Modeling and Verification of DSP Designs in HOL

Behzad Akbarpour

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

April 2005

Your file Votre référence
ISBN: 0-494-04055-6
Our file Notre référence
ISBN: 0-494-04055-6

ABSTRACT

Modeling and Verification of DSP Designs in HOL

Behzad Akbarpour, Ph.D.
Concordia University, 2005

In this thesis we propose a framework for the incorporation of formal methods in the design flow of DSP (Digital Signal Processing) systems in a rigorous way. In the proposed approach we model and verify DSP descriptions at different abstraction levels using higher-order logic based on the HOL (Higher Order Logic) theorem prover. This framework enables the formal verification of DSP designs which in the past could only be done partially using conventional simulation techniques. To this end, we provide a shallow embedding of DSP descriptions in HOL at the floating-point, fixed-point, behavioral, RTL (Register Transfer Level), and netlist gate levels. We make use of existing formalization of floating-point theory in HOL and introduce a parallel one for fixed-point arithmetic. The high ability of abstraction in HOL allows a seamless hierarchical verification encompassing the whole DSP design path, starting from top level floating- and fixed-point algorithmic descriptions down to RTL, and gate level implementations. We illustrate the new verification framework using different case studies such as digital filters and FFT (Fast Fourier Transform) algorithms.

To My Family

ACKNOWLEDGEMENTS

I have been very fortunate to have Dr. Sofiène Tahar as my supervisor. I am deeply grateful for his strong support and encouragement through out my Ph.D studies. His expertise and competent advice have shaped the character of my research.

Throughout my study in Concordia many people have encouraged and helped me through many obstacles. I have enjoyed studying and working with my colleagues in the HVG group in Concordia University, wishing to thank all of them for their support and the nice time we have spent together.

I would like to express my gratitute and thanks to Dr. Harrison from Intel for accepting to be my external examiner. I could not have a better expert than him world wide.

Many thanks also to the HOL community experts who helped me out throughout the thesis, in particular Dr. Hurd, Dr. Slind, Dr. Norrish, ... just to name a few.

I also wish to express my gratitude to the examination committee members, Dr. Hassan, Dr. Ait Mohamed, and Dr. Lynch, for reviewing my thesis and giving me invaluable feedback.

I would like to reserve my deepest thanks to my family for their perpetual love and encouragement. I can never thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ACL2	A Computational Logic Applicative Common Lisp
BDD	Binary Decision Diagram
CAD	Computer Aided Design
DFT	Discrete Fourier Transform
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FRIDGE	Fixed-point pRogrammIng DesiGn Environment
HDL	Hardware Description Language
HOL	Higher Order Logic
IEEE	Inistitute of Electrical and Electronics Engineers
IFT	Inverse Fourier Transform
LCF	Logic for Computable Functions
LSB	Least Significant Bit
ML	Meta Language
MSB	Most Significant Bit
OFDM	Orthogonal Frequency Division Multiplexing
PVS	Prototype Verification System
RTL	Register Transfer Level
SMV	Symbolic Model Verifier
SPW	Signal Processing WorkSystem
STE	Symbolic Trajectory Evaluation
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 General Objective: System Verification

Today, hardware and software systems are widely used in applications where failure is unacceptable: electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments, and other examples too numerous to list. We frequently read of incidents where some failure is caused by an error in a hardware or software system. A recent example of such a failure is the Ariane 5 rocket, which exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident found that it was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch, an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the computer to fail. The same error also caused the backup computer to fail. As a result incorrect altitude data was transmitted to the on-board computer, which caused the destruction of the rocket. The team investigating the failure suggested that several measures be taken in order to prevent similar incidents in the future, including the verification of the Ariane 5 software. Clearly, the need for reliable hardware and software systems is critical. As the involvement of such systems in our lives increases, so too does the burden for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. We are very much dependent on such systems for continuous operation; in fact, in some cases, devices are less safe when they are shut down. Even when failure is not life-threatening, the consequences of having to replace critical code or circuitry can be economically devastating. Because of the success of the Internet and embedded systems in automobiles, airplanes, and other safety critical systems, we are likely to become even more dependent on the proper functioning of computing devices in the future. In fact, the pace of change will likely accelerate in coming years. Because of this rapid growth in technology, it will become even more important to develop methods that increase our confidence in the correctness of such systems.

1.2 Specific Objectives: DSP Verification

Digital system design is characterized by ever increasing system complexity that has to be implemented within reduced time, resulting in minimum costs and short time-to-market. These characteristics call for a seamless design flow that allows to perform the design steps on the highest suitable level of abstraction. For most digital signal processing systems, the design has to result in a fixed-point implementation. This is due to the fact that these systems are sensitive to power consumption, chip size and price per device. Fixed point realizations outperform floating-point realizations by far with regard to these criteria. A typical DSP design flow is depicted in Figure 1.1 [45]. An algorithm design starts from a floating-point description. This allows to ignore the effects of finite wordlengths and fixed exponents and to abstract from all implementation details. Additionally, the use of floating-point models offers a maximum degree of reusability. On the fixed-point level, all operands are assigned a fixed word length and a fixed exponent, while the control structure and the operations of the floating point program remain unchanged. This description is used to analyze whether the fixed-point model fulfills the algorithmic system requirements. The transformation to the fixed-point is quite tedious and error-prone. On the implementation level, the fixed-point model of the algorithm has to be transferred into the best suited target description, either using a hardware description language (HDL) or a programming language. These requirements have been the motivation for the development of CAD tools for DSP design. Examples of such tools are SPW (Cadence) [12], CoCentric (Synopsys) [15], Matlab-Simulink (Mathworks) [49], and FRIDGE (Aachen UT) [45].

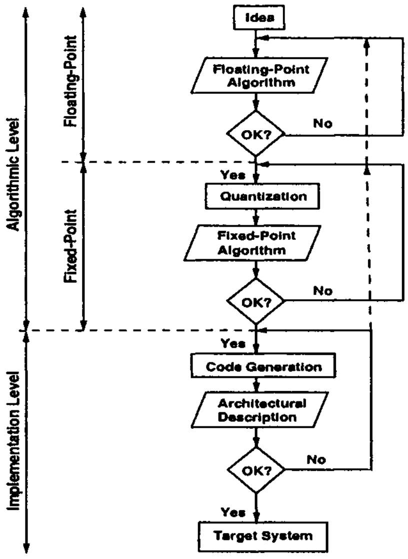


Figure 1.1: DSP design flow

Usually the conformance of the fixed-point implementation with respect to the descriptions in floating-point algorithmic, and RT and gate levels is verified by simulation techniques which cannot cover all design errors, especially for large systems. On the other hand, adopting formal verification [43] in system design generally means using methods of mathematical proof rather than simulation to ensure the quality of the design, to improve the robustness of a design and to speed up the development. The overall aim for the proposed research is to model the DSP descriptions at different abstraction levels based on the shallow embedding approach to enable their formal verification in the HOL theorem proving environment.

1.3 State of the Art: Simulation

Today, the usual validation method to discover the errors in the design flow of DSP systems is still simulation. In this method, a simulation run must be performed in each level of abstraction such as floating-point, fixed-point, RT and gate level to check if the required characteristics are preserved. With simulation, input signals are injected at certain points in the system and the resulting signals at other points are observed. These methods can be a cost-efficient way to find errors. However, in order to get full confidence in the design we would have to perform a complete simulation which covers all possible input combinations. Exhaustive simulation of even moderately-sized circuits is impossible, and partial simulation offers only partial assurance of correctness. This is an especially serious problem in safety-critical applications, where failure due to design errors may cause loss of life or extensive damage. In these applications, functional errors in circuit designs cannot be tolerated. But even where safety is not the primary consideration, there may be important economic reasons for doing everything possible to eliminate design errors, and to eliminate them early in the design process. A flawed design may mean costly and time-consuming refabrication, and mass-produced devices may have to be recalled and replaced.

