$ or MATLAB [12]. The algorithmic model forms the basis for verifying the correct behavior of the low-level designs. Both algorithm developers and hardware architects determine the necessary design parameters and performance constraints for the

overall signal processing application. These constraints ensure that the hardware designs are feasible in terms of IC cost, throughput performance, and power dissipation, especially since advances in technology are paving the way for smaller, more complex portable devices.

Distinguishing between the data communication and the computational elements of a system facilities the process for refining a high-level algorithm to hardware. The complexity of the algorithms and their applications means that different designers require different methods for separating the details of a design. Cai *et al.* [13] separated the design space into communication and computation axes, while establishing different levels of design abstraction. They defined six levels of design detail, four of which were classified as transaction level models (TLM). In the TLM, the details of computation are hidden from the details of communication. Figure 2.1 shows the system modeling graph and the transitions between the different levels of communication and computation defined by Cai.

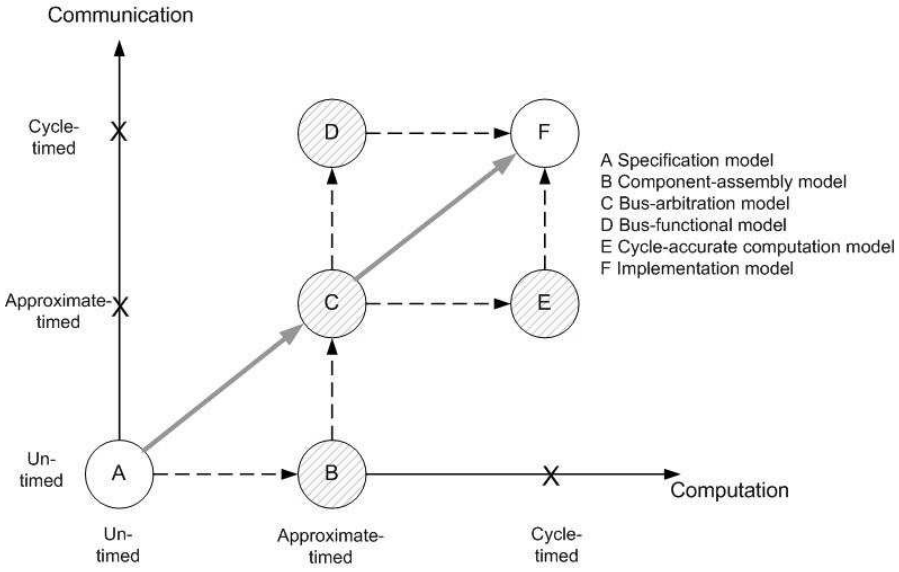



Figure 2.1: System modeling graph, *source* [13]

The $x$-axis on this graph represents the computations performed by the components of a design whereas the $y$-axis represents the communication between the components. Our work focuses on four levels of design detail that address the computational details for a design. The **specification level** is used to model the behavioral functionality of the

design with limited communication and architecture detail. MATLAB and $C/C^{++}$ are appropriate programming languages for efficiently modeling the behavioral functionality of a DSP design. The **bus-functional level** models cycle accurate communication of data, but approximates the execution time of the computational elements. SystemC is one example of a programming language that accurately models designs at that level of abstraction. The **cycle-accurate level** includes adequate details of the processing element in order to accurately estimate both execution times and communication cycles. The details at this level of design include pin accuracies between the computational elements, as well as bit accuracies for internal and external data communication. HDLs are appropriate design languages for detailing the components of this level of abstraction. The **implementation level** models the computational elements at the RTL which are used as inputs to synthesis tools. The synthesized designs are used to estimate area, throughput and power measurements which are essential for assessing the quality of the refined design.

Baganne *et al.* [14] defined four levels of abstraction within the realm of SystemC which is an extension to the $C^{++}$ class libraries [3]. The four levels of abstraction, initially described by Grotker *et al.* in [15], are:

1. **Untimed Functional Level:** Algorithms are modeled using high-level languages with as little code as necessary. The behavior of the algorithm is simulated in order to determine algorithmic optimizations. This level of abstraction is necessary for establishing reference models that are used to verify the functionality of low-level designs. Similar to Cai's specification model, languages such as MATLAB and $C/C^{++}$ can be used for modeling designs at this level of abstraction.

2. **Timed Functional Level:** Some of the functional models are designed with inherent processing or communication delays to analyze the latency effects on the behavior of the system. This analysis governs the allowable latencies and throughput when considering the final hardware implementation. SystemC originated with this level of abstraction in mind, and therefore is appropriately used for the timed functional level.

3. **Bus Cycle Accurate Level:** This level defines the accuracy of the bus interface. The algorithm may be partitioned into untimed behavioral processes. Designers can analyze the functionality of the system in terms of transferable data rates, throughput,

and more accurate clock latencies. This level of abstraction addresses the communication of the system components, and therefore, $C^{++}$-based design languages could be used for this level of abstraction.

4. **Cycle Accurate Level:** This is the lowest level of the design process before synthesis. It describes the modules of the system by means of a data transfer from sequential circuits through combinational logic. This level of abstraction is equivalent to the RTL. Any of the widely accepted HDLs, such as Verilog or VHDL, can be used for modeling designs at this level. However, SystemC has the capability of describing a system using a synthesizable subset of the SystemC language which include data-types that are used for RTL synthesis [16]. This eliminates the requirement for manually translating the high levels of the design from $C^{++}$ to a HDL language, saving on time and reducing interpretation errors.

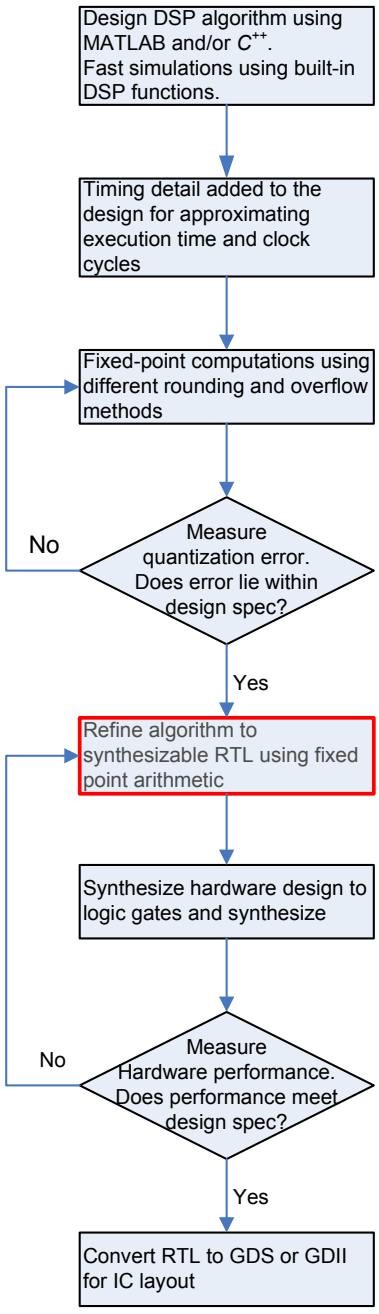
Alternatively to Cai *et al.* and Baganne's *et al.* definitions for the different levels of abstraction, Gemmeke *et al.* [17] also divided the design space but catered the design details more towards DSP algorithms. The **algorithmic level** can be as simple as a few lines of code that equate the output to a function of the input. The signal processing algorithm is initially implemented using floating-point computations for algorithmic verification. MATLAB and $C^{++}$ are attractive choices of tools for efficiently modeling signal processing algorithms and generating reference models. The **arithmetic level** is necessary when analyzing the quantization methods and overflow modes, which are vital when considering fixed-point implementations. DSP algorithm developers and hardware designers proceed to use MATLAB and SystemC for modeling the algorithms at this level of design abstraction [18]. Roy developed automated floating-point to fixed-point converters that address pertinent concerns when dealing with finite precision computations [19] [20]. The next level of abstraction defined by Gemmeke is the **architectural level**, where signal flow graphs (SFG) are used to show the transition of signals through a set of registers and mathematical blocks. This level of abstraction bridges the gap between the signal processing algorithm developer and the hardware designer. Low-level hardware optimizations are primarily the task of the hardware designer and therefore, a HDL is an appropriate design mechanism for the architectural level. The flexibility in design choices is reduced at the lower levels of abstraction. For example, at the **logic level**, the hardware designers can chose from a limited set of computational operations for implementing the signal processing algorithm. The

hardware performance of the algorithm is effected by the type of arithmetic blocks used. One of the lowest levels of abstraction is the **circuit level** which determines the depth of the combinational logic between sequential circuits. Designers utilize synthesis CAD tools, such as Synopsys Design Compiler [5], for generating the final chip layout from the HDL design which is described in Graphic Data System (GDS) format. The GDS or GDSII is a database format and is the standard format for fabricating an IC.

Figure 2.2 summarizes the levels of abstraction presented by Cai [13], Baganne [14] and Gemmeke [17] which we consider in our work for refining a DSP algorithm to hardware. The refinement process is generally iterative with design considerations made after each level of refinement. The separation between the algorithm developer and the hardware architect occurs when transitioning from the functional levels to the cycle accurate levels. This is similar to the hardware refinement process illustrated in Figure 1.1 of Chapter 1. The bottleneck in design productivity occurs at that critical transition where hardware designers typically explore multitudes of hardware implementations, seeking the most efficient structure in the design space.

## 2.2 EDA Tools for Modeling Hardware Designs

The goal of most emerging SoC designers is to provide an architecture that is specialized for a certain class of applications and can therefore deliver a higher level of throughput at lower power consumption than previously possible. The process for refining a DSP algorithm to a synthesizable hardware implementation requires several EDA tools, each one catered for a specific level of design abstraction. The separation in these EDA tools often results in large design gaps between the different levels of abstraction and therefore makes communication difficult between the different groups of designers. The overall effect reduces the design productivity. The right choice for EDA and CAD tools is necessary to ensure that designers can efficiently refine a DSP algorithm to a hardware model. We use a subset of academically and industrially accepted programming languages, EDA and CAD tools that aid us in refining DSP algorithms to synthesizable hardware models. Additionally, we use these tools to facilitate the process of searching the design space for hardware structures that perform well for specific design constraints.




| Cai | Gemmeke | Baganne |
|---|---|---|
| | Algorithmic Level | Untimed Functional Level |
| | | Timed Functional Level |
| Bus Functional Level | Arithmetic Level | Bus Cycle Level |
| Cycle Accurate Level | Architectural Level | Cycle Accurate Level |
| Implementation Level | Logic Level | |
| | Circuit Level | |

Figure 2.2: Levels of design abstraction: from algorithm to hardware layout.

### 2.2.1 MATLAB

The majority of DSP algorithm developers use MATLAB [12] as the initial stages for their system designs. MATLAB has gained popularity over the past several decades for modeling image and signal processing applications. MATLAB is an appropriate choice of tools for bridging the gap between the signal processing developers and hardware designers. The technological advances in digital systems permit complex signal processing algorithms to be implemented in hardware. As recent as a few years ago, MATLAB made it's way in the design flow for algorithm refinement and hardware modeling of ASIC and FPGA based designs [21].

### 2.2.2 SystemC

SystemC is a $C^{++}$ class library and a methodology that allows system designers to effectively create a cycle-accurate model for software algorithms, hardware designs, and interfaces for SoC and system-level designs. The designer utilizes SystemC to create a system-level model for the design and perform simulations in a relatively short time frame to validate and optimize the designs. Additionally, SystemC allows the designers to explore various algorithms and provide the hardware and software designers an executable specification of the system. The SystemC class library provides the necessary constructs to model system architectures which include hardware timing, concurrency and reactive behavior that may not be found in standard $C^{++}$. The $C^{++}$ object-oriented programming language provides the ability to extend the language through classes, without adding additional syntactic constructs.

#### SystemC Highlights

SystemC's hardware modeling capabilities allow designers to accurately design and synthesize hardware components using a $C/C^{++}$ design environment. It is unnecessary for the designers to migrate to a different design environment when refining all or parts of an algorithm to hardware components. This reduces errors that may be induced during the algorithm refinement process and therefore bridges the gap between the algorithmic and architectural levels of design abstraction. The following are highlights of SystemC that allow a designer to model a system at multiple levels of **cycle-accurate** detail, including

the RTL [3].

1. **Modules:**

   Modules are the basic blocks for partitioning a design. They allow designers to break complex systems into smaller, more manageable components, and to hide internal data representations and algorithms from other modules. In SystemC, a module is a hierarchical entity that can have other modules or processes contained in it. Modules contain ports, constructors, data members, function members, and channels.

2. **Processes:**

   Processes are used to describe the functionality and behavior of a module and are therefore contained inside modules. Processes are functions that contain a sensitivity list identified to the SystemC kernel. A process is executed when at least one of the signals in the sensitivity list changes it's value or state. There are three types of processes: `METHOD`, `THREAD`, and `CLOCKED THREAD` process. We primarily focus on `METHOD` processes for the level of design abstraction we adopt in our SystemC hardware designs. The `METHOD` process is equivalent to an `always` block in Verilog RTL code.

3. `METHOD` **Processes:**

   A `METHOD` process is the simplest form of SystemC's processes. Their execution is triggered by a signal event contained within the method's sensitivity list. The method is invoked when a signal in the method's sensitivity list changes state, i.e. an event occurs. Once the method is invoked, it continues running until a "return" is encountered or the end of the method function is reached, after which control is then passed back to the simulation kernel.

4. **Constructors:**

   Constructors are used to allocate memory and to instantiate the instances of a module. i.e. initialization of the signals and ports. The constructor creates the internal data structures that are used for the module and initializes these data structures to known values. The module constructors in SystemC are implemented such that the instance name of the module is passed to the constructor at instantiation time. A module destructor is provided in SystemC to free the allocated memory when the object associated with a module instantiation is destroyed.

5. **Ports:**

   Ports provide modules with a means for communicating with their surroundings by transferring data from processes within a module to processes within other modules. There are three types of ports defined in SystemC: `input`, `output`, and `bidirectional`. Ports are created from pre-defined SystemC template classes. A type passed as a template argument defines the type of data exchanged within the ports. This particular feature provides an advantage of using SystemC as a HDL over other HDLs such as Verilog or VHDL. Designers have the option of abstracting bit-true accuracy from hardware designs by utilizing other $C^{++}$ data-types or even user-defined data-types.

6. **Signals:**

   SystemC signals are similar to ports. They are used to provide data paths for connecting processes within a module. Signals are also used to connect modules at the top level of a design to modules at lower levels of the design hierarchy. Signals differ from ports in that they do not have a mode attribute and therefore cannot determine data flow through them. Direction of data flow is determined by the port to which they are connected.

7. **Rich set of data-types:**

   With the multiple levels of abstraction come SystemC's fixed precision data-types which allow for fast simulation, especially at the higher levels of abstraction. The arbitrary precision types can be used for computations with large numbers, and the fixed-point data-types can be used for bit-true DSP applications. Ports and signals may be defined using standard $C^{++}$ data-types, SystemC data-types, or user-defined types. This specific SystemC highlight is critical in improving the architectural design and exploration process we discuss in Chapter 3.

8. **Cycle-based simulation:**

   SystemC provides a small-scale cycle based simulation kernel that provides high-speed simulations with multiple or derived clocks. Prior to SystemC, a cycle-based circuit designed using an HDL would execute slower than it's system level counterpart written in $C/C^{++}$. SystemC expedites the development of cycle accurate models and the simulation times through the use of template classes. The work of Calazans *et al.* is an example of achieving considerable speedup when simulating a design using SystemC

versus VHDL for different levels of abstraction [22].

### 2.2.3 Verilog Hardware Description Language

Verilog is a HDL with specific syntax and semantics used to accurately model designs at the RTL. It was first standardized in December of 1995 by the Standards Group for Verilog; IEEE 1364-1995 [23]. The design language is commonly used as the back-end for EDA tools that generate synthesizable hardware designs from high-level description languages, as well as the front-end for commercially used synthesis tools such as Synopsys Design Compiler. Designs described at the RTL using Verilog are typically implemented using integer data-types, which provides bit-true accuracy for the hardware implementation.

The performance of a hardware design is governed by the style in which the Verilog code is written. "Behavioral Verilog" is a style of coding that uses Verilog constructs and syntax to model the behavior of a design with fewer detail than at the RTL [24]. The coding style is closer to using a high-level programming language such as *C*. The results of synthesizing Behavioral Verilog depends on the complexity of the design and the coding style, and in some cases, may not be synthesized. However, it is difficult to predict or even determine how the synthesis tools refine the behavioral code to a gate netlist.

### 2.2.4 Synopsys Design Compiler

Synopsys Design Compiler is a logic synthesis tool that synthesizes a design modeled using a HDL, such as Verilog, and generates a gate-level netlist of the circuit using the standard-cell library of IP's provided by the vendor. This netlist of logic gates performs the same function described by the RTL hardware design. Design constraints for area, delay, and operating conditions are used to guide the synthesis tool such that the logic gates are properly assembled to implement the design. The ability for the synthesis tools to meet the design constraints depend on several factors such as the complexity of the design and the description of the system at the RTL. In most cases, the hardware designer needs to verify whether the design constraints are met. While the Synopsys Design Compiler uses optimization algorithms to attempt to meet the design constraints, the hardware designer primarily controls the performance of the design according to his/her method for describing the hardware model.

We utilized Synopsys Design Compiler in conjunction with scripting techniques to specify design constraints and extract area, delay, and power measurements post synthesis. Our synthesis methodology, described in Chapter 3 was modeled using parts of the "Chip in a day flow" that automated the process for synthesizing hardware designs [25] [26]. We used Synopsys Design Compiler to determine the fastest allowable clock frequency by measuring the largest critical path delay in the design. We estimated the design area based on the standard cell blocks used to build the hardware model, and power was determined by simulating the synthesized hardware design and measuring the switching activity of the circuit netlist. Synthesis times depended primarily on the size and complexity of the hardware design. As a benchmark, a 32-tap, 16-bit direct form FIR filter synthesized for minimum delay required approximately 30 minutes using an Intel Pentium-4, 3 GHz CPU Linux machine. This design consists of 32 16 $\times$ 16-bit multipliers, 31 32-bit adders and 31 16-bit registers. We elaborate on the significance of synthesis times in Chapter 7.

### 2.2.5 Synopsys DesignWare DSP IP Library

Signal processing algorithms are refined to synthesizable hardware before they are mapped onto an ASIC. Synthesis tool makers, such as Synopsys, provide parameterized IP cores for basic DSP applications, examples of which include FIR and Infinite Impulse Response (IIR) filters [27]. Table 2.1 lists the DSP components in the DesignWare Building Block IP. The DesignWare DSP components primarily include FIR and IIR filters which are the basic building blocks for many signal and image processing applications.

Table 2.1: Synopsys DesignWare IP DSP components

| DSP Components | Description |
|---|---|
| DW_fir | High-Speed Digital FIR Filter |
| DW_fir_seq | Sequential Digital FIR Filter Processor |
| DW_iir_dc | High-Speed Digital IIR Filter with Dynamic Coefficients |
| DW_iir_sc | High-Speed Digital IIR Filter with Static Coefficients |

Parametrized IP cores reduces the overall design time. However, the Synopsys DSP IP cores lack additional optimization parameters that improve the performance of the design for certain applications. For example, the Synopsys DesignWare digital FIR filter components can be used for high-speed (DW_fir) applications or area-efficient (DW_fir_seq) applications. The high-speed DesignWare FIR filter is implemented using a basic TF structure shown in Figure 2.3 [28]. This structure uses $N$ multipliers and $N-1$ adders in parallel to compute each output, where $N$ is the number of filter taps. The data flow for this structure is one of the simplest forms to implement an FIR filter and requires minimal control logic. The coefficients are applied to the filter serially to reduce the input/output interconnects. This technique is useful for large order filters but may contribute to the overall initiation latency of the design. Alternative structures include a parallel input bus to supply the filter with the coefficients simultaneously.

The area-efficient DesignWare FIR filter is implemented using a single multiplier and adder shown in Figure 2.4 [28] and therefore requires additional control logic for correct signal synchronization and data transfer between the processing elements. This structure uses a multiply-accumulate (MAC) loop to perform the convolutional sum where each output sample is generated every $N$ clock cycles. Therefore, the throughput for this design is $N$-times slower than the former DesignWare FIR filter.

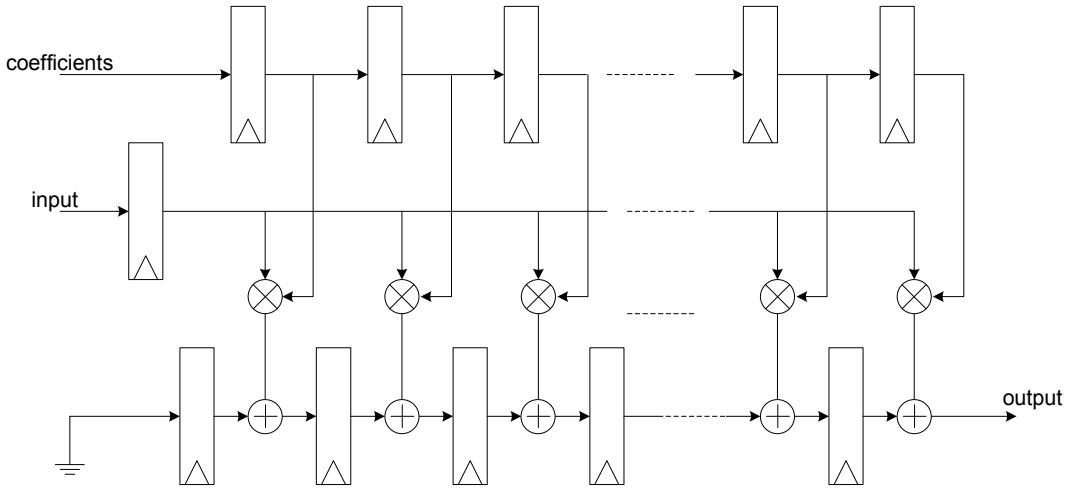



Figure 2.3: Synopsys DesignWare High-Speed Digital FIR Filter.

The correct choice for FIR filter structure depends primarily on the application and the design specifications. The hardware designer cannot alter the structure of the De-

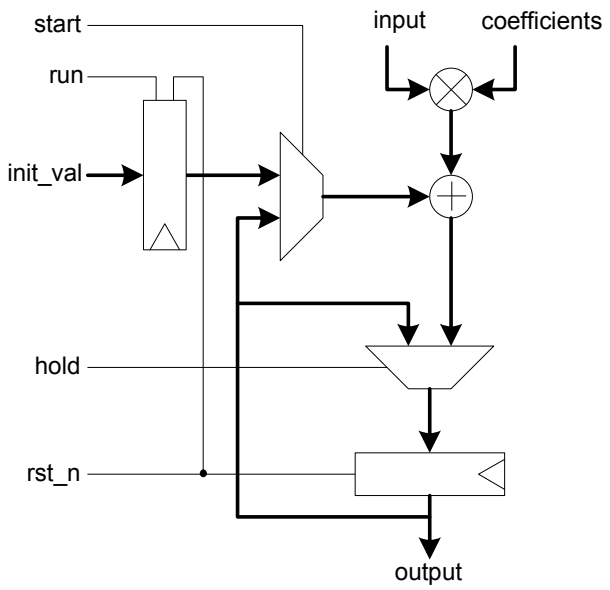



Figure 2.4: Synopsys DesignWare Area-Efficient Digital FIR Filter.

signWare FIR filters to include additional optimizations such as pipelining, parallelism and interleaving. This would require the hardware designer to manually construct the basic filter structure, and then proceed to iteratively modify the structure to implement additional optimizations. Our work focused on expanding the basic filter structures to include architectural optimizations that improve hardware performance based on the signal processing application. A detailed description of design optimizations is discussed in Chapter 4 where we explore the design space for different FIR filter structures.

## 2.3 Design Environments for DSP Hardware Blocks

Algorithm developers and hardware designers work collectively to refine a signal processing algorithm to a realizable hardware implementation. The algorithm developers concentrate on establishing the functionality of the system using high-level languages such as MATLAB and/or $C/C^{++}$. High-level languages permit them to efficiently define system parameters that ensure the algorithmic constraints are met. The figures of merit used to measure the effectiveness of the signal processing algorithm depend on the application. Examples of algorithmic figures of merit when considering image and signal processing

applications include signal-to-noise ratio, signal-to-interference ratio, bit and symbol error rates, and probability of error [29] [30]. Establishing the algorithm parameters as early as possible is detrimental for efficiently designing the hardware components.

The system design process begins once the functionality of the signal processing algorithm is established. During this step, system designers discuss detailed specifications necessary to meet the functionality, performance, cost, and development time for the system. This step involves both the customers and designers in order to accurately reflect the customers requirements in a way that can be clearly followed by the designers during the refinement process. Examples of issues presented during the system specification are the integrity of the signals, size, weight, speed, power consumption and cost for the final overall system. All designers involved in the system development are constrained by time-to-market deadlines to ensure that the final product(s) hits the market at just the right time.

A large gap tends to exist between the signal processing algorithm and the efficient hardware implementation. Hardware designers seek EDA tools and design methodologies that attempt to bridge the gap between the algorithm and the hardware model. However, generating an efficient tool or methodology that spans the entire signal processing design space at its multiple levels of abstraction is almost unrealizable. Therefore, designers are compelled to create in-house tools and methodologies that cater to specific types of signal processing applications. Of the slew of tools and products available to DSP algorithm developers and hardware designers, only a handful have gained popularity in both academic and industrial venues. The following subsections provide an overview of EDA tools and methodologies catered towards refining signal processing algorithms to realizable hardware implementations.

### 2.3.1 Synopsys Module Compiler

Synopsys Module Compiler allows designers to generate high-performance designs for data-intensive applications [31]. The design flow synthesizes the hardware design for ASIC or FPGA implementations. The tool also provides the means for designers to explore architectural variations by analyzing the hardwares performance results in terms of area and speed. Module Compiler enhances productivity by providing a path to the performance of the hardware design within minutes, depending on the complexity of the system. Architectural variations are performed at the arithmetic level by automatically analyzing

the hardware performance subject to different mathematical operators. The synthesis tools returns the optimized mathematical implementations for a particular design.

The designer provides Module Compiler the hardware model for the DSP algorithm which requires a skilled hardware designer. Any variations at the architectural level may result in contradicting performance results. This requires the designer to use Module Compiler once more to optimize the mathematical operators for the modified hardware model. Module Compiler is an example of an EDA tool geared towards increasing productivity by efficiently searching the design space for optimized designs, which is one of the highlights of our work. However, the user is burdened with having to manually model the various algorithms at the RTL in an attempt to obtain a design that has been optimized at both the architectural and circuit levels.

### 2.3.2 HYPER

HYPER is an automated high-level synthesis system that applies architectural optimizations to minimize power dissipation while maintaining acceptable throughput and area measurements [32]. The approach is to explore the design space using flowgraph transformations that alter the performance of a design while preserving the input/output behavior. The tool is primarily utilized for exploring the effects of power dissipation subject to the architectural transformations. Power optimization is achieved using several hardware and signal processing techniques. This includes critical path reduction by transforming a single cascaded structure to a parallel structure, an example of which is illustrated in Figure 2.5 for the case of an FIR filter.

This technique is applicable to structures primarily comprised of processing elements with similar hardware complexity. For example, if we assume that the multipliers in Figure 2.5 are of equal complexity to the adders, which is rarely the case, then the critical path delay is reduced by almost one half in the parallel structure. The clock frequency can be dropped to one half without effecting throughput. A common technique for slowing down the clock frequency is reducing the supply voltage, which in turn reduces the power dissipation. A numeric example is detailed in [32] highlighting this main feature. However, in the more realistic case, the multiplier is of greater complexity than the adder and, therefore, exhibits a much larger critical path delay. In the example illustrated in Figure 2.5, if we assume that the critical path delay of the multiplier is 10 times greater than that of the

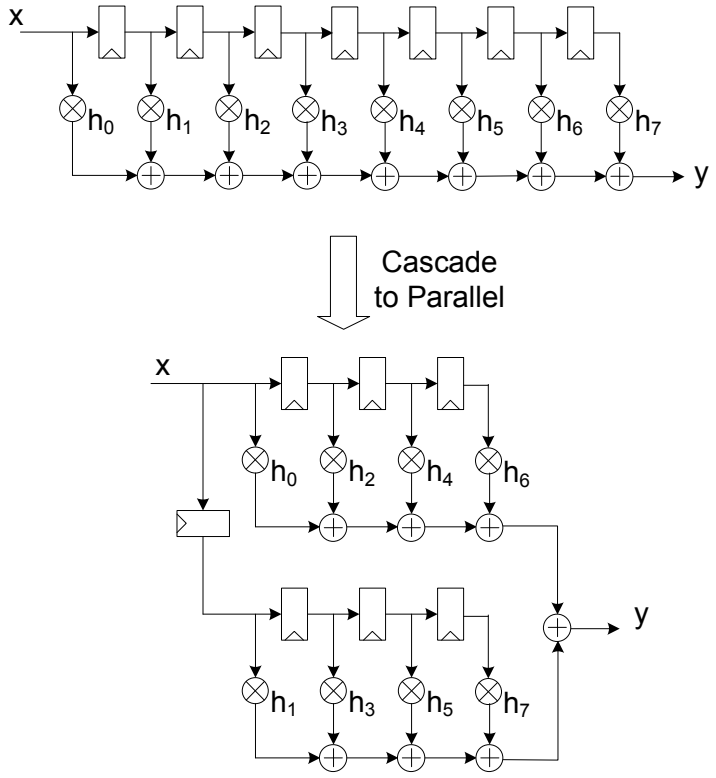



Figure 2.5: Cascade versus parallel structure for FIR filter example.

adder, then the critical path delay for the parallel structure is reduced by approximately $\frac{1}{5}$ the original critical path delay. Therefore, to maintain the same throughput, the clock frequency can only be reduced by 20%, which means that the supply voltage cannot be reduced by much.

More effective techniques for reducing the critical path delay include pipelining [33]. Other methods have combined both parallelism and pipelining to increase the effectiveness of power reduction [34]. While HYPER is useful in efficiently estimating power dissipation, the user is required to provide the tool with the initial hardware model for the signal processing algorithm described at the RTL. The example presented in [32] considered optimizations for the direct form II and cascade FIR filter structures. Unless optimizations are manually implemented, regardless of the architecture, the user has to contend with the limited architectures generated by the tool.

### 2.3.3 FIR Compilers

Hawley *et al.* presented an *FIR compiler* for generating different hardware designs for FIR filters targeting ASICs [35]. Their work discussed methods for improving the throughput of a design, such as pipelining, and decreasing the hardware complexity, such as symmetry properties in linear-phase FIR filters. The compilers focused on optimizing the arithmetic blocks in the filter structures while providing limited variations at the architectural level. The filter structures generated by the FIR compilers analyzed the performance in terms of area and delay without considering the effects of power dissipation. While some optimization techniques improved throughput, their effect on reducing power dissipation was not as effective, which complicated the process for choosing an efficient filter structure.

Utilizing the FIR filter compilers required the users to manually and exhaustively search the design space, a process which prolongs design time. While such tools employed useful hardware and throughput optimizations, most of these techniques required detailed signal processing and hardware design knowledge by the user to determine when to apply certain optimizations without altering the functionality of the design. Additionally, the FIR compilers have not made it into mainstream synthesis tools such as Design Compiler. The development and maintenance cost would be borne by the designers to optimize their filter structures rather than the EDA companies. The design optimizations discussed in Hawley's work remain an important asset to the type of optimizations we consider in our work.

### 2.3.4 SystemC for Hardware Design and Verification

SystemC was developed with both system designers and hardware architects in mind. Baganne *et al.* [14] used a case study for a wavelet based compression system design to illustrate how the different levels of design abstraction effect the simulation speeds for a system. The DWT was implemented using custom-generated FIR filter blocks modeled using different coding styles: functional to RTL. The design flow used for this system was built around the four main SystemC abstraction levels discussed earlier: Untimed Functional, Timed Functional, Bus Cycle Accurate and Cycle Accurate levels.

In Baganne's work, the DWT was integrated and simulated at the four levels of abstraction using SystemC, which provided an ease of transition from one level to the next. Different image sizes were used at the different levels of abstraction and the simulation

speeds for each were recorded. As the level of abstraction progressed from the untimed functional level to the cycle accurate level, the synchronization detail required for proper block communication increased which resulted in an increase in simulation time. The simulation results for the different image sizes and different levels of abstraction are tabulated in Figure 2.6 [14]. The results showed that simulation times increased linearly as image sizes increased, whereas the simulation times increased exponentially with lower levels of abstraction. The DWT system is similar to the level of design complexity for the applications we consider in our work. Therefore, the results of Figure 2.6 provide us with a sense of simulation times we can expect depending on the level of abstraction we model our designs. We have verified, using the design example of adaptive equalizers, that SystemC modeled at higher levels of abstraction simulated faster than Verilog designs modeled at the RTL [36]. Synopsys provides a set of synthesizable data-types that permit designers to model SystemC designs at lower levels of abstraction and synthesize them, which reduces overall refinement effort [16]. Therefore, SystemC is useful in verifying the architectural variations for designs that are modeled at levels of abstraction equivalent to the Verilog RTL.

### 2.3.5 AccelChip

MATLAB is an attractive tool for verifying the functionality of signal processing designs at different levels of abstraction. Including MATLAB in the design flow for generating and analyzing hardware designs bridges the gap between algorithm development and IP core generation. Researchers at Northwestern University have provided design methodologies and EDA tools for translating algorithms described in MATLAB to hardware models described using an HDL such as Verilog or VHDL. Examples of this include the work of Banerjee *et al.* [37] [38] [39] [21]. They developed tools and integrated verification flows for automatically mapping certain MATLAB functions onto FPGAs and ASICs using directives embedded within the MATLAB code. Their work founded the Xilinx tool "AccelDSP$^{TM}$ Synthesis Tool" [40] (formally "AccelChip"). Such tools were able to explore architectural variations for hardware designs, described in MATLAB, using metrics such as area, delay, and quantization error, which reduced the hardware development efforts. The hardware designs generated by these tools were ideal for general-purpose signal processing applications. Low-level optimization techniques for specific applications required hardware

| Abstraction Level | Image size (octets) | | | |
|---|---|---|---|---|
| | 16385 | 100K | 500K | 1M |
| Untimed Functional (UT) | 0.1s | 0.6s | 2.9s | 5.8s |
| Timed Functional (TF) | 2.4s | 15.0s | 70s | 146s |
| Bus cycle Accurate (BCA) | 52.7s | 312s | 1581s | 3192s |
| Cycle Accurate (CA) | 1H 23 | 3H 23 | 10H 50 | 17H 20 |

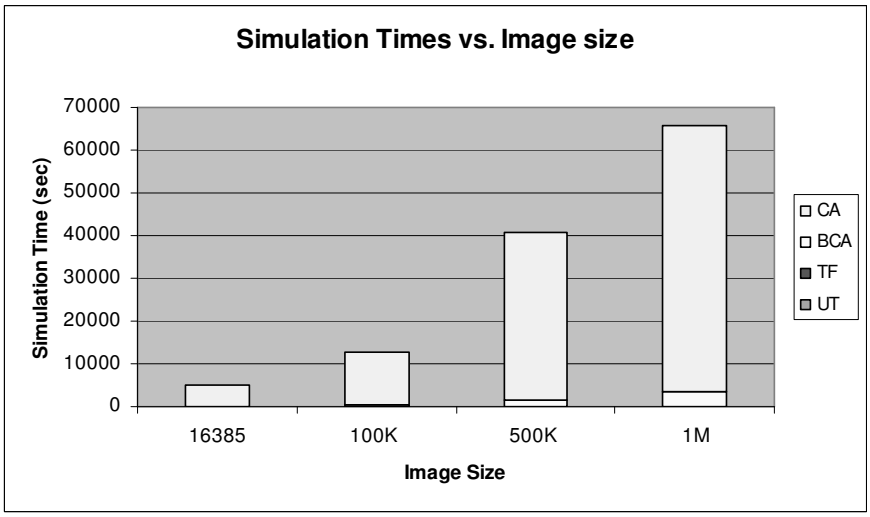



Figure 2.6: Simulation times for different image sizes at different levels of design abstraction, *source* [14]

designers to manually describe alternative structures using MATLAB as a hardware description language which may not be intuitive to hardware designers.

Figure 2.7 illustrates an overview of the "AccelChip" tool for converting a DSP algorithm, coded in MATLAB, to a synthesizable hardware description. The flow begins with a detailed MATLAB description for the signal processing algorithm with embedded directives that control the generation of the hardware code. Built-in MATLAB functions that efficiently implement a DSP algorithm, such as *conv* or $filter$, were not recognizable by the compiler. Therefore, such functions had to be elaborated into detailed MATLAB instructions that the compiler could translate. It is at that point where the algorithm directly impacts the outcome of the hardware design. Therefore, the "AccelChip" tool permitted the user to control the outcome of the hardware through algorithmic directives. For example, directives within the MATLAB code controlled the unwrap factor for iterative loops to implement parallelism. The advantage of using a MATLAB-to-hardware compiler reduced

the translation effort, but at a cost of detailed hardware description using a language not intended for hardware design. Any architectural optimizations would need to be implemented by a skilled hardware designer that is also knowledgeable of how algorithmic modifications may effect the output integrity.

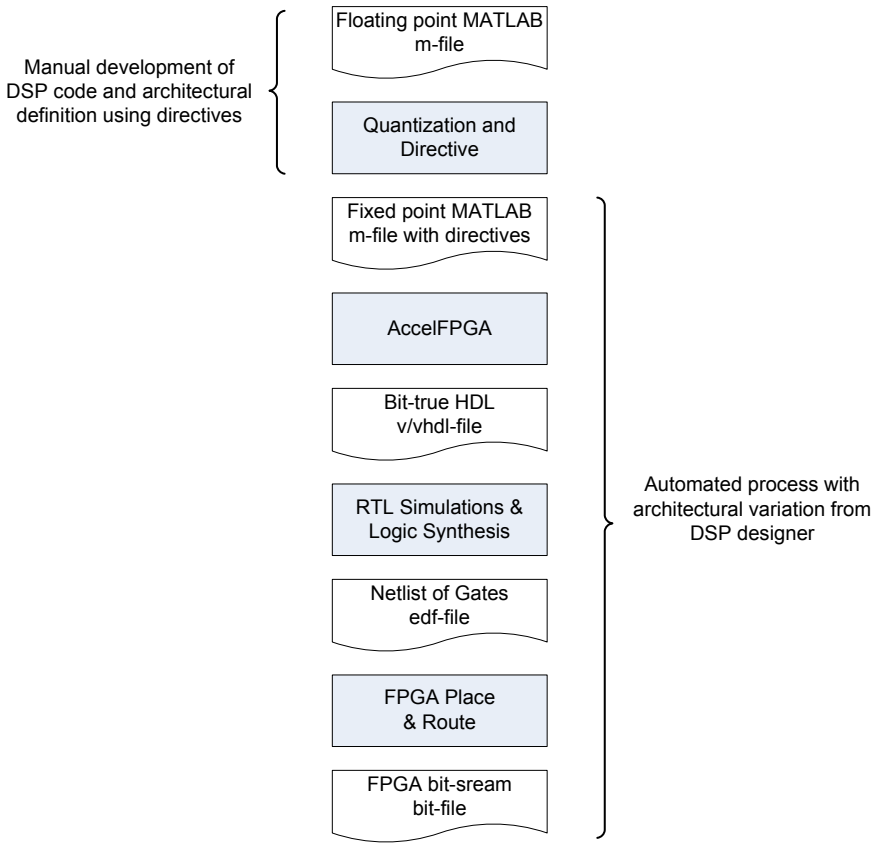



Figure 2.7: Automated design flow for MATLAB-to-hardware conversion, *source* [21].

While the automated conversion of MATLAB algorithms to synthesizable hardware increased design efficiency, the process for evaluating the tradeoffs between different hardware designs remained the responsibility of the user. Variations in simple algorithms, such as the $filter$ algorithm, could generate different performance measures for the hardware implementations. Manual modifications to the MATLAB DSP algorithm to explore the design space potentially increased the design time. The "AccelChip" tool lacked the ability to alleviate the user from manually designing and analyzing the different designs. However, it remains a useful tool for generating an elementary hardware model for the DSP

algorithm in a relatively short time.

## 2.4 Commercial Tools for Hardware Synthesis

High-Level Synthesis (HLS) originated in the late 1980's and early 1990's as a means for generating data paths and finite state machines (FSM) from high-level sequential code. The goal for HLS is to produce a hardware design that implements the behavior of the high-level code while meeting specific latency and throughput constraints. The initial step in HLS is to generate a control data flowgraph (CDF) from sequential code which captures the data dependencies between operations in the program and gives an indication for the degree of concurrency between the available operations. The CDF is then modified to determine the best allocation of resources such as adders and multipliers, and finally resource binding determines which resources should be used to implement each specific operation. One important aspect for HLS is its sensitivity to algorithmic coding styles on performance, area and power.

Behavioral code is one style of coding for HLS which describes the functionality of the design at a more abstract level than RTL. Behavioral style coding allows the use of loops, conditionals, and "wait until clock events" with behaviors embedded between the "wait" events. The advantages of behavioral style coding include a higher level of design with compact code that is easier to debug and faster to run. One example for behavioral coding adopted in the past by hardware designers is Behavioral Verilog [24]. Using Verilog as behavioral code allows hardware designers to use a familiar hardware language in a high-level design setting. However, Behavioral Verilog has not gained popularity since it limits variations to the final hardware design due to the lack of details that describe the flow of data between the processing elements. Additionally, Behavioral Verilog is not intuitive to the RTL coding styles a hardware designer is accustomed to. As a result, a new wave of tools started appearing based on high-level languages such as MATLAB and $C/C^{++}$. This section presents three commercial-based tools oriented toward generating hardware designs from high-level algorithm descriptions.

### 2.4.1 AccelDSP™ Synthesis Tool by Xilinx

AccelDSP™ Synthesis Tool [40] is a high-level MATLAB-based tool used for designing DSP blocks which are then synthesized and mapped onto FPGAs. As part of its design flow, the tool automatically converts the variables within the high-level code from floating-point to fixed-point values, taking the design from the algorithmic to the arithmetic level. Additionally, the tool generates synthesizable hardware code which is accompanied by testbenches for further architectural verification. The testbenches are necessary for verifying the design at the arithmetic level to make sure that design flaws, such as overflow and quantization errors, are addressed and adjusted by the designer. This requires the efforts of both the algorithmic developers and hardware designers to make sure that any modifications do not obstruct the behavior of the algorithm.