1.4 Proposed Solution: Formal Verification

A solution to these problems is one of the goals of formal methods [52] for verification of the correctness of hardware designs, sometimes just called hardware verification. With this approach, the behaviour of hardware devices is described mathematically, and formal proof is used to verify that they meet rigorous specifications of intended behaviour.

However, formal verification is not the golden rule in circuit testing because of some limitations. A correctness proof cannot guarantee that the real device will never malfunction; the design model of the device may be proved correct, but the hardware actually built can still behave in a way unintended by the designer (this is the case for simulation too). Wrong specifications can play a major role in this, because it has been verified that the system will function as specified, but it has not been verified that it will work correctly. Defects in physical fabrication can cause this problem too. In formal verification a model of the design is verified, not the real physical implementation. Therefore, a fault in the modeling process can give false negatives (errors in the design which do not exist). Although sometimes, the fault covers some real errors. Because of these limitations we can consider simulation and hardware verification as complementary techniques, the methods have to play together.

Formal verification methods can be categorized in two main groups [67], theorem proving and model checking. Theorem proving refers to the use of axioms and proof rules to prove the correctness of the systems. In this method, one expresses the system model and specifications in a suitable logic, and constructs a proof in the logic that the system model implies the specifications. The powerful mathematical techniques such as induction and abstraction are the strengths of theorem proving and make it a very flexible and powerful verification technique. It makes it possible to construct a model at almost every abstraction level and proves properties on all classes of systems. However, it is a time consuming process which can involve generating and proving literally hundreds of lemmas in painstaking detail. Model checking, on the other hand, is more limited in scope, but is fast and fully automated. The system model is in essence a finite state machine, and specifications are written in temporal logic. These logics are limited with respect to the very powerful logics handled by general theorem provers, but are quite simple and concise, and can express a wide variety of useful properties.

1.5 Proposed DSP Verification Framework

In this thesis we propose a methodology for applying formal methods to the design flow of DSP systems in a rigorous way. The corresponding commutating diagram is shown in Figure 1.2. Thereafter, we model the ideal real specification of the DSP algorithms and the corresponding floating- and fixed-point representations as well as the RT and gate level implementations as predicates in higher-order logic. The overall methodology for the formal specification and verification of DSP algorithms will be based on the idea of shallow embedding of languages [4] using the HOL theorem proving environment [23]. In the proposed approach, we first focus on the transition from real to floating- and fixed-point levels. For this, we make use of existing theories in HOL on the construction of real [27] and complex [32] numbers, the formalization of IEEE-754 standard based floating-point arithmetic [28, 29], and the formalization of fixed-point arithmetic. We use valuation functions to find the real values of the floating- and fixed-point DSP outputs and define the error as the difference between these values and the corresponding output of the ideal real specification. Then we establish fundamental lemmas on the error analysis of floating- and fixed-point roundings and arithmetic operations against their abstract mathematical counterparts. Finally, based on these lemmas, we derive expressions for the accumulation of roundoff error in floating- and fixed-point DSP algorithms using recursive definitions and initial conditions. While theoretical work on computing the errors due to finite precision effects in the realization of DSP algorithms with floating- and fixed-point arithmetics has been extensively studied since the late sixties [41], this thesis contains the first formalization and proof of this analysis using a mechanical theorem prover, here HOL. The formal results are found to be in good agreement with the theoretical ones.

After handling the transition from real to floating- and fixed-point levels, we turn to the HDL representation. At this point, we use well known techniques to model the DSP design at the RTL level within the HOL environment. The last step is to verify this level using a classical hierarchical proof approach in HOL [52]. In this way, we hierarchically prove that the DSP RTL implementation implies the high level fixed-point algorithmic specification, which has already been related to the floating-point description and the ideal real specification through the error analysis. The verification can be extended, following similar manner, down to gate level netlist either in HOL or using other commercial verification tools as depicted in Figure 1.2. This analysis is not covered in this thesis.

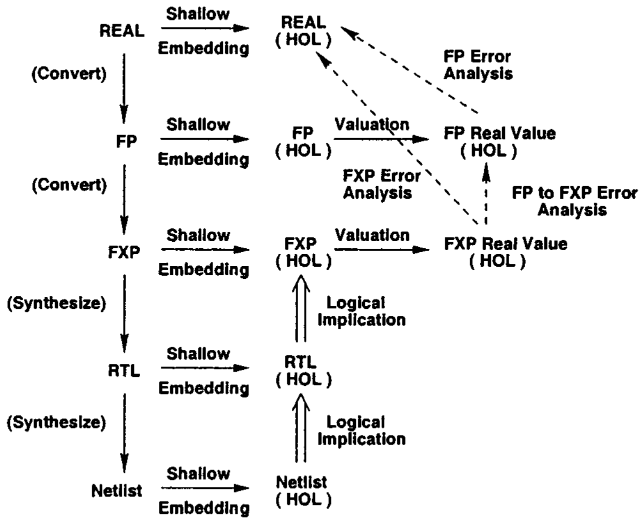


Figure 1.2: Propsed DSP specification and verification approach

The process of specifying a hardware description language in higher-order logic is commonly known as semantic embedding. There are two main approaches [4]: deep embedding and shallow embedding. In deep embedding, the abstract syntax of a design description is represented by terms, which are then interpreted by semantic functions defined in the logic that assign meaning to the design. With this method, it is possible to reason about classes of designs, since one can quantify over the syntactic structures. However, setting up HOL types of abstract syntax and semantic functions can be very tedious. In a shallow embedding on the other hand, the design is modeled directly by a formal specification of its functional behavior. This eliminates the effort of defining abstract syntax and semantic functions, but it also limits the proofs to functional properties. In this thesis, since our main concern is to check the correctness of the designs based on their functionality, we propose the shallow embedding for DSP descriptions: translate the intended meaning of DSP block desings as described in its documentation into HOL and then complete the formal proof in the HOL theorem prover.

In this thesis, we demonstrate how the methodology presented in this section can be used for the verification of a parametric L order digital filter and the fast Fourier transform (FFT) algorithms implemented in different canonical forms of realization. Similar discussion can be applied to other types of filtering and signal analysis algorithms.

When a linear recursive difference equation digital filter is realized with floating- and fixed-point arithmetic, on a computer or with special-purpose hardware, errors and constraints due to finite word length are unavoidable. The main categories of finite precision effects are errors due to roundoff in the arithmetic operations, errors due to quantization of input, and effects of coefficient inaccuracies. These error problems have already been studied extensively [47]. In this thesis, as the first case study we show how this error analysis can be mechanically performed using HOL theorem prover. We have used our verification methodology to derive expressions for the accumulation of roundoff error in a parametric L order digital filter, for each of the three canonical forms of realization: direct, parallel, and cascade.

The fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform with a substantial time saving over conventional methods. FFT algorithms are based on the fundamental principle of decomposing the computation of the discrete Fourier transform of a sequence of length N into successively smaller discrete Fourier transforms. The manner in which this principle is implemented leads to a variety of different algorithms, all with comparable improvements in computational speed. Two basic classes of FFT algorithms are the decimation-in-time and decimation-in-frequency. As the second case studty in this project, we consider the formal verification of the decimation-in-time and decimation-in-frequency FFT algorithms. We used our methodology to derive expressions for the accumulation of roundoff error in floating- and fixed-point FFT algorithms by recursive definitions and initial conditions, considering the effects of input quantization and inaccuracy in the coefficients. Based on the extensively studied theoretical work on computing the errors due to finite precision effects in the realization of FFT algorithms with floating- and fixed-point arithmetics [41], we perform a similar analysis using the HOL theorem proving environment. The formal results are found to be in good agreement with the theoretical ones.