System Generator [41], along with Simulink, provide a graphical user interface (GUI) for capturing and verifying the designs components generated by AccelDSP™ Synthesis Tool. It also provides the designer with dozens of DSP hardware IP blocks for constructing all or parts of the system. Examples of these IPs include FIR filters, decimators and interpolators from the DSP library. Most of these IPs are intended for general purpose DSP applications and provide limited means for any architectural modifications which may be applicable to specific applications. Therefore, any modifications to the hardware designs generated by the synthesis tool require a manual redesign of the architecture and reevaluation of its performance. This tool remains invaluable in the efforts of bridging the gap between the algorithmic and cycle-accurate levels of design when considering DSP algorithms.

### 2.4.2 Catapult-C by Mentor Graphics

Catapult-C [42] is a HLS tool from Mentor Graphics that generates cycle-accurate and synthesizable hardware code from an untimed $C/C^{++}$ specification [43] [44]. This tool allows the ASIC and hardware designers to rapidly produce efficient designs for computationally intensive applications such as image and video processing blocks and communication systems. Figure 2.8 illustrates the synthesis design flow used in the Catapult-C tool.

The flow begins from the high-level $C^{++}$ algorithm where simulations are performed to verify the correct functionality of the untimed model. Software constraints are

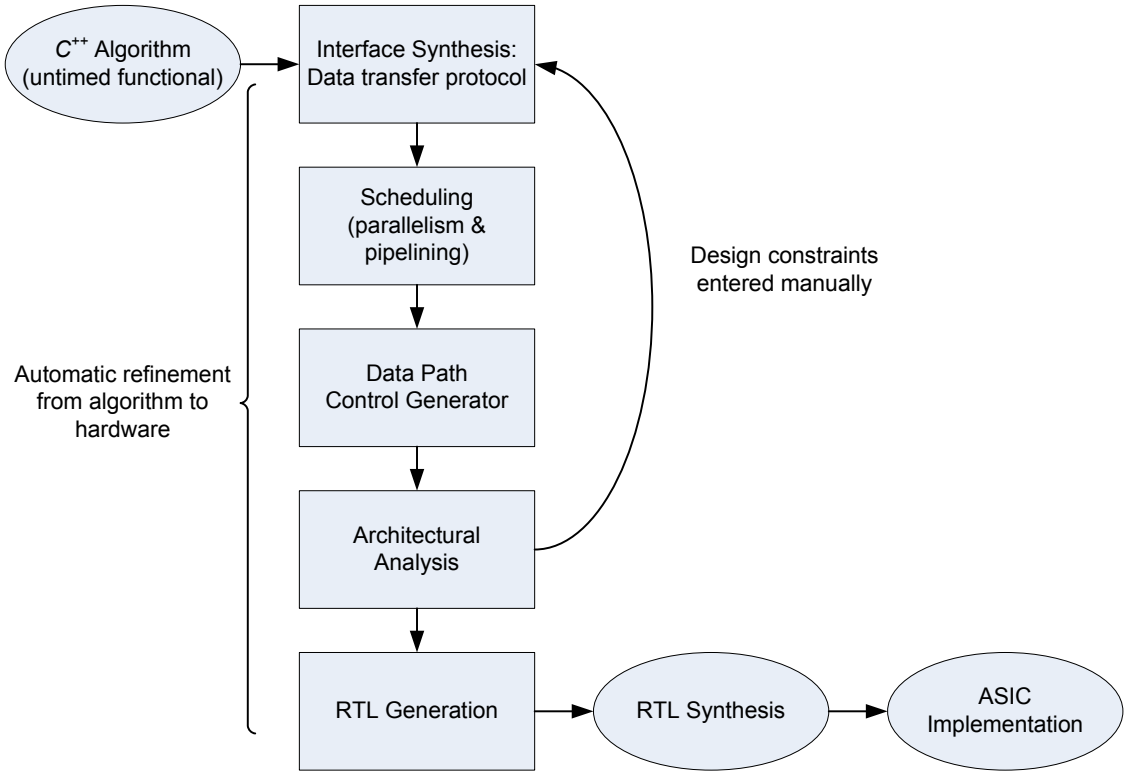



Figure 2.8: Catapult-C Synthesis Design Flow, *source* [42].

then added to the $C^{++}$ code which allows the compiler to correctly generate a hardware model. The interface synthesis imposes a protocol in the transfer of data to and from the top level function. Scheduling exposes the parallelism that exists in the algorithm in order to extract concurrency. The coding style for the algorithm impacts the level of parallelism and concurrency for the hardware design. As an example, the **for** loop in Figure 2.9(a) maps to a sequential MAC type design in hardware, similar in structure to Figure 2.3. However, unrolling the loop of the algorithm, as in Figure 2.9(b), translates into a hardware structure similar to Figure 2.4. The data path control generator ensures that the data flow between the processing elements is functionally equivalent to the $C^{++}$ algorithm. The architectural analysis using Catapult-C returns estimated performance measurements for the hardware code generated by the translators and compilers. The synthesizable RTL code is finally synthesized and mapped to a hardware platform such as an FPGA or ASIC.

This is one example of how a HLS tool can generate different hardware structures based on the coding style of the algorithm. Catapult-C allows a designer to indirectly ex-

```
for(i=0; i<8; i++)
{
   h += x[i]*b[i];
}
```

(a) Sequential code

```
h += x[0]*b[0];
h += x[1]*b[1];
 .
 .
 .
h += x[7]*b[7];
```

(b) Concurrent code

Figure 2.9: Sequential loop versus unrolled code for MAC algorithm

plore different hardware implementations by adding to or removing details from the software algorithm. While the savings in design time are predominantly improved by the automatic generation of hardware designs from high-level code, the manual process for architectural exploration remains the responsibility of the designer. Additionally, algorithm developers may not acquire the necessary hardware skills to fully utilize hardware optimizations described at the algorithmic level. In cases where the hardware structure generated by the synthesis tool cannot meet system specifications, the design is transferred to the hardware design team who is required to decipher the automatically generated hardware code before any additional optimizations can be applied.

### 2.4.3 FIR Compilers for FPGAs

The general term "FIR Compiler" is used to describe an EDA tool for automating the hardware design of FIR filters at the architectural level. The Xilinx LogicCORE $^{TM}$ FIR Compiler is one such FIR compiler used for generating filters [45]. This tool provides an interface for users to construct area-efficient high-performance FIR filters that perform well for specific design applications. A wide range of filter types can be implemented using the Xilinx CORE Generator which include: single-rate, half-band, and interpolated filters, in addition to multi-rate filters such as polyphase decimators and interpolators and half-band decimators and interpolators. The filter transfer function is used to determine the best filter structure. For example a linear-phase FIR filter takes advantage of the coefficient symmetries and results in a hardware implementation half the size of the cascaded DF structure. This method for automating the construction of FIR filters at the hardware

level reduces the design efforts and assists the user in arriving at a hardware structure designed for a specific application. However, the designer has to alter the parameters in order to analyze how various design options alter the hardware performance. This adds to the complexity of searching the design space for filter structures that perform well for specific design constraints.

## 2.5 Chapter Summary

In this chapter, we presented and discussed the process of refining a high-level description to a hardware implementation. This process typically undergoes several levels of design abstraction with specific tools catered for designing at each level. EDA tools and design frameworks generally expedite the refinement process for DSP algorithms to hardware models. Common EDA tools and design languages for implementing DSP algorithms and architectures at different levels of abstraction include MATLAB, $C/C^{++}$, SystemC, Verilog, and Synopsys synthesis tools. However, a large gap exists between the programming languages intended for algorithm development and the tools developed for hardware synthesis. This gap has forced the algorithm developers to hand over their algorithms to hardware designers who explore the architectural design space. This process contributes to an increase in design efforts. Recent work has focused on bridging the gap between the DSP algorithm and the hardware design by using high-level synthesis languages, such as MATLAB and Catapult-C. These tools are used to generate hardware designs from untimed functional descriptions. This process reduces refinement efforts but at a cost of substandard hardware performance. Additionally, most tools focus at either algorithmic variations at the higher levels of design, or gate-level optimizations at the lower levels of design. In either case, the architectural analysis for the different hardware implementations remains a substantial responsibility of the designer, whether it is the algorithm developer or the hardware architect. The goal is therefore to develop a framework and design environment that generates well understood and efficient hardware structures for common DSP applications while simultaneously optimizing performance metrics such as area, delay, and power. Such an approach is inspired by the tool Module Compiler, which owed its popularity to its ability to accelerate the design-space exploration that designers want to do, rather than implementing any behavior that the designer specifies. The next chapter presents our vision

of a framework and design methodology for improving the refinement process of a basic DSP function to realizable hardware models.

# Chapter 3

# Performance Analysis Framework

The goal of our performance analysis framework (PAF) is to efficiently generate, synthesize, and analyze the performance of different hardware representations for basic DSP algorithms. Additionally, our framework provides both algorithm developers and hardware designers cost functions that quantify the quality of a hardware design in terms of area, throughput, and power dissipation. The cost functions can be used to guide the designer in selecting a hardware implementation that performs well for certain design constraints. We accomplished these goals by defining a simplified user interface that paves a fast path to higher quality hardware designs for important DSP algorithms. We developed our methodology for designing and analyzing computationally intensive hardware designs that target ASICs. The concepts introduced in the development of this work are extendible to other hardware platforms such as FPGAs. Figure 3.1 illustrates a top-level view for the fundamental structure behind the development of our PAF in refining a DSP function to a synthesizable hardware design.

An additional goal of our work is to efficiently generate well-structured and recognizable hardware structures for basic, yet essential DSP functions. While the tools we presented in Chapter 2 were oriented toward generating hardware designs for DSP functions that meet specific constraints, we employ a design methodology that allows a designer to select from a pool of hardware designs that meet performance requirements determined by specific constraints. Our PAF provides the advantage of refining a DSP algorithm to a synthesizable hardware design. The designer simply selects the DSP algorithm he or she

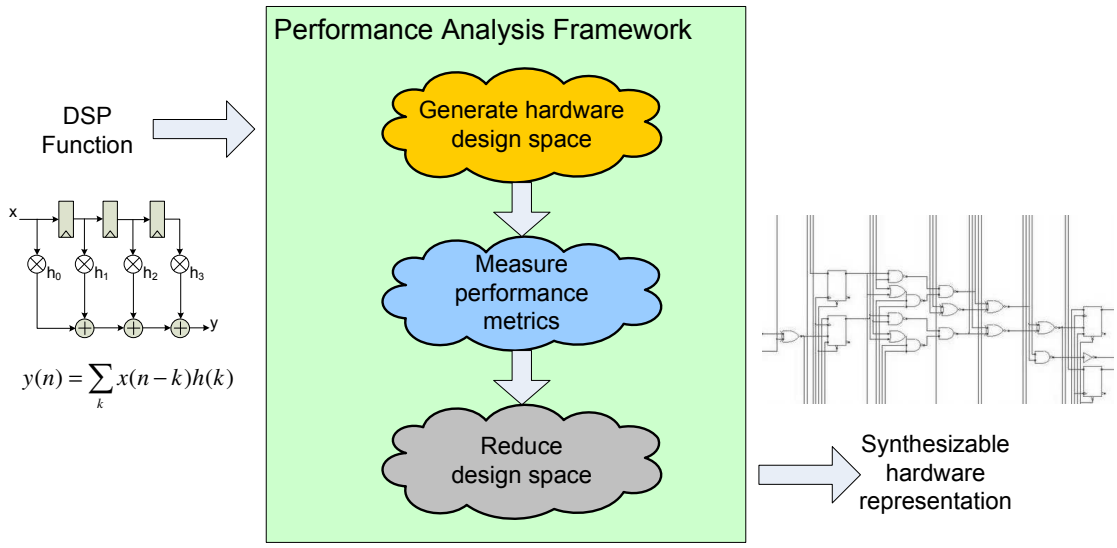



Figure 3.1: Concept of the Performance Analysis Framework

wishes to refine and provides several high-level design parameters that guide the framework in generating application-specific architectures that meet design constraints. Additionally, we provided cost functions to the designer that assess the quality of hardware designs generated using our framework. This is an advantage over other design tools and frameworks.

This chapter presents the details of our PAF for generating the hardware design space for a computationally intensive DSP algorithm such as an FIR filter. We begin by introducing our PAF methodology and illustrate the steps involved in constructing hardware designs and guiding the designer through the process of searching the design space. We then describe the pertinent hardware performance metrics and highlight a subset of cost functions used for evaluating the quality of hardware designs generated by our framework. Finally, we identify the different design options provided for the designer when considering efficient hardware representations for DSP applications.

## 3.1 Performance Analysis Framework Methodology and Flow

This work contributed to the design of a framework that efficiently refines a basic DSP function to several hardware implementations and developed a methodology for guiding a designer in selecting designs that perform well. The goal of this work was to improve the

overall design productivity by generating recognizable hardware structures that perform well while meeting specific design constraints defined by the user. Additionally, the goal of this work was to significantly reduce synthesis times, allowing the designer to explore the hardware design space in hours versus weeks. This allowed the designer to investigate alternate hardware structures in a relatively short time, should the designer change or enhance the system specifications.

We developed a method for estimating the performance metrics for the designs and included cost functions in our framework that can be used to evaluate trade-offs between area, throughput, power dissipation and hardware latency. The final output of our framework is a reduced set of synthesizable hardware designs with improved performance metrics compared to general purpose, "commercial off-the-shelf" designs. The structure of the designs generated by our framework allows proficient hardware designers to apply additional optimizations at multiple levels of design abstraction; a key advantage for further improving the design performance. The PAF reduces the number of designs to be considered by the designer by eliminating those that do not meet the specification.

### 3.1.1 EDA Tools used for the PAF

We combined well-known design tools and scripting languages accepted by both DSP algorithm developers and hardware designers. Our purpose in selecting common EDA tools was to facilitate the use of our framework, as well as allow designers to expand the set of hardware designs within the frameworks library. We chose MATLAB as the interface between the user and the hardware designs generated by our framework since algorithm developers are more comfortable using MATLAB for efficiently and accurately modeling their signal processing algorithms at the algorithmic level. The data generated by the MATLAB algorithmic models were used as reference data for validating the functionality of the low-level designs. Additionally, MATLAB is a good choice of tools for numerically and graphically analyzing the design space in search for hardware designs that perform well. We accomplished this through the use of cost functions that assess the quality the hardware designs using one or more performance metrics. Our framework provided the designers the freedom to select the cost functions they need to assist them in searching the design space for hardware structures that met design specifications.

We used SystemC as an additional design language in the development of our

work. SystemC extends the capabilities of $C/C^{++}$ by providing a set of class libraries capable of modeling designs at different levels of abstraction: from the algorithmic to the logic level [3] [14]. The main advantage of using SystemC in our work was to model the DSP algorithms at the architectural level, while using high-level data-types and communication interfaces in order to expedite the simulation process for the hardware designs. Our choice for using Verilog to model the hardware designs at the synthesizable RTL allowed designers to use our framework as the front end tool to their synthesis CAD tools. We chose the Synopsys Design Compiler for synthesizing the designs generated by our framework.

### 3.1.2 Framework Methodology and Flow

Figure 3.2 illustrates the underlying process of our work in generating and synthesizing efficient hardware designs for DSP functions. Currently, our framework includes variations of the FIR filter, a fundamental DSP function, which are used in adaptive filters and signal and image processing applications. Our framework starts with recognizable hardware structures for the filter, such as the direct form (DF) and transpose form (TF) filter structures for example, and proceeds to generate architectural variations of the basic structures $k = 1, ..., K$ where $K$ is the total number of structures in the design space. We present the details of the filter structures in Chapter 4. Each filter is modeled in hardware using SystemC and Verilog RTL code. Scripts within our framework generate a $C^{++}$ testbench for each SystemC filter, where a set of simulations are executed in order to validate the structural accuracy of the filter designs. Our previous experiments showed that simulations performed at the arithmetic level using the SystemC models were approximately five times faster than simulations performed using the Verilog models for designs generated by our framework [36].

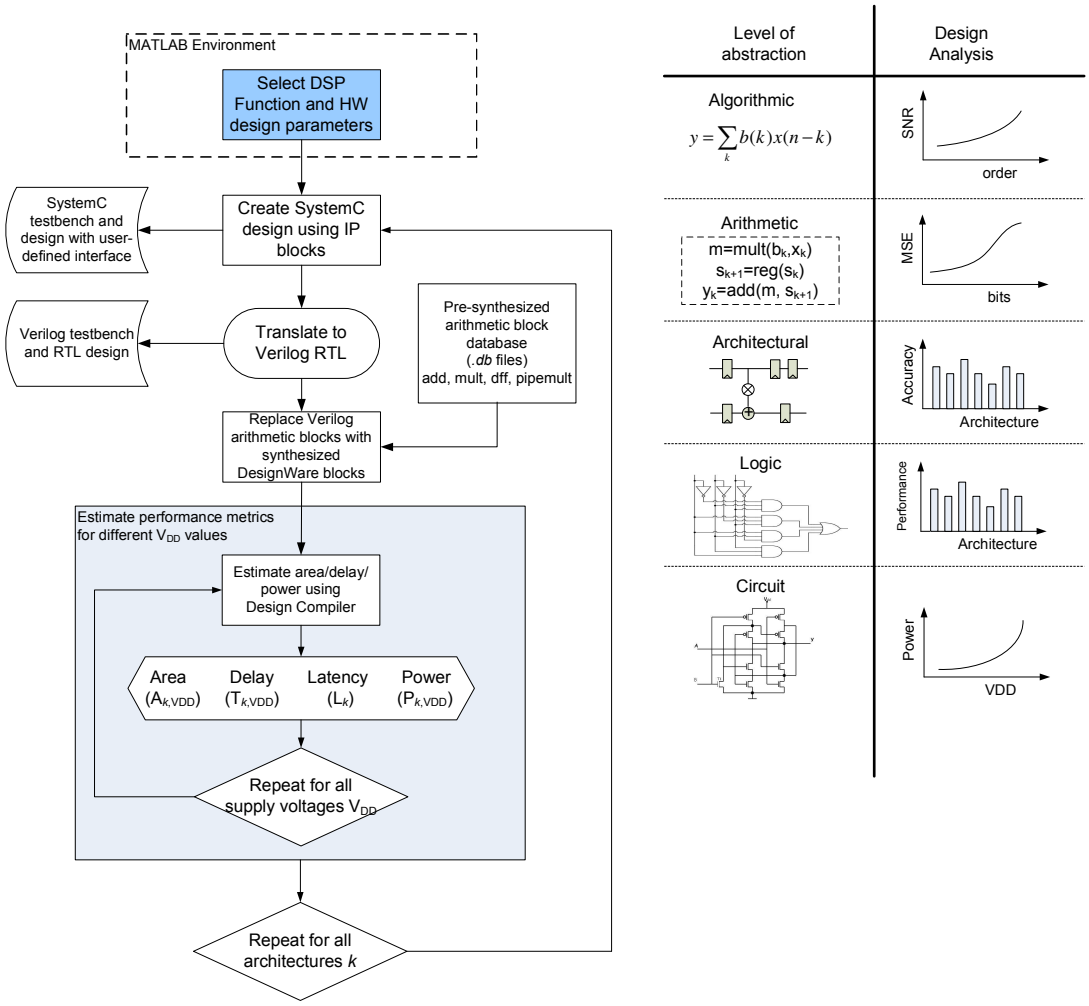



Figure 3.2: PAF Flow for refining a DSP algorithm and selecting an architecture

We utilized scripts and compilers that translate each SystemC filter structure to its Verilog RTL equivalent to ensure design consistencies between the architectural and RTL designs [46]. Each filter structure was generated using basic arithmetic cores such as multipliers, adders and delay units (registers). These blocks were used for functional verification of the filter structures and, therefore, were not synthesized. Scripts within our framework replaced the basic arithmetic cores with pre-synthesized and optimized Synopsys DesignWare equivalent arithmetic cores, relieving the user from having to manually optimize each arithmetic block at the gate level. We utilized a limited set of arithmetic blocks for

the signal processing algorithms considered in our work. We used the optimized Booth-recoded Wallace-tree multipliers and carry-look-ahead adders available in the Synopsys IP library. The same methodology can be used in cases where the user selects alternative circuits for the mathematical elements. The estimated performance results for the hardware designs obtained using the pre-synthesized cores were comparable to synthesizing the entire design [47]. The pre-synthesized cores significantly reduced synthesis times from tens of minutes to seconds for each design, depending on the complexity of the designs.

### 3.1.3 Hardware Design and Synthesis Process

We employed a bottom-up, modular design methodology to construct the hardware structures included in our framework. Figure 3.3 illustrates a single iteration for constructing a hardware design using basic arithmetic hardware elements and processing modules. The designer initially selects the type of DSP function from our library of DSP functions in our framework. The user then selects high-level parameters from the options in our framework. As an example, a designer wishing to implement an FIR filter in hardware can select the filter order and word sizes from our framework GUI. The user presses a button and our framework proceeds to generate each hardware design starting with common and well recognized filter structures. Scripts in our framework begin the design process by constructing the basic computational modules. The processing modules consist of the necessary instantiations of mathematical elements such as adders, multipliers and delay units.

Our framework generates an input module, output module, and computational module, for some filter structures. Each processing module consists of parameters that instantiate the necessary number of adders, multipliers, and registers. The parameters provided by the user and used to determine the appropriate number of each arithmetic block. The computational modules are instantiated a number of times to complete the filter design. Other filter designs can be implemented using a single computational module replicated as input and output cells. The mathematical elements in the modules are connected according to the type of hardware structure and design parameters. We currently include a finite set of processing cells to implement the different variations of filter designs.

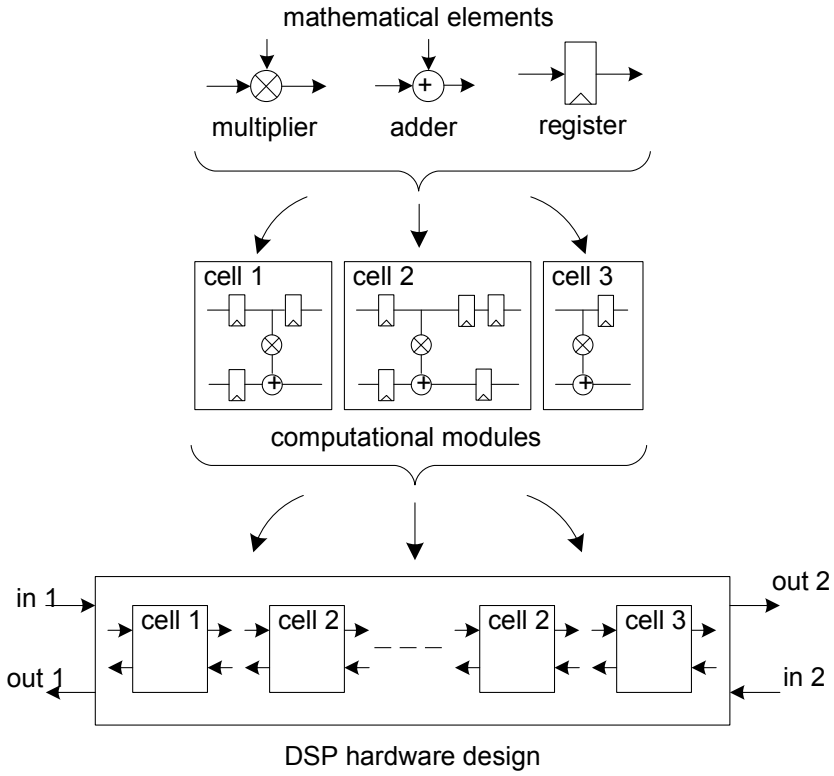



Figure 3.3: Bottom-up modular design for a single hardware structure

The next level of hardware generation focuses on connecting the computational modules to implement the DSP design. The method for connecting the computational modules also depends on the structure for the hardware design. The order in which the arithmetic computations are performed contribute to the various hardware implementations. Therefore, we include different methods for connecting the computational modules to provide architectural variations. The variation in hardware designs effects the low-level performance metrics such as area, throughput and power dissipation. The process continues until the possible architectural variations in the design space are considered for the chosen high-level parameters. Figure 3.4 illustrates an overview for implementing a filter design using different hardware implementations.

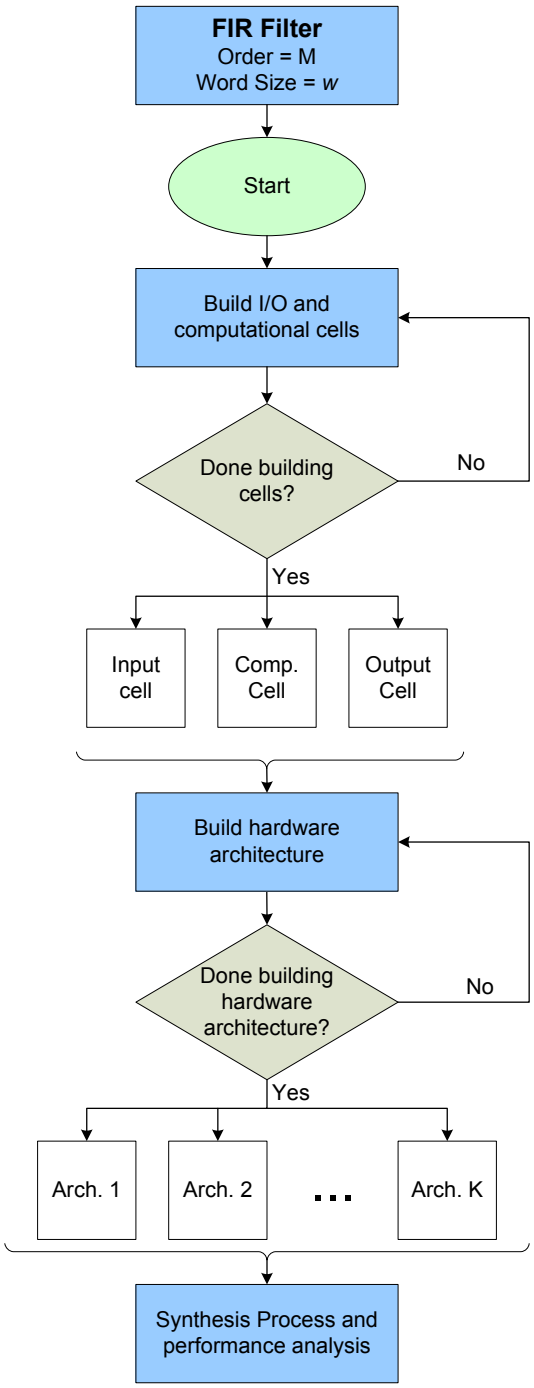



Figure 3.4: Process for generating architectural variations using computational cells

The methodology we employed for constructing the hardware designs is essential to promoting architectural reuse and design consistency. The design efforts are focused on several computational modules which are then instantiated to implement the hardware structures. The computational modules depend on the design parameters selected by the framework user. Furthermore, the filter hardware structures depend on high-level parameters such as filter order. Our PAF constructs a finite set of variations for filter structures while covering a wide range of DSP applications. This assists the framework developer in verifying the functionality of the various filter designs. We accomplish this by running scripts that compare the integrity of the hardware designs generated by our framework to the high-level DSP algorithms. This process is discussed in greater detail in Chapter 7. The final hardware structures conform to recognizable designs and are simple to modify by knowledgeable hardware designers if they wish to further improve the structures generated by the PAF.

We chose to use $C/C^{++}$ to construct each hardware design for the DSP algorithm. This allowed us to define structures, constructs and functions at a high level of abstraction which facilitates the reuse of code to build modules commonly found in DSP hardware designs. The coding style we employed in the construction of the hardware structures allows designers to contribute to the library of architectures in our PAF. This translates to a framework that is easily maintained and updated with state-of-the-art hardware designs. Additionally, the final $C/C^{++}$ executable file executes in a matter of seconds when constructing tens of hardware models for a DSP function.

### 3.1.4 Design Space Reduction using Cost Functions

Intuitively or analytically determining the critical path delay, area, or power dissipation for a hardware design is a complicated process which prolongs the task of analyzing a designs performance. Therefore, we use Synopsys Design Compiler to synthesize and estimate the performance metrics in a hardware design. The scripts we utilized to synthesize and optimize a design for timing requires an initial clock period. The scripts used in synthesizing a design selects a relatively small clock period as the initial timing constraint. The "SLACK" returned by the Synopsys `report_timing` command is then subtracted from the initial clock period. For cases where the timing is not met, i.e. the "SLACK" is negative, we increase the initial clock period. The process is repeated until the timing "SLACK" is

less than or equal to $0.01ns$. The design area is then estimated using the `report_area` command.

The scripts we use to estimate the power dissipation for a hardware design requires providing the operating clock period for the design. We use the same clock period derived for the timing report, which generates an upper bound on estimating the power dissipation for a hardware design. The process for estimating the minimum clock period and maximum power dissipation is repeated for cases where the supply voltage is changed. The synthesis scripts promptly return area, timing and power reports back to the MATLAB workspace where the user then applies cost functions for searching the design space for efficient hardware implementations.

Our framework estimates the performance metrics for each hardware structure by varying the $V_{DD}$ values using similar scripts and techniques described in [26] [48]. This provides the user alternative design options for improving both throughput and power dissipation. Our framework repeats the generation and synthesis process for each hardware structure in the design space. The performance values for each design are relayed back to MATLAB, and temporarily saved in an EXCEL spreadsheet for detailed architectural analysis using the cost functions. Our framework provides the designers the freedom to select the type of cost function they wish to use in assessing the quality of the hardware designs. For the sake of simplicity, we focused on one and two-parameter cost functions commonly used to measure hardware performance. Examples of two-parameter cost functions we used were area-delay product ($AT$), power-delay product ($PT$), and power density ($P/A$) which are discussed in detail in Section 3.3.

Our framework operates in one of two scenarios. The first is refining a DSP algorithm to tens of possible hardware permutations without specifying low-level performance constraints. For this case, the hardware design space includes a large number of designs with largely varying performance metrics. The user can use any of the cost functions provided by our framework, or use custom-defined cost functions to sift through the design space for architectures that perform well. Case studies for this scenario are discussed in Chapters 4 and 5. The second scenario is refining a DSP algorithm to an application-specific hardware design that includes specific design optimizations. For this case, our framework returns several closely matched designs with similar performance values rather than returning a single architecture that meets the design constraints. An example of this scenario is presented and discussed in Chapter 6. In either scenario, the framework user can make final architectural

selections, or possibly relax the performance constraints which may provide a better design option. The end product for our flow is a set of designs with specific details of hardware structure, supply voltage required to operate the design and the performance and efficiency values.

## 3.2 Hardware Performance Metrics

Designers use commercial synthesis CAD tools, such as Synopsys, to transform a design described using an HDL into a technology-dependent netlist of standard cells. A standard cell is a collection of transistors and interconnect structures that provide equivalent functionality to the corresponding high-level hardware modules. Examples of combinational standard cells include `NAND`, `NOR`, `XNOR` gates, and their compliments, whereas D-type Flip Flops (`DFF`) and latches are examples of sequential standard cells. The area, delay and power dissipation for the final synthesized design depends on the number of standard cells required to implement the design and how the cells are connected.

### 3.2.1 Area

The **area** of silicon ($A$) occupied by a standard cell module is proportional to the number of transistors used to implement the design. Wiring and netlist contacts are other factors that affect the final design area. However, silicon area is primarily governed by the mathematical cores used in computationally intensive hardware designs, similar to the designs generated by our framework. We used Synopsys Design Compiler to obtain an area estimate for designs described at the synthesizable RTL. The area estimate generated by Synopsys depends on the target library used as well as the input/output word sizes. We used the Synopsys "`report_area`" command to report the total cell area in $\mu m^2$ for designs that target ASICs. Minimizing the area for a design is important to ensure relatively low fabrication costs, high throughput results, and low power dissipation. While it is intuitive to predict how changes in the design can effect the overall area, it is still important to obtain an accurate estimate for the area using the Synopsys synthesis tools. As an example, Table 3.1 and Figure 3.5 illustrate the increase in area as the input word sizes increase for several Synopsys DesignWare blocks commonly used in DSP hardware designs. The

results show that the increase in area for the DFF is linearly proportional to the increase in input word sizes, whereas the increase in area for the adder is approximately proportional to the linear increase in input word sizes with a constant offset. However, the increase in area for the multiplier is approximately quadratically proportional to the input sizes, assuming a symmetric multiplier. The results in Table 3.5 were collected via our PAF using $0.18\mu m$ standard cell library and a supply voltage of 1.8 $V$. This information is useful for approximating the area for simple DSP designs using a constant standard cell library. However, the analyses becomes harder for complex designs and varying technology sizes. Therefore, we alleviate the process of analytically estimating the performance for a hardware design by synthesizing the design and generating the area report using Synopsys Design Compiler.

Table 3.1: Area for Basic DesignWare Mathematical Cores Measured for $0.18\mu m$ ($\mu m^2$)

| DesignWare | Input Word Sizes | | | | | |
|---|---|---|---|---|---|---|
| Block | 4 | 8 | 12 | 16 | 24 | 32 |
| Adder | 972 | 2208 | 3614 | 5297 | 8015 | 11142 |
| DFF | 976 | 1936 | 2896 | 3856 | 5792 | 7728 |
| Multiplier | 3227 | 12503 | 26333 | 47205 | 87328 | 150604 |

### 3.2.2 Critical Path Delay

The critical path refers to the slowest path of combinational logic that exists between two registers [4]. It is widely accepted that the delay ($T_d$) of a digital circuit is proportional to [33]

$$T_d \propto \frac{V_{DD}}{(V_{DD} - V_{th})^{\alpha}} \tag{3.1}$$

where $\alpha$ is the velocity saturation index with values ranging from 1 to 2, $V_{DD}$ is the supply voltage and $V_{th}$ is the threshold voltage of the CMOS circuits and is assumed constant for a given technology node. We used the Synopsys "`report_timing`" command to calculate and report the delay for the most critical path in the design. Synopsys Design Compiler

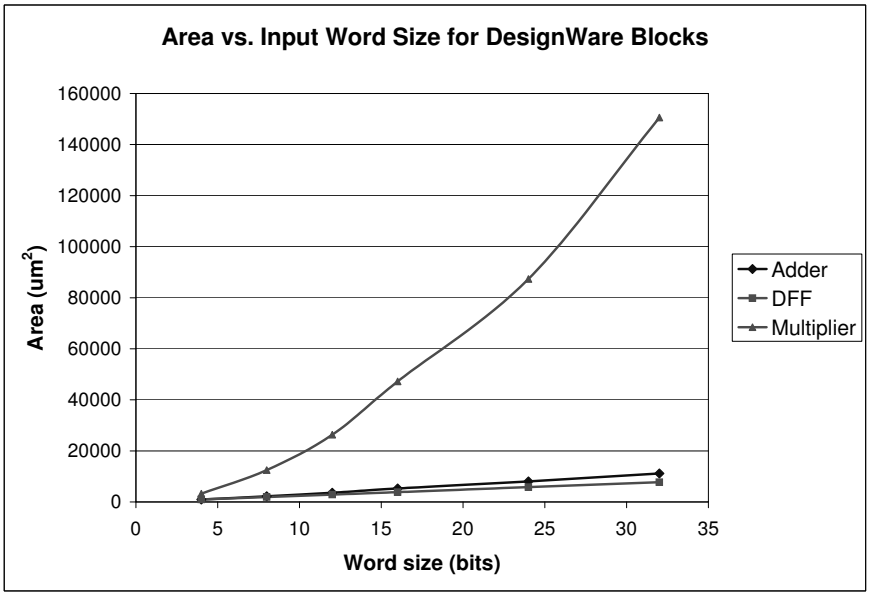


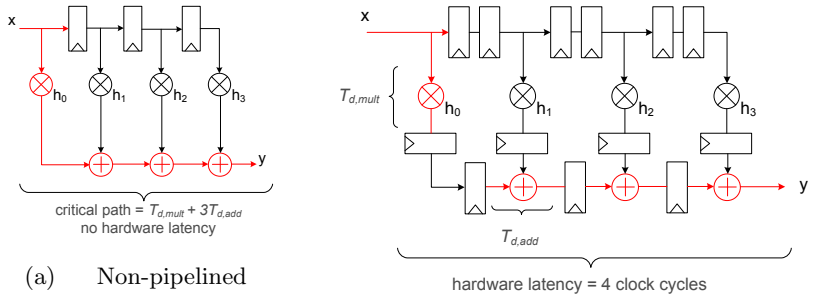

Figure 3.5: Area versus word size for mathematical cores

reports a "Negative Slack" if the delay does not meet the timing constraints and therefore a larger clock period would be required to synthesize the design. An additional performance metric which is derived from the critical path delay is operating frequency $f_{op}$ where we assume $f_{op} = 1/T_{d,max}$ and $T_{d,max}$ is the largest critical path delay. Therefore, designers iteratively vary the design clock period in order to find the fastest frequency at which the design can operate.

### 3.2.3 Hardware Latency

The latency for a design defines the amount of time required to complete an operation and is typically measured in clock cycles. We define **hardware latency** ($L$) as the number of clock cycles required to generate the output once the signal is applied at the input. Figure 3.6(a) illustrates an example of a DSP design consisting of adders and multipliers used to implement a $3^{rd}$ order DF FIR filter. The multipliers in the non-pipelined design must wait until all of the additions have been completed before they can begin another multiplication. This waiting time contributes to large delays and degrades the performance of the design. Pipelining is one method for reducing the "wait" time for the computational blocks, an example of which is illustrated in Figure 3.6(b). However,

hardware latency increases for larger stages of pipelining which may affect the designs execution time. The method for efficiently pipelining a DSP architecture depends primarily on the hardware structure and the performance specification for the final design.




(a) Non-pipelined design

(b) Pipelined design

Figure 3.6: Pipelining a $3^{rd}$ order DF FIR filter to improve critical path and hardware utilization

### 3.2.4 Design Throughput

The trade-off between the two designs in Figure 3.6 lie between the critical path delay and the hardware latency. Throughput is an appropriate performance metric that quantifies both critical path delay and the hardware latency into a single metric. We define **throughput** ($R$), within the context of our work, as the number of samples generated at the output per second, and can be computed as

$$R = \frac{1}{\left(\frac{Initial\ Latency}{N^o\ of\ samples} + 1\right) T_{d,max}}\ (samp/sec) \tag{3.2}$$

where $Initial\ Latency$ is the number of clock cycles required to fill the pipeline stages and $N^o\ of\ samples$ is the total number of data points processed by the hardware design. Therefore, the overall design throughput depends on both the critical path delay and the hardware latency. Figure 3.6(b) illustrates that we can increase the design throughput by placing registers within the paths of large clusters of combinational logic. For example, if the critical path delay for the multiplier and adder is $5ns$ and $2ns$ respectively, then the overall throughput for the design in Figure 3.6(a) is at most $91MHz$. However, the throughput for the pipelined design in Figure 3.6(b) is $200MHz$; a two fold improvement in

performance. This assumes that the *Number of samples* processed is considerably larger than the designs *Initial Latency*, which is the case for most practical DSP applications. Chapter 4 highlights the usefulness of pipelining for improving the performance in FIR filters.

### 3.2.5 Power Dissipation

Power dissipation and energy consumption are critical factors in the design of many hardware devices. Minimizing power dissipation is important for maximizing battery life and reducing heat dissipation in portable devices. **Dynamic power** $P_{dynamic}$ is the main source of power dissipation in CMOS circuits for feature sizes greater than $65nm$, and can be computed as

$$P_{dynamic} = \alpha C_{switch} V_{DD}^2 f_{op} \tag{3.3}$$

where $\alpha$ is the switching activity factor and $C_{switch}$ is the switching capacitance [49]. As equations 3.1 and 3.3 suggest, both power dissipation and operating frequency are affected by changes in $V_{DD}$. Therefore, hardware designers are burdened with the task of varying design parameters in an attempt to generate efficient hardware models that meet performance constraints. Section 3.3 presents several cost functions and figures of merit that assist the designer in achieving a balance between critical path delay and supply voltage for improving power dissipation and throughput in a hardware design.

### 3.2.6 Computational Precision

DSP algorithms are initially developed using high-level languages, where the data-types in the algorithm are usually specified as floating-point. The precision for the floating-point computations depends on the general purpose computer and operating system used to implement the algorithms. As a DSP algorithm is refined to lower levels of design abstraction, such as the arithmetic and architectural levels, designers rely on fixed-point computations to reduce hardware cost while increasing throughput rates. An important step in a top-down design flow is to determine the fixed-point data-types for all signals and ports in the design. This requires the designer to have acute knowledge of how word-lengths, quantization methods and overflow modes affect the functional behavior of the final hardware output(s) when compared to the high-level floating-point algorithms.

SystemC includes a set of data-types used to model computations using fixed-point representation. The general form for defining a fixed-point variable using SystemC data-types is `sc_fixed<w, iwl, q_mode, o_mode>` where `w` is the total word length, `iwl` is the integer word length, `q_mode` is the quantization mode, and `o_mode` is the overflow type [50]. We included options in our framework that allow a designer to utilize the various quantization modes and overflow types, four of which are:

1. `SC_RND`: **round** - Round to the nearest representable number.
2. `SC_RND_ZERO`: **fix** - Round to the nearest representable number towards zero.
3. `SC_RND_INF`: **ceil** - Round to the nearest representable number towards plus infinity.
4. `SC_RND_MIN_INF`: **floor** - Round to the nearest representable number towards minus infinity.

Each quantization method yields different quantization errors, with **round** being the best type of quantization for both positive and negative values. However, each quantization type has a different hardware representation, with **round** being slightly more complex than the other modes. Figure 3.7 illustrates several fixed-point, two's complement representations for a positive and negative irrational floating-point number that's $w$ bits wide, containing $f$ fractional bits, and a sign bit $s$. For this example, we assume that $w = 8$, $f = 4$, and $s = 1$ for representing $\sqrt{5}$ and $-\sqrt{5}$ in fixed-point.

Additionally, SystemC models the overflow in one of five ways:

1. `SC_SAT`: **Saturation** - The output is set to either the maximum or minimum representable number within the dynamic range.
2. `SC_SAT_ZERO`: **Saturation to Zero** - The output is forced to zero in case the minimum or maximum values are exceeded.
3. `SC_SAT_SYM`: **Symmetrical Saturation** - The output is saturated to the maximum values in case of a positive overflow. For the case of a negative overflow, the output is set to negative the maximum value for signed computations, or the minimum value for unsigned computations.
4. `SC_WRAP`: **Wrap-around** - The output is set to any representable number by ignoring the overflow bits.

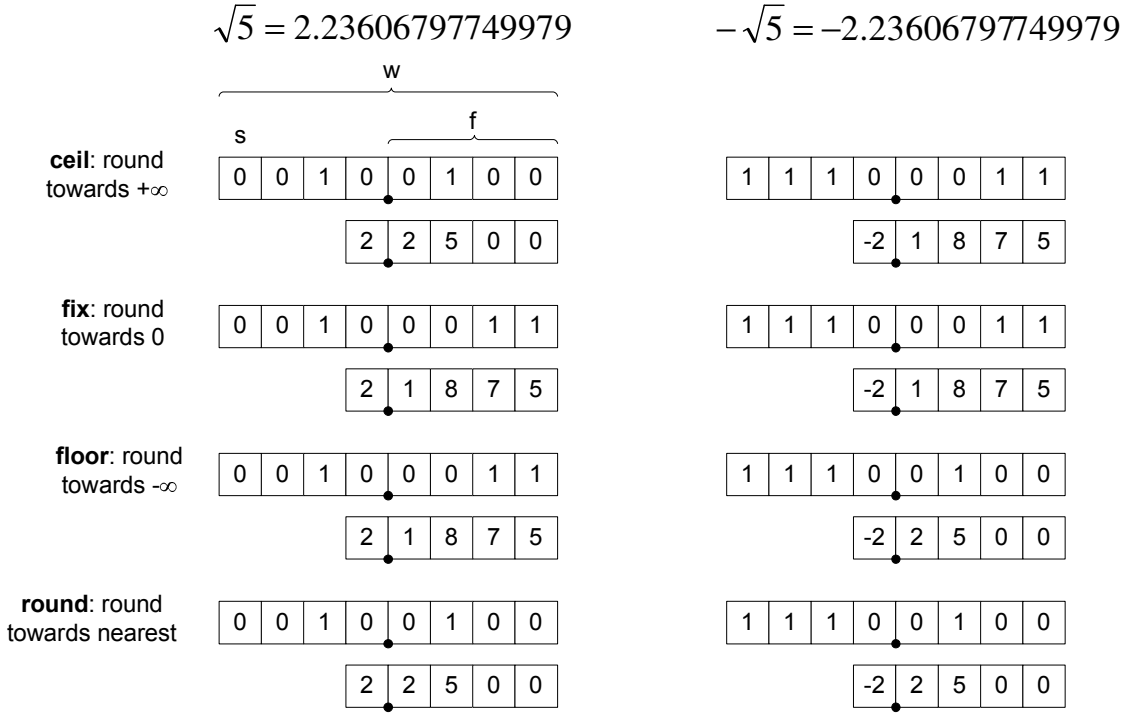
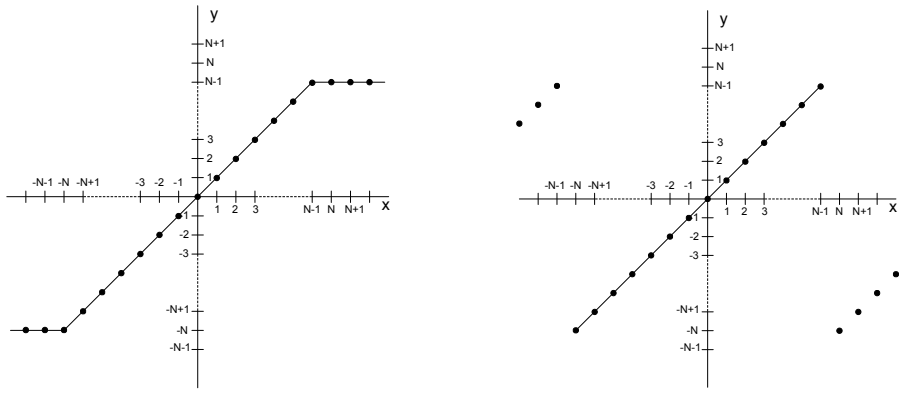



Figure 3.7: Binary representation for generalized fixed-point numbers

5. `SC_WRAP_SM`: **Sign Magnitude Wrap** - The output is sign-magnitude wrapped around in the event of an overflow.

Currently, our framework models the **Wrap** and **Saturate** overflow modes at the arithmetic and hardware levels. The natural overflow mode for integer computations in hardware is the **Wrap**. The **Saturate** mode slightly increases hardware complexity, but does a better job in maintaining signal integrity when compared to the **Wrap** mode. Figure 3.8 illustrates the two overflow modes we modeled in our framework.




(a) Saturation

(b) Wrap-around

Figure 3.8: Overflow modes *source*, [50]

## 3.3 Design Cost Functions and Figures of Merit

Designers find it useful to utilize two-dimensional cost functions in an effort to achieve a balance between area, throughput and power. Each two-dimensional cost function is important for determining the best structure that meets constrained design specifications. Figure 3.9 illustrates the trends in trade-off curves between pairs of performance metrics for a hardware design space [51]. In the case of digital circuits, trying to optimize one performance metric often results in a degradation of the others. We defined several cost functions that guide designers in selecting hardware models that perform well in the design space.

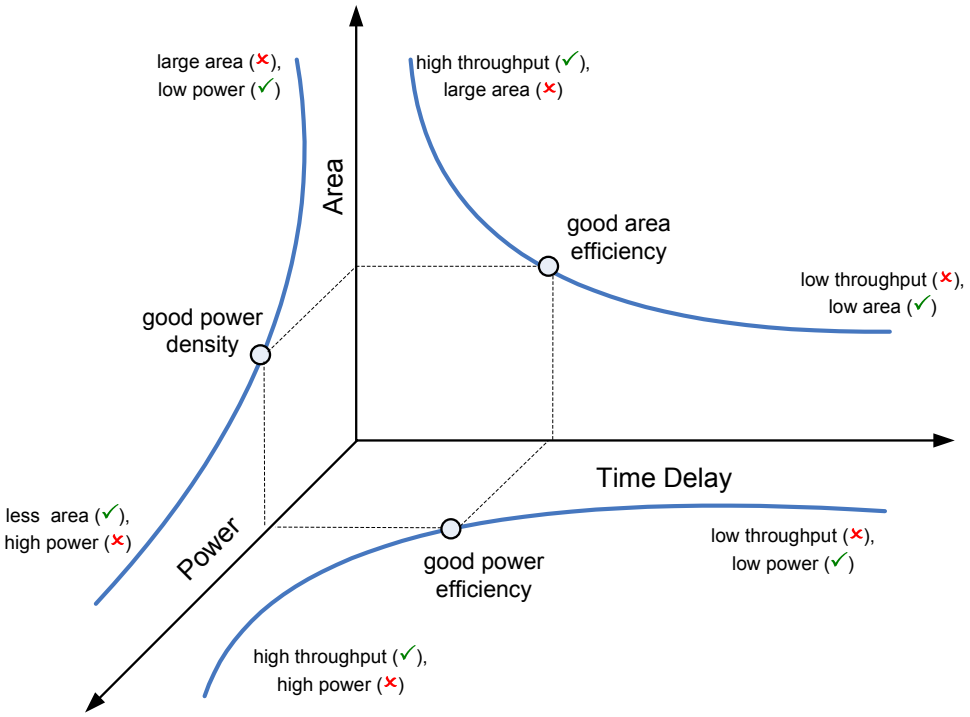



Figure 3.9: Trade-off curves for hardware designs

### 3.3.1 Power Density

**Power density** $(P/A)$ is an important metric that measures the power dissipation per unit squared of area and can be defined as [52]

$$Power\ Density = \frac{PowerDissipation}{DesignArea}\left(mW/mm^2\right) \tag{3.4}$$

Designers attempt to reduce the power density in portable devices in order to prolong battery life and minimize heat dissipation. A direct approach to reduce the power density is to reduce the supply voltage $V_{DD}$ in Equation 3.3. However, reducing the supply voltage increases the switching time, which in turn reduces the performance of the design in terms of throughput.

### 3.3.2 Power Efficiency

An alternative cost function that gives equal importance to both power dissipation and throughput is **power efficiency**, also referred to as **power-delay product** or simply

**energy**, and can be defined as

$$Power\ Efficiency = \frac{(Power\ Dissipation)}{(Throughput)}\,(mW/(Msamp/sec)) \tag{3.5}$$

Similar to power density, reducing power efficiency is important in hardware devices that require high throughput performances and limited power dissipation. The relationship between operating frequency and power dissipation shown in Equation 3.3 complicates the process for reducing power efficiency. Similar to the previous cost function, reducing $V_{DD}$ to reduce the power increases the switching time. However, the reduction in power due to $V_{DD}$ scaling has a larger impact than the increase in delay. Therefore, the overall effect for reducing supply voltage improves the power efficiency, but comes at a cost of degrading the design throughput.

Pipelining has been shown to reduce the power dissipation in designs with minimal degradation to throughput [33]. Equation 3.1 suggests that the delay increases as the supply voltage is reduced. Pipelining can effectively reduce the delay to the desired value by reducing the critical path. We observe from Equation 3.3 that reducing the supply voltage quadratically reduces power dissipation for a fixed throughput, which is highly desirable. Figure 3.10 illustrates how $V_{DD}$ scaling can be combined with pipelining techniques to achieve desirable throughput and power dissipation measurements. Manually pipelining the hardware structures along with $V_{DD}$ scaling contribute to the complexities of searching the design space for optimal structures. Therefore, we designed our framework to explore the effects of combined pipelining and $V_{DD}$ scaling on the performance of the hardware structures generated by our PAF.

### 3.3.3 Area Efficiency and Area-delay-product

Several cost functions exist that combine area and delay into a single cost function. For example, **area efficiency** is the area of the design divided the number of samples computed in one second, i.e.

$$Area\ Efficiency = \frac{Design\ Area}{Throughput}\left(mm^2/(Msamp/sec)\right) \tag{3.6}$$

The **area-delay product** ($AT$) is an alternative form to area efficiency that gives equal emphasis for both area and delay [53]. The goal for designers when using this cost function is to minimize both area and delay. However, techniques for reducing delay may

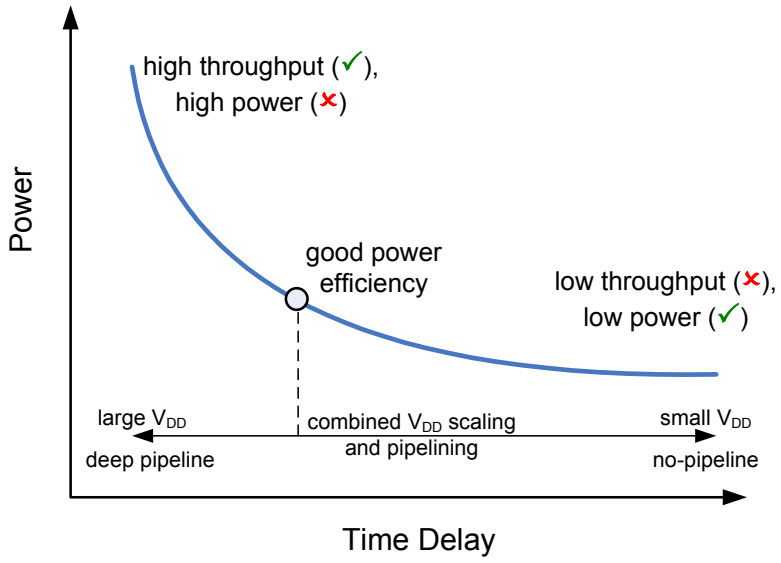



Figure 3.10: Power-Delay trade-off curve

come at a cost of increasing the area. Therefore, the area-delay product can be used to find the point at which the least amount of area is sacrificed for the most improvement in speed.

### 3.3.4 Additional Figures of Merit

The previous three cost functions give equal emphasis to two performance metrics at a time. Additional cost functions exist that emphasize one metric over the other. For example, the **energy-delay product** ($PT^2$), stresses the importance of minimizing the delay over reducing the power [54]. The **area-delay squared product** ($AT^2$), also referred to as the **second order area-delay product**, is similar in form to the area-delay cost function with an emphasis on throughput over area [53]. Gialhard *et al.* defined a method for linearly combining area, operating frequency and power dissipation into a single linear cost function used for module selection for a DWT hardware design [55]. Similarly, Pournara *et al.* defined a cost function, $ETA$, that combined energy, delay, and area which was used to select the parameters for designing efficient FPGA logic blocks at the circuit level [56]. Designers are continuously faced with the challenge of striking a balance between improving the designs in terms of area, throughput and power dissipation. Therefore, they may choose to define their in-house cost functions depending on the design specifications and target

platform.

The two dimensional cost functions defined earlier allow designers to search the design space for architectures that perform well while constraining at least one of the performance metrics. For example, throughput is a vital design constraint in ASICs, whereas hardware resource utilization is the driving force behind minimizing design area for FPGAs. Minimizing power dissipation for a given throughput constraint is imperative for heat-sensitive chip realizations and battery-operated devices [54]. Chapters 4, 5, and 6 present case studies where we apply various cost functions to several DSP hardware designs with different performance constraints. Our goal for the case studies presented in this work is to utilize the two dimensional cost functions in order to efficiently search the design space for hardware models that perform well.

## 3.4 Design Options Applied at Different Levels Abstraction

We designed our PAF to include several hardware design options that address architectural optimizations at different levels of abstraction. Designers with limited hardware experience can provide high-level design parameters to our framework. For this case, our framework considers a large design space that includes dozens of architectural variations with various types of optimizations. Additionally, we developed our framework to assist experienced hardware designers by providing options that address low-level optimizations. Examples of low-level design options our framework provides the designer include pipelining, interleaving, and parallelism. For these cases, designers can use our framework to select different optimization techniques that exclude certain structures from the design space. The result is a reduced set of designs that include hardware structures that meet specific performance constraints.

### 3.4.1 Algorithmic Level

The algorithmic level requires limited details to describe the behavior for the DSP algorithm and may be as simple as a few lines of $C/C^{++}$ code or a few MATLAB function calls. The outputs for the DSP algorithm are easily generated at the algorithmic level, which are used later on in the refinement process to assess the integrity of the low-level

output signals. Our framework provides the user several high-level options to construct the DSP function according to system level parameters. The designer focuses on the functional behavior of the algorithm within the system, and therefore the details for the design at this level are abstracted from the designer.

#### DSP Functions

We designed our framework to include several computationally intensive, yet important DSP functions. Currently, we developed our framework to address architectural variations and design optimizations applied to FIR filters. The FIR filters generated by our PAF are appropriate for other DSP algorithms such as the QMF banks used in signal and image compression and adaptive equalizers used in digital communication systems. The concepts introduced in the development of our work can be applied to DSP functions similar in nature to the FIR filters. Examples of appropriate functions that can be included in our work are Discrete Cosine Transforms (DCT), Fast Fourier Transforms (FFT), and DWT.

#### Multi-Rate and Multi-Signal Processing

Multi-rate and multi-signal DSP applications can be implemented efficiently by slightly altering the algorithm. The improvement in design performance by altering the algorithm is emphasized at lower levels of abstraction where improving performance metrics such as area, throughput and power dissipation is important. Therefore, we constructed our framework to include algorithmic optimizations that improve the hardware structure of the DSP design. This alleviates the framework user from having to manually alter the algorithm in order to model the low-level hardware efficiencies. Examples of algorithmic options that improve hardware performance currently included in our work are interleaving for multi-signal processing and polyphase filter structures for multi-rate filtering which are necessary operations in signal and image compression applications.

### 3.4.2 Architectural Level

Algorithm developers with limited hardware design experience may provide limited parameters to our framework. This results in a large design space that includes hardware

structures with substandard performance metrics. Hardware designers can use our framework to generate hardware structures that perform well for specific design constraints. This can be accomplished by selecting additional design options from our framework that confine the generation of the hardware designs to a smaller set. For either situation, both groups of designers are able to select an efficient implementation from the resultant design space by utilizing the cost functions defined earlier.

### Hardware Designs

The design space for a DSP algorithm typically includes tens of hardware implementations that are functionally equivalent. However, the structure for the hardware design and the layout of the mathematical blocks can affect the final performance in terms of area, delay and power dissipation. We included several hardware representations for basic DSP functions that have been presented in the literature and accepted in similar CAD tools that aim at refining a DSP algorithm to hardware blocks. In some cases, the hardware structure for the DSP algorithm can affect the functional behavior. This was discussed briefly in Chapter 1 where the output for an adaptive filter using the TF FIR filter is different than the output using a DF FIR filter, even though both filters perform the convolutional sum.

### Pipeline Type

Pipelining is a popular hardware design technique for improving the throughput for a computationally intensive design [33] [57]. We provide the user several pipelining techniques that employ pre-designed pipelined IP blocks within the Synopsys library of DesignWare IP blocks [27]. For example, a pipelined multiplier undergoes almost half the critical path that a non-pipelined multiplier undergoes. Simply replacing the non-pipelined multipliers within a hardware structure by pipelined multipliers may drastically alter the DSP algorithm and generate erroneous results. Therefore, our PAF strategically and appropriately places registers within a design while exploiting the DesignWare pipeline multipliers. We utilized single-stage pipeline multipliers since the higher stage pipelines resulted in minimal improvement in critical path delay of the multipliers.

The overall hardware latency due to pipelining depends on the hardware structure of the DSP algorithm. Designs consisting of many computational blocks within the path

from input to output may exhibit large hardware latencies after pipelining. This degrades the overall execution time for the design due to the time required to fill up the pipeline stages. We illustrate in Chapter 4 that the hardware latency for certain filter structures is minimally affected by pipelining. Our PAF allows the designer to assess the trade-offs in hardware latency and critical path delay subject to different pipelining methods. The details for the different pipelining methods are discussed in the next chapter within the realm of FIR filters.

### 3.4.3 Arithmetic Level

Designers analyze the computational precision at the arithmetic level, where both high-level languages as well as HDLs can be used to model bit-true computations. Therefore, we developed our framework to address the computational precision for the DSP designs using different data-types, quantization modes and overflow types within SystemCs class libraries. The design process at this level is iterative with each iteration assessing the trade-offs in computational precision and hardware complexity. Our framework allows designers to analyze signal integrity while providing an estimate for hardware utilization. We accomplished this by expanding the set of synthesizable data-types to include the **SystemC fixed-point** type. This relieves the designer from manually converting the fixed-point data-types to synthesizable types that the synthesis tools recognize.

#### Data-Type

Our framework provides the designer several options for representing the precision of the inputs, outputs, and internal signals. The SystemC class libraries allow us to model designs at the hardware detail using data-types included in other high-level languages. This permits the designers to integrate the hardware blocks into the high-level system designs for algorithmic analysis. Currently, our framework includes $C^{++}$ **floating-point**, **SystemC fixed-point**, and **SystemC synthesizable integer** data-types. Each data-type plays an important role in the hardware refinement and architectural selection process. The floating-point data-type can be used to analyze the design while performing simulations at a fraction of the time required for bit-level RTL designs [36]. We use SystemCs fixed-point data-type to analyze the affects of quantization and overflow modes, allowing designers to assess these

effects on the behavior of the design early in the refinement process. SystemC also includes a synthesizable subset of data-types that accurately models RTL data communication and bit-true computations [16].

### Input/Output Word Sizes

The performance for a hardware design is partially governed by the size of the input, output and internal signals. Generally speaking, larger word sizes result in fewer quantization errors. Therefore, our PAF includes a constrained set of word sizes that allows a designer to analyze performance trends as a function of input word sizes. The hardware structures generated by our framework conform to a well-structured and recognizable set of designs wherein the input, output and internal signals abide by a simple set of rules for determining their sizes. The following points summarize the rules we apply to assigning word sizes to ports and signals for a DSP hardware design:

1. Inputs and outputs can be either $n$ bits or $2n$ bits wide.
2. Inputs to multipliers are $n$ bits.
3. Word size reduction, when necessary, occurs after an arithmetic computation.
4. $n$-bit inputs are extended to $2n$ bits for adders with different sized inputs.
5. Inputs to adders of similar word sizes are preserved.

Figure 3.11 illustrates the rules we use for assigning the sizes of inputs, outputs, and internal signals to an arbitrary computational module.

### Computational Precision

Quantization and rounding errors induced in the hardware implementation of a design can effect the integrity of the signals. One method for analyzing the errors is comparing the output(s) of the hardware implementation to the output(s) of the high-level DSP algorithm. However, this requires implementing the design at the bit-wise RTL and converting the floating-point inputs to fixed-point values. This process increases design and analyses iterations which degrade the overall refinement process. Alternatively, computational errors can be analyzed using the floating-point design while emulating the quantization and

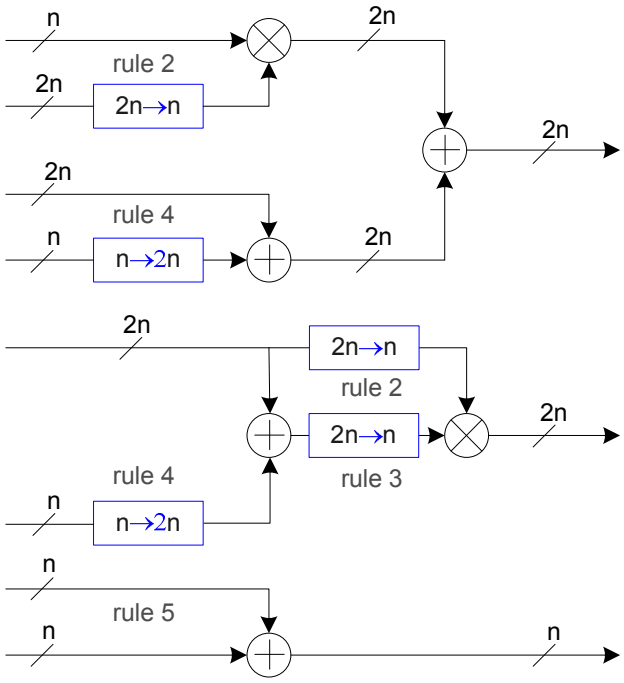
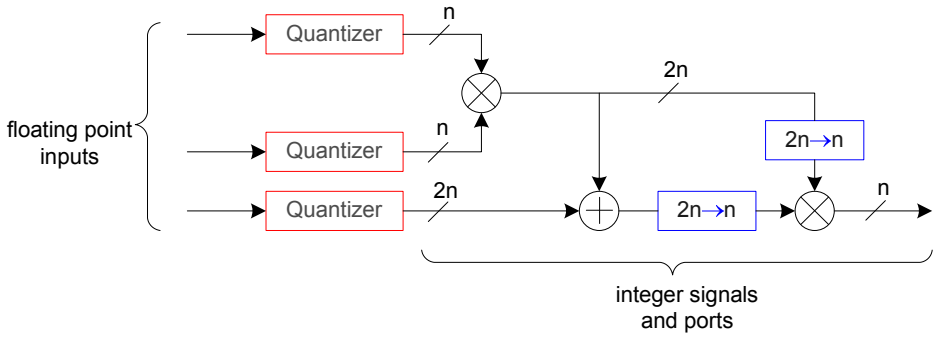



Figure 3.11: Input/output word sizes for DSP hardware designs

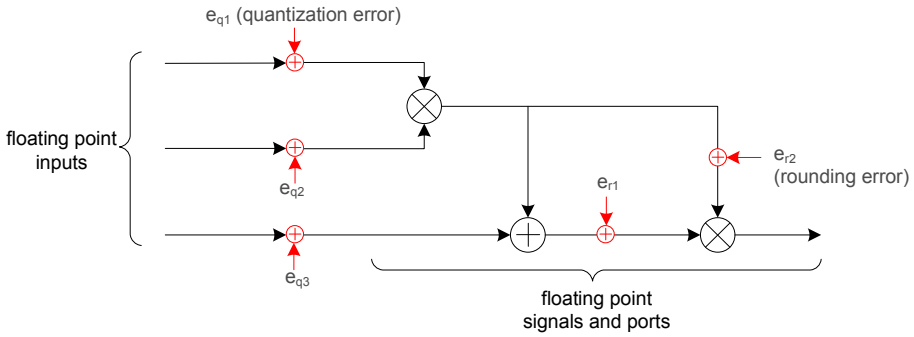
rounding errors [29]. Figure 3.12 illustrates a model for mathematically analyzing round-off errors induced into a floating-point hardware design. Our framework allows the designer to set the values for the error generators based on parameters such as statistical distribution and targeted word sizes. Additionally, we include options in our PAF that allow the designer to select input signals specific to the DSP algorithm for a more accurate analysis of quantization and rounding errors.

### 3.4.4 Circuit Level

The circuit level is one of the lowest levels of design abstraction where a designer can provide low-level parameters that alter the performance of the synthesized hardware designs. We utilized Synopsys Design Compiler to generate a low-level netlist that is functionally equivalent to the architecture generated in the previous refinement process. The layout for the design netlist is governed by the synthesis options such as optimizing the design for throughput or for area. We specified "medium effort" in the framework synthesis scripts for mapping the area of the standard cell blocks. Technology feature size and supply




(a) Actual integer system




(b) Mathematical model

Figure 3.12: Quantization and rounding errors for DSP hardware models. *source* [29]

voltage are two important parameters that effect design area, throughput and power dissipation. Therefore, we developed our framework to include Synthesis options for varying the supply voltage.

### Synthesis Options

Certain performance metrics, such as computational precision and hardware latency, can be analyzed without synthesizing the hardware design. Therefore, our PAF includes an option for generating hardware designs without synthesizing them. This substantially reduces the time to analyze precision and latency for tens of hardware implementations within the design space. Designers are still required to synthesize a hardware design when measuring circuit level performance metrics such as area, delay, and power

dissipation.

The synthesis time for a hardware design typically depends on the complexity of the architecture. Computationally intensive hardware designs that include a large number of complex blocks, such as multipliers, prolong synthesis times. Therefore, we incorporate a method for accelerating the synthesis process for such designs. The accelerated synthesis process is useful for searching a design space that includes tens of hardware permutations. Our framework additionally includes the conventional synthesis option that allows designers to accurately estimate the performance for one or two hardware structures in the design space. The accelerated synthesis process was discussed in detail in Section 3.1. We illustrate the improvements in synthesis times over conventional methods in Section 7.1.

**Technology Feature Size and Supply Voltage ($V_{DD}$) scaling**

Our framework currently characterizes area, delay, and power dissipation using different $V_{DD}$ values for the $0.18\mu m$ standard cell library from Oklahoma State University [58] which utilizes scalable CMOS rules. The designer can vary the supply voltage below or beyond the 1.8 $V$ nominal value in order to evaluate the effects of $V_{DD}$ scaling on all three performance metrics. This is especially useful when considering methods for reducing power dissipation and improving throughput using pipelining techniques [33].

## 3.5 Chapter Summary

We presented our GUI-based performance analysis framework used to guide a designer in selecting a DSP hardware design that performs well for specific constraints. Our framework is important in reducing the design efforts and guiding the designer through the architectural selection process. Users begin at the algorithmic level by selecting the DSP function from the framework's menu of functions. The framework scripts convert the DSP algorithm to fixed-point hardware models where the designer can analyze the effects of quantization and overflow on signal integrity. The user then selects design parameters which are used to shape the hardware structures at the architectural level. Additional scripts in our framework expand the design space to tens of possible hardware permutations with varying performance results. Our modular design methodology is necessary for promoting module reuse when considering alternative hardware structures. Additionally, the frame-

work hardware designs conform to recognizable structures that are simple to modify for further architectural refinement. Our framework estimates important performance metrics at the circuit level while varying low-level parameters such as pipeline depth and supply voltage. The framework cost functions are essential for efficiently assessing the quality of hardware structures in large design spaces. The next three chapters illustrate the usefulness of our framework in selecting hardware designs for DSP applications that require filtering as a major part of the algorithm.

# Chapter 4

# Case Study I: Digital Finite Impulse Response Filters

The previous chapter described the methodology and flow we used for refining a DSP algorithm to several hardware implementations that meet design constraints. This chapter presents the merits for our PAF using the basic, yet computationally intensive FIR filter. We illustrate that our framework expands the design space for an FIR filter implementation to a set of architectures that conform to a recognizable set of hardware designs with varying performance results. We applied our architectural selection methodology using cost functions that reduce the search space to a smaller set of hardware designs that meet specific performance constraints.

The chapter begins by presenting the details for the FIR filter at different levels of abstraction while highlighting the different filter structures our framework generates. Next, the chapter describes the important hardware optimizations applied using the PAF to the filter structures at both the arithmetic and architectural levels. The chapter then elaborates on the process of using the hardware performance metrics and cost functions to guide a system designer in selecting an FIR filter structure that performs well for given design constraints. The chapter concludes by exploring architectural variations for three design examples that require FIR filter blocks as part of a DSP algorithm.

## 4.1 Algorithmic Level

The essential task for a digital filter is to remove undesired components embedded within a signal or to extract useful information. The general algorithm for a digital filter can be modeled using the following difference equation:

$$y[n] = -\sum_{k=1}^{N} a(k)y[n-k] + \sum_{k=0}^{M} b(k)x[n-k] \tag{4.1}$$

where $N$ is the order of the feedback coefficients, $a(k)$ and are the feedback filter coefficients, $M$ is the order for the feed-forward coefficients, $b(k)$ are the feed-forward filter coefficients, $x[n]$ is the filter input signal, and $y[n]$ is the filter output. This class of filters uses the feedback from the output to the input and is referred to as an IIR or recursive filter. An alternative form for a digital filter assumes no feedback from output to input. This class of filters is referred to as an FIR filter and can be modeled using the following difference equation:

$$y[n] = \sum_{k=0}^{M} b(k)x[n-k] \tag{4.2}$$

The filter order is determined by the integer value $M$, which in turn depends on the parameters used to design the FIR filter. FIR filters are primarily used to attenuate undesired frequencies from a signal. Ideal filters are difficult to realize in practical systems. Therefore, several parameters exist for designing digital filters such that the frequency response closely matches the desired filter specifications. The following are examples of parameters used to compute the FIR filter coefficients [30]:

1. Filter type: Highpass, lowpass, bandstop, bandpass
2. Sampling frequency
3. Frequency band(s): stopband(s), passband(s).
4. Passband ripple
5. Stopband attenuation
6. Window method: examples include Bartlett, Blackman, Hamming, and Hanning

MATLAB includes a Signal Processing Toolbox that permits the algorithm developer to efficiently design the filter coefficients based on the above parameters. Additionally, designers can use MATLAB to analyze the algorithmic performance for the FIR filter in order to determine whether the filter meets high-level design constraints such as bit error rate (BER) and signal-to-noise ratio (SNR), depending on the DSP application.

Figure 4.1 illustrates the effects the filter parameters have on shaping the frequency response for a lowpass FIR filter as an example. The goal for the system designer is to generate the filter coefficients that meet the filter specifications while keeping the filter order $M$ to a minimum. The order governs the complexity of the filter design at lower levels of abstraction. In general, larger values of $M$ result in a frequency response that closely matches an ideal filter. However, this comes at a cost of increased design area, lower throughput, and excessive power dissipation.

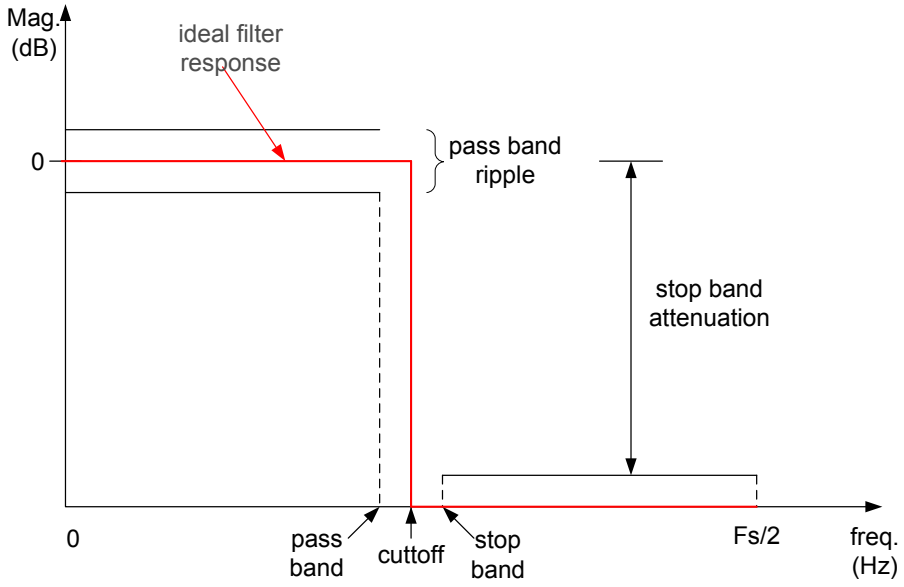



Figure 4.1: Lowpass FIR filter frequency response

Our PAF provides the designer several FIR filter algorithms catered towards specific DSP applications. The general form for the FIR filter can be used for initially analyzing the performance of a system that requires filtering. At this level of design abstraction, the designer is primarily concerned with meeting the filter specifications and assessing its algorithmic performance. We utilize MATLAB's built-in FIR filtering functions as a basis for

comparing the algorithmic accuracy for filter designs modeled at lower levels of abstraction. Therefore, our framework generates a MATLAB testbench as well as input and output reference signals. The signals generated by the testbench are used to assess design parameters at the arithmetic level where a designer focuses on maintaining fixed-point signal integrity.

## 4.2 Arithmetic Level

Digital FIR filters can be represented using the finite difference equation (4.2) assuming that the filter coefficients are computed using the parameters provided by the system designers. The filter coefficients can initially be computed using MATLAB's Filter and Design Analysis (FDA) Toolbox [59], which are represented using floating-point computations. The filter input and output can take on any floating-point value permitted by the general purpose processor. In a practical hardware system, the floating-point coefficients are represented in a form useful for hardware processors which are stored in data registers with finite lengths. Fixed-point arithmetic increases the computational efficiency when implementing a DSP algorithm on a hardware platform. However, errors associated with rounding and quantization are induced into the fixed-point filter implementation which can potentially degrade the overall system performance. Therefore, designers assess the appropriate word lengths, quantization modes, and overflow methods in order to reduce rounding and quantization effects.

### 4.2.1 Data-Type Representation

Our PAF provides the user options to select the data-type required for fixed-point computations. We use the $Qx.y$ notation to model the input, coefficients and output using fixed-point numbers, where $x$ represents the integer portion of the number and $y$ represents the fractional portion [60]. The sum $x+y+1$ represents the fixed-point word size, where the 1 is reserved for the sign bit. The $Q0.(w-1)$ form is a subset of the fixed-point representation which we use to represent the signals for the DSP algorithms we consider in our work. The fixed-point implementation for the FIR filter assumes the floating-point data and coefficients are normalized to lie in the range $(-1,\ 1)$. Our framework scales the data such that the normalized fixed-point values lie within the range $[-(1 - 2^{-(w-1)}),\ 1 - 2^{-(w-1)}]$. Our

framework allows the designer to select the finite word sizes used for implementing the fixed-point FIR filter. A $C^{++}$/SystemC testbench generated by the PAF permits the designer to measure rounding errors and assess the affects of the different types of quantization modes.

### 4.2.2 Data Scaling

The input data is quantized for most hardware implementations that require FIR filters. However, for cases where the input data into the FIR filter is represented using floating-point numbers, a simple method for converting the floating-point data samples to fixed-point form could be employed. This requires normalizing the floating-point data samples and then scaling the values by an integer factor of $C_1$ as the following equation illustrates:

$$f[n] = x[n] \cdot C_1 \tag{4.3}$$

where $f[n]$ is the fixed-point input data and $C_1$ is a non-integer scaling factor. This method for scaling the data assumes that the designer has prior knowledge of the input values. For the case that there is no prior knowledge of the input data, then the quantizer effectively bounds the input by clipping data samples that exceed the maximum or minimum representable fixed-point values. Therefore, the range of fixed point data samples lie between $[-(1 - 2^{(w-1)}),\ (1 - 2^{(w-1)}]$ in signed twos complement form. This form of clipping is equivalent to saturation when considering overflow modes in fixed-point computations.

The scaling factor for $C_1$ can be determined using several methods. Proakis and Manolakis present a "pessimistic constraint" for scaling the input data to guarantee the avoidance of overflow [30]. For this case, the filter coefficients can lie in the range $[-(1 - 2^{-(w-1)}),\ (1 - 2^{-(w-1)}]$ and the normalized input data is scaled by a factor of

$$C_1 < \frac{1}{\sum_{k=0}^{M} |b(k)|} \tag{4.4}$$

An alternate method for converting the floating-point input samples to fixed-point requires normalizing the data and then scaling the values by a factor equal to the largest representable positive fixed-point value. Therefore, the scaling factor for the data can simply be computed as

$$C_1 = 2^{w-1} - 1 \tag{4.5}$$

This method for scaling the data requires an improved approach for scaling the filter coefficients such that we guarantee avoiding overflow while minimizing the effects of rounding.

The following section summarizes the method we employed for data and filter coefficient scaling.

### 4.2.3 Coefficient Scaling to Avoid Overflow

The method we used for scaling the filter coefficients is slightly more involved than scaling the input data samples, since we assume the designer has prior knowledge of the floating-point values. Our framework uses the following methodology for scaling the filter coefficients while considering the dynamic range for the input data [30].

We consider the FIR filter difference equation presented earlier in Equation (4.2). If we assume that the upper limit for the dynamic range of the filter output is $C_3$ such that $0 < C_3 < \infty$, then the scaling conditions can be expressed as

$$\begin{aligned} |y[n]| &= \left| \sum_{k=0}^{M} b(k)x[n-k] \right| \\ &\leq \sum_{k=0}^{M} |b(k)x[n-k]| \\ &\leq \sum_{k=0}^{M} |b(k)|\,|x[n-k]| \end{aligned} \tag{4.6}$$

where we applied Parsevals theorem to obtain the inequalities. This can be further simplified such that

$$0 < |x[n]|\,C_1 < \infty \tag{4.7}$$

where $C_1$ is the largest positive value in the dynamic range for the input. Therefore, the output range can be represented as

$$|y[n]| \leq \sum_{k=0}^{M} |b(k)|\,C_1 < C_3 \tag{4.8}$$

where $C_3$ is the largest positive value in the output range. The condition for the FIR filter coefficients to avoid overflow can be expressed as

$$\sum_{k=0}^{M} |b(k)| < \frac{C_3}{C_1} \tag{4.9}$$

This method for scaling the filter coefficients can be integrated with scaling the input data samples such that the fixed-point implementation for the filtering algorithm

guarantees that the arithmetic computations do not result in overflow. The following process describes the method we use for scaling the input signals and filter coefficients to ensure the avoidance of overflow.

1. The input signal is scaled such that $f[n] = C_1 x[n]$,
2. The coefficients are scaled such that $\hat{b}(k) = C_2 b(k)$,
3. The scaled output is obtained by assuming $g[n] = C_1 C_2 y[n] = C_3 y[n]$,
4. The floating-point, unscaled output can be determined by $\tilde{y}[n] = \frac{g[n]}{C_1 C_2}$

We can determine the scale factors $C_1$ and $C_2$ by solving the system in the frequency domain. Therefore,

$$\begin{aligned} H(z) &= \frac{Y(z)}{X(z)} = \frac{C_1 Y(z)}{C_1 X(z)} \\ F(z) &= C_1 X(z) \\ H_2(z) &= C_2 H(z) = \frac{G(z)}{F(z)} \end{aligned} \tag{4.10}$$

where

$$\begin{aligned} G(z) &= C_1 C_2 Y(z) \\ Y(z) &= \frac{G(z)}{C_1 C_2} \end{aligned} \tag{4.11}$$

The modified transfer function $H_2(z)$ can be implemented as

$$\begin{aligned} H_2(z) &= C_2 H(z) = C_2 \sum_{k=0}^{M} b(k) z^{-k} \\ \frac{G(z)}{F(z)} &= \sum_{k=0}^{M} C_2 b(k) z^{-k} \end{aligned} \tag{4.12}$$

Since $C_2 b(k)$ are the scaled coefficients defined earlier as $\hat{b}(k)$, then the scaled transfer function for the output can be represented as

$$G(z) = F(z) \sum_{k=0}^{M} \hat{b}(k) z^{-k} \tag{4.13}$$

or can be represented using the following difference equation

$$g(n) = \sum_{k=0}^{M} \hat{b}(k) f(n-k) \tag{4.14}$$

We can scale the FIR filter coefficients to avoid overflow by selecting $C_2$ such that

$$y_{max} \leq x_{max} \sum_{k=0}^{M} |b(k)|\, C_2 \tag{4.15}$$

where $x_{max} = 2^{w-1} - 1 \geq |x[n]|$, and therefore, the maximum permitted value for $C_2$ is

$$C_2 = \frac{y_{max}}{x_{max} \sum_{k=0}^{M} |b(k)|} \tag{4.16}$$

The overflow method governs the integrity of the signal whereas the quantization method controls the amount of errors introduced into the fixed-point model for the FIR filter algorithm. We used the following setup to evaluate the computational precision for the fixed-point filter output compared to the floating-point output:

1. $12^{th}$ order FIR filter using the TF algorithm
2. Lowpass filter with normalized cut-off at 0.3 $rad/s$
3. Highpass filter with normalized cut-off at 0.7 $rad/s$
4. Hamming window (default using MATLABs `fir1` routine)
5. 1250-long floating-point input sample in the range $[7.5, \ -7.5]$
6. Input signal and coefficients scaled to 16-bit words
7. Output and internal signals maintained at 32-bit words

We used the absolute peak error and mean square error (MSE) to measure the quantization effects in the filter output. We defined the MSE as:

$$MSE = \frac{1}{K} \sum_{i=1}^{K} (y[n] - \hat{y}[n])^2 \tag{4.17}$$

where $y[n]$ is the floating point output for the FIR filter, $\hat{y}[n]$ is the scaled fixed-point output, and $K$ is the length of the filter output signal.

Table 4.1 summarizes different methods for converting the floating-point data and coefficient values to fixed-point in order to perform the computations. The tabulated results reflect the performance for the highpass filter since similar observations could be made using the lowpass filter. For simplicity, we constrained the quantization mode to **round** and varied

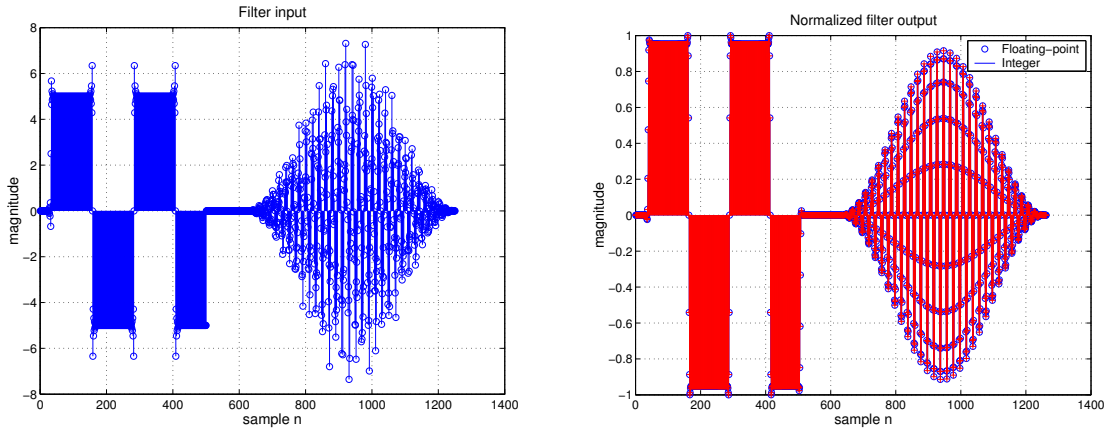
the scaling factors for Methods 1 and 2 until overflow was avoided. The results show that we lose one bit for scaling the coefficients in Method1 and 1 bit for scaling the data in Method 2 to avoid overflow. Methods 1 and 2 did not guarantee the avoidance of overflow for other filter parameters. Method 3 used the coefficient scaling method of Equation (4.4) which avoided overflow. However, this method was too constrained and therefore, we increased the output dynamic range using Method 4 and still avoided overflow. For sake of consistency, we employed Method 4 to quantize the filter input and coefficients generated by our framework.