1.6 Related Work

1.6.1 Error Analysis in Formal Verification

Previous work on the error analysis in formal verification was done by Harrison [29] who verified the floating-point algorithms such as the exponential function against their abstract mathematical counterparts using the HOL Light theorem prover. As the main theorem, he proved that the floating-point exponential function has a correct overflow behavior, and in the absence of overflow the error in the result is bounded to a certain amount. He also reported on an error in the hand proof mostly related to forgetting some special cases in the analysis. This error analysis is very similar to the type of analysis performed for DSP algorithms. The major difference, however, is the use of statistical methods and mean square error analysis for DSP algorithms which is not covered in the error analysis of the mathematical functions used by Harrison. In this method, the error quantities are treated as independent random variables uniformly distributed over a specific interval depending on the type of arithmetic and the rounding mode. Then the error analysis is performed to derive expressions for the variance and mean square error. To perform such an analysis in HOL, we need to develop a mechanized theory on the properties of random variables and random processes. This type of analysis is not addressed in this thesis and is a part of our future work. Huhn *et al.* [34] proposed a hybrid formal verification method combining different state-of-the-art techniques to guide the complete design flow of imprecisely working arithmetic circuits starting at the algorithmic down to the register transfer level. The usefulness of the method is illustrated with the example of the discrete cosine transform algorithms. In particular, the authors have shown the use of computer algebra systems like Mathematica or Maple at the algorithmic level to reason about real numbers and to determine certain error bounds for the results of numerical operations. In contrast to [34], we propose an error analysis for digital filters using the HOL theorem prover. Although the computer algebraic systems such as Maple or Mathematica are much more popular and have many powerful decision procedures and heuristics, theorem provers are more expressive, more precise, and more reliable [33]. One option is to combine the rigour of the theorem provers with the power of computer algebraic systems as proposed in [33].

1.6.2 Floating-Point Formal Verification

There exist several related work in the open literature on the formalization and verification of IEEE standard based floating-point arithmetic. For instance, Barrett [2] specified parts of the IEEE-754 standard in Z, and Miner [54] formalized the IEEE-854 floating-point standard in PVS. The latter defined the relation between floating-point numbers and real numbers, rounding, and some arithmetic operations on both finite and infinite operands. He used this formalization to verify abstract mathematical descriptions of the main operations and their relation to the corresponding floating-point implementations. His work was one of the earliest on the formalization of floating-point standards using theorem proving. His formal specification was then used by Miner and Leathrum [53] to verify in PVS a general class of IEEE compliant subtractive division algorithms.

Carreno [11] formalized the same IEEE-854 standard in HOL. He interpreted the lexical descriptions of the standard into mathematical conditional descriptions and organized them in tables, which were then formalized in HOL. He discussed different standard aspects such as precisions, exceptions and traps, and many other arithmetic operations such as addition, multiplication, and square-root of floating-point numbers.

Harrison [27] constructed the real numbers in HOL. He then developed in HOL a generic floating-point library [28] to define the most fundamental terms of the IEEE-754 standard and to prove the corresponding correctness analysis lemmas. He used this library to formalize and verify floating-point algorithms of complex arithmetic operations such as the square root, the exponential function [29], and the transcendental functions [30] against their abstract mathematical counterparts. He also used the floating-point library for the verification of the class of division algorithms used in the Intel IA-64 architecture [31].

Moore *et al.* [56] have verified the AMD-K5 floating-point division algorithm using the ACL2 theorem prover. Also, Russinoff [64] has developed a floating-point library for the ACL2 prover and applied it successfully to verify the floating-point multiplication, division, and square root algorithms of the AMD-K5 and AMD Athlon processors.

Aagaard and Seger [1] combined BDD-based model-checking and theorem proving techniques in the Voss hardware verification system to verify the IEEE compliance of the gate-level implementation of a floating-point multiplier. O'Leary *et al.* [62] reported on the specification and verification of the Intel Pentium® Pro processor's floating-point execution unit at the gate level using a combination of model-checking and theorem proving. Leeser *et al.* [46] verified a subtractive radix-2 square root algorithm and its hardware implementation using the higher-order logic theorem proving system Nuprl. Chen and Bryant [14] used word-level SMV to verify a floating-point adder. Cornea-Hasegan [17] used iterative approaches and mathematical proofs to verify the correctness of the IEEE floating-point square root, divide, and remainder algorithms.

More recently, Daumas *et al.* [19] have presented a generic library for reasoning about floating-point numbers within the Coq system. This library was then used in the verification of IEEE-compliant floating-point arithmetic algorithms [8] and hardware units [7]. Berg *et al.* [3] have formally verified a theory of IEEE rounding presented in [57] using the theorem prover PVS. They have used a formal definition of rounding based on Miner's formalization of the standard [54]. This theory was then used to prove the correctness of a fully IEEE compliant floating-point unit used in the VAMP processor [6]. Sawada and Gamboa [65] formally verified the correctness of a floating-point square root algorithm used in the IBM Power4^{TM} processor. The verification was carried out with the ACL2(r) theorem prover which is an extension of the ACL2 theorem prover that performs reasoning on real numbers using non-standard analysis. The proof required the analysis of the approximation error on Chebyshev series by proving Taylor's theorem. Kaivola *et al.* [38, 40, 42] presented the formal verification of the floating-point multiplication, division, and square root units of the Intel IA-32 Pentium® 4 microprocessor. The verification was carried out using the Forte verification framework, a combined model-checking and theorem-proving system built on top of the Voss system. Model checking was done via symbolic trajectory evaluation (STE), and theorem proving was done in the ThmTac proof tool.

While all of the above work are concerned with floating-point representation and arithmetic, there is no report in the open literature on any machine-checked formalization of properties of fixed-point arithmetic. Therefore, the formalization presented in this thesis is to our best knowledge, the first of its kind. Our formalization of the fixed-point arithmetic has been inspired mostly by the work done by Harrison [29] and Carreno [11] on floating-point. Harrison's work was more oriented towards verification purposes. Indeed, we used an analogous set of lemmas to his work, to check the validity of operation results and to carry out the error analysis of the quantized fixed-point result. For exception handling which is not covered by Harrison [29], we followed Carreno [11] who formalized floating-point exceptions and their handling in more details.

1.6.3 Error Analysis of Digital Filters

Work on the analysis of the errors due to the finite precision effects in the realization of the digital filters has always existed since their early days, however, using theoretical paper-and-pencil proofs and simulation techniques. For digital filters realized with the fixed-point arithmetic, error problems have been studied extensively. For instance, Knowles and Edwards [44] proposed a method for analysis of the finite word length effects in fixed-point digital filters. Gold and Radar [25] carried out a detailed analysis of the roundoff error for the first-order and second-order fixed-point filters. Jackson [37] analyzed the roundoff noise for the cascade and parallel realizations of the fixed-point digital filters. While the roundoff noise for the fixed-point arithmetic enters into the system additively, it is a multiplicative component in the case of the floating-point arithmetic. This problem is analyzed first by Sandberg [66], who discussed the roundoff error accumulation and input quantization effects in the direct realization of the filter excited by a deterministic input. He also derived a bound on the time average of the squared error at the output. Liu and Kaneko [47] presented a general approach to the error analysis problem of digital filters using the floating-point arithmetic and calculated the error at the output due to the roundoff accumulation and input quantization. Expressions are derived for the mean square error for each of the three canonical forms of realization: direct, cascade, and parallel. Upper bounds that are useful for a special class of the filters are given. Oppenheim and Weinstein [60] discussed in some details the effects of the finite register length on implementations of the linear recursive difference equation digital filters, and the fast Fourier transform (FFT) algorithm. Comparisons of the roundoff noise in the digital filters using the different types of arithmetics have also been reported in [71].