Table 4.1: Different methods for converting filter input and coefficients from floating-point to fixed-point

| Method | Input Sample Scale Factor | Filter Coefficients Scale Factor | Filter Output Scale Factor | MSE $\times 10^{-9}$ | Peak Error $\times 10^{-4}$ | No Overflow Guaranteed |
|---|---|---|---|---|---|---|
| 1 [30] | $\frac{2^{-15}-1}{x_{max}}$ | $\frac{2^{-14}-1}{b_{max}}$ | 556111842 | 1.26 | 1.39 | No |
| 2 [19] | $\frac{2^{-14}-1}{x_{max}}$ | $\frac{2^{-15}-1}{b_{max}}$ | 556160774 | 3.86 | 2.34 | No |
| 3 [30] | $\frac{2^{-15}-1}{x_{max}}$ | $\frac{2^{-15}-1}{\sum_{k=0}^{M}\|b(k)\|}$ | 302284720 | 18.80 | 3.03 | Yes |
| 4 | $\frac{2^{-15}-1}{x_{max}}$ | $\frac{2^{31}-1}{x_{max}\sum_{k=0}^{M}\|b(k)\|}$ | 604612728 | 2.38 | 1.56 | Yes |

Figure 4.2(a) illustrates the input sample used for experimentally measuring quantization and rounding errors. Figure 4.2(b) compares the floating-point filter output to the fixed-point output quantized using the **round** mode. The plot for the fixed-point filter output closely matched the floating-point output and any discrepancies would be difficult to assess visually. Therefore, Table 4.2 summarizes the MSE and peak errors for the different quantization modes used in the fixed-point implementation for the FIR filter algorithm. The results show that the **round** mode generated the least MSE and peak error for both the lowpass and highpass filters. The **floor** mode generated the largest errors for the lowpass filter whereas the **ceil** mode generated the largest errors for the highpass filter. While it seems intuitive to use the **round** mode in the hardware implementation, this tends to be the most complex of the four quantization modes. The **floor** mode is the simplest quantization method to implement in hardware and is therefore commonly used for fixed-point

DSP implementations.




(a) Filter input signal

(b) Filter output

Figure 4.2: Input and output plots for an $12^{th}$ order FIR filter - floating point versus fixed-point computations

Table 4.2: Effects of quantization on fixed-point implementation for $12^{th}$ order FIR filter

| Quantization Mode | Lowpass Filter | | Highpass Filter | |
|---|---|---|---|---|
| | MSE $\times 10^{-10}$ | Peak Error $\times 10^{-5}$ | MSE $\times 10^{-10}$ | Peak Error $\times 10^{-5}$ |
| round | 1.39 | 4.76 | 8.91 | 8.93 |
| ceil | 26.64 | 11.04 | 315.00 | 36.19 |
| fix | 17.43 | 7.59 | 71.29 | 20.19 |
| floor | 47.39 | 16.50 | 135.66 | 26.82 |

## 4.3 Architectural Level

The main computational modules required for the hardware implementation of an FIR filter are the multipliers, adders, and delay units or registers. Area, delay, throughput, computational precision and power dissipation can be used as one-dimensional measures of performance to assess the quality of a filter design. Additional metrics can be used to quantify the performance for alternative FIR filter hardware structures. In Chapter 3, we discussed the use of two-dimensional cost functions that evaluate the quality of a hardware design while constraining at least one of the performance metrics. The filter structures generated by our PAF conform to a regular set of structures that utilize basic computational modules constructed in a manner that optimizes one or more performance metrics. The use of recognizable hardware structures allows a designer to modify the filter architectures if they wish to further optimize the performances at lower levels of abstraction.

This section presents the vast filter structures generated by our framework. The designer specifies high-level parameters using our GUI-based interface. Examples of high-level parameters include filter order and fixed-point word sizes. The framework proceeds to generate the various filter structures in the design space and estimates the performance for each design. The designer can then use our framework to select the necessary cost functions to compare the performance of the filter structures in the design space. Our method for architectural evaluation assists the designer in choosing appropriate filter designs without overwhelming him or her with laborious decision making.

### 4.3.1 FIR Filter Structures

Several FIR filter structures exist that behaviorally implement the convolutional sum of Equation 4.2. The hardware structure for a filter can initially be represented using a SFG which determines the order of mathematical operations. The SFG for the DF FIR filter, illustrated in Figure 4.3(a), is a direct implementation for the convolutional sum. Each output is generated by simultaneously computing the product between the delayed input samples and the filter coefficients. The chain of adders in the DF structures accumulates the multiplier outputs. Therefore, the time interval between two output samples is determined by the designs critical path delay given as

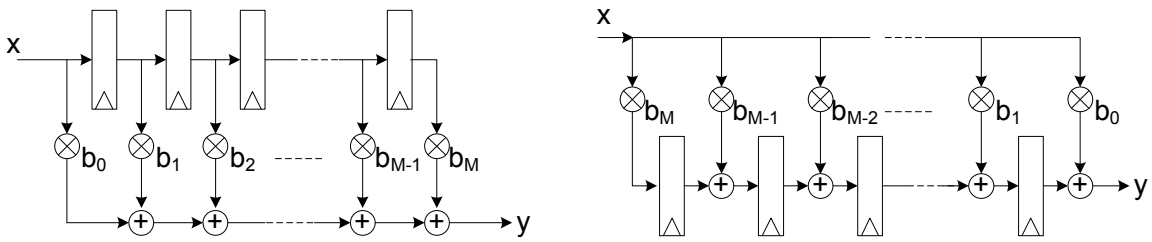
$$T_{d,max}^{DF} = T_{d,mult} + MT_{d,add} + T_{d,reg} \tag{4.18}$$

where $T_{d,mult}$ is the critical path delay in the multiplier, $T_{d,add}$ is the critical path delay in the adder, and $T_{d,reg}$ is the setup and hold times required by the register. The smallest clock period must therefore be at least equal to $T_{d,max}$ for correct operation. The throughput for the DF FIR filter can simply be computed as $1/T_{d,max}$.

An alternative filter structure can be derived by transposition of the SFG in Figure 4.3(a). The TF FIR filter is obtained by reversing the order for the filter coefficients and alternating the placement of registers in the chain of adders. Figure 4.3(b) shows the SFG and block diagram for the TF FIR filter. The TF structure features single memory elements separated by adders whereas the DF structure is characterized by a continuous shift register. The sizes for the registers in the TF structure are twice as large as the registers in the DF structure, which results in a slightly larger area. However, the critical path delay in the TF structure is reduced to just a single multiplier, adder and register and can be computed as

$$T_{d,max}^{TF} = T_{d,mult} + T_{d,add} + T_{d,reg} \tag{4.19}$$

For large order filters, the reduction in critical path delay for the TF is less than the DF filter. However, the TF filter structure exhibits large broadcasting in the input which requires a larger supply voltage than the DF structure to drive the input signal to all the multipliers, which increases the power dissipation.



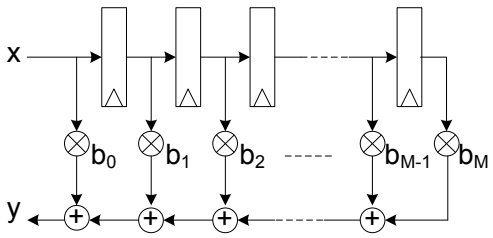

(a) Direct Form FIR filter

(b) Transpose Form FIR filter

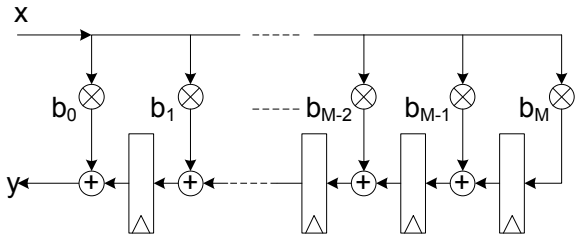
Figure 4.3: SFG for basic FIR filter structures

The SFG for the two basic FIR filter structures illustrated in Figure 4.3 form the basis for additional filter structures with alternate performance metrics. The TFII and DFII filter structures, shown in Figure 4.4(a) and 4.4(b) respectively, are obtained by reversing the direction of signal flow in all branches and interchanging the input and output. The critical path delay in the chain of adders for the DF structure can be reduced by utilizing a

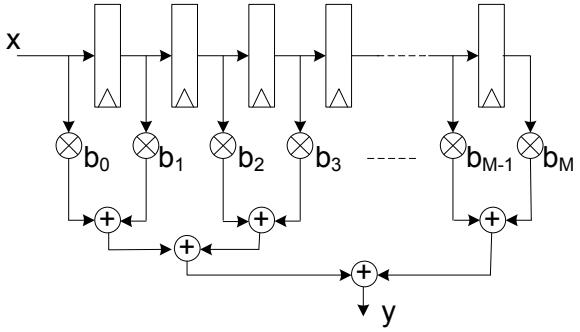
tree adder structure, shown in Figure 4.4(c). Our PAF constructs the FIR filter structures presented in this section and applies additional optimizations to further improve the area, delay, hardware latency, throughput, and power dissipation.




(a) TFII FIR filter




(b) DFII FIR filter




(c) DF FIR filter with Tree Adder

Figure 4.4: SFG for TFII, DFII, and Tree-adder DF FIR filter structures

### 4.3.2 Second Order Sections

The dynamic range for fixed-point signals effects round-off errors, especially in high-order filters. Second Order Sections (SOS) reduce the sensitivity to coefficient quantization by increasing the dynamic range of the coefficients [61]. However, modeling a filter using SOS increases the overall hardware complexity for the filter and may potentially degrade the throughput and power dissipation, depending on the structure of the filter. For example, an $8^{th}$ order FIR filter implemented in cascade form using 16/32-bit word sizes results in a MSE in the order of $\left(8 \times (2^{-16})\right)^2 = 10^{-8}$ while using a total of 9 multipliers. Whereas a SOS design for the same filter parameters results in quantization errors in the

order of $\left(2 \times (2^{-16})\right)^2 = 10^{-10}$ while using a total of 12 multipliers. Therefore, the SOS design reduces round-off errors by 2 orders of magnitude for a 33% increase in area. Table 4.3 summarizes the performance results collected for both filter structures: direct implementation and SOS structure. We used our PAF to construct an $8^{th}$ order lowpass FIR filter, with 16/32-bit word sizes, synthesized using the $0.18\mu m$ standard cell library. We used the input signal displayed earlier in Figure 4.2(a) to experimentally measure the MSEs. The results show that the SOS design exhibits 33% more area than the direct implementation while reducing the MSE by one order of magnitude. This closely matched the analytical conclusions we made. The advantage of using our PAF allows a designer to efficiently compare both arithmetic and hardware performances for various design options while avoiding laborious analytical methods.

Table 4.3: Performance metrics for $8^{th}$ order SOS FIR filter compared to direct implementation

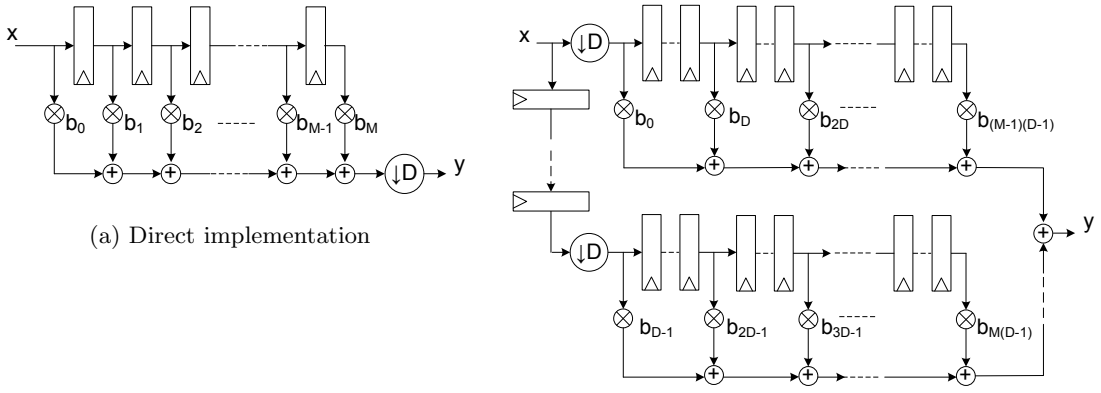
| Filter Structure | Area ($\mu m^2$) | Delay ($ns$) | Hardware Latency | Throughput ($Msamp/sec$) | Power ($mW$) | MSE $\times 10^{-10}$ |
|---|---|---|---|---|---|---|
| Direct | 513027 | 13.59 | 1 | 74 | 40 | 13.02 |
| SOS | 684036 | 24.63 | 1 | 41 | 53 | 1.19 |

### 4.3.3 Polyphase Structures for Multirate Signal Processing

Multirate filtering is ideally used for FIR filters that exhibit narrow transition bands, narrow passbands or wide passbands. These filters are generally not practical to implement as basic cascaded FIR filters since they tend to exhibit large filter lengths and inefficient computations. Polyphase structures are effective design options for multirate signal processing where up-sampling and down-sampling filters are required, such as quadrature mirror filter banks [62]. A down-sampling filter can easily be implemented using a regular filter followed by a downsampler, as shown in the SFG of Figure 4.5(a). However, this design degrades the hardware utilization, since only $1/D$ of the filtered signal is retained, where $D$ is the down-sample factor. Similarly, an up-sampling filter can be implemented

by upsampling the input signal prior to filtering, which also introduces inefficiencies in the hardware utilization. The polyphase structure increases the hardware utilization by using subfilters that filter the signal in parallel. This is illustrated in Figures 4.5 and Figure 4.6 for the down-sampling and up-sampling FIR filters, respectively. The polyphase filter structures are of comparable complexity to the direct implementation; the advantage of using the polyphase structure is improved hardware utilization, which reduces power dissipation for a given throughput constraint.

Each subfilter for the polyphase structure can be implemented using any of the filter structures in Figures 4.3 and 4.4. Our framework generates different structures for the subfilters used to implement the polyphase architecture. The dimension for the polyphase structure depends on the down-sampling (or up-sampling) factor. Our framework alerts the designer for cases where the factor is not a divisor of the filter order. The designer can over-ride this warning by forcing the framework to proceed with constructing the filters. Currently, we simplify the process for dimension mismatch my extending the filter coefficients with zeros in order to correctly construct the polyphase filters.



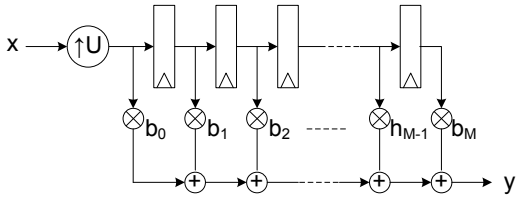

(a) Direct implementation

(b) Efficient decimation filter

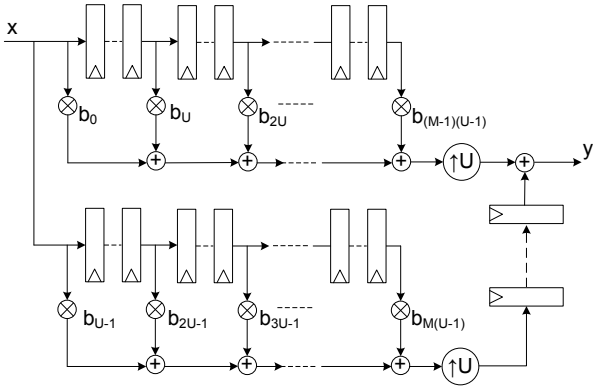
Figure 4.5: SFG for down-sampling filters: down-sample rate = $D$

### 4.3.4 Symmetric FIR Filters

Linear-phase FIR filters are widely used in DSP systems where phase distortion in the vicinity of the cutoff frequency cannot be tolerated. Linear phase results in a con-




(a) Direct implementation




(b) Efficient interpolation filter

Figure 4.6: SFG for up-sampling filters: up-sample rate = $U$

sistent delay for all frequencies that pass through the filter. Therefore, the filter does not cause phase (or delay) distortion. The distortionless property is developed by imposing a symmetric relationship on the coefficients for the linear-phase FIR filters [63]:

$$b(k) = b(M-k) \quad or \tag{4.20}$$

$$b(k) = -b(M-k) \tag{4.21}$$

where $M$ can be even or odd. The symmetry in Equation (4.21) is known as anti-symmetry. The linear-phase property can be derived using the transfer function for the FIR filter and extracting a delay term corresponding to half the duration of the unit-sample response [64]

$$H(e^{j\omega T}) = e^{-j\omega TM/2} \sum_{k=0}^{M} b(k) e^{-j\omega T(k-M/2)} \tag{4.22}$$

We can make a substitution for $b(k)$ in Equation (4.22) using Equation (4.20):

$$H(e^{j\omega T}) = e^{-j\omega TM/2} \sum_{k=0}^{M} b(M-k) e^{-j\omega T(k-M/2)} \tag{4.23}$$

Substituting $r = n - k$ in Equation (4.23) results in

$$\begin{aligned} H(e^{j\omega T}) &= e^{-j\omega TM/2} \sum_{r=0}^{M} b(r) e^{-j\omega T(-r+M/2)} \\ &= e^{-j\omega TM/2} \sum_{r=0}^{M} b(r) e^{j\omega T(r-M/2)} \end{aligned} \tag{4.24}$$

We can combine Equations (4.22) and (4.24) since they are complex conjugates. The result after adding the two equations and dividing by 2 is

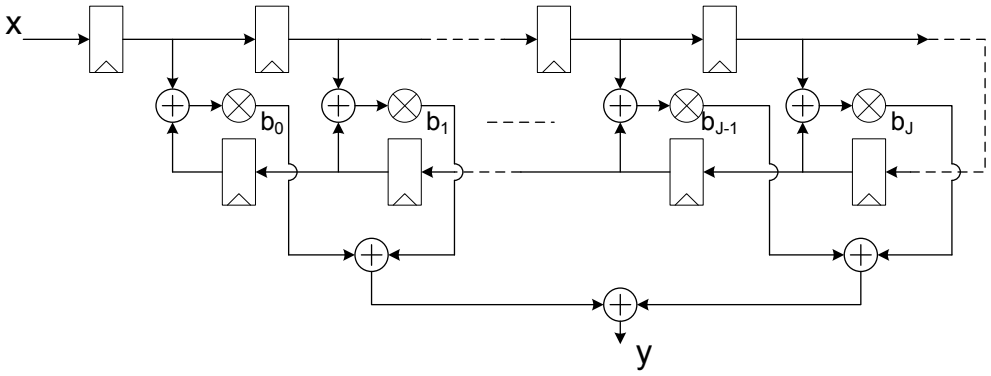
$$H(e^{j\omega T}) = e^{-j\omega TM/2} \sum_{k=0}^{M} b(k) \cos\left(k - M/2\right) \tag{4.25}$$

The exponential function in front of the real-valued sum represents a linear-phase transfer function that causes a signal time delay of $MT/2$ which does not influence the magnitude.

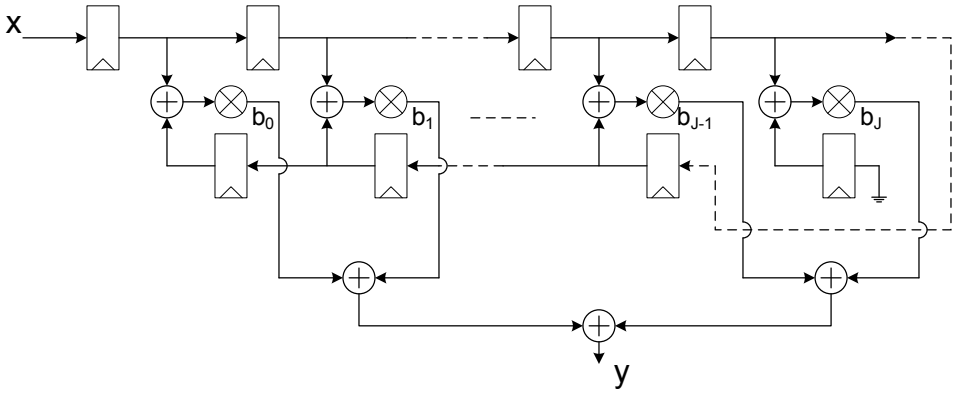
Our framework generates both even and odd symmetric and anti-symmetric FIR filters for applications that require linear-phase filtering. The symmetry in the linear-phase FIR filters is commonly exploited to reduce the number of multipliers by half, which significantly reduces design area and power dissipation. Figure 4.7 illustrates one of several structures generated by our framework for even and odd tap linear-phase FIR filters. Both even and odd filter structures are similar with one minor distinction in the way the data at the end of the filter is connected to the sequence of registers. The designer can use our framework to generate the linear-phase filter structures according to the DSP application. Biorthogonal filters are examples of symmetric FIR filters commonly used in signal and image compression implementations such as the DWT block in the JPEG2000 compression engine [65].

### 4.3.5 Interleaving for Multi-Signal Processing

Data interleaving is one method for efficiently processing multiple independent signals using a single filter structure. A filter having transfer function $H(z)$ can be interleaved




(a) Even-tap: $J = \frac{M-1}{2}$




(b) Odd-tap: $J = \frac{M}{2}$

Figure 4.7: SFG for even and odd linear-phase FIR filters using DF structure

by replacing the $z^{-1}$ in the transfer function of Equation (4.2) by $z^{-k}$ where $k$ is an integer $\geq 2$ [66]. The interleaved $Z$-transform can be modeled as

$$Y(z^k) = X(z^k)H(z^k) \tag{4.26}$$

This effects the throughput for the individual input and output sequences. This can be observed by analyzing the $Z$-transform for the input sequence $x[n]$ where

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{4.27}$$

We can define a new sequence $\hat{x}[n]$ with $Z$-transform $X(z^k)$ given by

$$\begin{aligned} \hat{X}(z) &= X(z^k) = \sum_{n=0}^{\infty} x[n] z^{-nk} \\ &= \sum_{n=0,k,2k,...}^{\infty} x[\frac{n}{k}] z^{-n} = \sum_{n=0}^{\infty} \hat{x}[n] z^{-n} \end{aligned} \tag{4.28}$$

Therefore, the sequence $\hat{x}[n]$ can be defined as

$$\hat{x}[n] = \begin{cases} x[\frac{n}{k}], & \text{for n=0, k, 2k} \\ 0, & \text{elsewhere} \end{cases} \tag{4.29}$$

The modified input sequence is a zero-interleaved sequence of the signal $x[n]$. Similarly, the filtered output $\hat{y}[n]$ contains $(k-1)$ zero samples between successive samples of $y[n]$. Alternatively, the zeros in the interleaved input sequence can be replaced by additional independent sequences $x_2[n], x_3[n], ..., x_k[n]$. This generates $k$ independent output sequences $y_1[n], y_2[n], ..., y_k[n]$ where the throughput for each output is reduced by a factor of $k$. We provide the designer the option to interleave the filter structures generated by our framework by any integer factor $k$.

The effect of placing additional registers in the FIR filter structure introduces extra delays in the transfer function which can be used for pipelining the hardware design. This is achieved by shifting the extra delays in the filter SFG to branches where data-broadcasting and excessively long critical paths need to be eliminated. The next section discusses the importance of pipelining the FIR filter structures and how we combine data interleaving with pipelining to minimize redundant placement of registers in the filter hardware designs.

### 4.3.6 Pipelining Types

Pipelining is an essential design technique for improving the throughput in DSP hardware designs such as FIR filters. The delay time for the slowest computational block is decisive in determining the throughput for the design. Pipelining reduces the design delay by placing registers in the paths of heavily clustered combinational logic. For large hardware systems, care must be taken when pipelining a design to avoid altering the behavior of the DSP algorithm.

Cut-set retiming is one method for efficiently pipelining an FIR filter while maintaining the external behavior for the algorithm [33]. Cut-set retiming separates a set of

processing elements (PE) by applying the following rules [62]:

1. All paths leading to the PEs receive a positive delay ($z$)
2. All paths leading from the PEs receive a negative delay ($z^{-1}$)

Alternatively, for the hardware structures considered in this work, we can apply the following set of modified rules to the FIR filter computational modules:

1. A cut that intersects all arcs going in the same direction receive a positive delay ($z$)
2. A cut that intersects all arcs going in the opposite direction receive a negative delay ($z^{-1}$)

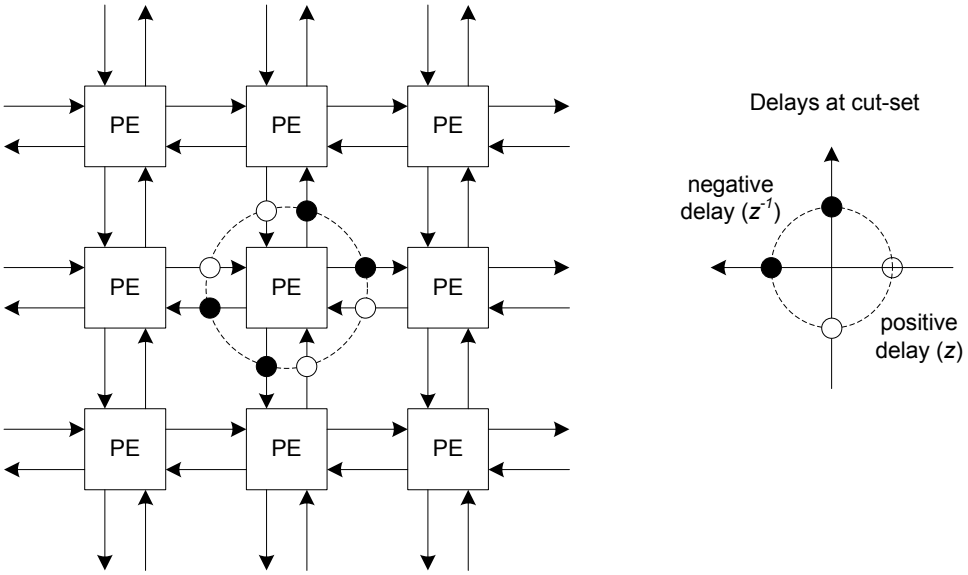
The modified rules apply for each cut only. However, it is important that the cut intersect the entire computational module for the correct placement of registers. An example for applying cut-set retiming to a processing element or computational module is illustrated in Figure 4.8.

It is not practical to add a positive delay for time invariant systems and therefore must be compensated for by delays within the computational module. This can be achieved by one or a combination of two methods:

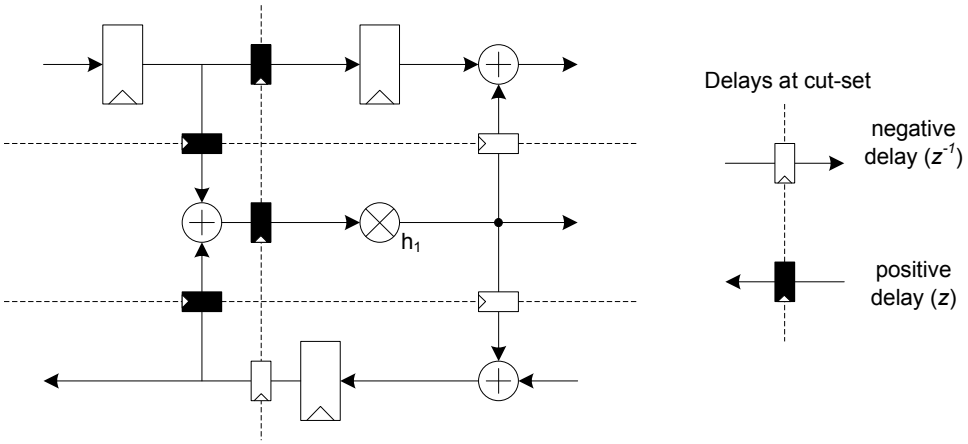
1. Shift existing delays within the computational module
2. Double the delays (interleave be a factor of two), then remove one delay after a cut

The second rule applies to computational modules that do not contain a delay in the path from input to output. The TFII and DFII require doubling the delays before we can apply cut-set retiming to the computational modules. Ideally, it is more effective to double delays, then add negative delays to arcs while removing delays from interleaved arcs. Doubling and shifting delays may be necessary to achieve an efficiently pipelined design. Our PAF relieves the designers from having to manually pipeline the filter structures by applying the pipeline rules described above.

An additional application to pipelining a design using cut-set retiming is to place registers in the outputs of computational modules to minimize data broadcasting. We provide the designer options to apply cut-set retiming to the computational modules for the FIR filters generated by our framework. However, pipelining a design tends to increase the hardware latency from input to output. We overcome this by including several options




(a) Processing elements




(b) Computational module

Figure 4.8: Pipelining hardware designs using cut-set retiming

for pipelining each computational module to ensure the FIR filters exhibit high throughput and minimal hardware latencies. This allows the designer to utilize arithmetic IP blocks provided by the synthesis tools such as adders, multipliers, and pipeline multipliers. Figure

4.9 presents the different pipelining options for each filter structure generated by our PAF.

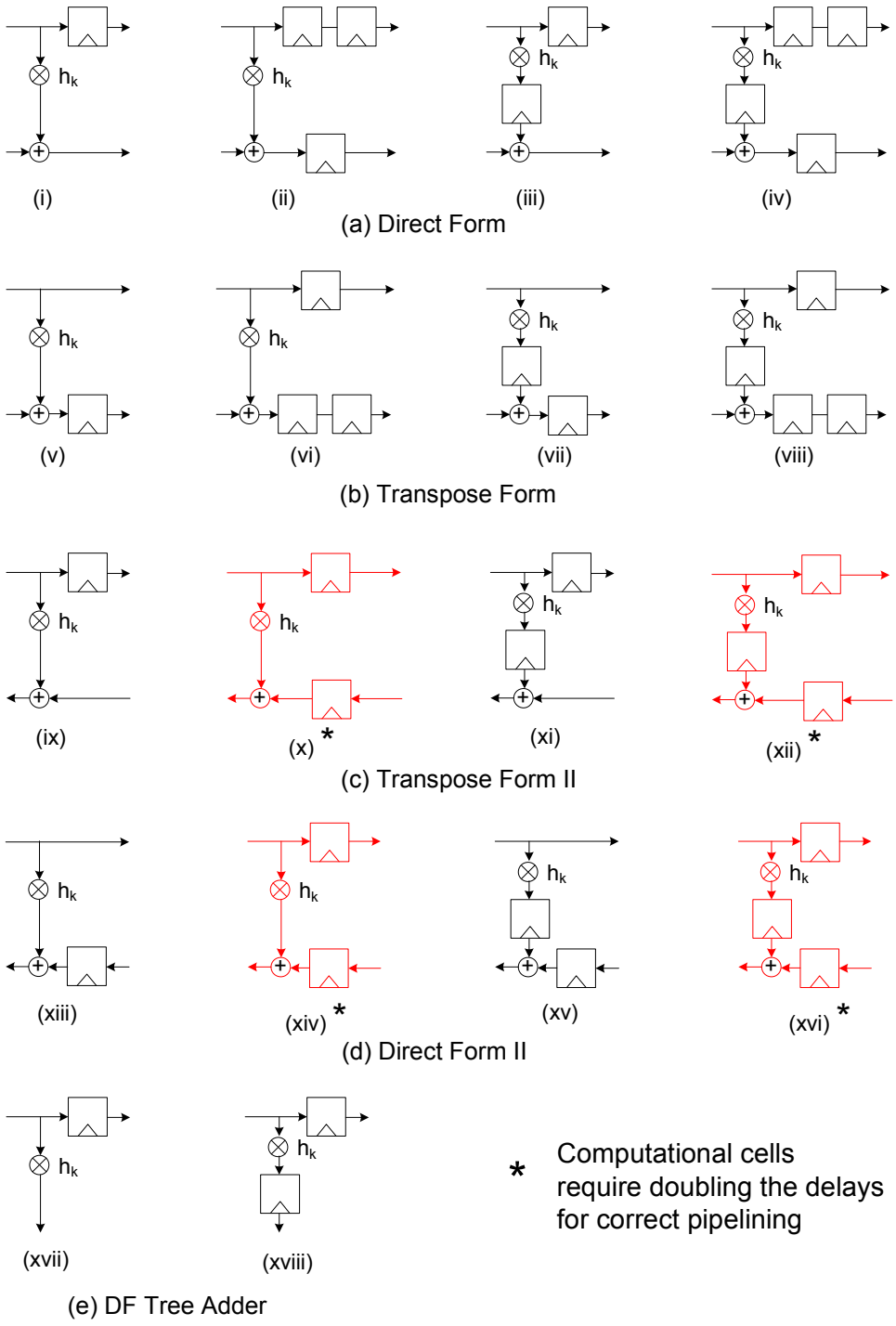



Figure 4.9: Different types of pipelined computational modules for FIR filter structures

## 4.4 Circuit Level

Designers can use our framework to specify detailed synthesis parameters at the circuit level when measuring area, delay and power dissipation. The filter structure is unaltered at this level of design abstraction and any variations in performance are governed by the layout of the arithmetic blocks that perform the filter operation. The standard cell library used to create the design netlist affects the area, delay and power dissipation. Whereas, the supply voltage provided to the hardware circuit affects the delay and power only. Therefore, different permutations in technology size and supply voltage, combined with the dozens of different filter structures, increase the design space to hundreds of possible options. This necessitates the use of cost functions to reduce the design space to a smaller, more manageable set of hardware designs that meet and possibly exceed performance specifications.

### 4.4.1 Circuit Architectures for Arithmetic Blocks

The architectures for the mathematical cores contribute to governing the overall performance of the FIR filter structures. We use optimized Booth-recoded Wallace-tree multipliers and carry-look-ahead adders available in the Synopsys IP library to construct the FIR filter structures. Designers wanting to utilize alternative mathematical circuits can accomplish this by simply altering the syntheses scripts. The performance results may differ using the alternative IP cores, however the methodology for applying the cost functions in order to reduce the design space remains effective.

### 4.4.2 Combined Pipelining and $V_{DD}$ scaling

The pitfalls of the large delay path for the DF structure and the large fan-in for the TF can be overcome using pipelining and $V_{DD}$ scaling [33] [67]. One effective method for reducing power dissipation is reducing the supply voltage. However, equation 3.1 suggests that the delay increases as $V_{DD}$ is reduced which degrades the overall design throughput. Pipelining can be applied to the hardware designs to increase the throughput to a specific value while simultaneously reducing $V_{DD}$. We observe from Equation 3.3 that reducing the supply voltage quadratically reduces power dissipation for a fixed throughput, which is highly desirable. However, manually pipelining the filter structures along with

$V_{DD}$ scaling contribute to the complexities of searching the design space for optimal filter implementations. Therefore, we use our framework to investigate the effects of combined pipelining and $V_{DD}$ scaling on the hardware performance for the filter structures.

### 4.4.3 Performance Metrics and Cost Functions

Our framework allows a designer to alter both technology size and $V_{DD}$ in order to assess different design permutations and operating parameters. We provide the designer performance metrics and cost functions for all synthesizable hardware designs generated by our PAF. Currently, our framework synthesizes the hardware designs using the $0.18\mu m$ standard cell library and characterizes the delay and power using any or all eight supply voltages $\{1.0, 1.2, 1.5, 1.8, 2.0, 2.1, 2.5, 2.7\}$. Designers wanting to analyze the performance for a specific design using a single supply voltage can do so by selecting the appropriate parameters from the PAF's menu of options.

## 4.5 Executing the Flow

We developed our framework to include numerous design options at the architectural level that result in tens of variations for the hardware implementation of a basic FIR filter algorithm. Combining this with the different circuit level synthesis options expands the design space to hundreds of design permutations with varying performance metrics. We provide the designers cost functions that guide them in reducing the design space and selecting filter structures that meet design specifications. The goal of our framework and architectural selection process is to increase the quality of hardware designs without degrading the algorithmic refinement process.

Our design and architectural selection methodology can be applied to DSP algorithms of equivalent complexity to filters. We make the assumption that designers determine the high-level parameters for the DSP algorithm at the algorithmic and arithmetic levels. For the case of the filter designs, both filter order and word sizes are determined prior to invoking our PAF. Therefore, the framework user simply specifies filter order and word sizes to our framework, and with a press of the button, our framework proceeds to expand the architectural design space. Figure 4.10 illustrates an overview for executing the flow using

our framework to design and analyze the performance for different FIR filter designs.

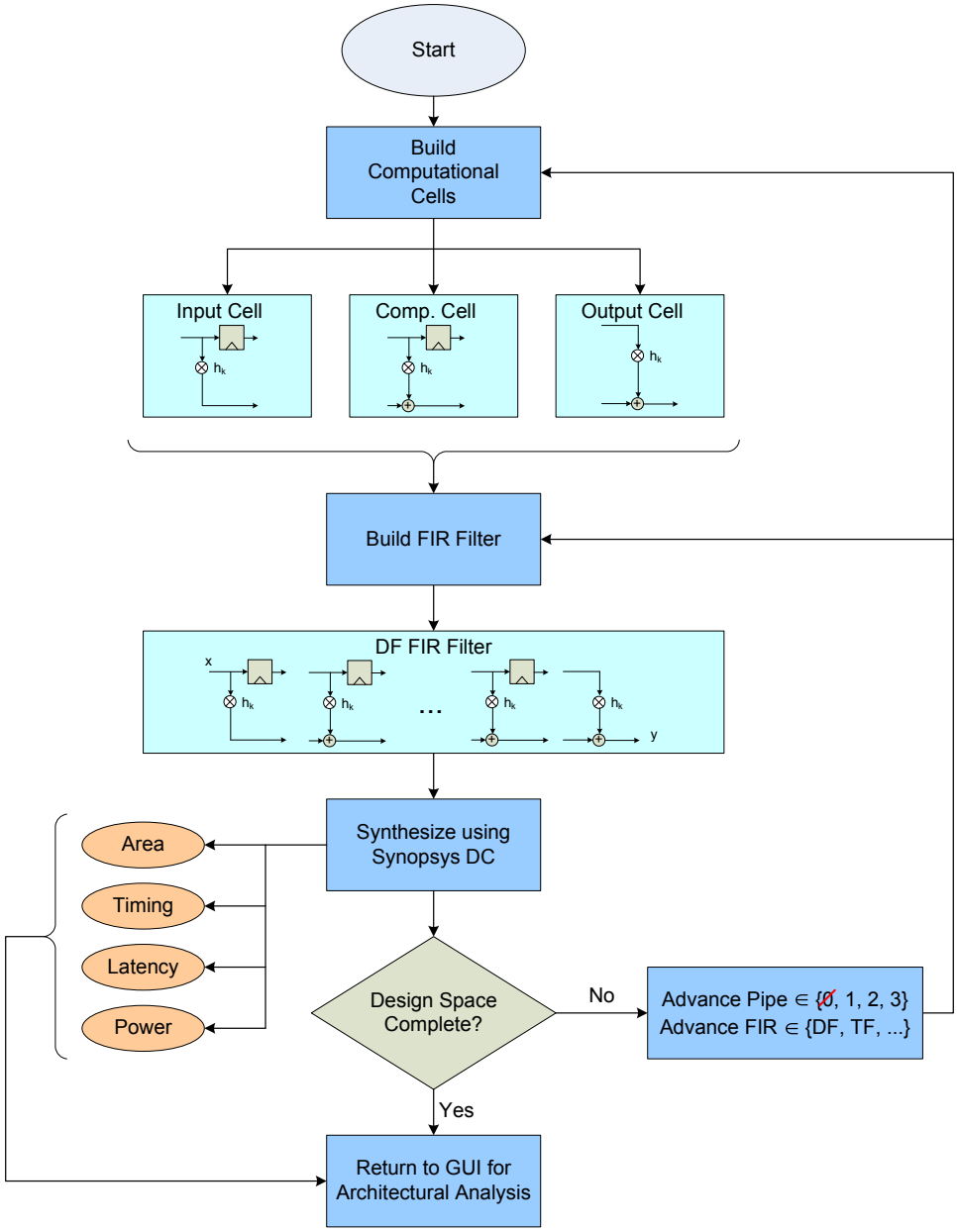



Figure 4.10: Executing the PAF flow for the architectural analysis of FIR filters

The scripts in our framework begin by generating the computational modules for the basic non-pipelined FIR filter. For example, the arithmetic blocks in the computational module of Figure 4.9 are connected to implement a single multiply/accumulate operation for

a DF FIR filter. The $C^{++}$ scripts in our framework then proceed to replicate and connect the computational modules to construct the complete DF FIR filter of Figure 4.9 (a)-(i). The design, modeled at the synthesizable RTL using Verilog, passes to the next phase of the framework which is the design synthesis. A Perl script within our framework invokes the Synopsys Design Compiler to synthesize the design and estimate the area, throughput, and power dissipation for different values of $V_{DD}$. The performance metrics are stored in MATLAB for architectural analysis using the cost functions, which commences once the framework completes generating the design space. This completes the construction and analysis for the first filter structure. The PAF then advances to the next FIR filter design by altering the mathematical elements in the computational modules to include one of the pipelining techniques. Furthermore, our framework progresses to the next FIR filter structure, such as the TF, once all possible pipelining techniques have been implemented for a specific filter structure. Table 4.4 summarizes the computational modules used to implement the different FIR filter structures. The complete flow executes in seconds or minutes for each design, depending on the filter order and the word sizes. Chapter 7 presents a more detailed analysis of the execution times for running our PAF on different design examples.

Table 4.4: Pipelined FIR filter structures used for adaptive equalizer design

| Filter Structure | Pipelined Type and Computational Module | | | |
|---|---|---|---|---|
| | Type 0 | Type 1 | Type 2 | Type 3 |
| DF | Fig. 4.9 (a)-(i) | Fig. 4.9 (a)-(ii) | Fig. 4.9 (a)-(iii) | Fig. 4.9 (a)-(iv) |
| TF | Fig. 4.9 (b)-(v) | Fig. 4.9 (b)-(vi) | Fig. 4.9 (b)-(vii) | Fig. 4.9 (b)-(viii) |
| DFII | Fig. 4.9 (c)-(ix) | Fig. 4.9 (c)-(x) | Fig. 4.9 (c)-(xi) | Fig. 4.9 (c)-(xii) |
| TFII | Fig. 4.9 (d)-(xiii) | Fig. 4.9 (d)-(xiv) | Fig. 4.9 (d)-(xv) | Fig. 4.9 (d)-(xvi) |
| TreeDF | Fig. 4.9 (e)-(xvii) | - | Fig. 4.9 (e)-(xvii) | - |

## 4.6 FIR Filter Applications and Design Examples

We demonstrate, through several filtering applications, how the designer can use our framework to derive filter structures that perform well for specific performance constraints. The design optimizations presented in the previous sections provide a more comprehensive set of hardware designs with variations in performance. We considered three design examples to illustrate the effectiveness of our framework and the usefulness of the cost functions. The first two design examples focused on FIR filters used for data equalization. For these two cases, we observed the design space for high-throughput FIR filters and then considered area-efficient/high-throughput filter designs, respectively. For the third design example, we considered a matched FIR filter design used in signal communication applications where we analyzed the design space for low-power/high-throughput filter structures. It must be noted that the results reported here do not consider place and route results which would result in different area, timing and power measurements. However we make the supposition that the resulting values would scale in the same relative manner for the different implementations given that these designs are logic dominated and not interconnect dominant. Moreover, we compare the designs generated by our framework to designs presented in the literature by manually re-implementing the hardware structures and analyzing their performances.

### 4.6.1 High-Throughput Filters

The first example we considered was a high-throughput FIR filter for magnetic recording read channel applications [68]. Current read channels use partial response maximum likelihood (PRML) equalizers which require an FIR filter for fine signal equalization. Efficient time-recovery in a read channel is necessary for fast phase and frequency acquisition. The PRML channel requires an FIR filter to fully equalize the data samples which are used to extract timing information. The FIR filter output samples are further processed using a maximum likelihood sequence detector in order to increase timing robustness. Therefore, it is crucial to utilize filters that exhibit minimal delay in order to provide sufficient bandwidth. An 8-tap, non-pipelined transpose form FIR filter structure was presented in [69] which was sufficient for achieving tolerable mean square errors in the output. We manually designed the filter structure of [69] at the Verilog RTL using Booth-recoded

Wallace-tree multipliers and carry-look-ahead adders in order to measure the performance. The design specifications are summarized in Table 4.5.

Table 4.5: 8-tap Non-pipelined transpose form FIR filter specifications

| | |
|---|---|
| technology | $0.18\mu m$ CMOS |
| supply voltage | 1.8 V |
| coeff. and in-bits | 8 bits |
| throughput | 317 $Msamp/sec$ |
| area | 0.18 $mm^2$ |
| power | 86 $mW$ |
| power density | 478 $mW/mm^2$ |

We used our framework to generate the various hardware implementations for an 8-tap, 8-bit FIR filter. We selected to vary supply voltage to explore the effects of combined pipelining and $V_{DD}$ scaling since throughput was emphasized for this design example. At this point, a designer simply would select the order of the filter and the size of the filter input and coefficients. The framework proceeded to generate the different filter structures and characterize the performance of each design for different supply voltages. The resultant design space consisted of 112 filter design permutations with varying throughput, power and area metrics. Figure 4.11 illustrates the reduced design space for structures that met both throughput and power performance metrics. Only a small subset of the structures lie in the accepted power-efficiency region. Figure 4.12 illustrates the Pareto-optimal set of the reduced design space. A set of designs that perform well in the entire design space can be interpreted as a Pareto-optimum set. This term was first proposed by Francis Y. Edgeworth [70] and later generalized by Vilfredo Pareto [71]. A design $D$ is Pareto-optimal if there exists no other design in the design space such that $D'$ performs better than $D$. This concept was applied to design spaces for analyzing the performance of power-delay curves [34].

Manually sifting through all designs to select efficient filter implementations would

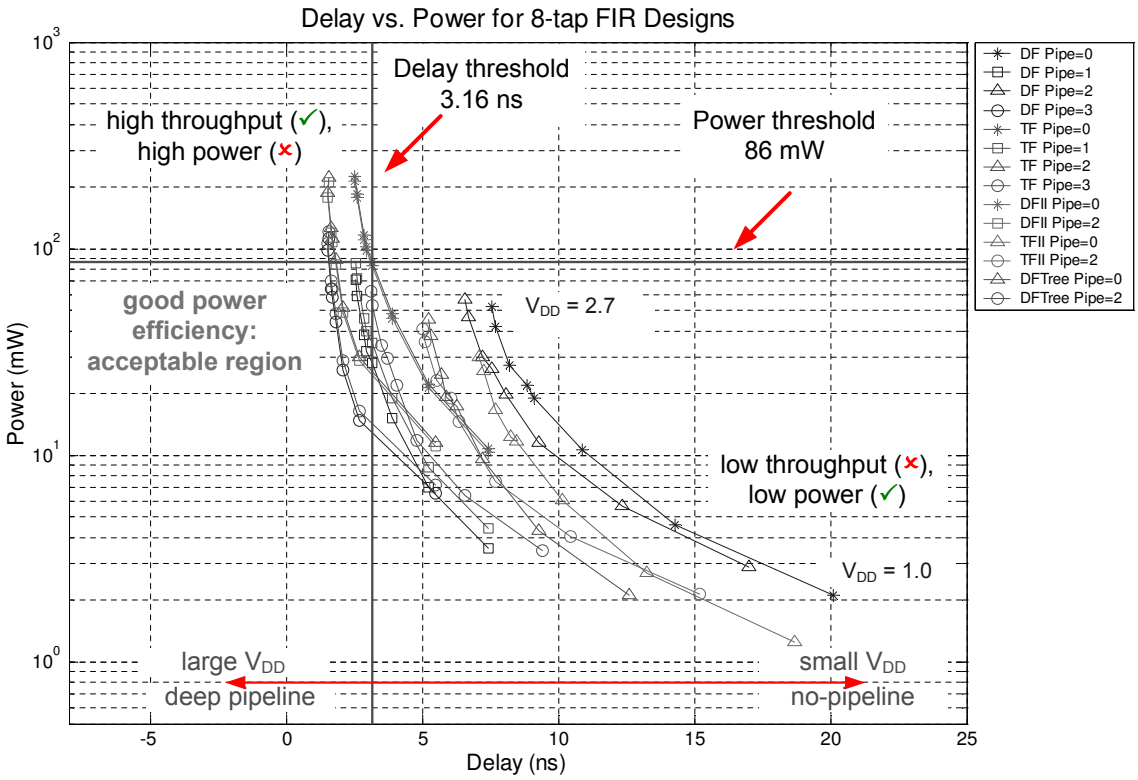



Figure 4.11: Design permutations for high-throughput FIR filters

be a laborious task. Therefore, we selected to reduce the design space to filter structures that met both the throughput and power constraints. Of the 112 filter designs and permutations, only 28 operated at 317 $Msamp/sec$ or greater and exhibited less than 86 $mW$ of power dissipation. We selected the power density cost function from our framework options to analyze the power dissipation per square unit of area [52]. Table 4.6 summarizes the results for the top five design options in the reduced design space. The first four design options are highly pipelined filter structures that lie on the Pareto-optimal curve illustrated in Figure 4.12. A designer can choose any one of the filter designs returned by our framework with comparable performance to the non-pipelined transpose form filter. Comparing the performance of the first filter design option in Table 4.6 to the performance of the non-pipelined TF filter in Table 4.5, we observed that we can achieve a 17% increase in throughput while reducing power density by approximately 8 fold. However, this came at a cost of 50% increase in area and an increase in hardware latency. Figure 4.13 illustrates the filter structure we arrived ar using our PAF and compares it to the structure presented

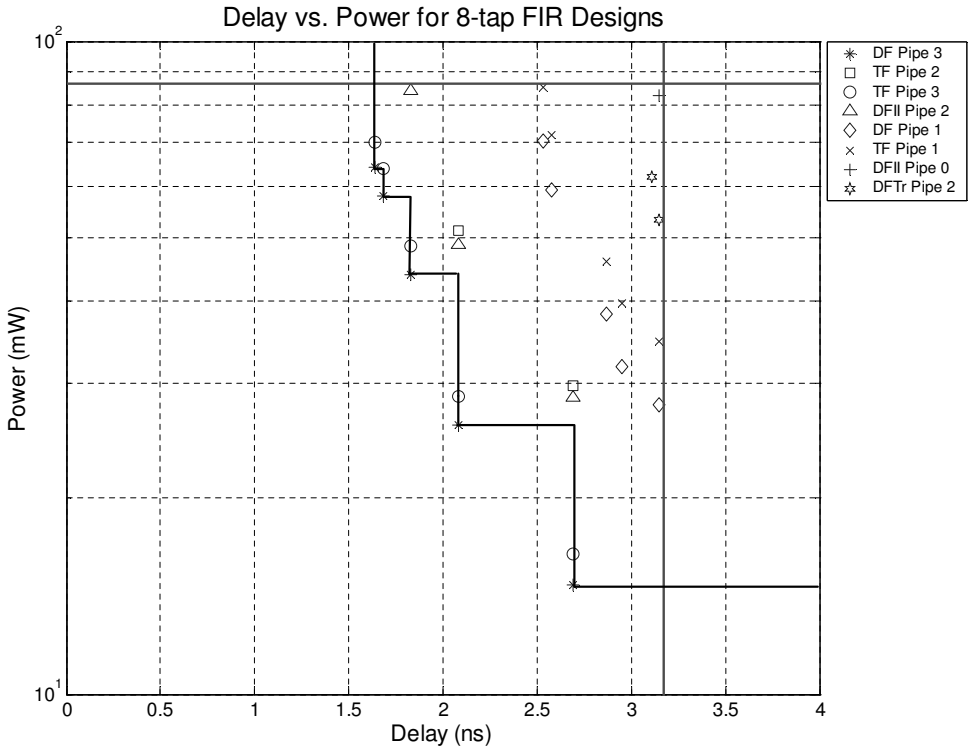
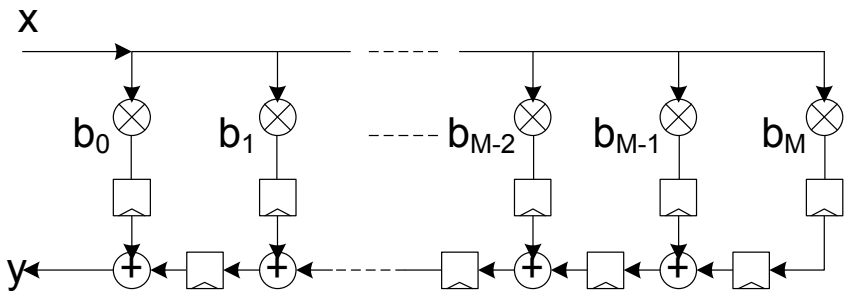



Figure 4.12: Pareto-optimal power-delay curve for high-throughput FIR filter

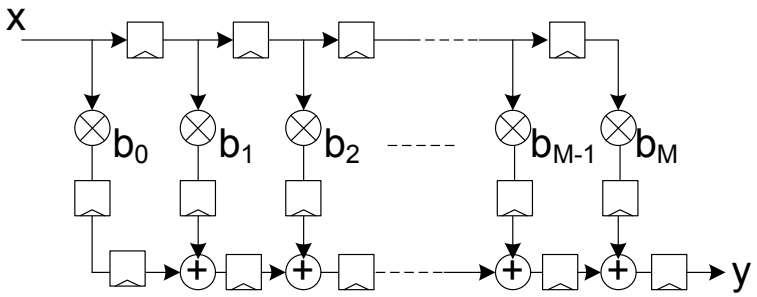
in [69].

Table 4.6: Performance results for 8-tap high-throughput FIR filters

| Structure | Pipe | $V_{DD}$ $(V)$ | Area $(\mu m^2)$ | Delay $(ns)$ | Throughput $(Msamp/sec)$ | Latency (cycles) | Power $(mW)$ | Power Density $(mW/mm^2)$ |
|---|---|---|---|---|---|---|---|---|
| DF | 3 | 1.2 | 253896 | 2.69 | 372 | 10 | 14.71 | 58 |
| TF | 3 | 1.2 | 263680 | 2.69 | 372 | 10 | 16.40 | 62 |
| DF | 3 | 1.5 | 241952 | 2.08 | 481 | 10 | 25.86 | 107 |
| TF | 3 | 1.5 | 251736 | 2.08 | 481 | 10 | 28.55 | 113 |
| DFII | 2 | 1.2 | 226086 | 2.69 | 372 | 3 | 28.51 | 126 |




(a) Non-pipelined TF used in [69]




(b) Filter generated using PAF (Pipelined DF)

Figure 4.13: Comparison of high-throughput filter structures

### 4.6.2 Area-Efficient and High-Throughput Filters

The next design example we considered was a reduced-area, high-throughput FIR filter used in data equalization for communication systems. The QAM modulation standard employs an equalizer for minimizing channel distortions such as ISI. Yu *et al.* presented a popular area-efficient scheme for implementing a 64-QAM system by time-multiplexing (or interleaving) a 16-tap FIR filter by a factor of 4 [72]. The authors implemented the interleaved FIR filter using a non-pipelined TF structure. We manually designed and synthesized that filter structure in order to measure the performance which is summarized in Table 4.7.

In a similar manner to the previous design example, we provided filter order and bit sizes to our framework. Additionally, we set the interleaving factor to 4 since we provided that design option in our framework. Our framework returned 16 hardware designs

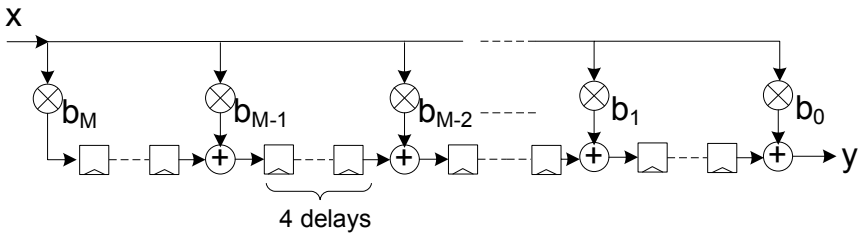
Table 4.7: 16-tap area and throughput-efficient, time-multiplexed FIR filter specifications

| | |
|---|---|
| technology | 0.18 $\mu m$ CMOS |
| supply voltage | 1.8 V |
| coeff. and in-bits | 12 bits |
| power | 314 $mW$ |
| throughput | 190 $Msamp/sec$ |
| area | 0.85 $mm^2$ |
| latency | 2 cycles |
| power efficiency | 1.65 $mW/(Msamp/sec)$ |
| area efficiency | $4.74 \times 10^{-3}$ $mm^2/(Msamp/sec)$ |

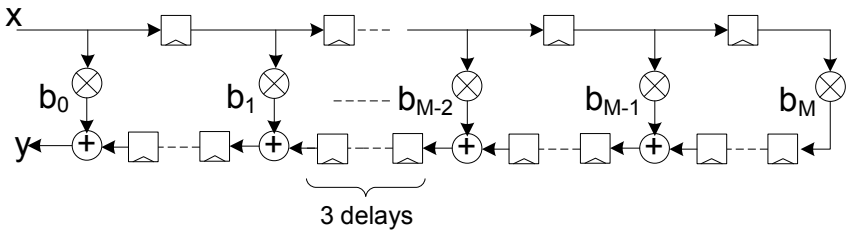
characterized for performance using 8 supply voltages for a total of 112 permutations. We selected to reduce the design space to filter structures that met area, throughput and power constraints. Only 4 of the 112 design options met all performance constraints, one of the design options being the non-pipelined TF structure operating at 1.8 V. We used both power efficiency and area efficiency to assess the quality of the remaining designs. From the results in Table 4.8, we notice a 3.5 fold improvement in power efficiency and a 5% increase in area efficiency when using the pipelined DFII structure operating at 1.8 V compared to the transpose form structure. Figure 4.14 compares the framework filter implementation to the filter structure presented in [72]. We observe the the critical path for both designs are equivalant and therefore, our framework could not significantly improve the throughput for the filter parameters used in this specific design example, which may be one drawback to using our framework in generating area and throughput-efficient designs.

Table 4.8: Performance results for 16-tap area and throughput-efficient, multiplexed FIR filters

| Structure | Pipe | $V_{DD}$ ($V$) | Area ($\mu m^2$) | Delay ($ns$) | Throughput ($\frac{Msamp}{sec}$) | Latency (cycles) | Power ($mW$) | Power Eff. ($\frac{mW}{Msamp/sec}$) | Area Eff. ($\frac{mm^2}{Msamp/sec}$) |
|---|---|---|---|---|---|---|---|---|---|
| DFII | 1 | 1.8 | 802896 | 5.26 | 190 | 2 | 90 | 0.471 | $4.2 \times 10^{-3}$ |
| DFII | 1 | 1.8 | 760529 | 4.95 | 202 | 2 | 116 | 0.574 | $3.8 \times 10^{-3}$ |
| DFII | 0 | 1.8 | 836680 | 5.26 | 190 | 2 | 310 | 1.63 | $4.4 \times 10^{-3}$ |
| TF | 0 | 1.8 | 844872 | 5.26 | 190 | 2 | 314 | 1.65 | $4.5 \times 10^{-3}$ |




(a) Non-pipelined TF used in [72]




(b) Filter generated using PAF (Pipelined DFII)

Figure 4.14: Comparison of area-efficient filter structures

### 4.6.3 Low-Power and High-Throughput Filters

The third design example we considered was a binary pseudo-random code matched filter for IS-2000 CDMA systems constructed using a 32-tap, low-power, high-throughput

FIR filter. CDMA communication systems are based on spreading the passband frequencies generated by multiple users, allowing for multi-channel utilization. Each signal is uniquely coded using a pseudo-random code generator to avoid interference between users occupying the same channel. Receivers utilize matched filters that despread or decode each users frequency bandwidth in order to remove undesired interference from other transmissions. The matched filter in the receiver maximizes the signal-to-interference ratio using different signal processing optimization techniques. The hardware overhead introduced in maximizing signal recovery leaves little room for over-designing the basic, yet vital, FIR filter. Chen *et al.* proposed a low power reconfigurable 32-tap FIR filter used in the CDMA's receiver [73]. The structure was pipelined every 8 taps to achieve a reasonable performance in throughput while maintaining minimal power dissipation. We normalized the performance results using a method suggested in [73] to match the technology size, word sizes and $V_{DD}$ values we used in analyzing the performance of the filter designs generated by our framework. The performance constraints for the filter structures is summarized in Table 4.9.

Table 4.9: 32-tap low-power, high-throughput FIR filter specifications

| | |
|---|---|
| technology | 0.18 $\mu m$ CMOS |
| supply voltage | 1.8 $V$ |
| coeff. and in-bits | 16 bits |
| power | 129 $mW$ |
| throughput | 134 $Msamp/sec$ |
| area | 2.33 $mm^2$ |
| latency | 12 cycles |
| power efficiency | 0.96 $mW/(Msamp/sec)$ |

We utilized our framework to search the design space for FIR filters that met both the power and throughput design specifications. Figure 4.15 illustrates the 112 design permutations that lie within the accepted power/throughput region. We observed a similar

trend in the power-delay curve to that of Figure 3.10. The designs that lie in the lower left region yielded good power efficiency results. Of the 112 FIR filter design options, only 14 filters dissipated 129 $mW$ of power or less, while operating at 134 $Msamp/sec$ or more. However, most of these filter structures required some form of pipelining, and therefore, exhibited large latencies. Therefore, we further reduced the design space by selecting pipelined filter structures that met both power and throughput constraints, while requiring low latencies. We selected power efficiency to re-order the filter structures and the results for the top five design options in terms of power efficiency are summarized in Table 4.10.

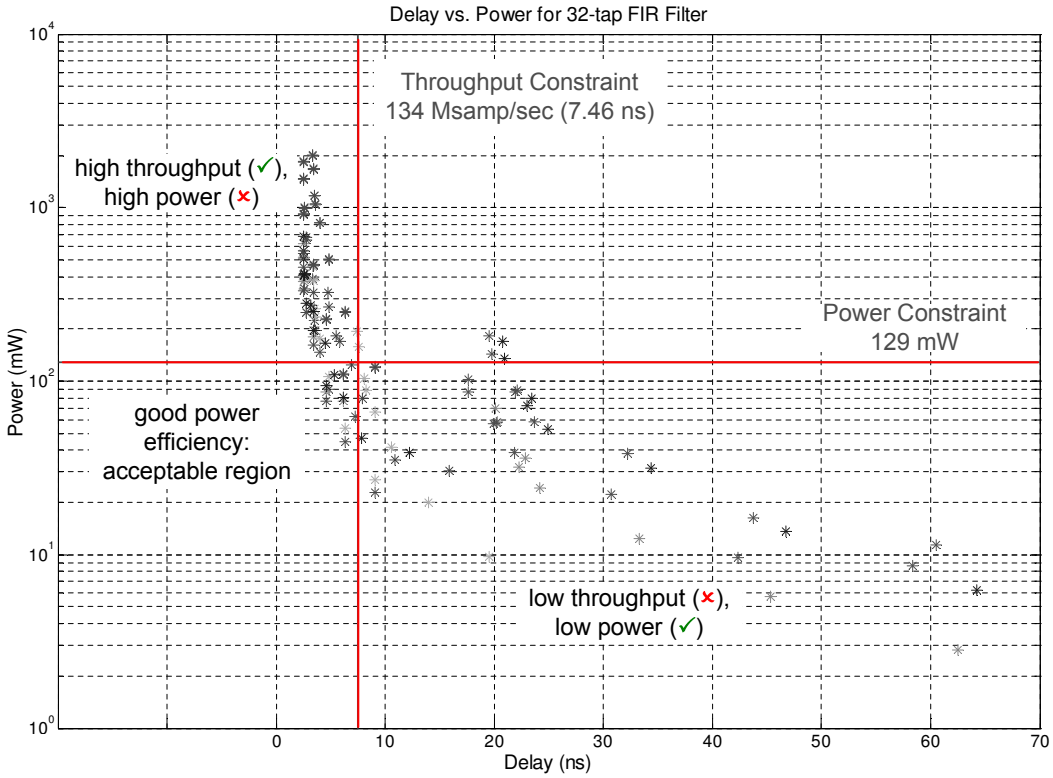



Figure 4.15: Design permutations that lie within the acceptable power/throughput region

All 14 filter structures in the reduced design space met the design specifications for area, throughput and power. The designer can use power efficiency to find a design that is optimized for both power and throughput, since these were the two crucial design constraints for this example. In this case, the DF tree-adder FIR filter with pipelined multipliers operating at 1.2 V was an appropriate design option which exhibited only 4 clock cycles of latency versus 12, and achieved a 2.7 fold increase in power efficiency compared to

Table 4.10: Performance results for 32-tap low-power, high-throughput FIR filters

| Structure | Pipe | $V_{DD}$ ($V$) | Area ($\mu m^2$) | Delay ($ns$) | Throughput ($\frac{Msamp}{sec}$) | Latency (cycles) | Power ($mW$) | Power Efficiency ($\frac{mW}{Msamp/sec}$) |
|---|---|---|---|---|---|---|---|---|
| TreeDF | 3 | 1.2 | 2338520 | 4.58 | 218 | 4 | 76.82 | 0.35 |
| TreeDF | 3 | 1.0 | 2191672 | 7.23 | 138 | 4 | 62.83 | 0.46 |
| TreeDF | 2 | 1.5 | 2158008 | 5.37 | 186 | 3 | 109.10 | 0.59 |
| TF | 2 | 1.0 | 2193152 | 6.19 | 162 | 3 | 110.20 | 0.68 |
| TreeDF | 1 | 1.8 | 2185272 | 6.90 | 145 | 2 | 123.60 | 0.85 |

the filter structure we manually designed.

## 4.7 Chapter Summary

This chapter illustrated how our PAF can be used to guide the designer in selecting specific hardware structures for FIR filters. The FIR filter was analyzed at multiple levels of abstraction and we illustrated how our framework addresses the pertinent design optimizations at different levels of design detail. The chapter discussed the vast design options included as part of our PAF for generating specific filter structures. The design options are essential for allowing the user to shape the hardware structure for the FIR filter algorithm and improve performance. The cost functions are important for eliminating structures from the design space that do not meet performance specifications. This improves the process of searching the reduced design space for structures that perform well.

We included a set of well-structured filters that comply to a modular-based design methodology allowing experienced designers to further improve the filter structures if they wish to do so. We illustrated, through the use of three FIR filter examples, that our framework was capable of guiding the designer in selecting FIR filter structures with comparable performance metrics to manually designed structures. We accomplished this by using our

framework to generate and synthesize the hardware designs in a short period of time. The results of this case study are essential in illustrating that our automated design and analysis methodology generated filter structures that performed well with minimal design effort from the user. The next chapter illustrates how similar types of design optimizations can be applied to a second computationally intensive DSP algorithm, namely the adaptive channel equalizer.

# Chapter 5

# Case Study II: Adaptive Channel Equalizers

The previous chapter presented an essential case study for illustrating hardware design improvements when considering a computationally intensive DSP algorithm such as an FIR filter. The methodology we used for generating the filter hardware blocks can be extended to other DSP applications that require filtering. This chapter proceeds beyond the implementation of FIR filters and considers its application in adaptive channel equalizers. The chapter addresses whether the design methodology used to construct FIR filters can be used to construct alternative hardware structures for the adaptive equalizer. Additionally, the chapter investigates the process for applying low-level design optimizations to improve the hardware performance of the equalizer design beyond those presented in the literature.

The chapter begins by outlining the algorithm for the equalizer using the LMS technique for coefficient adaptation [74]. The fixed-point model for the equalizer design is analyzed at the mathematical level to determine the appropriate word sizes for hardware implementation. Next, the chapter presents the methodology employed by the PAF in constructing the various equalizer designs. The refinement process from algorithm to hardware demonstrates the merits of our work in efficiently refining the equalizer algorithm to a hardware implementation. The chapter concludes by analyzing several case studies for

equalizer structures presented in the literature with specific design constraints. We illustrate, through the use of the PAF cost functions, how we reduce the design space to a set of designs that meet the performance specifications.

## 5.1 Adaptive Equalizer using the LMS Algorithm

Adaptive channel equalizers are used in mobile communication systems for the purpose of increasing the data rate transmission subject to a specified probability of error. As the name suggests, channel equalizers adapt to the statistics of the communication channel that corrupt the original signal due to intersymbol interference (ISI). This corruption is in the form of noise originating from the transmit filter or from channel distortion. The main function of the channel equalizer is to remove as much of the distortion as possible. While this is possible via filters with constant coefficients, better results are attainable with adaptive filters, especially when prior statistics of the data signal and noise are not known.

The details for the adaptive equalizer implementations are governed by the type of algorithm used to update the filter coefficients. The LMS is the simplest algorithm for adapting the coefficients to the inverse of the channel. It requires the least number of multiplications and additions per iteration, and therefore is an attractive design option for power and throughput constrained implementations. The following three equations describe the algorithm for adapting the coefficients to the inverse channel using the LMS technique [74]:

$$f_k = \sum_{i=0}^{M-1} w_i^{k-i} u_{k-i} \tag{5.1}$$

$$e_k = d_k - f_k \tag{5.2}$$

$$w_i^{k+1} = w_i^k + 2\mu e_i^k u_i^k, \quad i = 0, 1, ..., M \tag{5.3}$$

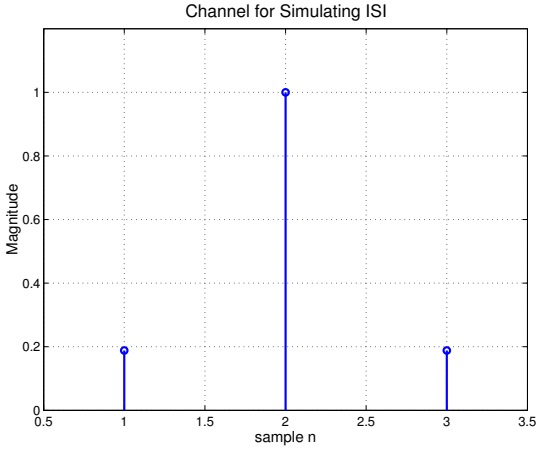
where $f_k$ is the filter output, $u_k$ is the $M \times 1$ corrupted input vector, $w_k$ is an $M \times 1$ vector of FIR filter coefficients, $M$ is the number of filter coefficients, $e_k$ is the estimator error, $d_k$ is the uncorrupted training sequence, and $\mu$ is the adaptation step size which controls the speed and stability of channel equalization.

The effects of the adaptive channel equalizers at the different levels of abstraction are best analyzed using an experimental setup that emulates a transmitted data corrupted by ISI. MATLAB is an appropriate tool for modeling the equalizer at the algorithmic level

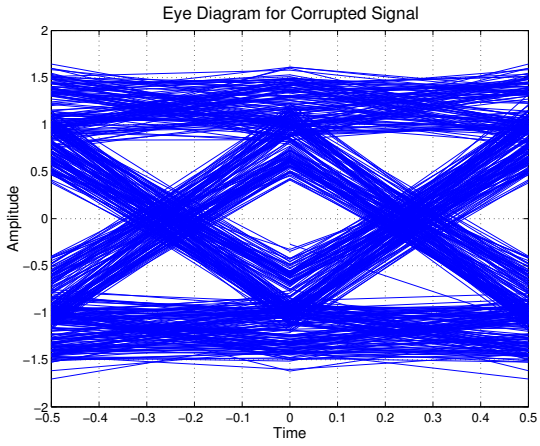
since it requires limited detail to perform the computations. We developed an experimental model similar to the model used by Haykin in [75] when addressing the algorithmic performances using computer-based simulations. Therefore, we modeled the channel using a raised cosine impulse response:

$$h(n) = \frac{1}{2}\left[1 + \cos\left(\frac{2\pi\,(n-2)}{\beta}\right)\right], \quad n = 1, 2, 3 \tag{5.4}$$

where $\beta$ is the roll-off factor. The training sequence $d(n)$ consists of a *Bernoulli Sequence* ($\pm 1$) with zero mean and unit variance. The impulse response for the channel is shown in Figure 5.1(a). The result is a corrupted sequence $u(n)$ whose level of corruption can be measured using an eye diagram, illustrated in Figure 5.1(b). Eye diagrams are used to plot multiple traces of a modulated, pulse shaped signal to analyze system characteristics. The opening of the eye is an indication of how much ISI is present in the system. The goal of the equalizer is to remove as much of the ISI in order to generate a clear stream of data which exhibits a large eye opening. The channel impulse response simulates ISI by smearing neighboring pulses onto the center pulse and was used as a benchmark for evaluating the algorithmic performances of our designs.




(a) Channel impulse response




(b) Corrupted signal

Figure 5.1: Channel impulse response ($\beta = 2.80$) and eye diagram for corrupted signal

We used the channel model in Figure 5.1(a) to determine an appropriate filter order for adapting the filter coefficients. Figure 5.2 shows a plot for the filter order vs. the

signal-to-interference ratio (SIR) for a fixed adaptation step size $\mu = 0.01$. We collected this data by modeling the communication system in MATLAB. The plot of filter taps versus SIR illustrates that a filter length of 12-taps is adequate for equalizing the channel and removing the effects of ISI. Therefore, this chapter analyzes the performance for the different adaptive equalizer designs for a 12-tap system. The methodology employed using these specifications can be extended to adaptive filters of different orders.

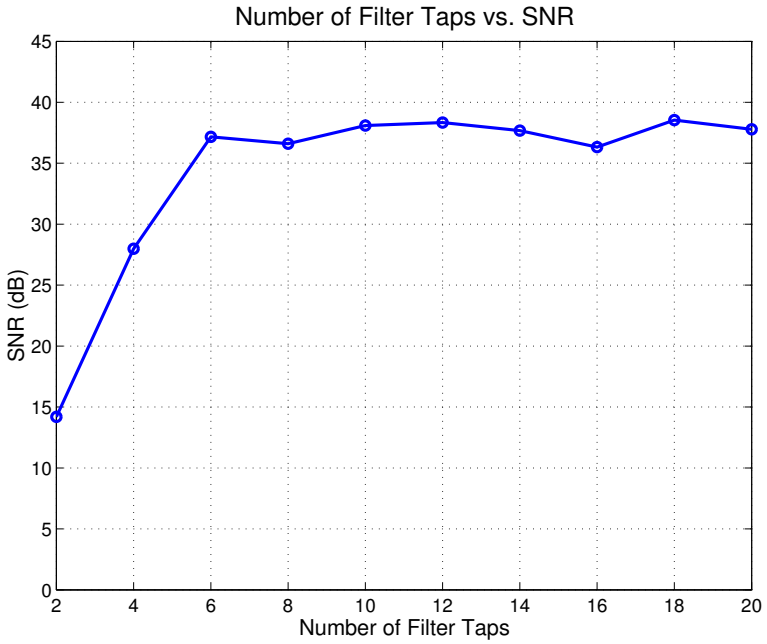



Figure 5.2: Adaptive equalizer Filter taps vs SNR: $\mu = 0.01$ and $\beta = 2.8$

The computational order in the FIR filter structure affects the adaptation speed and algorithm stability, which are two important concerns in the algorithmic analysis for the adaptive equalizer. D. L. Jones illustrated that the structure for the FIR filter plays a pivotal role in controlling the adaptation speed for the LMS algorithm [11]. His work analyzed the MSE generated at the comparator block for both the DF and the TF FIR filter algorithms. Both FIR filter algorithms perform the filtering operation. However, when used in an adaptive algorithm where the coefficients of the filter change, only the DF implements the exact dot product operation in the LMS algorithm. The TF algorithm yielded slightly different results due to the delays storing past values of the updated coefficients. The performance for each FIR filter structure depended on whether the signals $u_k$ and $d_k$ in

Equation (5.1) are correlated. The signal $d_k$ was corrupted by the smearing effects of the linear channel to produce the signal $u_k$ and therefore, the two signals were correlated.

Additional optimizations, such as pipelining to increase throughput, affect the performance of the algorithm due to inherent delays. Jones derived a stochastic gradient algorithm for an adaptive TF FIR filter to ensure that the LMS algorithm performs adequately and converges to a solution [11]. The error function after $k_0$ is

$$e_{k-k_0} = d_{k-k_0} - f_{k-k_0} = d_{k-k_0} - \sum_{i=0}^{M-1} w_i^{k-k_0} u_{k-k_0-i} \tag{5.5}$$

where $e_k$ is the different between the desired output $d_k$ and the TF filter output $f_k$ at time $k$. Using the square error $e_{k-k_0}^2$ as an instantaneous estimate of the mean squared error, a stochastic gradient can be determined as

$$\nabla_i^k = \frac{\partial e_{k-k_0}^2}{\partial w_i^k} = 2e_{k-k_0} u_{k-k_0-i} \frac{\partial w_i^{k-k_0-i}}{\partial w_i^k} \tag{5.6}$$

where we assume a relatively slow update and approximate

$$w_i^{k-k_0-i} \approx w_i^k \tag{5.7}$$

and Equation (5.6) reduces to

$$\nabla_i^k \approx -2e_{k-k_0} u_{k-k_0-i} \tag{5.8}$$

The filter coefficients are updated according to

$$w_i^{k+1} = w_i^k + 2\mu e_{k-k_0} u_{k-k_0-i} \tag{5.9}$$

which is a similar form to the update algorithm of Equation (5.1) where the adaptation delay in updating the coefficients is $D = k_0 + i$, $i = 0, 1, ..., M-1$. The adaptation delay is bounded between $k_0 \leq D \leq k_0 + M - 1$ and it is generally determined experimentally, depending on the filter structure and the hardware latency in the pipelined designs.

We investigated how adaptation speed was effected using alternate filter structures other than the DF. We used our framework to generate the different equalizer structures using SystemC data-types that can be integrated into the MATLAB environment. We analyzed the effects of filter structure on adaptation speed using the five non-pipelined filter structures presented earlier in Figures 4.3 and 4.4. The results for adaptation speed using the different filter structures are shown in Figure 5.3. We generated these results in MATLAB

and SystemC by running an ensemble of 200 experiments and averaging the mean square errors. We used a roll-off factor $\beta = 2.8$ and an adaptation step size $\mu = 0.01$. The MSE for the five basic filter structures converged, but at different rates. The convergence rate using the TF FIR filter was slightly faster than the DF filter, which matched the conclusions derived by Jones in his work [11]. This information is important when considering the algorithmic performance of the system and whether the order of computations affected the integrity of the signals.

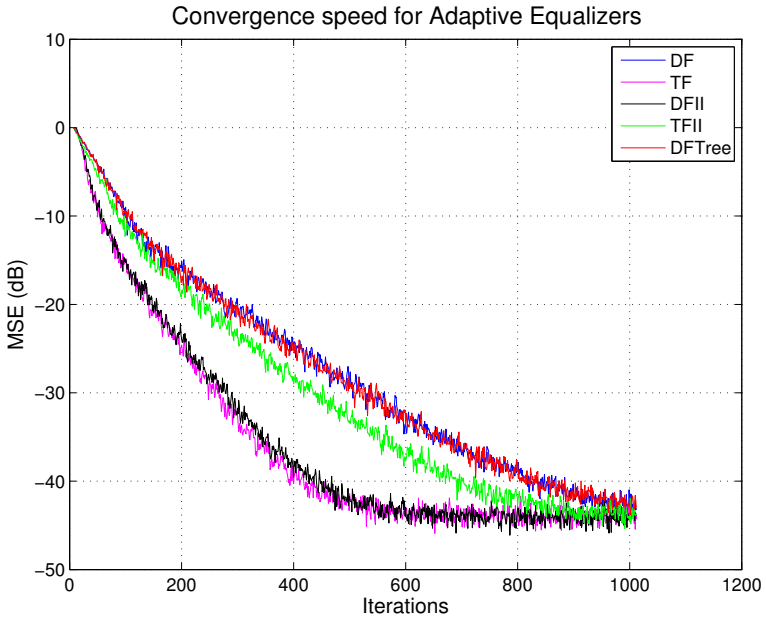



Figure 5.3: Convergence rate for non-pipelined FIR filter structures

## 5.2 Finite Word Length Effects

A fixed-point model for the adaptive equalizer is necessary when refining a DSP algorithm to synthesizable hardware models. Therefore, we modeled the equalizer algorithm using SystemC fixed-point computations to analyze the effects of finite word lengths. SystemC allowed us to model the DSP algorithm at a level of abstraction comparable to the MATLAB code with the added bonus of investigating different word lengths, quantization modes, and overflow types. The adaptive equalization algorithm is susceptible to noises

induced by quantization effects. Furthermore, due to the potentially unstable adaptation algorithm, overflow values could set the algorithm in a mode of divergence that would be difficult to recover from. Therefore, we found it useful to limit the overflow type to saturate at the maximum positive value and minimum negative value. The various quantization modes resulted in negligibly varying quantization errors. Therefore we chose to implement the fixed-point computations using the **floor** quantization mode since it was the default mode when considering the hardware implementation. Figure 5.4 illustrates the convergence rate for various word sizes using the **floor** quantization mode. The results illustrate that 12-bit word sizes for the input samples and filter coefficients were adequate for implementing the adaptive equalizer algorithm using fixed-point computations.

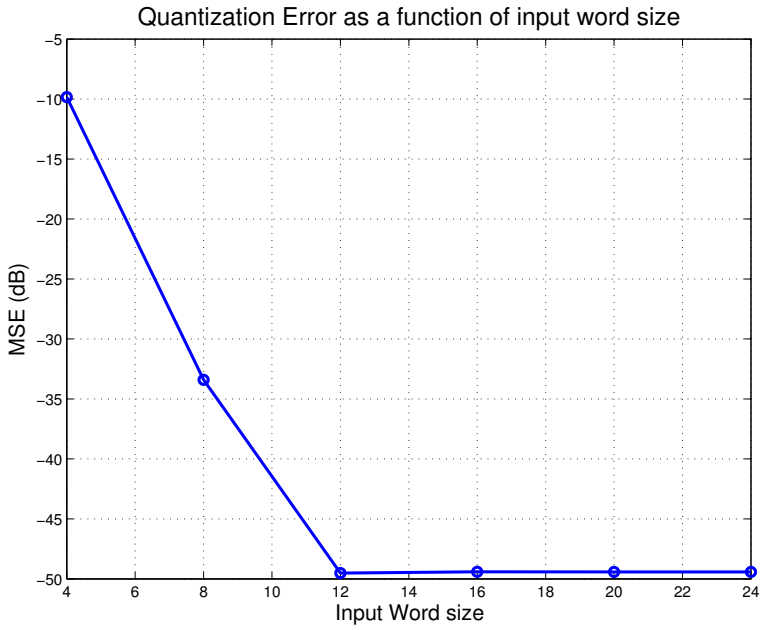



Figure 5.4: MSE due to quantization versus input word sizes

## 5.3 Architectural Implementation

We analyzed the hardware performance for the equalizer structures once the filter order and the word sizes were determined at the algorithmic and arithmetic levels. We proceeded to the hardware refinement process by using our PAF to construct the design

space for the adaptive equalizer using the FIR structures discussed previously in Chapter 4. We utilized our PAF to estimate area, throughput and power dissipation for each design. We applied our analysis methodology for efficiently sifting through the hardware designs in search for structures that performed well for specific design constraints such as high throughput or low-power designs. The user does not need extensive hardware knowledge of equalizer designs to use the PAF to accomplish this task.

Figure 5.5 illustrates the SFG graph for a basic hardware implementation of the adaptive channel equalizer. The DF filter design is one method for implementing the convolutional sum in Equation (5.1) and was used in [75] for the analysis of adaptive algorithms. However, we notice that the bottleneck in the throughput performance for the design lies in the long critical path through the FIR filter, error comparator and coefficient update blocks. This motivated us to explore different hardware structures to improve the hardware performance of the equalizer design.