In order to validate the error analysis, most of the above work compare the theoretical results with corresponding experimental simulations. In this thesis, we show how the above error analysis can be mechanically performed using the HOL theorem prover, providing a superior approach to validation by simulation. Our focus will be on the process of translating the hand proofs into equivalent proofs in HOL. The analysis we propose is mostly inspired by the work done by Liu and Kaneko [47], who defined a general approach to the error analysis problem of digital filters using the floating-point arithmetic. Following a similar approach, we have extended this theoretical analysis for fixed-point digital filters. In both cases, a good agreement between the HOL formalized and the theoretical results are obtained.

Through our work, we confirmed and strengthened the main results of the previously published theoretical error analysis, though we uncovered some minor errors in the hand proofs and located a few subtle corners that are overlooked informally. For example, in the theoretical fixed-point error analysis it is always assumed that the fixed-point addition causes no error and only the roundoff error in the fixed-point multiplication is analyzed [60]. This is under the assumption that there is no overflow in the result and also the input operands have the same attributes as the output. Using a mechanical theorem prover, we provide a more general error analysis in which we cover the roundoff errors in both the fixed-point addition and multiplication operations. On top of that, for the floating-point error analysis, we have used the formalization in HOL of the IEEE-754 [28], a standard which has not yet been established at the time of the above mentioned theoretical error analysis. This enabled us to cover a more complete set of rounding and overflow modes and degenerate cases which are not discussed in earlier theoretical work.

1.6.4 Error Analysis of FFT Algorithms

Analysis of errors in FFT realizations due to finite precision effects has traditionally relied on paper-and-pencil proofs and simulation techniques. The roundoff error in using the FFT algorithms depends on the algorithm, the type of arithmetic, the word length, and the radix. For FFT algorithms realized with fixed-point arithmetic, the error problems have been studied extensively. For instance, Welch [73] presented an analysis of the fixed-point accuracy of the radix-2 decimation-in-time FFT algorithm. Tran-Thong and Liu [68] presented a general approach to the error analysis of the various versions of the FFT algorithm when fixed-point arithmetic is used. While the roundoff noise for fixed-point arithmetic enters into the system additively, it is a multiplicative component in the case of floating-point arithmetic. This problem is analyzed first by Gentleman and Sande [22], who presented an upper bound on the mean-squared error for floating-point decimation-in-frequency FFT algorithm. Weinstein [72] presented a statistical model for roundoff errors of the floating-point FFT. Kaneko and Liu [41] presented a detailed analysis of roundoff error in the FFT decimation-in-frequency algorithm using floating-point arithmetic. This analysis is later extended by the same authors to the FFT decimation-in-time algorithm [48]. Oppenheim and Weinstein [60] discussed in some detail the effects of finite register length on implementations of digital filters, and FFT algorithms.

In order to validate the error analysis, most of the above work compare the theoretical results with experimental simulation. In this thesis, we show how the above error analyses for the FFT algorithms can be mechanically performed using the HOL theorem prover, providing a superior approach to validation by simulation. Our focus will be on the process of translating the hand proofs into equivalent proofs in HOL. The analysis we develop is mainly inspired by the work done by Kaneko and Liu [41], who proposed a general approach to the error analysis problem of the decimation-in-frequency FFT algorithm using floating-point arithmetic. Following a similar idea, we have extended this theoretical analysis for the decimation-in-time and fixed-point FFT algorithms. In all cases, good agreements between formal and theoretical results were obtained.

1.6.5 Formalization and Verification of FFT Algorithms

Related work on the formalization and mechanical verification of the FFT algorithm was done by Gamboa [21] using the ACL2 theorem prover. The author formalized the FFT as a recursive data-parallel algorithm, using the powerlist data structure. He also presented an ACL2 proof of the correctness of the FFT algorithm, by translating the hand proof taken from Misra's seminal paper on powerlists [55] into a mechanical proof in ACL2. In the same line, Capretta [10] presented the formalization of the FFT using the type theory proof tool Coq. To facilitate the definition of the transform by structural recursion, Capretta used the structure of polynomial trees which is similar to the data structure of powerlists introduced by Misra. Finally, he proved its correctness and the correctness of the inverse Fourier transform (IFT).

Bjesse [5] described the verification of FFT hardware at the netlist level with an automatic combination of symbolic simulation and theorem proving using the Lava hardware development platform. He proved that the sequential pipelined implementation of the radix-4 decimation-in-time FFT is equivalent to the corresponding combinational circuit. He also proved that the abstract implementation of the radix-2 and the radix-4 FFT are equivalent for sizes that are an exponent of four. While [21] and [10] prove the correctness of the high level FFT algorithm against the DFT, the verification of [5] is performed at the netlist level. In contrast, our work tries to close this gap by formally specifying and verifying the FFT algorithm realizations at different levels of abstraction based on different data types. Besides, the definition used for the FFT in [21, 10] is based on the radix-2 decimation-in-time algorithm. We cover both decimation-in-time and decimation-in-frequency algorithms, and radices other than 2. The methodology we propose in this paper is, to the best of our knowledge, the first project of its kind that covers the formal specification and verification of integrated FFT algorithms at different abstraction levels starting from real specification to floating- and fixed-point algorithmic descriptions, down to RT and netlist gate levels.

1.7 Contributions of the Thesis

In light of the above related work review and discussions, we believe the contributions of the thesis can be specified as follows:

1. Formalization in higher-order logic of fixed-point arithmetic. We encoded the fixed-point number system and specified the different quantization and overflow modes and exceptions. An error analysis is then performed to check the correctness of the quantized result of basic arithmetic operations.

2. Mechanical analysis of finite word length effects in digital filters using HOL theorem prover. We derived expressions for the accumulation of roundoff error in parametric Lth-order digital filters, for each of the three canonical forms of realization: direct, parallel, and cascade. The HOL formalization and proofs are found to be in a good agreement with existing theoretical paper-and-pencil counterparts.

3. Formal specification and verification of fast Fourier transform (FFT) algorithms at different abstraction levels based on the HOL theorem prover. We

derive expressions for the accumulation of roundoff error in FFT designs. Finally, we use a classical hierarchical proof approach in HOL to prove that the FFT implementations at the register transfer and gate levels imply the corresponding high level fixed-point and floating-point algorithmic specifications taking into account the finite precision effects.

1.8 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 describes the fixed-point arithmetic and the details of its formalization in higher-order-logic. Chapter 3 describes the error analysis of digital filters using HOL theorem proving. Chapter 4 presents the verification of FFT algorithms in HOL from real specification to gate level implementation. Chapter 5 concludes the thesis and outlines the future research directions.

Chapter 2

Formalization of Fixed-Point Arithmetic in HOL

2.1 Introduction

Usually the conformance of the fixed-point implementation with respect to the floating-point specification is verified by simulation techniques which cannot cover the entire input space yielded by the floating-point representation. The objective of this work is to formalize the fixed-point arithmetic in higher-order logic as a basis for checking the correctness of the implementation of DSP designs against higher level algorithmic descriptions in floating-point and fixed-point representations.

Unlike floating-point arithmetic which is standardized in IEEE-754 [35] and IEEE-854 [36], current fixed-point arithmetic does not follow any particular standard and depends on the tool and the language used to design the DSP chip. For instance, in SPW (Signal Processing Worksystem), a fixed-point number is defined as a binary string and a set of attributes. Attributes specify how the binary string is interpreted using three arguments for the total number of bits, the number of integer bits, and the sign format. For arithmetic operations, it supports three kinds of exceptions such as *loss-of-sign* or *overflow*, two overflow modes, and five quantization modes.