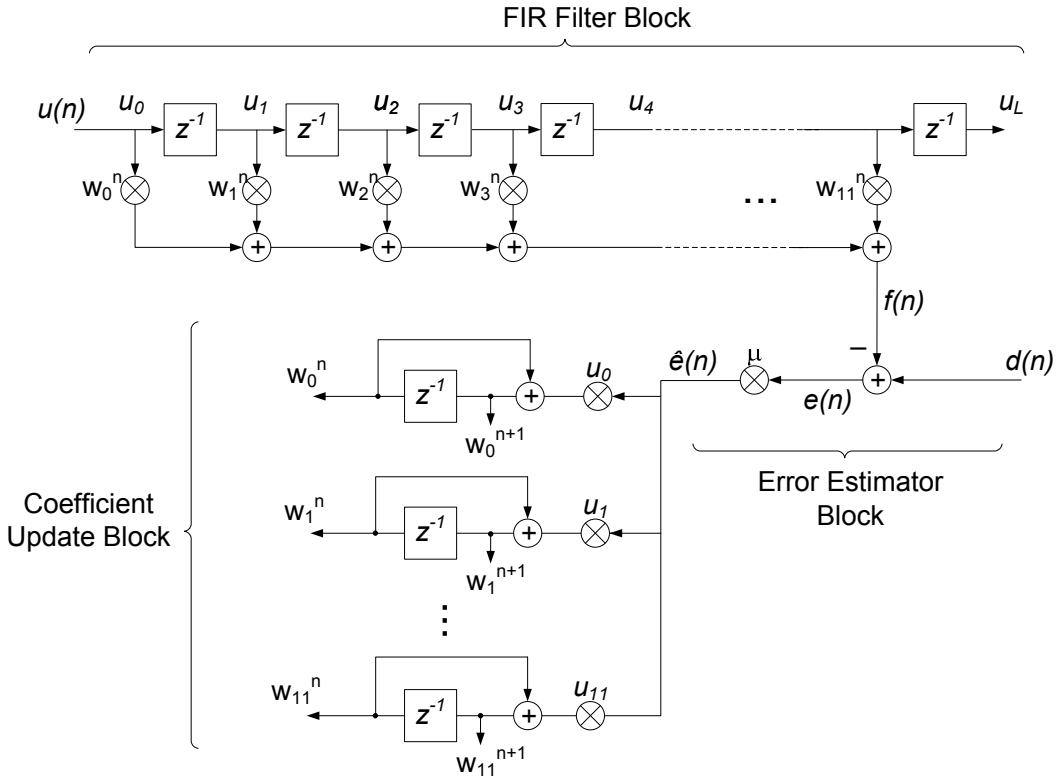



Figure 5.5: Hardware implementation for adaptive equalizer using DF FIR filter

We included the non-pipelined and pipelined FIR filter structures in an effort to

improve the hardware performance metrics. It must be noted that the pipelined DFII and TFII FIR structures were limited to using pipelined multipliers only since we did not include interleaving in the implementation. Furthermore, the computational components for the error estimator and coefficient update blocks are similar to the ones used for the FIR filter. Therefore, we utilized our framework to investigate the effects of low-level optimizations applied to the error estimator and the coefficient update blocks to further improve the performance for the overall system. Figure 5.6 illustrates the methodology employed by our PAF to design and analyze the quality of various hardware implementations for the equalizer design.

The design methodology constructed the basic cells for the FIR filter which included a computational module and possible input/output modules. Scripts in our framework generated the filters according to specific structures. A similar process was repeated for constructing the coefficient update cell using mathematical elements such as adders, multiplier and delay units. The coefficient update cells were instantiated $M$ times to construct the coefficient update block. The error estimation block was the final computational module constructed before the PAF scripts assembled the complete equalizer design. Each equalizer design was synthesized by utilizing a Perl script that invoked the Synopsys Design Compiler to estimate area, throughput, and power dissipation. The performance metrics were relayed back to MATLAB for architectural comparisons with other equalizer designs. The process was repeated for variations of the filter, error estimator, and coefficient update blocks.

Our framework generated four pipelined variations for the DF and TF FIR filters each. Our framework also generated two pipelined variations for the DFII, TFII, and DF-Tree adder FIR filters each, for a total of 14 different pipelined and non-pipelined filters. Additionally, our framework generated two pipelined versions for the error estimator and coefficient update blocks, each illustrated in Figure 5.7. This resulted in a total of 56 design permutations for the entire equalizer structure. Figure 5.8 summarizes the various combinations for pipelined and non-pipelined designs for the three major computational blocks used in the equalizer design.

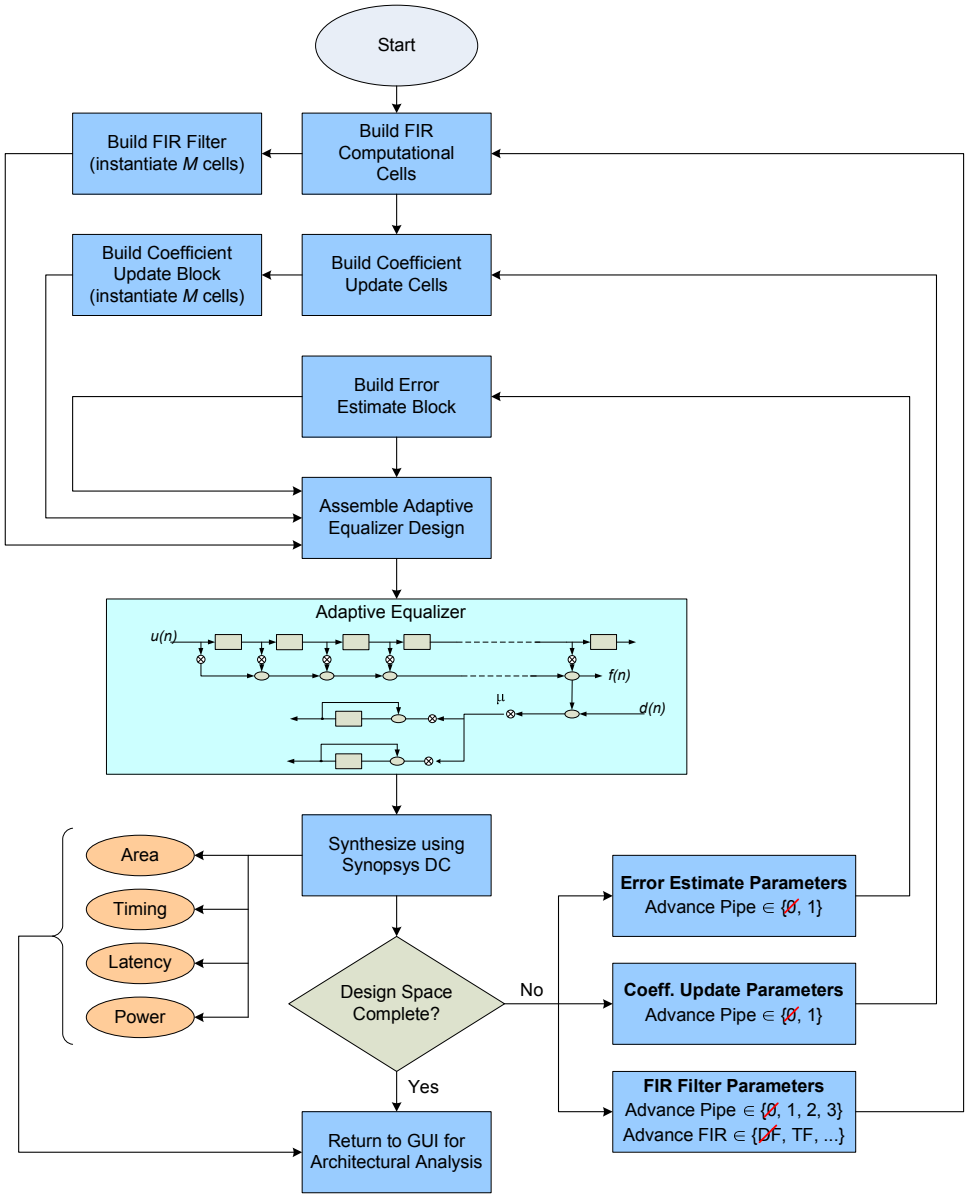
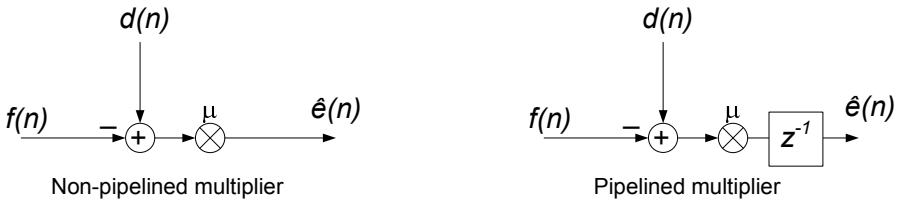


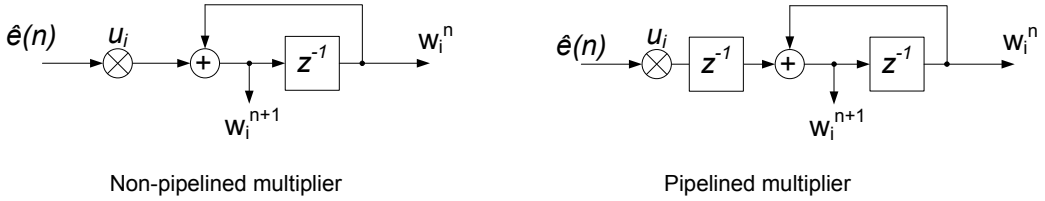

Figure 5.6: Executing the PAF flow for the architectural analysis of adaptive equalizer designs

## 5.4 Circuit-Level Performance Metrics

The performance of a synthesizable hardware design requires a specific standard-cell library and supply voltage to analyze circuit area, critical path delays and power dissipation. Therefore, we constrained our analysis to using $0.18\mu m$ for technology feature size and $1.8V$ for the supply voltage. This limited the design space to the original 56 architec-




(a) Error Estimator




(b) Coefficient Update Cell

Figure 5.7: Hardware structures for error estimator and coefficient update blocks

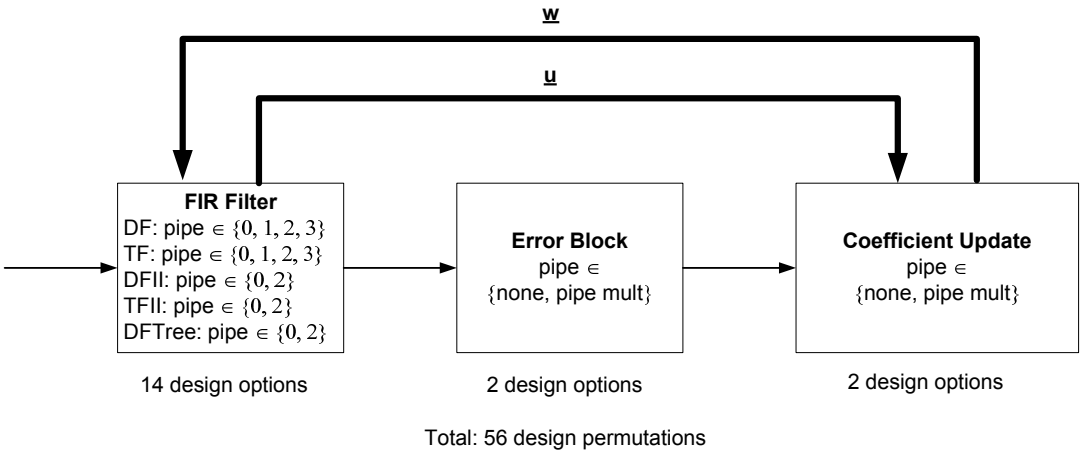



Figure 5.8: Design permutations for pipelined and non-pipelined adaptive equalizer computational blocks

tural permutations conveyed in Figure 5.8. Any variations in technology size and/or supply voltage would expand the design space to hundreds of feasible parameters. Our framework and cost functions are still applicable to large design spaces with varying circuit-level parameters. The hardware implementation for the adaptive equalizer designs consisted of mathematical elements similar to the elements used for FIR filters. Therefore, we utilized

our framework to generate the hardware structures and measure the performances using the pre-synthesized Synopsys DesignWare mathematical blocks. This considerably reduced the synthesis times for the entire design space. Our framework relayed the performance metrics for each equalizer implementation to MATLAB where we used our framework to apply multi-parameter cost functions to assess the quality of the hardware designs.

## 5.5 Equalizer Applications and Design Space Analysis

Area, throughput and power dissipation are examples of essential circuit-level performance metrics used to assess the quality of a hardware design. Our goal for the three case studies presented in this section was to search the design space for equalizer structures that performed well for constrained area, throughput, or power values. We used two-dimensional cost-functions to reduce the design space to a smaller set of designs that performed well. This was discussed in detail in Section 3.3. We compared the designs generated using our framework to adaptive LMS equalizer designs presented in the literature. We ensured the accuracy of our comparisons by manually designing and synthesizing the architectures in order to obtain the constrained performance metrics.

### 5.5.1 Area-Efficient Designs

Designers strive to minimize circuit area when mapping a design onto a platform with limited hardware resources. Yu *et al.* presented an area-efficient equalizer design for the QAM modulators used in digital video broadcasting systems to remove ISI [72]. We considered this structure previously in Chapter 4 when using our framework to generate area-efficient FIR filters only.

The authors utilized the non-pipelined TF FIR structure in their equalizer design due to its high throughput and low area. Alternative designs considered in [72] required some form of pipelining the DF filter structure which increased the area beyond that of the TF equalizer design and therefore the authors disregarded pipelined filter designs in their analysis. Furthermore, the error estimator and coefficient update blocks were not pipelined in Yu's hardware implementation since pipelining increases the area due to the addition of registers. We manually modeled the hardware structure presented in [72] at the Verilog

RTL, illustrated in Figure 5.9, and synthesized the design to assess the performance metrics. Table 5.1 summarizes the performance metrics obtained from the non-pipelined hardware design.

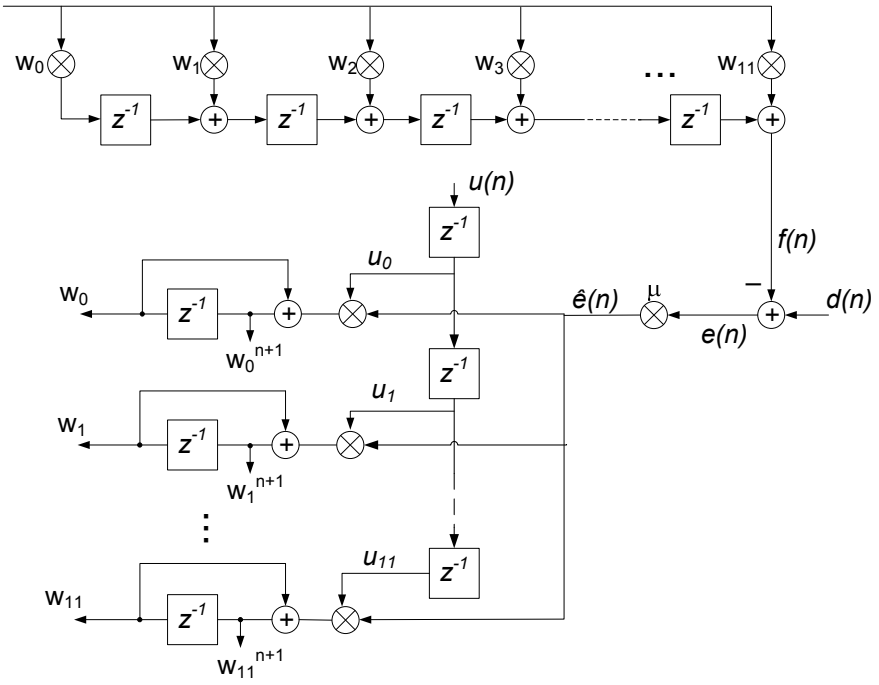



Figure 5.9: Area-efficient equalizer design presented in [72]

Our framework constructed the equalizer design space according to the filter order and word size derived earlier. We used the cost functions in our framework to reduce the design space to equalizer designs with area comparable to $0.96mm^2$ and throughput comparable to $109Msamp/sec$. Figure 5.10 displays the area versus power efficiency curve for the hardware implementations considered in the equalizer design space. A subset of the designs lied in the acceptable area range with varying power efficiency numbers. We applied the power efficiency cost function to assess the quality of the architectures in the reduced design space. Table 5.2 presents the first five implementations in the reduced design space in order of lowest power-efficiency. The results indicate that the equalizer design with the best power-efficiency could be implemented using a non-pipelined DF FIR filter with tree adder, a pipelined error estimator, and a non-pipelined coefficient update block. This structure, illustrated in Figure 5.11, resulted in a 6 fold increase in power efficiency compared to the

Table 5.1: Area-efficient adaptive equalizer specifications

| | |
|---|---|
| technology | 0.18 $\mu m$ CMOS |
| supply voltage | 1.8 V |
| coeff. and in-bits | 12 bits |
| area | 0.96 $mm^2$ |
| throughput | 109 $Msamp/sec$ |
| power | 157 $mW$ |
| latency | 2 cycles |
| power efficiency | 1.440 $mW/(Msamp/sec)$ |

manually constructed equalizer design.

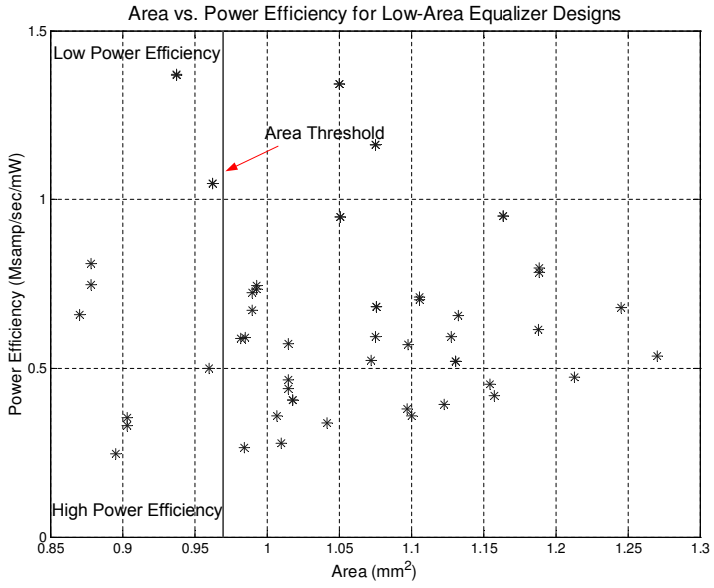



Figure 5.10: Area versus power-efficiency design space for area-efficient equalizer designs

Table 5.2: Performance results area-efficient equalizers

| Filter structure | Filter Pipe | Error Pipe | Coeff. Pipe | Area ($\mu m^2$) | Delay ($ns$) | Latency ($cycles$) | Thruput. ($\frac{Msamp}{sec}$) | Power ($mW$) | Power Efficiency ($\frac{mW}{Msamp/sec}$) |
|---|---|---|---|---|---|---|---|---|---|
| DFTr | 0 | 1 | 0 | 895168 | 9.49 | 1 | 105 | 26 | 0.249 |
| TFII | 2 | 0 | 0 | 903360 | 14.20 | 1 | 70 | 23 | 0.330 |
| DF | 2 | 0 | 0 | 903360 | 14.00 | 1 | 71 | 25 | 0.355 |
| DF | 0 | 0 | 1 | 959999 | 9.20 | 12 | 109 | 54 | 0.500 |
| DFTr | 0 | 0 | 0 | 870423 | 14.60 | 1 | 68 | 45 | 0.658 |

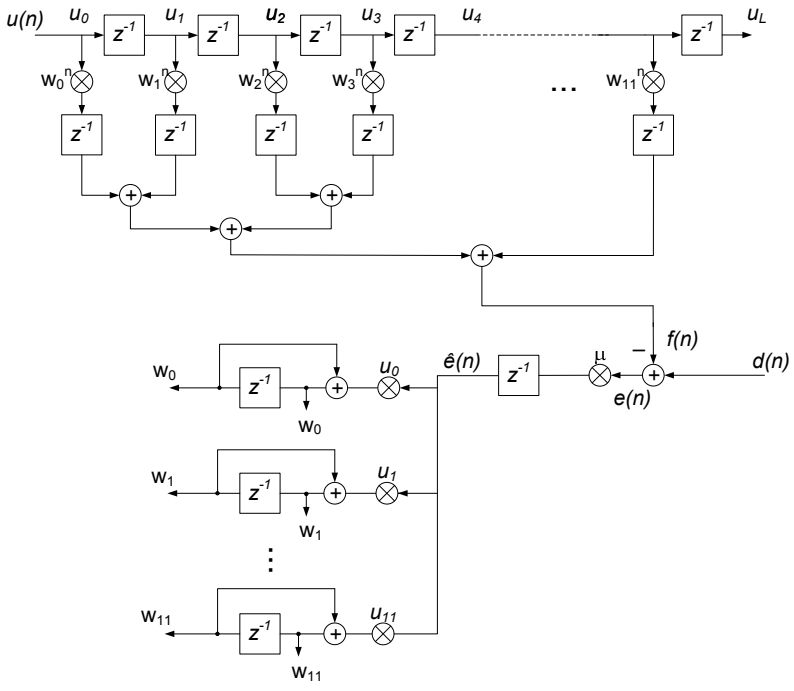



Figure 5.11: Area-efficient equalizer design generated by PAF

### 5.5.2 High-Throughput Designs

The next example we considered explored the equalizer design space for high-throughput hardware implementations. Hardware pipelining is one method for improving

the throughput of a computationally intensive DSP design. However, pipelining a design may alter the algorithmic functionality due to delays introduced in the design. Perry *et al.* presented a pipelined equalizer structure wherein the multipliers in a TF FIR filter and coefficient update block were replaced by single-stage pipeline multipliers [76]. The equalization algorithm was slightly modified to accommodate for the hardware latencies after pipelining the design. This was compensated for by introducing delays in the training sequence which had minimal effects on design performance. We designed the pipelined adaptive equalizer manually and synthesized the design using our frameworks synthesis scripts. The performance metrics are summarized in Table 5.3.

Table 5.3: High-throughput adaptive equalizer specifications

| | |
|---|---|
| technology | 0.18 $\mu m$ CMOS |
| supply voltage | 1.8 V |
| coeff. and in-bits | 12 bits |
| throughput | 255 $Msamp/sec$ |
| area | 1.19 $mm^2$ |
| power | 206 $mW$ |
| latency | 3 cycles |
| power density | 175 $mW/mm^2$ |

We used our framework to reduce the design space to equalizer implementations that met or exceeded the throughput constraint. We accomplished this by selecting a single-parameter cost function from our framework. This resulted in a reduced design space that consisted of only 4 hardware designs. The equalizer architecture presented by Perry was a highly pipelined design and therefore our framework found it difficult to generate hardware designs that exceeded a throughput of 255 $Msamp/sec$. The power density cost function allowed us to compare the quality of the remaining designs and assess the structures that performed well in terms of area and power dissipation. Table 5.4 summarizes the

performance results for the reduced design space that met the throughput specification. The designs are ordered in terms of lowest power density. The results show that one design option for the equalizer could be implemented by pipelining the DF FIR filter, error estimator and coefficient update blocks. This design option could reduce the power dissipation by approximately 43% when compared to the design presented in [76].

Table 5.4: Performance results for throughput-efficient equalizer

| Filter structure | Filter Pipe | Error Pipe | Coeff. Pipe | Area ($\mu m^2$) | Delay ($ns$) | Latency ($cycles$) | Thruput. ($\frac{Msamp}{sec}$) | Power ($mW$) | Power Density ($mW/mm^2$) |
|---|---|---|---|---|---|---|---|---|---|
| DF | 3 | 1 | 1 | 1212936 | 3.86 | 13 | 256 | 123 | 101 |
| TF | 3 | 1 | 1 | 1268328 | 3.86 | 14 | 255 | 138 | 109 |
| TF | 2 | 1 | 1 | 1186344 | 3.86 | 3 | 258 | 206 | 173 |
| DFII | 2 | 1 | 1 | 1186344 | 3.86 | 2 | 259 | 206 | 174 |

### 5.5.3 Low-Power Designs

The final equalizer design example we considered emphasized reducing the power dissipation while maintaining high throughput. Muhammad *et al.* presented different pipelined versions for the DF FIR filter used in the adaptive equalizer for partial response maximum likelihood (PRML) equalization of magnetic recording read channels to reduce the power dissipation [77]. The design presented by Muhammad, illustrated in Figure 5.12, employed a DF FIR filter with pipelined multipliers and pipelined tree-adder structure which we manually designed and synthesized for 12 taps and 12-bit inputs/coefficients. The performance results are summarized in Table 5.5.

Our goal was to use the PAF to construct an equalizer structure that exhibited low power while meeting the throughput constraint of 259 $Msamp/sec$. Therefore, we reduced the design space using our PAF to designs that met both power and throughput metrics. Table 5.6 displays the performance results for the equalizer structures generated by our framework that exhibited high throughput while maintaining low power dissipation.

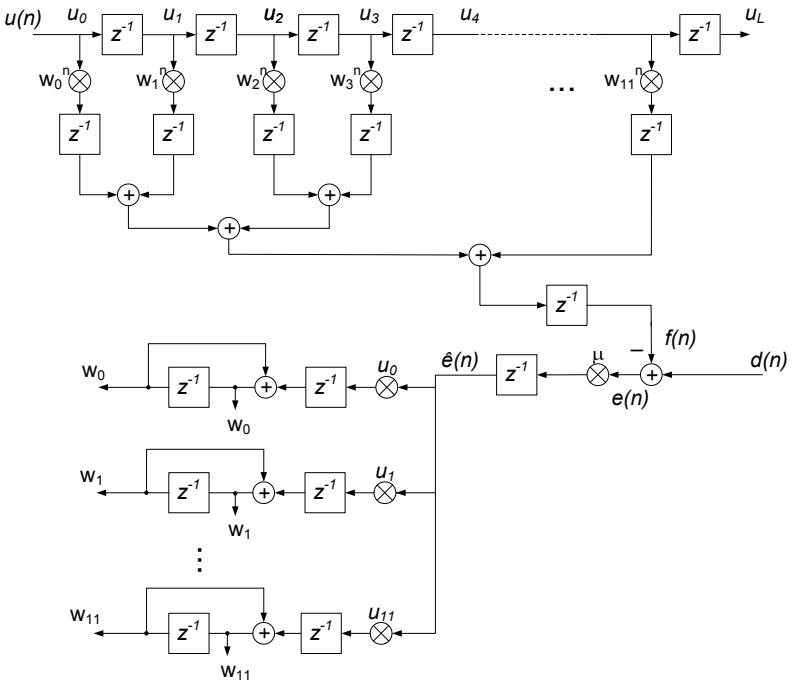



Figure 5.12: Low-power equalizer design presented in [77]

Table 5.5: Low-power adaptive equalizer specifications

| | |
|---|---|
| technology | 0.18 $\mu m$ CMOS |
| supply voltage | 1.8 V |
| coeff. and in-bits | 12 bits |
| power | 212 $mW$ |
| throughput | 259 $Msamp/sec$ |
| area | 1.22 $mm^2$ |
| latency | 5 cycles |
| power efficiency | 0.853 $mW/(Msamp/sec)$ |

The results show that the equalizer structure generated by our framework with best power efficiency was a highly pipelined design, illustrated in Figure 5.13. The power-efficient design utilized pipeline multipliers, similar to the structure presented in [77]. We achieved a 2 fold improvement in power efficiency using the framework architecture compared to the design we constructed manually.

Table 5.6: Performance results low-power equalizers

| Filter structure | Filter Pipe | Error Pipe | Coeff. Pipe | Area ($\mu m^2$) | Delay ($ns$) | Latency ($cycles$) | Thruput. ($\frac{Msamp}{sec}$) | Power ($mW$) | Power Efficiency ($\frac{mW}{Msamp/sec}$) |
|---|---|---|---|---|---|---|---|---|---|
| DF | 3 | 1 | 1 | 1212940 | 3.86 | 13 | 259 | 123 | 0.473 |
| TF | 3 | 1 | 1 | 1270549 | 3.86 | 14 | 259 | 139 | 0.535 |
| TF | 2 | 1 | 1 | 1188567 | 3.86 | 3 | 259 | 206 | 0.797 |
| DFII | 2 | 1 | 2 | 1188562 | 3.86 | 2 | 259 | 206 | 0.797 |

## 5.6 Chapter Summary

This chapter illustrated the usefulness of our PAF in generating and analyzing hardware structures for a DSP algorithm that requires filtering. We considered an adaptive equalizer system using the LMS algorithm as a case study for a computationally intensive algorithm beyond FIR filters. The optimizations applied to FIR filters were applicable to algorithms with equivalent computational complexity, such as the coefficient update algorithm in the adaptive equalizer design. We analyzed the equalizer algorithm at the algorithmic and arithmetic levels in order to determine high-level design parameters such as equalizer taps and finite word lengths. We then utilized our PAF to generate the architectural design space and applied the cost functions to search for hardware implementations that performed well for three specific performance constraints respectively: area, throughput and power dissipation. Our framework was useful in expanding the equalizer design space to tens of hardware implementations with minimal design effort from the user. This was essential in bridging the

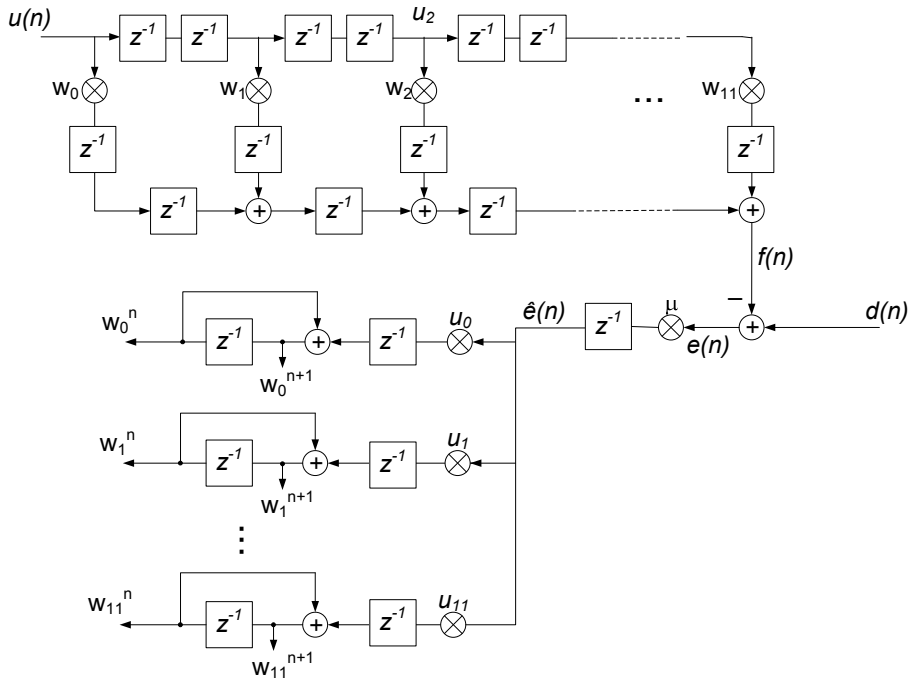



Figure 5.13: Low-power equalizer design generated by PAF

gap between the high-level equalizer algorithm and the efficient hardware implementation. In some cases, our framework could not outperform high-throughput designs presented in the literature. This would require further investigation into alternative pipeline methods to further reduce the critical path delays. The advantage of our framework, however, was the exploration of the design space with low effort while obtaining designs that performed comparably well to hardware structures designed manually. The next chapter illustrates how our design and analysis methodology addresses architectural improvements for specific filter designs used for wavelet transforms.

# Chapter 6

# Case Study III: Discrete Wavelet Transforms

FIR filters are essential building blocks for DSP algorithms such as signal and image compression techniques that rely on DWT blocks. Hardware designs constructed for specific DSP applications generally perform better than general purpose hardware blocks. This chapter presents a case study for using the PAF to generate application-specific FIR filters that perform well for quadrature mirror filter banks. We illustrate how a skilled hardware designer can use the PAF to select design optimizations suitable for the hardware implementation of the DWT system. The chapter begins by briefly formulating the DWT algorithm as a set of filter banks. The chapter then discusses the basic hardware implementation for a multi-level DWT system. We then proceed to highlight the types of suitable hardware optimizations that can be applied for the efficient implementation of the DWT block and how our framework generates the various application-specific filters. We conclude the chapter by analyzing the performance of the DWT hardware design space.

## 6.1 Discrete Wavelet Transform Algorithm

Wavelets are functions that are generated from a basis function $\psi(t)$, also known as a mother function. Other wavelets can be defined from the mother function as

$$\psi_{a,b}(t) = \frac{1}{\sqrt{|a|}}\psi\left(\frac{t-a}{b}\right) \tag{6.1}$$

where $a$ and $b$ are two arbitrary real numbers and represent the parameters for dialation and translation respectively [78]. The wavelet transform can be discretized by representing the dialation and translation parameters as discrete values

$$a = a_0^m, \qquad b = nb_0a_0^m \tag{6.2}$$

where $m$ and $n$ are integers. Therefore, the discrete wavelets can be represented as

$$\psi_{m,n}(t) = a_0^{-m/2}\psi\left(a_0^{-m}t - nb_0\right). \tag{6.3}$$

The dyadic decomposition is a popular method for sampling the wavelets and can be formed by selecting $a_0 = 2$ and $b_0 = 1$. In general, for the dyadic decomposition, the wavelet coefficients for a function $f(t)$ can be represented as

$$c_{m,n}(f) = 2^{-m/2}\int f(t)\psi\left(2^{-m}t - n\right)dt \tag{6.4}$$

where $f(t)$ can be reconstructed from the discrete wavelet coefficients as

$$f(t) = \sum_{m=-\infty}^{\infty}\sum_{m=-\infty}^{\infty} c_{m,n}(f)\psi_{m,n}(t). \tag{6.5}$$

Multiresolution analysis is used to decompose a single into two parts; one part is an approximation of a signal and the other part contains the detailed components of a signal due to the approximation. This can be mathematically represented as

$$f_m(t) = \sum_n a_{m+1,n}\phi_{m+1,n} + \sum_n c_{m+1,n}\psi_{m+1,n} \tag{6.6}$$

where $\phi_{m+1,n}$ and $\psi_{m+1,n}$ are the dilation and wavelet basis functions, respectively. $c_{m+1}, n$ is the detail information and $a_{m+1,n}$ is the approximation of the signal and both signals can be represented as

$$\begin{aligned} c_{m,n}(f) &= \sum_k g_{2n-k}a_{m-1,k}(f) \\ a_{m,n}(f) &= \sum_k h_{2n-k}a_{m-1,k}(f) \end{aligned} \tag{6.7}$$

where $g[n]$ and $h[n]$ are the highpass and lowpass filter respectively. Therefore, a signal $x(n)$ can be decomposed into approximation and detail components, or alternatively, into highpass and lowpass signals as

$$\begin{aligned} y_L(n) &= \sum_{i=0}^{\tau_L-1} h(i)x(2n-i) \\ y_H(n) &= \sum_{i=0}^{\tau_H-1} g(i)x(2n-i) \end{aligned} \tag{6.8}$$

where $\tau_L$ and $\tau_H$ are the lengths of the lowpass ($h$) and highpass ($g$) filters, respectively [78].

The formulation of the DWT as a set of FIR filters establishes the foundation for modeling a complex DSP algorithm as a hardware architecture. We used MATLAB's Wavelet Toolbox to define a set of 16-tap highpass and lowpass Daubechies [79] filter coefficients to decompose a signal into its respective frequency bands. The MATLAB floating-point DWT algorithm formed the basis for validating the outputs of the DWT system modeled at lower levels of design abstraction. We chose to use MATLAB to analyze the DWT system at the algorithmic level due to the efficient formulation of the code.

## 6.2 Fixed-Point Algorithmic Analysis

The next task after defining the DWT parameters was to model the algorithm using fixed-point data types and computations. The details of the wavelet algorithms in the MATLAB DWT Toolbox were hidden from us which made it challenging to analyze the effects of fixed-point operations. Therefore, we modeled the details of the DWT filter banks in MATLAB using a similar approach to Banerjee *et al.* [21]. We made the assumption that the input data we were processing was quantized using 8 bits, which was the case for the speech signal we used as a test signal. We quantized the inputs and coefficients using 8, 12, 16 bit words. Table 6.1 summarizes the quantization errors for the different word sizes. This data was collected with the assistance of a Masters student working on a one dimensional, three-level DWT hardware implementation for her thesis project. The peak errors and MSE improved for larger word sizes. However, for the case of the speech signal, we concluded upon listening to the reconstructed wavefile that 12-bit input and coefficient word sizes were adequate for implementing the DWT algorithm using fixed-point

computations. We proceeded with the hardware implementation using our PAF once we established the necessary high-level parameters and adequate words sizes.

Table 6.1: Quantization results for fixed point DWT implementation

| Input/Coeff. Bits | Output Bits | Peak Error | | | MSE | | |
|---|---|---|---|---|---|---|---|
| | | $1^{st}$ Level | $2^{nd}$ Level | $3^{rd}$ Level | $1^{st}$ Level | $2^{nd}$ Level | $3^{rd}$ Level |
| 8 | 16 | 0.162 | 0.363 | 0.376 | $6.48 \times 10^{-3}$ | $1.74 \times 10^{-2}$ | $2.49 \times 10^{-2}$ |
| 12 | 24 | 0.0095 | 0.0193 | 0.0392 | $1.87 \times 10^{-5}$ | $8.73 \times 10^{-5}$ | $4.20 \times 10^{-4}$ |
| 16 | 32 | 0.0010 | 0.0013 | 0.0015 | $1.72 \times 10^{-7}$ | $2.98 \times 10^{-7}$ | $4.19 \times 10^{-7}$ |

## 6.3 DWT Hardware Architectures

The vast sizes of data transmitted over a channel demands improved methods for compressing and encoding a signal to ensure better bandwidth utilization. The DWT is a relatively new and computationally intensive method for decomposing a signal into several frequency bands and compressing the signals using various coding schemes. The non-even distribution of the energy of the signal in the different subbands allows the original signal to be compressed using wavelets and later on reconstructed with minimal loss of information. An efficient method for implementing subband decomposition is by filtering the input using a quadrature mirror filter bank [62]. The output of each filter is downsampled by a factor of two to remove redundancies in the signal and to maintain a constant data rate.

### 6.3.1 Basic DWT Implementation

We chose to analyze the one dimensional DWT hardware architecture due to its moderate computational complexity and its utilization of FIR filter blocks. The design optimizations presented in Chapter 4 that were applied to the FIR filters were appropriate for improving the performance of the DWT hardware design. Figure 6.1 shows a three-level, eight-channel subband decomposition of a signal $x[n]$ which we used as a case study of a

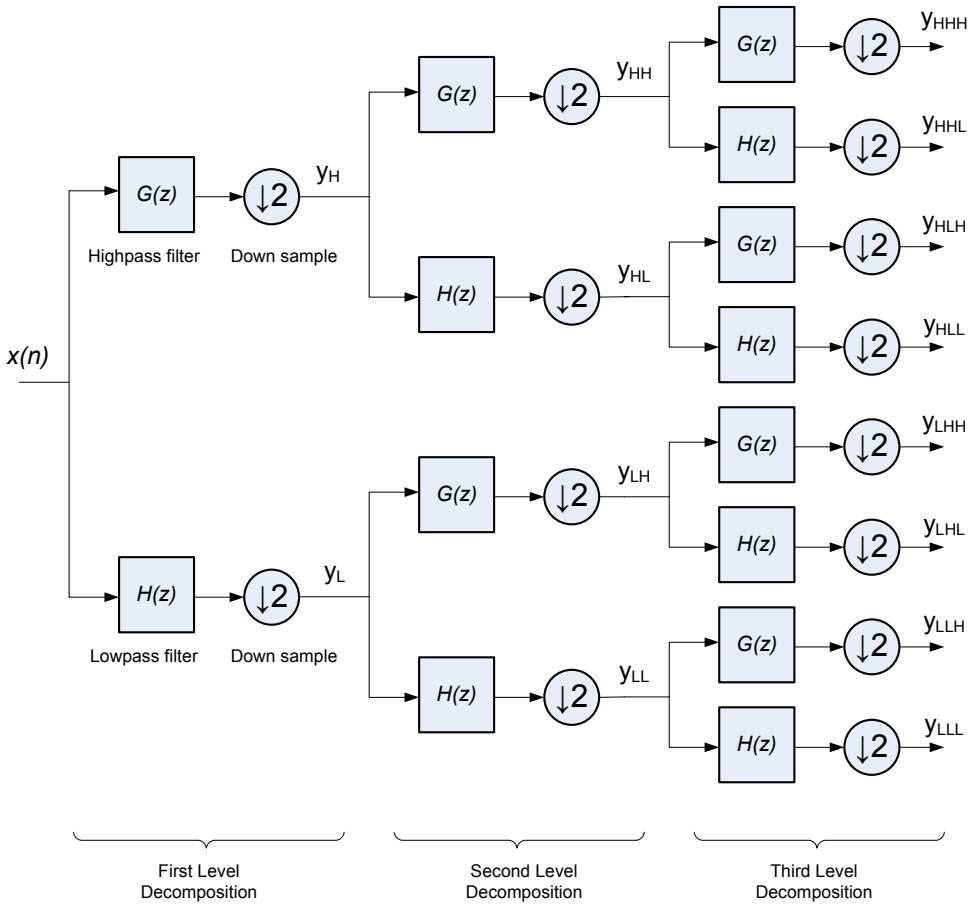



Figure 6.1: Three-level, eight-channel subband decomposition

multi-level, one dimensional DWT hardware block. The analysis filter stage decomposes the signal into two frequency bands using lowpass ($H(z)$) and highpass ($G(z)$) filter blocks, respectively. Each of the two subbands can be further decomposed into smaller frequency subbands at higher stages of decomposition.

The hardware implementation for the DWT architecture consists primarily of filter blocks with fixed coefficients defined by the wavelet function. The SFG in Figure 6.1 is a basic multi-level DWT architecture and can easily be implemented using any one of the FIR filter blocks presented in Figures 4.3 and 4.4. This DWT architecture, while simple, consist of several inefficiencies that designers over the years have addressed in order to improve the hardware performance. We addressed these inefficiencies using our PAF and compared the designs generated using our framework to several implementations presented

in the literature.

### 6.3.2 Survey of DWT Architectures

The three-level, one dimensional DWT architecture utilized by Baganne *et al.* [14] consisted of pairs of identical filter blocks with a single delay tap line feeding into both the highpass and lowpass FIR filter coefficients. This reduced the number of registers in the design. However, the DF FIR filter blocks exhibited a large critical path delay which prolonged the execution time of the design. Additionally, the filter outputs were downsampled which increased the power dissipation due to performing unnecessary computations. The advantage of this basic implementation was the minimal computational latency in the different levels of decomposition. The hardware architecture for the DWT presented by the authors was primarily used to illustrate the process of modeling a DSP algorithm at multiple levels of abstraction. It was assumed that a basic hardware implementation of the DWT block would be adequate to analyze the computational complexity of the system at the RTL.

Marino *et al.* [80] presented a pipelined TF FIR filter structure while employing a coefficient-interleaved design to reduce the number of multipliers. Additionally, a polyphase structure was used for the decimation filter to reduce the number of computations and improve the power dissipation. Each level of decomposition required a slightly different filter structure to accommodate the different types of interleaving. The method used to pipeline the design reduced the critical path from one stage of decomposition to the next. This was accomplished by simply placing registers at the outputs of the FIR filters at each stage. One design consideration that was overlooked was the broadcasting problem in the structure of the TF FIR filter which can potentially increase the power dissipation in the VLSI implementation of the design. However, the optimizations presented in Marino's work is an example of how application-specific FIR filter blocks are necessary to improve the performance of the DWT architecture.

An interleaved DWT architecture was presented by Premkumar *et al.* [81] where the authors implemented a binary tree structure for a three level DWT design. An improved scheme was employed where a single DWT block was used to perform the different levels of decomposition. This scheme is referred to as the folded architecture where the output of the subsequent levels of decomposition are interleaved with the original input in order to

use the single DWT block. Similar folded architectures were presented by Denk *et al.* [82] and Zhang *et al.* [83]. The folded architectures required a fairly complex controller to correctly adjust the flow of data from one level of decomposition to the next. Additionally, using a folded architecture increased the processing time required to filter a signal, which effects the overall throughput of the design. An advantage of this method is a considerable reduction in the number of multipliers required to process a signal through the three stages of decomposition.

### 6.3.3 Efficient Filter Blocks Using the PAF

The types of optimizations included as part of our framework allowed a designer to select specific FIR filter blocks to implement the DWT system. Polyphase filters, interleaving and pipelining techniques can be implemented using our PAF to improve the power dissipation, hardware area and overall throughput, respectively. Manually constructing a hardware design that combines all three design optimizations can require hundreds or thousands of man hours (weeks or months of design time). Chapter 4 highlighted several design optimizations adequate for multi-rate filter banks such as the ones required in the DWT application. Therefore, we used our framework to generate polyphase decimator filters that combined interleaving and pipelining. Each filter generated by our framework was accompanied by a testbench that allowed the designer to verify the correct operation of the filter using test vectors of their choice. Our goal was to use the PAF to efficiently generate application-specific FIR filters for the DWT block and to use the cost functions to guide us in selecting a design that performed well in terms of area, throughput and power dissipation.

Figure 6.2 illustrates the design options we selected through our PAF GUI to design the specific 16-tap FIR filter blocks used in the multiple levels of decomposition. The case study presented in this chapter is an example of how the expertise of a hardware designer is necessary to select the appropriate framework options to generate efficient filter blocks. The next task was to connect the filter blocks in order to implement the entire DWT system. This required manually designing the interfaces between the different levels of decomposition for the correct alignment of data. The final hardware architecture utilized any of the DF, TF, DFII, and TFII FIR filter structures with various pipelining methods. The interface between the different levels of decomposition remained the same for various

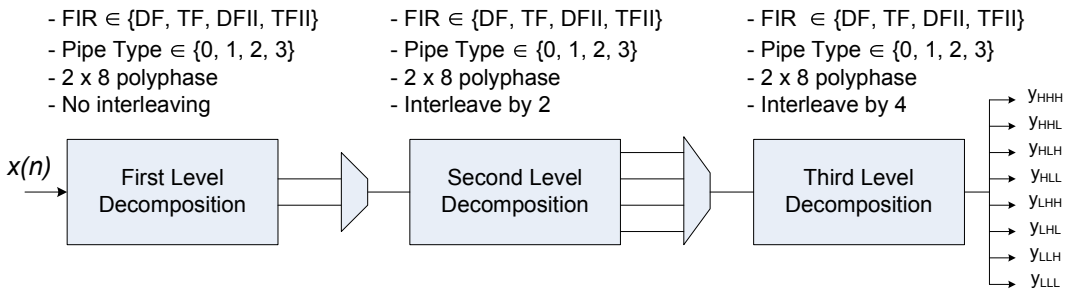



Figure 6.2: PAF design options to generate efficient filter blocks for DWT architecture

filter structures and pipelined cells. The task of combining the framework filters to construct a three-level one dimensional DWT architecture was primarily that of a Masters student seeking to investigate improved DWT hardware implementations. The student was familiar with DSP hardware designs and the PAF through a graduate level DSP Architecture course. She completed and verified the basic hardware implementation of the three-level DWT in less than 100 man hours (less than two weeks) using the PAF.

## 6.4 DWT Design Space Analysis

The three types of optimizations we included in the design of the DWT system included the polyphase decimation filter for reduced power dissipation, pipelining for higher throughput, and data interleaving for lower design area. While each optimization was chosen to improve the performance of a specific metric, the overall effect may degrade the performance of the others. We used our framework to analyze the effects of the three types of optimizations on the performance of the DWT system. Table 6.2 summarizes the impact for each optimization. This data was collected using a supply voltage of 1.8 V and we measured power dissipation using a three-second wavefile sampled at 8 KHz which resulted in an input signal approximately 24,000 samples long. The improvement in power dissipation when we utilized the polyphase decimation filter reduced the power dissipation by approximately 25% rather than the predicted 50%. Pipelining significantly reduced the critical path delay but at a cost of increased area due to the additional registers. Additionally, we observed a considerable increase in power dissipation due to the increase

in switching activity in the enlarged design. The interleaved FIR filter structures reduced the area and power dissipation to values comparable to the basic architecture. However, we maintained a significant improvement in the critical path delay. The overall effect of the three combined optimizations was a 2.5 fold reduction in critical path delay and a 30% reduction in area. The initial results in Table 6.2 allowed us to assess the impact of the optimizations on the overall DWT system. We proceeded with the implementation of the DWT design using the three optimizations and investigated the effect the filter structures had on the overall performance.

Table 6.2: Performance results for 1D DWT Design Space

| Architecture Label | Polyphase | Pipelining | Interleaving | Power ($mW$) | Delay ($ns$) | Area ($mm^2$) |
|---|---|---|---|---|---|---|
| Basic | - | - | - | 645 | 12.83 | 8.1 |
| PlyPhs | $\surd$ | - | - | 484 | 10.78 | 8.2 |
| PlyPhs_Pipe | $\surd$ | $\surd$ | - | 2,088 | 3.70 | 12.6 |
| PlyPhs_Pipe_Intrlv | $\surd$ | $\surd$ | $\surd$ | 658 | 4.98 | 5.7 |

We utilized our framework to construct the FIR filter structures for each level of decomposition, illustrated in Figure 6.3. The first level decomposition split the input signal into highpass and lowpass signal components using a polyphase decimated filter structure. The output of each signal was half the rate of the original signal due to the downsampling in the DWT algorithm. This allowed us to interleave the two signals by alternating each signal at a rate equivalent to the input data rate. The interleaved signal was then processed using a single lowpass and highpass filter, modified to accommodate the interleaved data. Similarly, the output of the second level decomposition consisted of two signals, highpass and lowpass, which we interleaved once more prior to the third level decomposition. We deinterleaved the final output of each filter and obtained eight separate channels, each at one eighth the original data rate. This method of interleaving can be extended to higher levels of decomposition with fairly simple control logic since most of the sequential circuitry

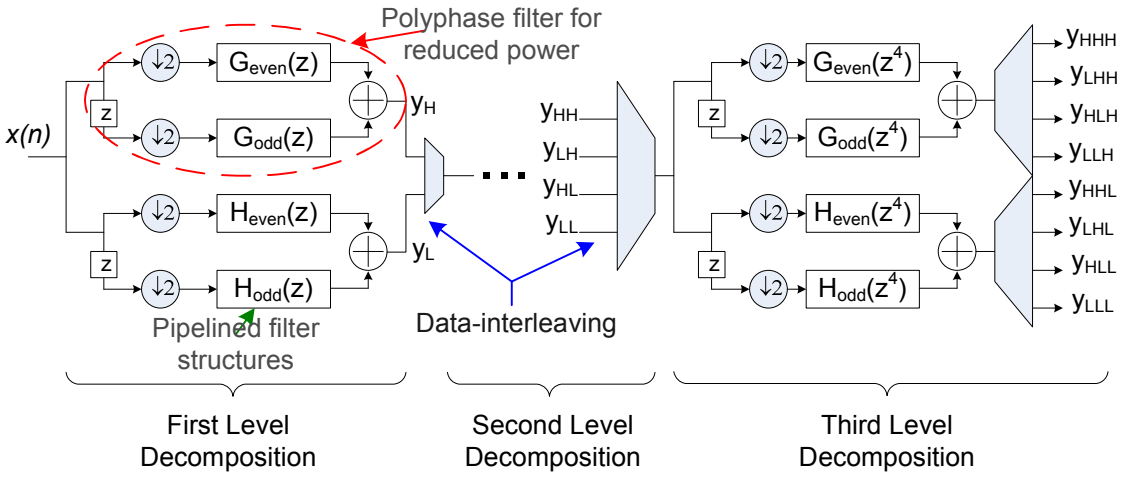



Figure 6.3: Efficient implementation for 1D multi-level DWT architecture

is operating using a single clock rate. We manually designed the interfaces between the different levels of decomposition using synthesizable Verilog semantics.

Figure 6.3 illustrates our method for combining several design optimizations to implement an efficient three-level 1D DWT architecture. We utilized two-phase polyphase FIR filters to implement each FIR filter in the three levels of decomposition. We used interleaved filter structures in the second and third levels of decomposition. Furthermore, we varied the FIR filter structures using the implementations in Figures 4.3 and 4.4. Each structure was pipelined four different ways using the computational modules in Figure 4.9. This resulted in a total of 16 architectural variations that we modeled at the Verilog RTL and verified for correct algorithmic performance. The interface between the outputs and inputs at each level of decomposition was independent of the filter structure and pipeline type which simplified our architectural modifications.

## 6.5 Performance Evaluation

We synthesized each DWT architecture using Synopsys Design Compiler and the $0.18\mu$ standard cell library. We measured the area for each design and varied supply voltage to measure the critical path timing delay and power dissipation. Values for supply voltage included $V_{DD} \in \{1.2, 1.5, 1.8\}$ which further expanded the design space to 48 design permu-

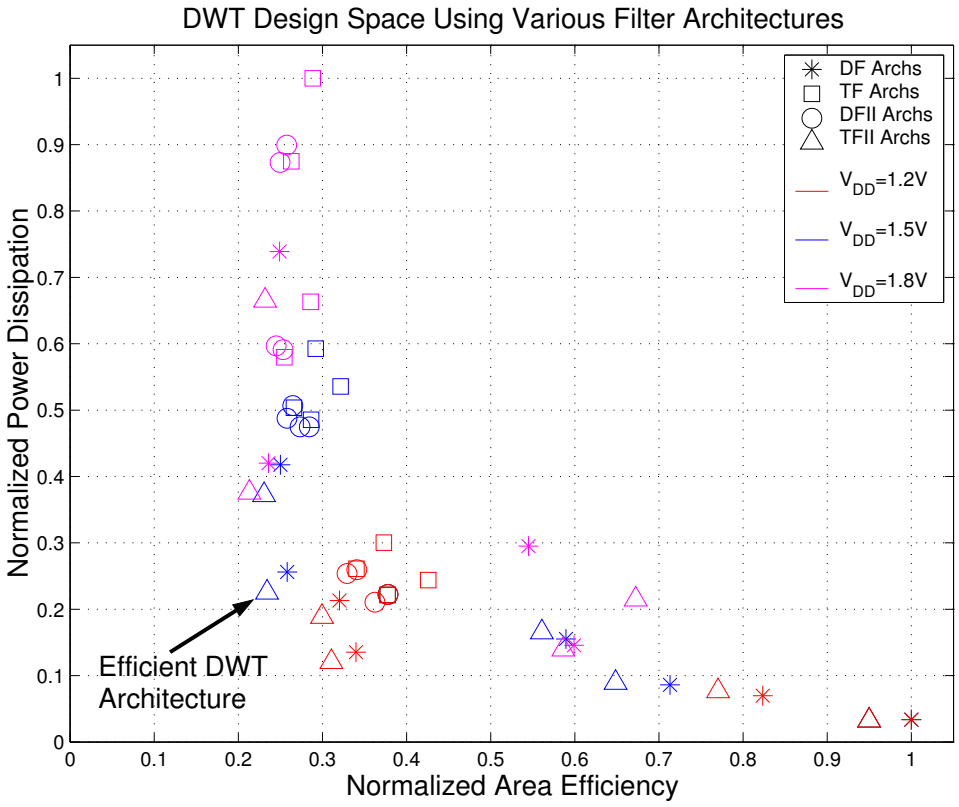



Figure 6.4: Performance metrics for DWT design space

tations. This was necessary in order to employ pipelining techniques with $V_{DD}$ scaling to reduce power while maintaining comparable throughput performance. Each architectural permutation exhibited unique performance metrics. Therefore, we compared the performance of the architectures in the design space by analyzing the area efficiency versus the power dissipation, illustrated in Figure 6.4. We searched the design space for a DWT architecture and supply voltage that resulted in a low power dissipation at a comparable area efficiency. We concluded that an efficient DWT design could be implemented using the pipelined TFII FIR filter, illustrated in Figure 6.5, operating at $1.5V$. The overall DWT architecture using the TFII filter structure exhibited the best area, throughput and power dissipation metrics in the design space.

We analytically compared the area and throughput of our proposed DWT architecture to previously designed architectures which are summarized in Table 6.3. Latency refers to the clock cycles required per computation. Our design performed comparably well

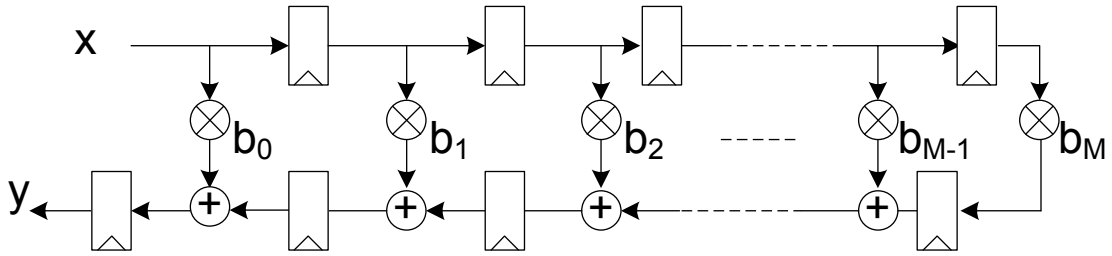



Figure 6.5: Pipelined TFII FIR filter used in DWT design

to Baganne's design [14] in terms of throughput and required less than half the multipliers than Baganne's implementation. Marino's design [80], which used a coefficient interleaving scheme, exhibited an equivalent number of multipliers to our design, but required twice as many clock cycles to process each sample. Denk [82] and Premkumar's [81] folded architectures operated at one fourth the throughput of our architecture, however, required one third the number of multipliers. While Zhang's [83] semi-folded architecture required fewer multipliers than our proposed architecture, we were still able to achieve a two fold improvement in throughput. Our proposed high-throughput architecture required a larger design area compared to the folded architectures and future work is needed to reduce the number of multipliers without sacrificing latency.

## 6.6 Chapter Summary

This chapter presented an example of how our PAF can be used to generate specific FIR filter blocks appropriate for the hardware implementation of a multi-level DWT system. We assumed that the designer had detailed hardware knowledge and can therefore set specific parameters within our framework GUI to generate the desired filter structures. We illustrated that several architectural optimizations can be combined to generate a pipelined FIR filter using a polyphase structure along with interleaving. Most hardware implementations of the DWT combine one or two design improvements due to the complexities involved in adding additional optimizations. Our PAF allowed us to quickly and easily assess the impact of specific design optimizations on the performance of the overall DWT system. This was important for constructing an initial implementation of the DWT design without sacrificing performance or increasing design efforts. The next chapter discusses the quality

Table 6.3: Performance comparison for 16-tap, 3-level DWT Designs

| DWT Design | No. of Mults. | No. of Adders | No. of Regs. | Latency (cycles) | Critical Path |
|---|---|---|---|---|---|
| Baganne [14] | 224 | 210 | 105 | 1 | $T_{mult} + 15T_{add}$ |
| Denk [82] | 32 | 32 | 96 | 4 | $T_{mult} + T_{add}$ |
| Marino [80] | 96 | 90 | 224 | 2 | $T_{mult} + T_{add}$ |
| Premkumar [81] | 32 | 30 | 472 | 4 | $T_{mult} + T_{add}$ |
| Zhang [83] | 64 | 60 | 120 | 2 | $T_{mult} + T_{add}$ |
| Proposed | 96 | 90 | 448 | 1 | $T_{mult} + T_{add}$ |

and usefulness of our PAF by addressing several design issues that concern both algorithm developers and hardware designers.

# Chapter 7

# Framework Evaluation

Chapter 3 presented the methodology employed by our framework to generate and analyze well-structured hardware designs for basic DSP functions. Chapters 4, 5 and 6 presented three computationally intensive DSP applications where we utilized our framework to search the design space for hardware implementations that performed well for specific design constraints. In this chapter, we evaluate the effectiveness of our work in addressing some of the pertinent issues designers are concerned with when refining a DSP algorithm to a hardware design. Some of these issues include the efficiency of the algorithmic refinement process, accuracy in estimating the performance metrics, and the validity of the hardware designs generated by our PAF.

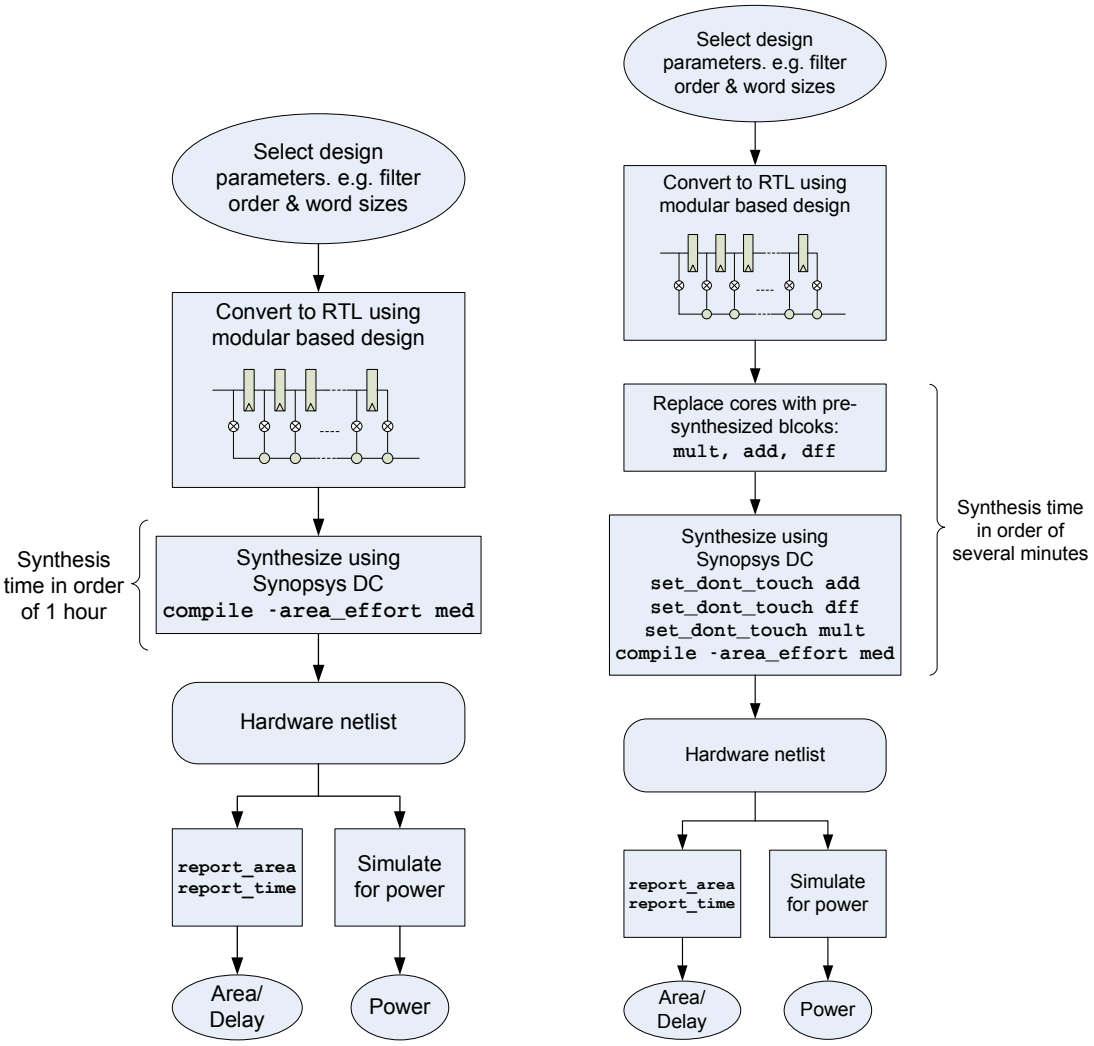
## 7.1 Synthesis Times Using Pre-Synthesized Blocks

The process we developed for estimating area and timing metrics for a design used several Synopsys commands compiled into a single script. The first command in the script read in the design generated by the PAF described at the Verilog RTL. The next Synopsys command performed logic and gate level synthesis on the current design in an attempt to implement a combination of standard cell blocks that meet the functional, area and speed requirements of the DSP system. We used `medium effort` for area when synthesizing a design. The `report_area` and `report_timing` commands were used to estimate the cell

count and timing for the design, respectively. The scripts utilized by our framework repeated the synthesis process while varying the designs clock period until the timing "SLACK" was 0.00 or 0.01 $ns$. We applied this process to a design of medium complexity, such as a 32-tap DF FIR filter with 16/32 bits for input/output running on an Intel Pentium-4, 3 GHz CPU Linux machine, which executed in approximately 55 minutes.

We utilized the Synopsys DesignWare mathematical blocks for reducing the synthesis times of the hardware designs generated by our framework. We identified the core blocks required to construct a DSP hardware design, such as adders, multipliers and registers, and applied a bottom-up modular design methodology to model the DSP algorithm. The area, timing, and power information was primarily determined by the hardware layout of the mathematical blocks. Therefore, we characterized the performance for each mathematical block and used this information to estimate the performance for the entire design. Our method for characterizing area and delay for each block consisted of synthesizing the DesignWare blocks and generating a database *.db* files. The PAF synthesis scripts generated the entire designs netlist by instantiating each of the *.db* core files. The Synopsys `report_area` and `report_timing` commands were used once more to estimate the designs area and delay. We automated this process as part of the PAF which executed in approximately 4 minutes for the 32-tap 16/32-bit FIR filter considered earlier. Figure 7.1 summarizes the two methods provided by our framework for synthesizing a design and estimating performance metrics.

We estimated the performance of the designs presented in the previous chapters using the pre-synthesized blocks. We recorded the times required to fully synthesize the designs in order to quantify the speed up in utilizing the pre-synthesized blocks for estimating the metrics. Table 7.1 summarizes the synthesis times for the case studies presented in Chapter 4. We achieved a 14 fold improvement in analysis times for the designs considered in Chapter 4 and observed a similar speed-up for the design examples considered in Chapters 5 and 6. It should be noted that the synthesis times using the full-synthesis method were recorded using several hardware architectures in the design space. The duration was then approximated to include the rest of the architectures in the design space.




(a) Complete synthesis

(b) Utilizing pre-synthesized blocks

Figure 7.1: Synthesis methods employed by the PAF

## 7.2 Estimating Performance Metrics Using Pre-Synthesized Blocks

The designs generated by the PAF conformed to a well-structured and regular set of hardware implementations modeled at the synthesizable RTL using Verilog. The

Table 7.1: Synthesis times for different order FIR filters

| Design Example | Filter Length | Coefficient/ Input Word Sizes | Output word size | Synthesis times | | | |
|---|---|---|---|---|---|---|---|
| | | | | with pre-synthesized cores | | without pre-synthesized cores | |
| | | | | per design | design space | per design | design space |
| High-thruput | 8 | 8 bits | 12 bits | 13.6 secs | 29 mins | 3 mins | 6 hrs 30 mins |
| Low-area & high-thruput | 16 | 12 bits | 24 bits | 1 min 50 secs | 240 mins (4 hrs) | 26 mins | 55 hrs |
| Low-power & high-thruput | 32 | 16 bits | 32 bits | 3 min 45 secs | 480 mins (8 hrs) | 50 mins | 107 hrs |

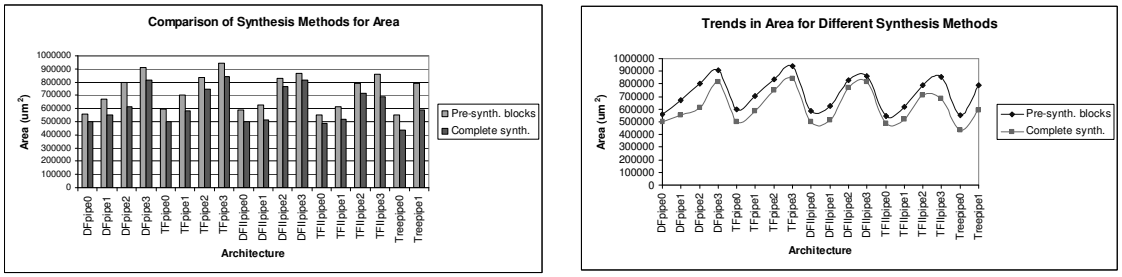
framework scripts invoked Synopsys Design Compiler in order to synthesize the designs and estimate the hardware performance metrics such as area, delay and power dissipation. The layout of the synthesized designs using the standard cell blocks maintained the original SFG structure which resulted in different performance metrics for the same algorithmic behavior. For example, both the DF and TF structure can be used to implement the convolutional sum of the FIR filter. However, the registers between the chain of adders in the TF structure resulted in different metrics compared to the DF structure. The data-flow structure for the designs remained approximately the same after they were synthesized. Therefore, the designer can guide the layout for the low-level design netlist according to the structure of the Verilog RTL code.

The performance metrics we estimated using the pre-synthesized blocks differed slightly than the metrics obtained using the full-synthesis method. This is illustrated in Figure 7.2 for the case study of the 16-tap 12/24-bit FIR filter which was analyzed in Chapter 4 as a design example. The bar charts for area and delay illustrate that the performance metrics estimated using the pre-synthesized blocks were slightly larger than the estimates obtained by fully synthesizing the designs. These results were expected since the pre-synthesized blocks restricted the Synopsys optimization algorithms from rearranging the standard cell blocks to further reduce the designs area and delay. A slightly different effect was observed for the power metric where several designs that were refined using the full-synthesis method exhibited slightly greater power dissipation than designs that used the pre-synthesized blocks. However, the performance trends in all three bar-charts of Figure 7.2 using the pre-synthesized blocks were similar to the trends for estimating the metrics using the full-synthesis method. Therefore, designers may find it beneficial to use the pre-synthesized blocks when analyzing large design spaces.

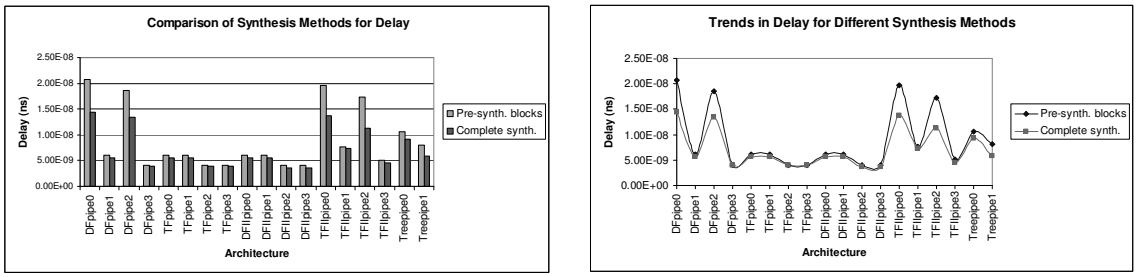
## 7.3 Validating PAF Architectures

The architectural variations in implementing a DSP design were necessary for providing the designers options in selecting hardware implementations that performed well. The multitudes of hardware designs generated by our PAF to implement a DSP algorithm, such as the FIR filter, differ in the type of pipelining method and level of parallelism. The methodology used to construct the hardware designs employed by the PAF was developed by several research students, who we refer to as the framework developers. The human element in developing the PAF contributes to the potential existence of errors in the hardware designs. Any minor deviation in design from the intended SFG could easily generate erroneous results, leaving the framework user with little confidence in the validity of the hardware designs generated by the PAF and with doubts in the accuracy of the performance metrics.

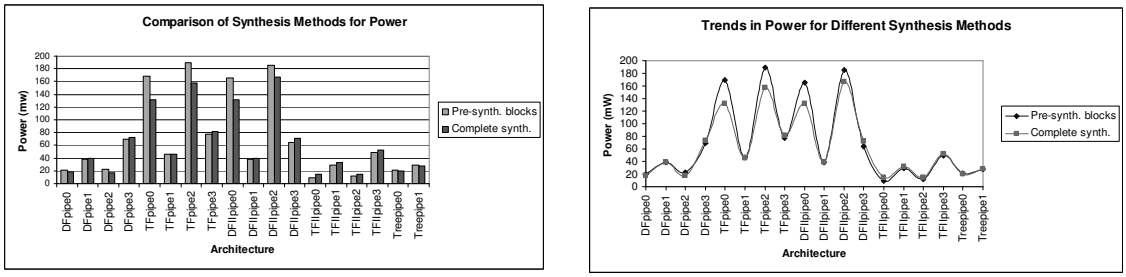
We addressed this concern by including a "self test" script written in MATLAB that continuously generated various hardware implementations and simulated the designs at the RTL. We automated this process to relieve the framework developer from manually verifying the designs generated by the PAF and to reduce human errors in the steps required




(a) Area




(b) Delay




(c) Power

Figure 7.2: Comparison of accuracy in metric estimation: complete synthesis versus using pre-synthesized blocks

to simulate the hardware models. The simulation outputs for the designs generated at the architectural level were compared to the output of the DSP function generated at the algorithmic level. We utilized SystemC's floating-point data-types for this process of architectural verification to expedite the simulation times, which we observed in [36]. Additionally, the use of floating-point data-types ensured compatibility in computational precision between the outputs of the hardware designs and the outputs of the high-level algorithm.

Figure 7.3 illustrates the process we employed in verifying the designs generated by our PAF. The PAF initially implemented the DSP function, FIR filter for example, at the algorithmic level where sample input sequences were used to generate an output response. A script written in MATLAB randomly generated the necessary hardware parameters needed to construct the DSP algorithm. Examples of hardware parameters included:

- filter structure $\in$ {DF, TF, DFII, TFII, DF with Tree adder}
- pipeline type $\in$ { none, low, medium, high} (for high throughput designs)
- level of parallelism $\in$ {integer ¡ order} (for polyphase structures)
- interleave rate $\in$ {integer $\geq 1$} (for multi-signal applications)
- decimation factor $\in$ {integer ¡ order} (for multi-rate DSP algorithms)
- SOS structure $\in$ { none, non-pipelined, pipelined}

The input test sequences were used once more for the inputs to the hardware designs in order to generate an output response. The MATLAB script compared the architectural output to the algorithmic output response using the peak error and MSE. Designs that generated a peak error greater than $10^{-15}$ were considered to contain errors in the implementation and the parameters used to construct that particular design were reported back to the framework developer for detailed debugging. We determined the peak error threshold based on the machine used to simulate the designs. A floating-point `double` precision computation performed on a 64-bit machine is accurate to $2^{-52} \approx 10^{-16}$, where 52 of the 64 bits represent the fractional portion in a 64-bit IEEE 754 double. The architectural verification process continued until the framework developer terminated the process or until the maximum allowable time to reserve a Linux machine was exceeded; in which case, the verification process can be repeated under a new reservation. The verification results were temporarily logged into a file as a record of designs that were verified along with the results for the peak error and MSE. To date, we verified approximately 20,000 FIR filter hardware permutations. Design errors generated from these runs were addressed and corrected. We included a step in our verification process wherein we **deliberately** induced design errors after every 1,000 designs that passed the verification process. This was necessary to confirm that design errors were correctly reported to the framework developer. The framework developer was

made aware that the design errors were deliberate and that no action to correct the design was necessary.

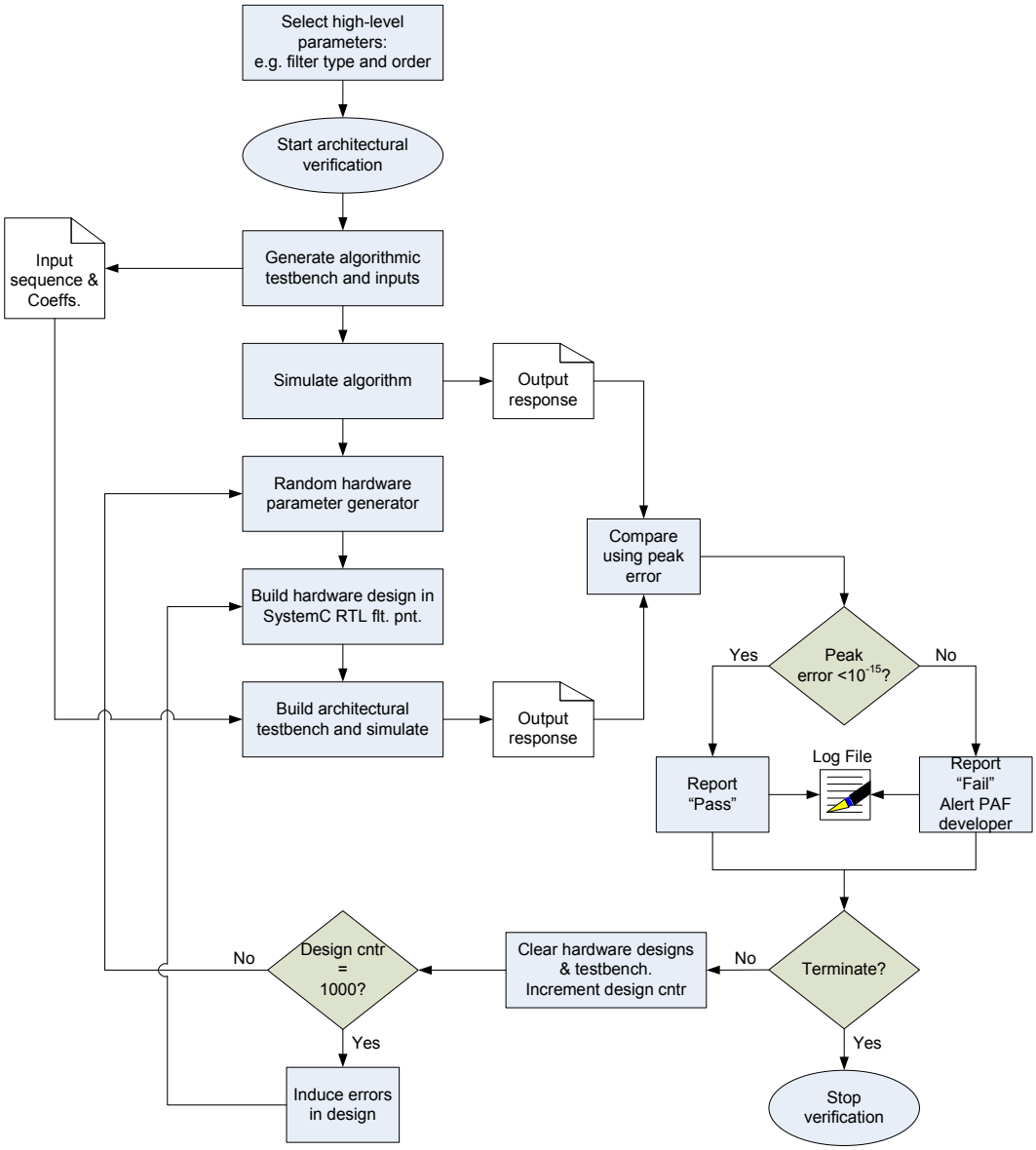



Figure 7.3: Flow chart for automated architectural verification process

## 7.4 PAF versus Similar Frameworks

Several tools exist that address various methods for refining a DSP algorithm to a synthesizable hardware design. The common goals for such tools are to efficiently refine a hardware design from a high-level of design detail and to ensure that the resultant design meets the final system specifications. We attempted to address both these goals through the development of our PAF. We compared the effectiveness of our methodology to those that have been presented in the academic domain as well as the industrial venues. The goal in developing our framework was not to compete or challenge existing work, but more so to address pertinent architectural design and analyses features that were not emphasized in similar tools. The following three subsections highlight the merits for three CAD tools used for DSP algorithmic refinement and how our work attempted to address some of their limitations.

### 7.4.1 Synopsys DesignWare IP Blocks

Synopsys Design Compiler is a well-established industrial tool used to synthesize hardware designs described at the Verilog or VHDL RTL. The range of hardware designs considered for synthesis depends primarily on the final system performance requirements. Computationally intensive DSP systems are generally considered for hardware implementation due to their fast computational times, low circuit area and reduced power dissipation when compared to using GPPs. Therefore, Synopsys provides a limited library of commonly used DSP IPs that designers can utilize for constructing their hardware systems. The advantage of providing pre-designed IPs relieves the designer from manually describing the DSP design at the lower levels of abstraction such as the RTL. However, as the name signifies, IPs are generally hard cores which are reusable blocks in the form of an encrypted box that designers incorporate into a hardware platform [84]. The Synopsys DesignWare FIR filter is an example of a parameterized IP core that designers can simply instantiate into their Verilog descriptions. The Synopsys vendors establish a fixed architecture for the DSP IP core before it is made available to the customer, which the designer has to contend with. In most cases, the IP cores are carefully designed in order to include optimizations that increase their performance.

The TF structure is used to implement the Synopsys FIR filter IP core due to its

high throughput compared to the conventional DF structure. Therefore, we compared the hardware performance of the Synopsys FIR filter to the non-pipelined TF filter generated by our framework, the results of which are summarized in Table 7.2 for the three FIR filter design examples analyzed in Chapter 4. The results show that the performance for the non-pipelined TF filter generated by our framework closely matched the performance of the Synopsys DesignWare FIR filter. This provided us with a sense of validation in the methodology we developed for designing and synthesizing the hardware architectures generated by our framework. The effort required to generate and utilize the PAF filters was comparable to that of the Synopsys IP filter core with the added advantage that the hardware designer can alter the structure generated by our framework if he or she wished to further optimize the hardware performance.

Table 7.2: Performance comparison between Synopsys TF FIR filter and PAF TF filter

| Filter | Word | DesignWare TF Filter | | | PAF TF Filter | | |
|---|---|---|---|---|---|---|---|
| Length | Size | Area ($\mu m^2$) | Delay ($ns$) | Power ($mW$) | Area ($\mu m^2$) | Delay ($ns$) | Power ($mW$) |
| 8 | 8 | 135235 | 4.75 | 53 | 149265 | 4.51 | 64 |
| 16 | 12 | 529607 | 5.20 | 212 | 533463 | 4.98 | 235 |
| 32 | 16 | 1627640 | 5.81 | 679 | 1545071 | 5.57 | 634 |

### 7.4.2 High-Level Synthesis

Designers use HLS tools, such as Catapult-C, to synthesize a design described using a high-level coding style. The FIR filter algorithm can be used to illustrate the basic example of synthesizing a design from a high-level description. We used the experimental setup for a 15-tap constant coefficient filter presented in [85] where an FIR filter algorithm described using Catapult-C was synthesized to assess the area and throughput of the final hardware design. We applied the design methodology described by Catapult-C [44] using a behavioral Verilog coding style since we did not have access to the Catapult-C tools. Our goal was to illustrate how a designer can efficiently model a hardware block using a

high-level coding style, but at a potential sacrifice in performance.

Table 7.3 illustrates the performance for the FIR filter synthesized from a high-level of design detail comparable to the Catapult-C coding style. We compared the performance metrics to the non-pipelined filters generated by our PAF. We measured the performance metrics with the $0.18\mu m$ standard cell library and a supply voltage of 1.8 $V$. The results show that the delay for the filter described at the algorithmic level outperformed the DF and TFII filters, while we were able to achieve better results with the TF and DFII designs, and comparable results using the DF tree adder structure. Similarly, the DF, TFII and DF tree adder structures generated by our PAF performed better than the behavioral model in terms of power dissipation. One advantage for using the behavioral coding style to design an FIR filter in hardware is the efficiency in the number of lines of code. However, this style of modeling a hardware architecture limits the designer from applying additional optimizations for further improving the hardware performance. We developed our framework to address both design efficiency and architectural performance. The designer can utilize our framework to generate hardware filters modeled at the RTL that perform well while keeping design efforts to a minimum.

Table 7.3: Behavioral FIR filter design versus framework filters modeled in RTL: 14-tap, 16-bit FIR filter

| Filter Structure | Lines of code | Area ($\mu m^2$) | Delay ($ns$) | Power ($mW$) |
|---|---|---|---|---|
| DF | 261 | 772067 | 13.22 | 0.04555 |
| TF | 267 | 771786 | 5.4 | 0.3249 |
| DFII | 264 | 764569 | 5.37 | 0.3221 |
| TFII | 276 | 737021 | 12.72 | 0.03837 |
| DF Tree | 309 | 672393 | 8.81 | 0.04845 |
| Behavioral | 29 | 708002 | 8.47 | 0.1746 |

### 7.4.3 AccelDSP and FIR Compilers

The Xilinx LogicCORE $^{TM}$ FIR Compiler is an EDA tool that automatically generates different FIR filter structures using synthesizable RTL based on design options provided by the user. The filter structures generated by this tool are part of the "AccelDSP$^{\text{TM}}$ Synthesis Tool" [40] which originated as a research project by Prith Banerjee and several of his students [21] [37]. The tool exploits transfer function characteristics to determine the best implementation for the filter design. One important characteristic that significantly alters the hardware performance for a filter structure is coefficient symmetry which we used as an example to compare the FIR Compiler designs with our framework designs. Linear-phase FIR filters can be implemented using a folding technique in the filter coefficients that reduces the number of multipliers by approximately half. This optimization feature is included in the Xilinx FIR Compiler which generates the non-pipelined DF linear-phase FIR filter illustrated in Figure 7.4. The structure improves hardware performance in terms of area over the direct implementation. However, other linear-phase FIR filters exist in the design space that can further improve performance in terms of throughput and power dissipation. Table 7.4 displays the performance results for various linear-phase FIR filters generated using our PAF which includes the structure of Figure 7.4. We analyzed the performance of different structures of a 16-tap symmetric FIR filter which we synthesized using the $0.18\mu m$ standard cell library and a supply voltage of 1.8 V. The results show that the non-pipelined DF structure exhibited the least area but the largest critical path delay. The TF structure with pipelined multiplier exhibited the least critical path delay, but at the expense of large area and power dissipation. A compromise for area, delay and power dissipation would be the DF symmetric filter with pipelined multipliers. Our framework generated various symmetric filter structures in a similar design effort to the Xilinx FIR Compiler. The advantage of using our framework was that the designer was not limited to a single symmetric filter structure and had the ability to make architectural decisions based on performance constraints.