In Matlab Simulink Fixed-Point Blockset [50], fixed-point numbers are stored in data types that are characterized by their word size (up to 128 bits), a *radix point*, and whether they are signed or unsigned. The *radix point* is used to support integers, fractionals, and generalized fixed-point data types. The Matlab Blockset provides four quantization modes corresponding to those supported by SPW. It also supports saturation and wrapping to deal with overflow for all fixed-point data types. Another example is the Synopsys CoCentric tool, which uses fixed-point as described in the SystemC language [61]. It supports signed and unsigned fixed-point data types, as well as limited precision (53 bits mantissa) fixed-point, called *fast fixed-point* to speed up simulation. SystemC supports seven quantization modes, of which four correspond exactly to the quantization modes of SPW. The other three modes are specific to SystemC and are not supported by the other tools. SystemC supports five overflow modes covering those of SPW. With the objective of providing a general methodology for the formalization and verification of fixed-point arithmetic using higher-order logic, we define in this chapter a complete common set of fixed-point arithmetic as supported by most of the DSP tools, in particular SPW and SystemC.

Based on higher-order logic, we propose to encode a fixed-point number by a pair composed of a Boolean word, and a triplet indicating the word length, the length of the integer portion, and the sign format. Then, we formalize the concepts of valuation and quantization as functions that convert respectively a fixed-point number to a real number and vice versa, taking into account different quantization and overflow modes. Fixed-point arithmetic operations are formalized as functions performing operations on the real numbers corresponding to the fixed-point operands and then applying the quantization on the real number result. Finally, we prove various lemmas regarding the error analysis of the fixed-point quantization and correctness of the basic operations like addition, multiplication, and division. The higher-order logic formalization and proof were done using the HOL theorem prover [26]. They were developed into a full fixed-point arithmetic library, which was recently included in the last release of HOL (HOL4, Kananaskis-2).

The rest of this chapter is organized as follows: Section 2.3 describes the fixed-point arithmetic definitions adopted in this thesis including the format of the fixed-point numbers, arithmetic operations, exceptions detection and their handling, and the different overflow and quantization modes. Section 2.4 describes in detail their formalization in HOL. In Section 2.5, we discuss the verification of basic fixed-point arithmetic operations, such as addition and multiplication. Section 2.6 presents an illustrative example on how this formalization can be used through the modeling and verification of an Integrator circuit. Finally, Section 2.7 concludes the chapter.

2.2 HOL Preliminaries

The HOL theorem prover is a mechanizaed proof-assistant developed by Mike Gordon at the University of Cambridge for conducting proofs in higher-order logic [26]. It was explicitly designed for the formal verification of hardware, though it has also been applied to other areas including software verification and formalization of pure mathematics.

HOL is based on LCF approach to interactive theorem proving and has many features in common with LCF systems developed at Cambridge and Edinbergh [24]. Like LCF, the HOL system supports secure theorem proving by representating its logic in the strongly-typed functional programming language ML [63]. Propositions and theorems of the logic are represented by ML abstract data types, and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. In addition to the usual programming language expressions, ML has expressions that evaluate to terms, types, formulas, and theorems of HOL's deductive apparatus. The HOL system supports a natural deduction style of proof, with driven rules formed from eight primitive inference rules. All inference rules are implemented using ML functions, and their application is the only way to obtain theorems in the system. Once proved, theorems can be saved and used in future proofs.

There are four types of HOL terms: constants, variables, function applications, and lambda-terms (denoted function abstractions). Polymorphism, types containing type variables, is a special feature supported by this logic. Semantically, types denote sets and terms denote members of these sets. Formulas, sequences, axioms, and theorems are represented by using terms of Boolean types. The main task of the higher-order logic theorem prover is the derivation of proofs. Accepting defined types and functions of new types, give us the ability to prove properties of those types and functions. The sets of types, type operators, constants and axioms available in HOL are organized in the form of theories. There are two main primitive theories, *bool* and *ind*, for booleans and individuals (a primitive type to denote distinct elements), respectively. Theorems can be derived based on these two main theories and added to the system.

HOL supports two styles of interactive proof: forward proof and backward proof. In the forward proof style, inference rules are simply applied in sequence to previously proved theorems until the desired theorem is obtained. The user specifies which rule to be applied at each step of the proof, either interactively or by writing an ML program that calls the appropriate sequence of procedures. Forward proof is not the easiest way of doing a proof, since the exact details of a proof are rarely known in advance. An important advance in proving using HOL was made by Robin Milner in the early 1970s when he invented the notion of *tactic*, introducing a new proof methodology called the backward, or goal-directed, proof style. A tactic is an ML function that breaks goals down into increasingly simple subgoals, until the subgoals obtained can be proved directly from theorems already derived. Again, the user specifies which tactic to use at each step. In addition to breaking a goal down into subgoals, a tactic also constructs a sequence of forward inference steps which can be used to prove the goal, once the subgoals have themselves been proved.

This is necessary because all theorems in the system must ultimately be obtained by forward proof. Table 2.2 summarizes some of the HOL symbols used in this thesis and their meanings [26]. The HOL type system does not support subtypes, so the real numbers ($\mathbb{R}$) have formally a different type from the natural numbers ($\mathbb{N}$). Therefore, the unary operator *ampersand* (&) is used to map between them. Thus the real number *numerals* can be written as $\&0, \&1$, etc [29].

HOL Symbol	Standard Symbol	Meaning
$@x.\ t$	$\varepsilon x.\ t$	An x such that $t\ (x)$ holds
$\lambda x.\ t$	$\lambda x.\ t$	Function that maps x to $t\ (x)$
&	(none)	Natural map operator ($\mathbb{N} \rightarrow \mathbb{R}$)
$\neg\ t$	$\neg\ t$	Not t
$\neg\ x$	$-\ x$	Unary negation of x
inv (x)	x^{-1}	Multiplicative inverse of x
abs (x)	$\mid x \mid$	Absolute value of x
x *pow* n	x^n	Real x raised to natural number power n
m *EXP* n	m^n	Natural number m raised to exponent n

Table 2.1: HOL Symbols

2.3 Fixed-Point Arithmetic

In this section we describe the fixed-point arithmetic definitions on which we base our formalization. While we tried to keep these definitions as general as possible, the fixed-point numbers format, arithmetic operations, overflow and quantization modes, and exception handling adopted are to some extent influenced by the fixed-point arithmetic defined by Cadence SPW [12] and Synopsys SystemC [61].

2.3.1 Fixed-Point Numbers

A fixed-point number has a fixed number of binary digits and a fixed position for the decimal point with respect to that sequence of digits. Fixed-point numbers can be either unsigned (always positive) or signed (in two's complement representation). For example, consider the case of four bits being used to represent the fixed-point numbers. If the numbers are unsigned and if the decimal point or, more properly, the binary point is fixed at the position after the second digit ($XX.XX$), the representable real values range from 0.0 to 3.75. In two's complement format, the most significant bit is the sign bit. The remaining bits specify the magnitude. If four bits represent the fixed-point numbers, and the binary point is fixed at the position after the second digit following the sign bit ($SXX.X$), the real values range from -4.0 to $+3.5$.

Fixed-point numbers are expressed as a pair consisting of a binary string and a set of attributes, ($Binary\ String, Attributes$). The attributes specify how the binary string is interpreted. Generally, the attributes are specified in the following format:

$$(wl, iwl, sign) \tag{2.1}$$

which consists of the following parameters:

- **wl:** Total word length, specifying the total number of bits used to represent the fixed-point binary string, including integer bits, fractional bits, and sign bit, if any. Word length must be in the range of 1 to 256.

- **iwl:** Integer word length, specifying the number of integer bits (the number of bits to the left of the binary point, excluding the sign bit, if any). If this number is negative, repeated leading sign bits or zeros are added to generate the equivalent binary value. If this number is greater than the total word length, trailing zeroes are added to generate the equivalent binary value.

- **sign:** A letter specifying the sign format: "u" for unsigned, and "t" for two's complement.