## 7.5 Chapter Summary

This chapter presented several EDA tools for refining a computationally intensive DSP algorithm, such as an FIR filter, to a synthesizable hardware block. The goals of

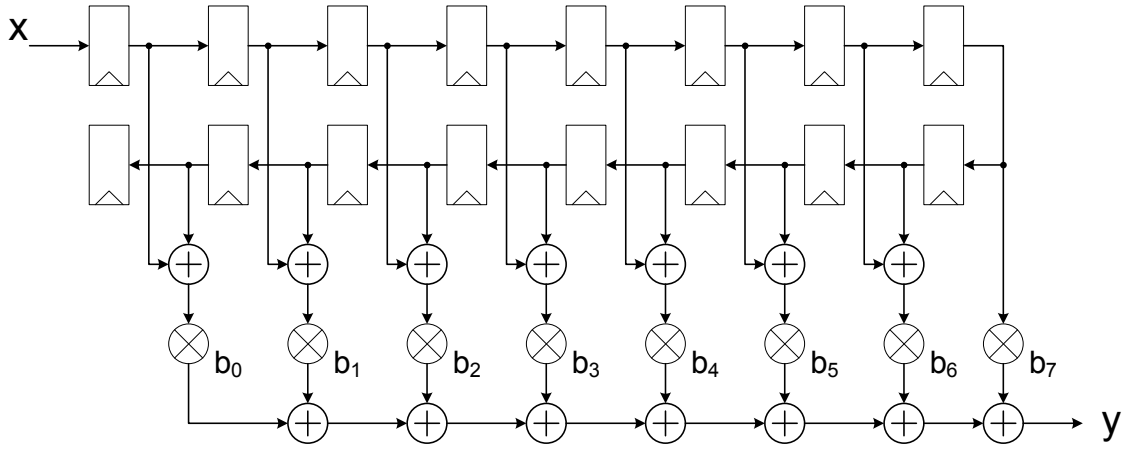



Figure 7.4: Symmetric FIR filter structure included in the Xilinx LogicCORE $^{TM}$ FIR Compiler

Table 7.4: Comparing performance metrics of Xilinx FIR Compiler to PAF FIR filter for 16-tap, 12-bit symmetric FIR structures

| Filter Structure | Pipeline Mult. | Area ($\mu m^2$) | Delay ($ns$) | Latency | Power ($mW$) |
|---|---|---|---|---|---|
| Direct Form | no | 259829 | 7.75 | 2 | 21 |
| Direct Form | yes | 365367 | 4.75 | 3 | 40 |
| Transpose Form | no | 344166 | 5.04 | 2 | 147 |
| Transpose Form | yes | 553017 | 3.36 | 3 | 181 |

these tools are to relieve the designer from manually describing the design at the RTL and to generate a design that meets the hardware performance specifications. We developed our performance analysis framework to address both issues and evaluated the effectiveness of our methodology. We demonstrated that the pre-synthesized blocks utilized by our performance estimation methodology reduced synthesis times from hours to minutes. These results are important when analyzing large design spaces consisting of hundreds of design permutations. We compared the effectiveness of our work to several similar tools that focused on a methodology of refining and synthesizing a DSP hardware design from a level of abstraction higher than the RTL. Table 7.5 summarizes the merits of these tools and how we attempted to address some of the areas they lacked in.

Table 7.5: Survey of Frameworks

| EDA Tool | Data-Type | Architectural Exploration | Performance Estimation | Modifiable HW Code | Designer Expertise | Target (ASIC/FPGA) |
|---|---|---|---|---|---|---|
| Synopsys DW Blocks | fxd-pnt | None | None | None | High | ASIC |
| Catapult-C | fxd/flt-pnt | Limited | $A$,$T_d$ | Difficult | Medium | both |
| AccelDSP/FIR Compiler | fxd/flt-pnt | Limited | $A$,$T_d$, $P$ | Difficult | Medium | both |
| This work | fxd/flt-pnt | Extended | $A$,$T_d$, $P$ | Simple | Hi-Lo | ASIC |

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

This work presented a framework for guiding a designer in selecting hardware designs for basic DSP algorithms that performed well for specific design constraints. We clearly defined a flow and methodology for generating and analyzing the performance of well-structured hardware designs subject to various types of optimizations. We used our PAF to generate and analyze the design space for three DSP applications that required computationally intensive operations. The results of the case studies are important in illustrating the efficiency of generating synthesizable hardware designs that meet specific design constraints. The design and analyses concepts presented in this work are extendable to other DSP algorithms similar in computational complexity to that of the FIR filter.

We established concise levels of design abstraction and identified the appropriate languages and tools to utilize at every level. This was useful for generating different hardware designs while applying different optimization techniques. The three case studies we analyzed in this work proceeded from the algorithmic level, through the arithmetic and architectural levels, down to the circuit level where we estimated area, throughput, and power dissipation using commercial synthesis tools. Figure 8.1 summarizes the goals and contributions of our design flow in refining a DSP algorithm to several hardware implementations and analyzing their performance.

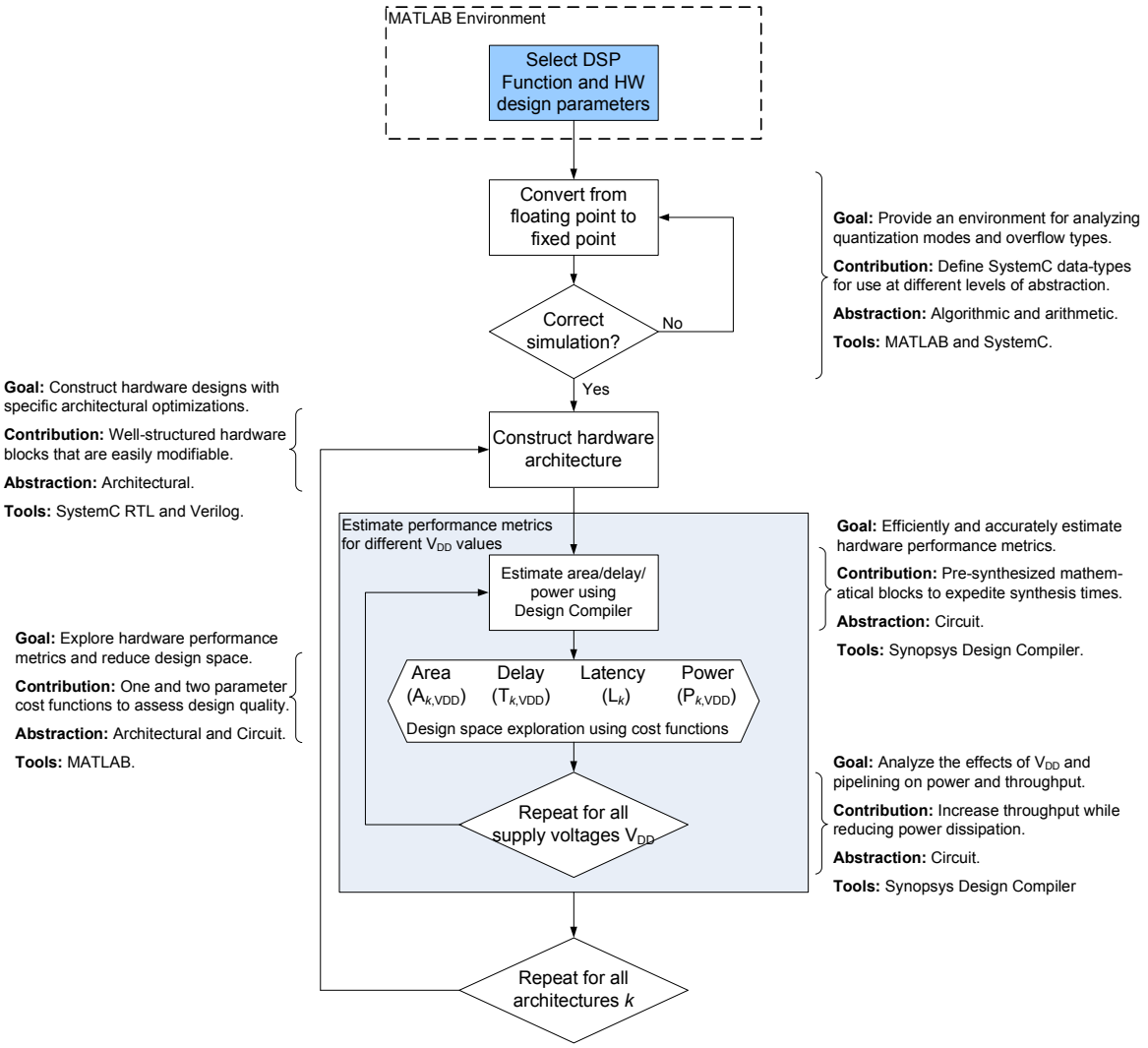



Figure 8.1: Summary of goals and contributions for the PAF flow

The overall goal of our work was to **bridge the gap** between the high-level DSP algorithm and the **efficient** hardware architecture. This single goal statement presents two main design questions concerning our framework: 1) Did we successfully reduce the design efforts from an algorithm to a hardware implementation? and 2) Are the hardware designs generated by our framework efficient? We answered the first question by comparing our flow to methodologies and tools presented in the literature and those that are commercially available. "AccelDSP$^{TM}$ Synthesis Tool" [40] (formally "AccelChip") is one tool of

particular interest that initially started as an academic research project and eventually became part of the Xilinx DSP suite. While this tool was invaluable for efficiently generating hardware designs from high-level MATLAB descriptions, the user had to contend with the limited set of DSP hardware implementations in the design space. Any further architectural improvements were the responsibility of the user who had to implement the optimizations using MATLAB as an HDL; a language not intended for hardware design.

Our framework adopted the concept of generating synthesizable hardware blocks that can easily be instantiated into a larger hardware system. The Synopsys Design Compiler [5] utilizes this concept in an effort to improve the hardware design effort at the RTL. Several IP hardware blocks exist in the Synopsys tool that can be used to implement basic DSP algorithms such as filtering. The IP blocks can simply be instantiated into a hardware system using the appropriate interfaces. However, this constrains the designer from modifying the IP block to further improve the performance of the designs for specific DSP applications. Our framework can be used to generate hardware blocks similar in performance to the Synopsys IP blocks with the added advantage of allowing the designer to modify the HDL source files if required.

We attempted to address the second design question concerning algorithmic refinement tools: How does the designer know that the hardware architectures generated by an automated design flow are efficient? We introduced the importance of cost functions that assess the quality of the hardware implementations in the design space to address this issue with our framework. We left it up to the designer to select the cost functions he or she wished to use to evaluate the performance of a design. Our framework generated multitudes of feasible hardware implementations and with the use of the cost functions, we were able to reduce the design space without ruling out any of the generated designs. We utilized pre-synthesized mathematical elements to estimate the performance of the hardware designs without increasing the design efforts and analyses times. This method of expediting synthesis times proved useful and accurate for the types of hardware designs considered in this work.

The tools we analyzed for refining a DSP algorithm to a hardware design assumed that the user had limited hardware experience and therefore may not be too inclined to modify the hardware designs generated by the tool. The assumptions we made in the development of our PAF included users with various levels of hardware expertise. Therefore, the hardware designs generated by our framework conformed to a well-structured and regular

design technique that allowed a designer to easily modify the resultant hardware architectures if they wished to do so. This was the case for the third design example discussed in this work where an experienced hardware designer modified the outputs of the PAF filter blocks to establish a concise interface with successive hardware blocks.

The contributions of this work are in the method we developed our design and analysis flow. We successfully illustrated how we can integrate several commonly used design languages and tools into a single automated flow that improved the refinement process of a DSP algorithm to a hardware implementation. We accomplished this by providing entry and exit points to our framework flow for each level of abstraction. This promoted adaptability in our flow should any of the design languages, synthesis tools, DSP applications, or hardware platforms change. The current state of our PAF is still in its infancy. The architectural design techniques and hardware analysis concepts used in this work can be conveyed to students in various electrical and computer engineering venues to improve the quality of our framework.

## 8.2 Future Work

The development of our PAF must be utilized for DSP algorithms beyond FIR filters to increase the appreciation of our design and analysis methodology. We successfully demonstrated an automated flow for designing various FIR filter structures and analyzing their performance in order to guide designers in selecting a hardware design that performs well for specific constraints. Additionally, we illustrated that our framework can be used for generating efficient filter structures used in several computationally intensive DSP algorithms. As the hardware system extends beyond the FIR filter and its applications, the designer may abandon our automated design tool and resort to the traditional and manual process of refining DSP algorithms to hardware implementations. Furthermore, restricting an automated design tool to a single hardware platform, such as ASICs, limits the use of our framework for larger SoC designs. This section presents several important perspectives involved for continuing the growth and expansion of our PAF.

### 8.2.1 Mapping DSP Algorithms onto FPGAs

Currently, our PAF is catered to synthesizing a basic DSP algorithm to hardware architectures suitable for ASICs. However, the drawbacks for using ASICs for the hardware implementation of a DSP application are primarily due to the high cost in designing an ASIC and the prolonged time required for final chip fabrication. ASICs are used for hardware implementations of DSP algorithms when several millions of units are introduced into the hardware market, which spreads out the design cost to many units. Additionally, ASICs require a time-consuming test process before they can be manufactured in bulk quantities.

The FPGA can overcome the problems of high-cost and prolonged design and verification time. An FPGA is a semiconductor device that contains programmable logic components and programmable interconnects also referred to as configurable logic blocks (CLBs). The CLBs can be programmed to match the functionality of basic logic gates such as NOT, AND, OR or more complex combinational and mathematical functions. In most FPGAs, the CLBs also include memory elements, which may be simple flip-flops or blocks of memory. The popularity of using FPGAs for implementing on-chip hardware designs has increased over the past decade and is therefore mass-produced reasonably inexpensively. Targeting our work for generating FPGA hardware designs increases the utilization of our PAF.

Our PAF must support types of pipelining and levels of parallelism to ensure the creation of a hardware design that performs well for a specific FPGA technology. One method for catering our framework to FPGA designs is to model the FPGA mathematical blocks at different levels of abstraction. This ensures that the algorithmic model for the DSP algorithm matches the functionality of the synthesized hardware design and therefore improves the design and verification process. Our methodology for applying cut-set retiming needs to be applicable to the types of pre-designed IP blocks provided by the various FPGAs. This improves the quality of hardware designs generated by our framework for multiple DSP applications that are mapped onto the FPGA platforms.

One challenge for integrating FPGA designs into our framework lies in the efficient method for estimating the hardware performance metrics for large design spaces. One possible solution would be to employ the complete synthesis methodology illustrated earlier in Figure 7.1(a) for estimating area, throughput and power. This may be a sufficient method for analyzing an FPGA design since synthesis and place & route times for FPGAs

are considerably less than synthesis times for ASICs.

One research area that has gained popularity over the past several years is mapping image processing algorithms onto FPGAs. The computational intensity of such algorithms are suitable for hardware implementations that exhibit high throughput and low power dissipation rather than implementing them using a GPP. The two dimensional DWT using both the convolution-based and lifting-based algorithms are challenging imaging applications with various venues of research for improving the hardware performance. Optimizations such as decimation filters, pipelining and interleaving can be applied to the two dimensional DWT to significantly improve the hardware performance using an FPGA. Therefore, the two dimensional DWT is an attractive system for initiating the foundation of developing a methodology that refines and maps imaging algorithms onto FPGAs.

### 8.2.2 Additional DSP Functions and Algorithms

We utilized the basic, yet important, FIR filter as the basis for analyzing different hardware implementations of a computationally intensive DSP algorithm. Designers can arrive at a more complete implementation of the hardware system by providing a larger library of DSP functions and hardware blocks. However, the types of DSP functions and algorithms seems endless with the continuous increase in state-of-the-art complex signal processing systems. One way to maintain the value of our PAF is to focus our efforts on hardware designs for a specific class of DSP applications. Hardware implementations for imaging algorithms are gaining popularity in the area of digital image and video processing. Therefore, the types of DSP functions and algorithms suitable for hardware refinement can be limited to the types required for imaging applications such as image compression, object classification, segmentation and identification.

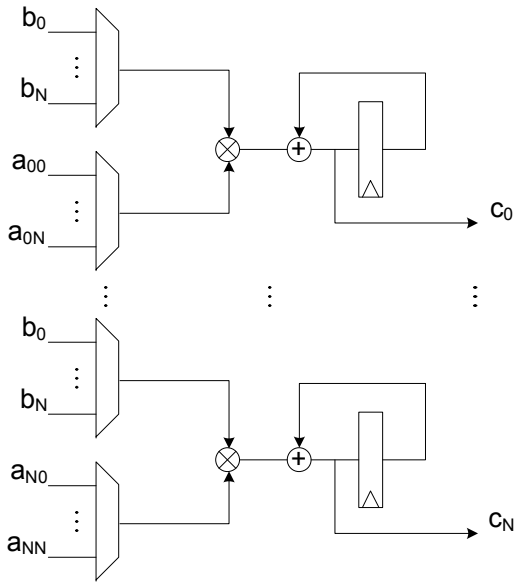
Matrix-vector multiplication is a necessary operation for signal and image processing applications and multi-channel communication systems. A hardware implementation of this operation can significantly improve the performance of a system in terms of area, throughput and power dissipation. However, this depends on the hardware design and optimizations used to implement the matrix-vector operations. The matrix-vector operation

can be represented mathematically as

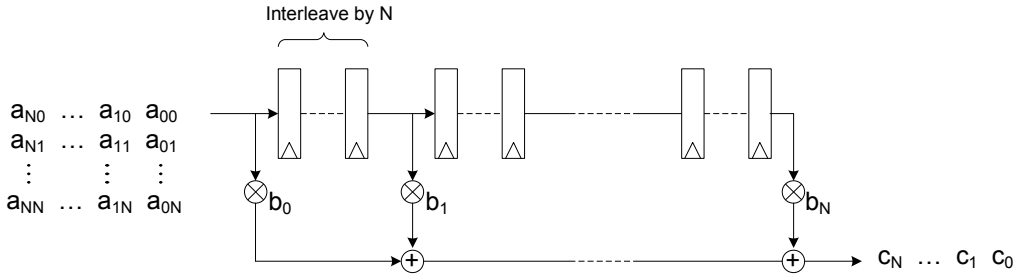
$$\begin{bmatrix} a_{00} & a_{01} & \dots & a_{0N} \\ a_{10} & a_{11} & \dots & a_{1N} \\ \vdots & \vdots & \ddots & \ddots \\ a_{N0} & a_{N1} & \dots & a_{NN} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} \tag{8.1}$$

A direct hardware implementation of this equation is impractical due to the large number of multipliers, especially for large matrices and vectors. Therefore, a designers goal would be to reduce the number of multipliers while maintaining adequate throughput and power dissipation measures. The design optimizations available through our framework need to be applicable to matrix-vector operations since they utilize multipliers and adders. The performance of a hardware design depends primarily on the number of processing blocks and the method in which they are connected together. Figure 8.2 illustrates an example of two different hardware implementations for a matrix-vector multiplication. Both designs require the same number of basic mathematical elements. The design in Figure 8.2(a) performs the computations in $N$ cycles for an $N \times N$ system, whereas the design in Figure 8.2(b) requires $2N$ cycles. However, the design in Figure 8.2(a) requires additional control logic and large multiplexers which contribute to an increase in area and power dissipation. The matrix-vector designs can be optimized using different forms of pipelining and $V_{DD}$ scaling to improve throughput and power dissipation in a similar manner that was applied to the FIR filters. Additionally, the design in Figure 8.2(b) is a direct application of data interleaving to make use of hardware resource sharing. Our design methods and architectural analyses techniques can be extended to the hardware implementation of the matrix-vector and other similar mathematical operations.

IIR filters are alternative filtering algorithms to FIR filters. The IIR filter requires a smaller order of coefficients than the FIR filter to implement a particular transfer function and would be a useful algorithm to include as part of the frameworks functions. The modular methodology used to construct the FIR filters is applicable to hardware structures for the IIR filters. Additionally, the types of design parameters and optimizations used for the FIR filter design space are suitable for the IIR filter designs. Therefore, a framework developer can utilize the modular design methodology to construct the basic processing modules required for a particular hardware design of the IIR filter. Figure 8.3 illustrates this modular design methodology applied to the TF IIR filter. The framework developer programs the




(a) Architecture 1




(b) Architecture 2

Figure 8.2: Various hardware implementations for matrix-vector multiplication

necessary connections to describe the functionality of the input, output and computational modules. The computational modules consist of the same three mathematical elements used for constructing the FIR filter designs, namely the adder, multiplier and delay unit. The processing modules are then connected to implement the IIR filter design.

The modular design technique is important since it allows the framework developer to modify the processing elements in order to describe different types of optimizations. This is accomplished using parameters that describe the placement of the mathematical elements in each of the processing modules. The processing modules promote design consistency

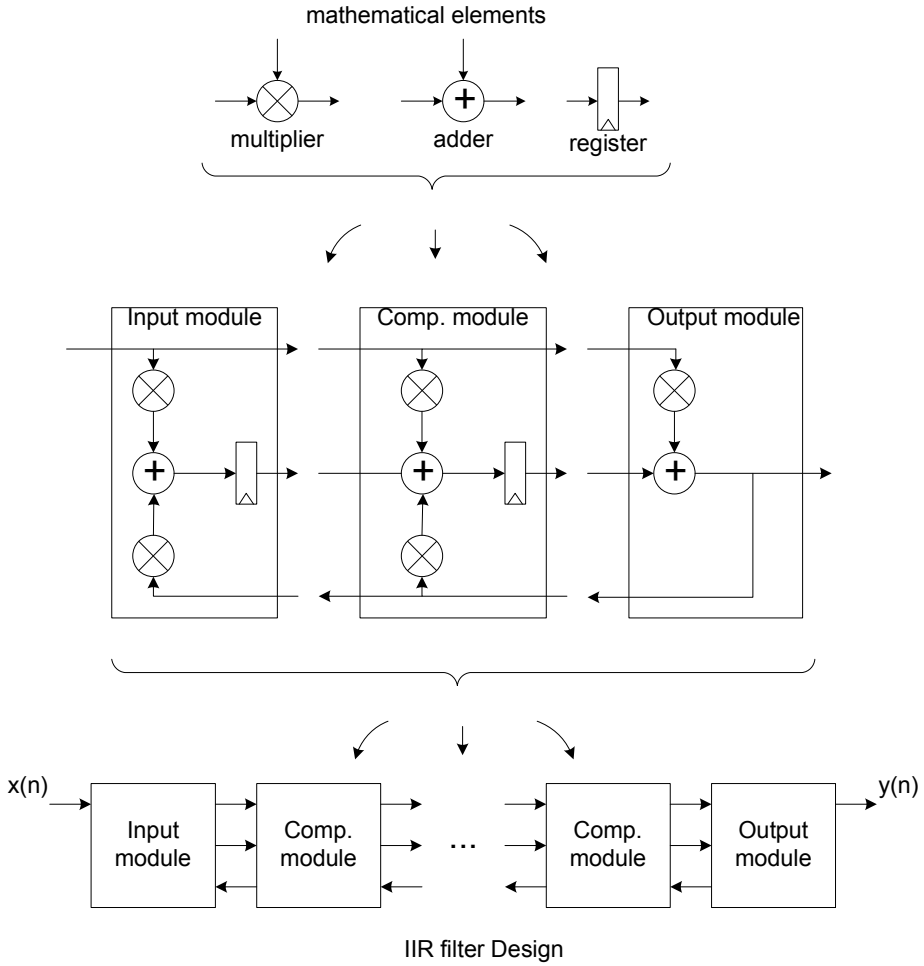



Figure 8.3: Modular design methodology applied to IIR filter structure

and allow the framework developer to easily verify the designs for correct functionality. Additionally, different IIR filter implementations can be included in the framework library by connecting the modules in various techniques. The modular technique is applicable to data-intensive designs with a regular flow of data. The regular flow of data permits the framework developer to construct a regular hardware design for the DSP algorithm. Therefore this design methodology can be applied to algorithms beyond filters, as long as the algorithm consists of a regular flow of data.

### 8.2.3 Educational Venues

The concepts we applied for designing and analyzing various hardware implementations were based on graduate courses offered by the ECE department at NC State University (NCSU). Algorithmic development for DSP applications is part of the *Digital Signal Processing* course offered at NCSU, whereas the hardware architectural considerations are discussed in the *DSP Architecture* course, and the concepts used to analyze the performance of the digital hardware designs at the circuit-level are explained in the *VLSI System Design* and *Digital Hardware Design* courses. These concepts can be conveyed to students in a manner that puts theoretical concepts to practical applications. For example, the various hardware designs used to implement a simple FIR filter algorithm can be analyzed and addressed by equating performance numbers to the specific filter structures. This allows a student to appreciate simple design techniques such as how cut-set retiming can be applied to pipeline a filter structure and how the throughput, area, power dissipation and hardware latency are affected. Figure 8.4 lists several graduate courses that students may benefit from when considering additional improvements in the development our PAF.

We currently use a MATLAB-based GUI to interface our design and analysis methodology to the designer. We make the assumption that the designer is knowledgeable of the specific types of design optimizations and synthesis parameters. However, students may not have acquired the necessary knowledge to understand how the framework design and synthesis parameters can effect the creation and performance of the hardware designs. One way to address this issue is to create a GUI-based interface with limited design options that confine a student to utilizing specific design options they are more familiar with. Additionally, a detailed users manual explaining the methodology employed by our framework for creating and synthesizing hardware designs would be useful for students wanting to broaden their knowledge in this area.

The creation of this framework was primarily the focus of one graduate student with the invaluable assistance of his colleagues and advisors. The quality of this work improves considerably for every student that contributes his or her efforts to the creation of this framework. Therefore, students would need to acquire the necessary tutelage to ensure the constant maintenance and expansion of this work. We can address this by assigning tasks to students in order to contribute their ideas and efforts to portions of the framework through independent course projects which can then be formulated into possible

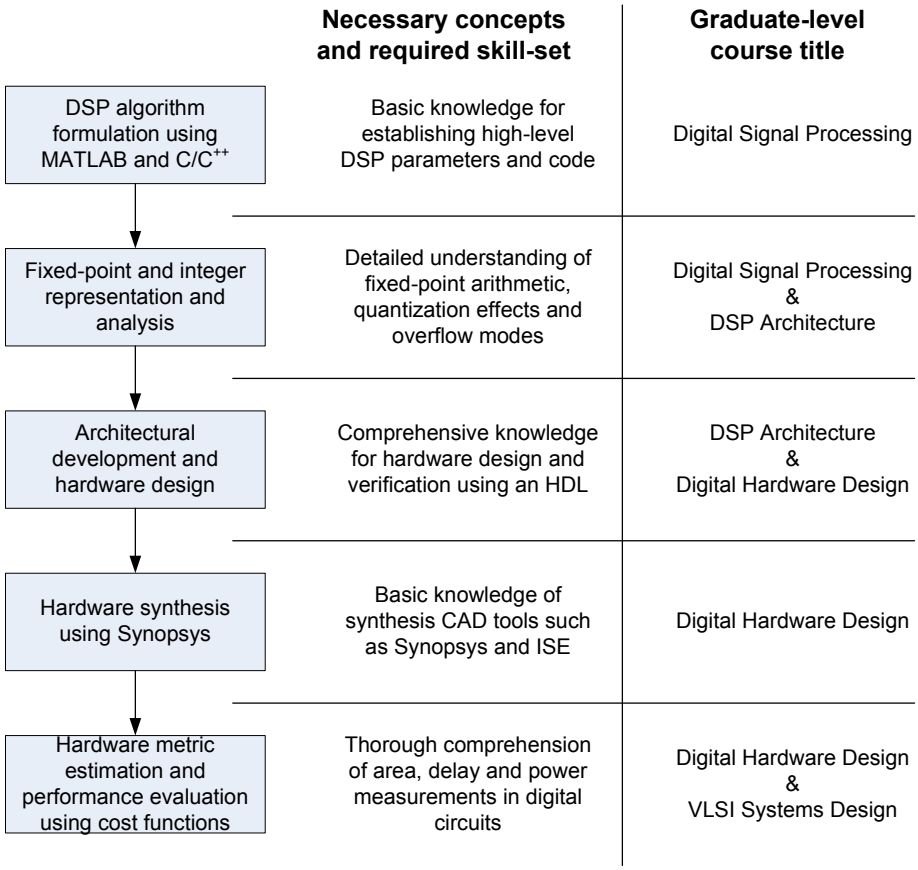



Figure 8.4: Graduate courses utilized in the development of the PAF

undergraduate and graduate level research topics.

## 8.3 Chapter Summary

The extensions of this work can improve the value of our PAF to the point where it may be utilized in graduate-level engineering courses and could possibly be integrated into existing EDA tools for both ASIC and FPGA designs. One such tool presented and analyzed in this work was the efforts of Prith Banerjee and his students [21] [37] who went onto to develop their research ideas and projects into a marketable tool that is currently utilized by Xilinx for their FPGA platforms - "AccelDSP$^{TM}$ Synthesis Tool" [40]. Our

goal for developing the PAF was not to compete with such tool providers, but to contribute our efforts to the EDA community in order to assist designers in moving forward with their innovative designs and creations. As the electronic design process continues to advance, so do the EDA tools and therefore, it is important that we establish a clear and concise manner in which our methodology can also mature and grow accordingly.

# Bibliography

[1] W. Shockley, J. Bardeen, and W. Brattain, "Electronic theory of the transistor," *Science*, pp. 678–679, 1948.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan. 1998.

[3] OSCI, "Systemc version 2.0 user's guide." Website, Jan. 2002. `http://www.systemc.org`.

[4] D. Smit and P. Franzon, *Verilog styles for synthesis of digital systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[5] "Synopsys design compiler." `http://www.synopsys.com/products/logic/design_compiler.html`.

[6] P. Foldesy, "Trends in design of massively parallel coprocessors implemented in digital ASICs," *2004. Proceedings. 2004 IEEE International Joint Conference on Neural Networks*, vol. 4, pp. 3131–3135, July 25–29, 2004.

[7] W. R. Davis, "A hierarchical, automated design flow for low-power, high-throughput digital signal processing ics," *Ph.D. Dissertation, University of California, Berkeley, Electronics Research Laboratory*, Spring 2004.

[8] ITRS, "The international technology roadmap for semiconductors." Website. `http://public.itrs.net`.

[9] L.-K. Ting, R. Woods, and C. F. N. Cowan, "Virtex FPGA implementation of a pipelined adaptive LMS predictor for electronic support measures receivers," *IEEE*

*Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, pp. 86–95, Jan. 2005.

[10] P. Pirsch, *Architectures for Digital Signal Processing*. New York, NY: John Wiley and Sons, 1998.

[11] D. L. Jones, "Learning characteristics of transpose-form LMS adaptive filters," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 39, pp. 745–749, Oct. 1992.

[12] "The mathworks, matlab and simulink for technical computing." `http://www.mathworks.com`.

[13] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pp. 19–24, Oct. 1–3, 2003.

[14] A. Baganne, I. Bennour, M. Elmarzougui, R. Gaiech, and E. Martin, "A multi-level design flow for incorporating IP cores: case study of 1D wavelet IP integration," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 250–255, 2003.

[15] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Boston, MA: Kluwer Academic Publishers, 2002.

[16] "Describing synthesizable RTL in systemc." `http://www.synopsys.com`, Jan. 2002.

[17] T. Gemmeke, M. Gansen, H. J. Stockmanns, and T. G. Noll, "Design optimization of low-power high-performance DSP building blocks," *IEEE Journal of Solid-State Circuits*, vol. 39, pp. 1131–1139, July 2004.

[18] "Fixed-point toolbox." `http://www.mathworks.com/access/helpdesk/help/toolbox/fixedpoint/`.

[19] S. Roy and P. Banerjee, "An algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design," in *Design Automation Conference, 2004. Proceedings. 41st*, pp. 484–487, 2004.

[20] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, pp. 886–896, July 2005.

[21] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. Uribe, "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 312–324, Mar. 2004.

[22] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From VHDL register transfer level to SystemC transaction level modeling: a comparative case study," in *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pp. 355–360, Sept. 2003.

[23] "Ieee p1364-1995, IEEE standard verilog hardware desciption language, VSG - IEEE standards group for verilog, 1995."

[24] M. Becker, "Faster Verilog simulations using a cycle based programming methodology," in *Verilog HDL Conference, 1996. Proceedings., 1996 IEEE International*, (Santa Clara, CA), pp. 24–31, Feb. 26–28, 1996.

[25] W. R. Davis, N. Zhang, K. Camera, D. Markovic, T. Smilkstein, M. J. Ammer, E. Yeo, S. Augsburger, B. Nikolic, and R. W. Brodersen, "A design environment for high-throughput low-power dedicated signal processing systems," in *Solid-State Circuits, IEEE Journal of*, vol. 37, (San Diego, CA), pp. 420–431, Mar. 2002.

[26] W. Davis, "Getting high-performance silicon from system-level design," in *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 238–243, Feb. 2003.

[27] "Synopsys designware intellectual property." `http://www.synopsys.com/products/designware/designware.html`.

[28] "Designware building block IP, high-speed digital FIR filter." `http://www.synopsys.com/`.

[29] J. G. Proakis, *Digital communications*. New York: McGraw-Hill, 1995.

[30] J. G. Proakis and D. G. Manolakis, *Digital signal processing : principles, algorithms, and applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[31] "Synopsys module compiler 1999, synopsys corporation." `http://www.synopsys.com`.

[32] A. P. Chandrakasan, M. Potkonjak, J. Rabaey, and R. W. Brodersen, "HYPER-LP: a system for power minimization using architectural transformations," in *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, (Santa Clara, CA), pp. 300–303, Nov. 1992.

[33] K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. New York, NY: John Wiley and Sons, 1999.

[34] W. R. Davis, A. M. Sule, and H. Hua, "Multi-parameter power minimization of synthesized datapaths," in *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pp. 151–157, Feb. 2004.

[35] R. A. Hawley, B. C. Wong, T.-J. Lin, J. Laskowski, and H. Samueli, "Design techniques for silicon compiler implementations of high-speed FIR digital filters," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 656–667, May 1996.

[36] R. Hourani, W. Alexander, , and T. Raithatha, "A hardware performance analysis framework for architectural exploration of DSP systems," in *Global Signal Processing Expo GSPX*, (San Jose, CA), Oct. 2005.

[37] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee, "FPGA hardware synthesis from MATLAB," in *VLSI Design, 2001. Fourteenth International Conference on*, (Bangalore), pp. 299–304, Jan. 2001.

[38] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Automated synthesis of pipelined designs on FPGAs for signal and image processing applications described in MATLAB," in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, (Yokohama), pp. 645–648, Jan./Feb. 2001.

[39] P. Banerjee, "An overview of a compiler for mapping MATLAB programs onto FPGAs," in *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pp. 477–482, Jan. 2003.

[40] "AccelDSP synthesis tool." `http://www.xilinx.com/ise/dsp_design_prod/acceldsp/index.htm`.

[41] "System generator for DSP." http://www.xilinx.com/ise/optional_prod/system_generator.htm.

[42] "Catapult synthesis." http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthe%sis/index.cfm.

[43] S. McCloud, "Catapult C synthesis-based design flow: Speeding implementation and increasing flexibility." Mentor Graphic, White Paper 2004.

[44] "Mentor takes out timing to provide hardware from C," *Electronics Systems and Software*, vol. 2, pp. 46–46, Aug./Sept. 2004.

[45] "Xilinx LogicCORE FIR compiler." http://www.xilinx.com, 2004.

[46] "OpenSoCDesign." http://www.opensocdesign.com/.

[47] R. Hourani, R. Jenkal, W. R. Davis, and W. Alexander, "Automated Architectural Exploration for Signal Processing Algorithms," in *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, (Banff, Alta.), pp. 274–279, Oct. 2006.

[48] "Methodologies for user-friendly system-on-a-chip experimentation." http://www.ece.ncsu.edu/muse/sshaft/.

[49] Q. Yue, L. Zhancai, and W. Qin, "Low-power FIR filter based on standard cell," in *ASIC, 2005. ASICON 2005. 6th International Conference On*, vol. 1, pp. 208–211, Oct. 2005.

[50] "IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual - section 7.10.9," in *Design Automation Standards Committee of the IEEE Computer Society*, 2006.

[51] D. Markovic, B. Nikolic, and R. W. Brodersen, "Power and area efficient VLSI architectures for communication signal processing," in *Communications, 2006 IEEE International Conference on*, vol. 7, (Istanbul), pp. 3223–3228, June 2006.

[52] S. Velusamy, W. Huang, J. Lach, M. Stan, and K. Skadron, "Monitoring temperature in FPGA based SoCs," in *Computer Design: VLSI in Computers and Processors, 2005.*

*ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pp. 634–637, Oct. 2005.

[53] M. R. Stan, "Low-power CMOS with subvolt supply voltages," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 394–400, Apr. 2001.

[54] M. Pedram and Q. Wu, "Battery-powered digital CMOS design," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, (Munich), pp. 72–76, Mar. 9–12, 1999.

[55] S. Gailhard, N. Julien, J. P. Diguet, and E. Martin, "How to transform an architectural synthesis tool for low power VLSI designs," in *VLSI, 1998. Proceedings of the 8th Great Lakes Symposium on*, (Lafayette, LA), pp. 426–431, Feb. 19–21, 1998.

[56] H. Pournara, V. Kalenteridis, I. Pappas, N. Vassiliadis, S. Nikolaidis, S. Siskos, and D. J. Soudris, "Energy efficient fine-grain reconfigurable hardware," in *Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean*, vol. 1, pp. 209–212, May 12–15, 2004.

[57] S. Y. Kung, *VLSI Array Processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[58] "The OSU standard cell library." `http://www.ece.iit.edu/~jgrad/osucells/`.

[59] T. Mathworks, "Signal processing toolbox, introduction to FDATool." Website, 2007. `www.mathworks.com/products/signal/`.

[60] S. M. Kuo and W.-S. Gan, *Digital Signal Processors : Architectures, Implementations, and Applications*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2005.

[61] C. Tsai, "Floating-point roundoff noises of first- and second-order sections in parallel form digital filters," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 44, pp. 774–779, Sept. 1997.

[62] P. P. Vaidyanathan, *Multirate systems and filter banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[63] A. Benkrid, K. Benkrid, and D. Crookes, "Design and implementation of a novel architecture for symmetric FIR filters with boundary handling on Xilinx Virtex FPGAs," in *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pp. 356–359, Dec. 16–18, 2002.

[64] D. Schlichtharle, *Digital filters : basics and design*. Berlin, Germany: Springer - Verlag Berlin Heidelberg, 2000.

[65] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," vol. 18, pp. 36–58, Sept. 2001.

[66] M. Dabbagh and W. Alexander, "Pipelining of digital filter structures for VLSI implementation," in *Southeastcon '89. Proceedings. 'Energy and Information Technologies in the Southeast'., IEEE*, (Columbia, SC), pp. 1185–1189, Apr. 1989.

[67] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473–484, Apr. 1992.

[68] H.-J. Ki, W.-H. Paik, J.-S. Yoo, and S.-W. Kim, "A high speed, low power 8-tap digital FIR filter for PRML disk-drive read channels," in *Solid-State Circuits Conference, 1997. ESSCIRC '97. Proceedings of the 23rd European*, pp. 312–315, Sept. 1997.

[69] R. B. Staszewski, K. Muhammad, and P. Balsara, "A 550-MSample/s 8-tap FIR digital filter for magnetic recording read channels," *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 1205–1210, Aug. 2000.

[70] F. Y. Edgeworth, *Mathematical Psychics: An Essay on the Application of Mathematics to the Moral Sciences*. C. Kegan Paul and Co., 1881.

[71] V. Pareto, *Cours d'Economie Politique*, vol. 2. F. Rouge and Cie., Lausanne, Switzerland, 1897.

[72] H. Yu, B. W. Kim, Y. G. Cho, J. D. Cho, J. W. Kim, J. K. Lee, H. C. Park, and K. W. Lee, "Area-efficient and reusable VLSI architecture of decision feedback equalizer for QAM modem," in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, (Yokohama), pp. 404–407, Jan./Feb. 2001.

[73] K. H. Chen and T. D. Chiueh, "A low-power digit-based reconfigurable FIR filter," vol. 53, pp. 617–621, Aug. 2006.

[74] N. R. Shanbhag and K. K. Parhi, "Pipelined adaptive DFE architectures using relaxed look-ahead," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 43, pp. 1368–1385, June 1995.

[75] S. Haykin, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice Hall, 3rd ed., 1996.

[76] R. Perry, D. R. Bull, and A. Nix, "Pipelined DFE architectures using delayed coefficient adaptation," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 45, pp. 868–873, July 1998.

[77] K. Muhammad, R. B. Staszewski, and P. T. Balsara, "Low power techniques and design tradeoffs in adaptive FIR filtering for PRML read channels," in *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pp. 262–267, 2000.

[78] T. Acharya and P.-S. Tsai, *JPEG2000 standard for image compression : concepts, algorithms and VLSI architectures*. Hoboken, NJ: Wiley-Interscience, 2005.

[79] I. Daubechies, "Wavelets: a tool for time-frequency analysis," in *Multidimensional Signal Processing Workshop, 1989., Sixth*, (Pacific Grove, CA), Sept. 6–8, 1989.

[80] F. Marino, D. Guevorkian, and J. T. Astola, "Highly efficient high-speed/low-power architectures for the 1-D discrete wavelet transform," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 47, pp. 1492–1502, Dec. 2000.

[81] A. B. Premkumar and A. S. Madhukumar, "An efficient VLSI architecture for the computation of 1-D discrete wavelet transform," in *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, pp. 1180–1184, Sept. 9–12, 1997.

[82] T. C. Denk and K. K. Parhi, "Systolic VLSI architectures for 1-D discrete wavelet transforms," in *Signals, Systems & Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on*, vol. 2, (Pacific Grove, CA), pp. 1220–1224, Nov. 1–4, 1998.

[83] C. Zhang, C. Wang, and M. O. Ahmad, "A VLSI architecture for a high-speed computation of the 1D discrete wavelet transform," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pp. 1461–1464, May 23–26, 2005.

[84] A. M. Rincon, G. Cherichetti, J. A. Monzel, D. R. Stauffer, and M. T. Trick, "Core design and system-on-a-chip integration," *IEEE Design & Test of Computers*, vol. 14, pp. 26–35, Oct./Dec. 1997.

[85] B. Buyukkurt, Z. Guo, and W. A. Najjar, "Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to vhdl compiler for fpgas.," in *ARC* (K. Bertels, J. M. P. Cardoso, and S. Vassiliadis, eds.), vol. 3985 of *Lecture Notes in Computer Science*, pp. 401–412, Springer, 2006.