Example: According to the above definitions, the real value -0.75 is represented by $(111101, (6, 3, t))$. If we consider the same bit string with unsigned attributes

$(111101, (6, 3, u))$, then the equivalent number is 111.101 or +7.625. On the other hand, $(111101, (6, -3, u))$ represents the value .000111101 which is +0.119140625.

2.3.2 Fixed-Point Operations

A DSP design tool usually provides a library including basic fixed-point signal processing blocks such as adders, multipliers, delay blocks, and vector blocks. It also supports fixed-point hardware blocks such as multiplexers, buffers, inverters, flip-flops, bit manipulation and general-purpose combinational logic blocks. These blocks accurately model the behavior of fixed-point digital signal processing systems. In this thesis, we will focus on the arithmetic and logic operations, but the idea can be generalized to the remaining operations. Operations performed on fixed-point data types are done using arbitrary and full precision. After the operation is complete, the resulting operand is cast to fit the fixed-point data type object. The casting operation applies the quantization behavior of the target object to the new value and assigns the new value to the target object. Then, the appropriate overflow behavior is applied to the result of the process which gives the final value. In addition to the parameters corresponding to the input operands and output result, the arithmetic operations take specific parameters defining the overflow and quantization (loss of precision) modes. These parameters are as follows:

- **q_mode:** Quantization mode. This parameter determines the behavior of the fixed-point operations when the result generates more precision in the least significant bits (LSB) than is available.
- **o_mode:** Overflow mode. This parameter determines the behavior of the fixed-point operations when the result generates more precision in the most significant bits (MSB) than is available.
- **n_bits:** Number of saturated bits. This parameter is only used for overflow mode and specifies how many bits will be saturated if a saturation behavior

is specified and an overflow occurs.

Example: Consider a block that serves as a primitive fixed-point multiplier, which truncates the results when loss of precision occurs and wraps the result when overflow occurs. We can make a call to the multiplier routine through the function *fxpMul* (*Wrap* | *Truncate*, *In1*, *In2*, *Out*), in which *In1* and *In2* are the input fixed-point operands, *Out* is a parameter corresponding to the output attributes, and *Wrap* and *Truncate* indicate the overflow and quantization modes, respectively.

Fixed-Point Exception Handling

Fixed-point arithmetic operations that do not compute and return an exact result resort to an exception-handling procedure. This procedure is controlled by the exception flags. There are three kinds of exceptions that can be tested [12]:

- **Loss of Sign:** The result was negative but the result storage area was unsigned. Zero is stored.
- **Overflow:** The result was too big to be represented in the result storage area. The overflow mode determines the returned value.
- **Invalid:** No result can be meaningfully represented (e.g., divide by zero). This error can also occur if the fixed-point number itself is invalid.

Fixed-Point Quantization Modes

Quantization effects are used to determine what happens to the LSBs of a fixed-point type when more bits of precision are required than are available. The quantization modes are listed in Table 2.2.

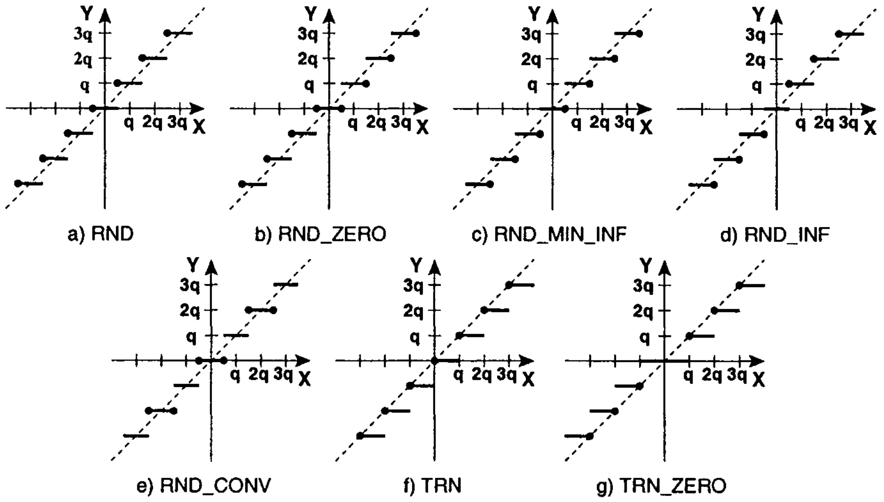
Figure 2.1 shows the behavior of each quantization mode. The X axis is the result of the previous arithmetic operation and the Y axis is the value after

Quantization Mode	**Name**
Quantization to Plus Infinity	RND
Quantization to Zero	RND_ZERO
Quantization to Minus Infinity	RND_MIN_INF
Quantization to Infinity	RND_INF
Convergent Quantization	RND_CONV
Truncation	TRN
Truncation to Zero	TRN_ZERO

Table 2.2: Fixed-Point Quantization Modes

quantization. The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the quantization. Any value of the X axis within the range of the line will be converted to the value of the Y axis. The symbol q in the figure refers to the quantization step, that is, the resolution of the data type. Each non integer value on the X axis is located in a quantization interval surrounded by two successive integer multiples of q as its closest representable quantized numbers, one greater and one smaller than the original value. If the value is exactly in the middle of the quantization interval, then the two closest representable numbers are equally distanced apart from the original value. As shown in this figure modes *RND*, *RND_ZERO*, *RND_MIN_INF*, *RND_INF*, and *RND_CONV* will quantize a value to the closest representable number if the two nearest representable numbers are not equally distanced apart from the original value. Otherwise, quantization towards plus infinity, to zero, towards minus infinity, towards plus infinity if positive or minus infinity if negative, and towards nearest even will be performed, respectively (Figure 2.1 (a-e)). The *TRN* mode is the default for fixed-point types and will be used if no other value is specified. The result is always quantized towards minus infinity (Figure 2.1 (f)). In other words, the result value is the first representable number lower than the original value. Finally, for *TRN_ZERO* the result is the nearest representable value to zero (Figure 2.1 (g))

[61].



a) RND b) RND_ZERO c) RND_MIN_INF d) RND_INF

e) RND_CONV f) TRN g) TRN_ZERO

Figure 2.1: The behavior of fixed-point quantization modes

Fixed-Point Overflow Modes

In addition to quantization modes, we can use overflow modes to approximate a higher range for fixed-point operations. Usually, overflow occurs when the result of an operation is too large or too small for the available bit range. Specific overflow modes can then be implemented to reduce the loss of data. Overflow modes are specified by the *o_mode* and *n_bits* parameters, and are listed in Table 2.3.

Overflow Mode	**Name**
Saturation	SAT
Saturation to Zero	SAT_ZERO
Symmetrical Saturation	SAT_SYM
Wrap-Around	WRAP
Sign Magnitude Wrap-Around	WRAP_SM

Table 2.3: Fixed-Point Overflow Modes

Figure 2.2 shows the behavior of each overflow mode for a 3 bit fixed-point data type. The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the original value and the Y axis is the result. From this figure, it can be seen that $MAX = 3$ and $MIN = -4$ for a 3 bit fixed-point data type. The *SAT* mode will convert the specified value to *MAX* for an overflow or *MIN* for an underflow condition (Figure 2.2 (a)). The *SAT_ZERO* mode will set the result to 0 for any input value that is outside the representable range of the fixed-point type. If the result value is greater than *MAX* or smaller than *MIN*, the result will be 0 (Figure 2.2 (b)). In the *SAT_SYM* mode, positive overflow will generate *MAX* and negative overflow will generate $-MAX$ for signed numbers or *MIN* for unsigned numbers (Figure 2.2 (c)). With the *WRAP* mode, the value of an arithmetic operand will wrap around from *MAX* to *MIN* as *MAX* is reached. There are two different cases within this mode. The first is with the *n_bits* parameter set to 0 or having a default value of 0. All bits except for the deleted bits are copied to the result number (Figure 2.2 (d)). The second is when the *n_bits* parameter is a nonzero value. In this case the specified number of most significant bits of the result number are saturated with preservation of the original sign, the other bits are simply copied. Positive numbers remain positive and negative numbers remain negative. A graph showing this behavior with *n_bits* = 1 is given in Figure 2.2 (e). Note that positive numbers wrap around to 0 while negative values wrap around to -1. The *WRAP_SM* overflow mode uses sign magnitude wrapping. This overflow mode behaves in two different styles depending on the value of the *n_bits* parameter. When *n_bits* is 0, no bits are saturated. This mode will first delete any MSB bits that are outside the result word length. The sign bit of the result is set to the value of the least significant deleted bit. If the most significant remaining bit is different from the original MSB, then all the remaining bits are inverted. If the MSBs are the same, the other bits are copied from the original value to the result value. A graph showing the result of this overflow mode is provided in Figure 2.2

(f). As the value of X increases, the value of Y increases to MAX and then slowly starts to decrease until MIN is reached. The result is a sawtooth like waveform. With *n_bits* greater than 0, *n_bits MSB* bits are saturated to 1. A graph showing this behavior with *n_bits* = 1 is given in Figure 2.2 (g). Note that while the graph looks somewhat like a sawtooth waveform, positive numbers do not dip below 0 and negative numbers do not cross -1 [61].

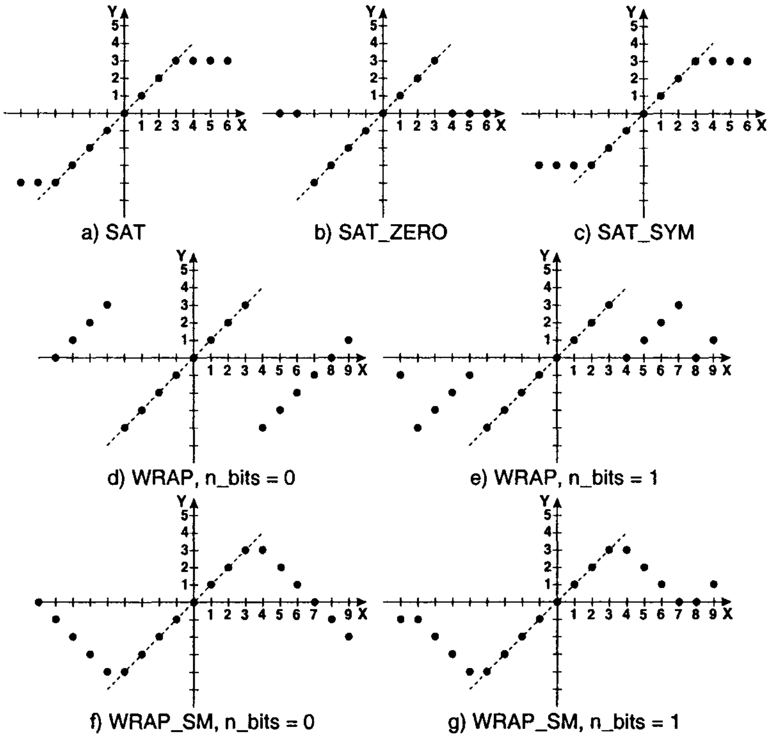

a) SAT
b) SAT_ZERO
c) SAT_SYM
d) WRAP, n_bits = 0
e) WRAP, n_bits = 1
f) WRAP_SM, n_bits = 0
g) WRAP_SM, n_bits = 1

Figure 2.2: The behavior of fixed-point overflow modes

2.4 Formalizing Fixed-Point Arithmetic in HOL

In this section, we present formalization of the fixed-point arithmetic in higher-order logic, based on the general purpose HOL theorem prover. The HOL system supports both forward and backward proofs. The forward proof style applies inference rules to existing theorems to obtain new theorems and eventually the desired theorem. Backward or goal oriented proofs start with the goal to be proven. Tactics are applied to the goal and subgoals until the goal is decomposed into simpler existing theorems or axioms. The system basic language includes the natural numbers and Boolean type. It also includes other specific extensions like reals library [27], which was proved to be essential for our fixed-point arithmetic formalization.

2.4.1 Fixed-Point Numbers Representation

The actual fixed-point numbers are represented in HOL by a pair of elements representing the binary string and the set of attributes. The extractors for the two fields of a fixed-point number are defined as follows:

$\vdash_{def}$ `string (s,a) = s`

$\vdash_{def}$ `attrib (s,a) = a`

The binary string is treated as a Boolean word (type: *bool word*). For example, the bit string 1010 is represented by *WORD [T;F;T;F].* In this way, we use the definitions and theorems already available in the HOL word library [70] to facilitate the manipulation of binary words. The attributes are represented by a triplet of natural numbers for the total number of bits, the integer bits and the sign format.

In HOL, we define functions to extract the primitive parameters for arbitrary attributes.

$\vdash_{def}$ `wordlength (w,iw,s) = w`

$\vdash_{def}$ `intbits (w,iw,s) = iw`

$\vdash_{def}$ `sign (w,iw,s) = s`

We also define predicates partitioning the fixed-point numbers into signed and unsigned numbers.

$\vdash_{def}$ `is_signed X = (sign X = 1)`

$\vdash_{def}$ `is_unsigned X = (sign X = 0)`

The number of digits on the right hand side of the binary point of a fixed-point number is defined as *fracbits.* It can be derived as the difference between the total number of bits and the number of integer bits, considering the sign bit in the case of signed numbers.

```
⊢def  fracbits X =
       if (is_unsigned X) then (wordlength X − intbits X)
       else (wordlength X − intbits X − 1)
```

Two useful derived predicates test the validity of a set of attributes and a fixed-point number based on the definition in Section 2.3.1. In a valid set of attributes, the *wordlength* should be in the range of 1 and 256, the *sign* can be either 0 or 1, and the number of integer bits is less than or equal to the *wordlength.* A valid fixed-point number must have a valid set of attributes and the length of its binary string must be equal to the *wordlength.*

```
⊢def  validAttr X =
       wordlength X > 0 ∧ wordlength X < 257 ∧
       intbits X < wordlength X + 1 ∧ sign X < 2
```

```
⊢def  is_valid a =
       validAttr (attrib a) ∧ (WORDLEN (string a) = wordlength (attrib a))
```

where *WORDLEN* is a predefined function of the HOL *word* library, which returns the size of a word.

2.4.2 Fixed-Point Type

Now we define the actual HOL type for the fixed-point numbers. The type is defined to be in bijection with the appropriate subset of $(bool\ word \times \mathbb{N}^3)$, with the bijections written in HOL as $fxp : (bool\ word \times \mathbb{N}^3) \rightarrow fxp$, and $defxp : fxp \rightarrow (bool\ word \times \mathbb{N}^3)$. The bijection maps the set of all elements of type $(bool\ word \times \mathbb{N}^3)$ to the set of valid fixed-point numbers specified by the function *is_valid* as defined in the previous section. For this purpose, we make use of built-in facilities in HOL for defining new bijection types [69]. A similar technique was used in [29] for defining type bijections for the floating-point numbers (*float,defloat*) in HOL.

```
   fxp_tybij =
⊢  (∀a. fxp (defxp a) = a) ∧ (∀r. is_valid r = (defxp (fxp r) = r))
```

We specialize the previous functions and predicates to the *fxp* type, as follows:

```
⊢def  String a = string (defxp a)
⊢def  Attrib a = attrib (defxp a)
⊢def  Wordlength a = wordlength (Attrib a)
⊢def  Intbits a = intbits (Attrib a)
⊢def  Fracbits a = fracbits (Attrib a)
⊢def  Sign a = sign (Attrib a)
⊢def  Issigned a = is_signed (Attrib a)
⊢def  Isunsigned a = is_unsigned (Attrib a)
⊢def  Isvalid a = is_valid (defxp a)
```

Note that we start the name of the functions manipulating fixed-point numbers by capital letters to distinguish them from those taking pairs and triplets as argument.

2.4.3 Fixed-Point Valuation

Now we specify the real number valuation of fixed-point numbers. We use two separate formulas for signed and unsigned numbers:

- **Unsigned:**

$$(1/2^M) * (\sum_{n=0}^{N-1} 2^n \ * \ v_n) \qquad (2.2)$$

- **Signed:**

$$(1/2^M) * [\sum_{n=0}^{N-1} 2^n \ * \ v_n - 2^N * v_{N-1}] \qquad (2.3)$$

where v_n represents the n^{th} bit of the binary string in the fixed-point number[1], and M and N are respectively *fracbits* and *wordlength*. In HOL, we define the valuation function *value* that returns the corresponding real value of a fixed-point number.

```
⊢def  value a =
       if (Isunsigned a) then &(BNVAL (String a)) / 2 pow Fracbits a
       else (&(BNVAL (String a)) - &((2 EXP Wordlength a) *
            BV (MSB (String a)))) / 2 pow Fracbits a
```

where $BNVAL$ is a function which returns the numeric value of a Boolean word, BV is a function for mapping between a single bit and a number, and *MSB* is a constant for the most significant bit of a word, available in the HOL *word* library.

We also define the real value of the smallest (*MIN*) and largest (MAX) representable numbers for a given set of attributes. The maximum is defined for both signed and unsigned numbers using the following formula:

$$MAX = 2^a - 2^{-b} \tag{2.4}$$

where a is the *intbits* and b the *fracbits*. The minimum value for unsigned numbers is zero and for signed numbers is computed using the following formula:

$$MIN = -2^a \tag{2.5}$$

Thereafter, we obtain the corresponding functions in HOL.

```
⊢def  MAX X = 2 pow intbits X - inv (2 pow fracbits X)
⊢def  MIN X = if (is_unsigned X) then 0 else ¬(2 pow intbits X)
```

The constants for the smallest (*bottomfxp*) and largest (*topfxp*) representable fixed-point numbers for a given set of attributes can be defined as follows:

[1] We adopt the convention that bits are indexed from the right hand side.

```
⊢def  topfxp X =
        if (is_unsigned X) then fxp (WORD (REPLICATE (wordlength X) T),X)
        else fxp (WCAT (WORD [F],WORD (REPLICATE (wordlength X - 1) T)),X)
```

```
⊢def  bottomfxp X =
        if (is_unsigned X) then fxp (WORD (REPLICATE (wordlength X) F),X)
        else fxp (WCAT (WORD [T],WORD (REPLICATE (wordlength X - 1) F)),X)
```

where *WCAT* denotes the concatenation of two words, and *REPLICATE* makes a list consisting of a value replicated a specified number of times, which are predefined functions in HOL.

2.4.4 Exception Handling

Operations on fixed-point numbers can signal exceptions as described in Section 2.3.2. These are declared as a new HOL data type.

```
⊢def  Exception = no_except | overflow | invalid | loss_sign
```

where *no_except* is reserved for the case without exception.

Five overflow modes are also represented via an enumerated type definition.

```
⊢def  overflow_mode = SAT | SAT_ZERO | SAT_SYM | WRAP | WRAP_SM
```

According to the definition of overflow modes in Section 2.3.2 for *Saturation*, if the number is greater than *MAX* or less than *MIN*, we return *topfxp* and *bottomfxp*, as the closest representable values to the right result, respectively. For *Saturation to Zero* overflow, we will return zero in any case. For *Symmetrical Saturation*, if the number is greater than *MAX*, we return *topfxp*. If the number is less than *MIN*, we return the two's complement of the maximum value, defined by the function *minustopfxp* for signed, and *bottomfxp* for unsigned numbers, respectively. For *Wrap-around* and *Sign magnitude*, we must first convert the real number to a binary format. Then we discard the extra bits according to the output attributes, and saturate the required bits based on the parameter *n_bits*. The details are defined as functions *WRAP_AROUND* and *WRAP_AROUND_SM*. Therefore, we define the fixed-point overflow function in HOL as follows:

```
⊢def fxp_overflow X o_mode n_bits x =
      if (x > MAX X) then
            if (o_mode = SAT) then topfxp X
            else if (o_mode = SAT_ZERO) then
                  fxp (WORD (REPLICATE (wordlength X) F),X)
            else if (o_mode = SAT_SYM) then topfxp X
            else if (o_mode = WRAP) then
                  WRAP_AROUND X n_bits x
            else WRAP_AROUND_SM X n_bits x
      else if (x < MIN X) then
            if (o_mode = SAT) then bottomfxp X
            else if (o_mode = SAT_ZERO) then
                  fxp (WORD (REPLICATE (wordlength X) F),X)
            else if (o_mode = SAT_SYM) then
                  if (is_unsigned X) then bottomfxp X
                  else minustopfxp X
            else if (o_mode = WRAP) then
                  WRAP_AROUND X n_bits x
            else WRAP_AROUND_SM X n_bits x
      else Null
```

where *Null* is a constant that represents the result of an invalid operation, defined as:

```
⊢def Null = @a. ¬ (Isvalid a)
```

Note that if the number is in the representable range of the given attributes, i.e. its value is neither greater than *MAX* nor less than *MIN*, then the overflow is meaningless and *Null* will be returned as the result.

2.4.5 Quantization

Fixed-point quantization takes an infinitely precise real number and converts it into a fixed-point number. Seven quantization modes are specified in Section 2.3.2, which we formalize using the following data type.

```
⊢def  quantization_mode =
        RND | RND_ZERO | RND_MIN_INF | RND_INF | RND_CONV | TRN | TRN_ZERO
```

Then we define the fixed-point quantization operation by a function, which is defined case by case on the quantization modes as follows:

```
⊢def  fxp_quantize X q_mode x =
        if (q_mode = RND) then
              closest value (λ a. value a ≥ x)
              {a | (Isvalid a) ∧ (Attrib a = X)} x
        else if (q_mode = RND_ZERO) then
              closest value (λ a. abs (value a) ≤ abs x)
              {a | (Isvalid a) ∧ (Attrib a = X)} x
        else if (q_mode = RND_MIN_INF) then
              closest value (λ a. value a ≤ x)
              {a | (Isvalid a) ∧ (Attrib a = X)} x
        else if (q_mode = RND_INF) then
              closest value
              (λ a. (if 0 ≤ x then value a ≥ x else value a ≤ x))
              {a | (Isvalid a) ∧ (Attrib a = X)} x
        else if (q_mode = RND_CONV) then
              closest value (λ a. LSB (String a) = F)
              {a | (Isvalid a) ∧ (Attrib a = X)} x
        else if (q_mode = TRN) then
              closest value (λ a. T)
              {a | (Isvalid a) ∧ (Attrib a = X) ∧ (value a ≤ x)} x
        else closest value (λ a. T)
             {a | (Isvalid a) ∧ (Attrib a = X) ∧
             (abs (value a) ≤ abs x)} x
```

The fixed-point quantization function takes as arguments a real number, a quantization mode, and an output attributes, and returns the corresponding fixed-point number. Similar to the floating-point case [29], its definition is based on the following predicate meaning that a is an element of the set s that provides a best approximation to x, assuming a valuation function v:

```
⊢def  is_closest v s x a =
        ((a IN s) ∧ ∀b. (b IN s) ⟹  (abs (v a − x) ≤ abs (v b − x)))
```

However, we still need to define a function that picks out a best approximation in case there are more than one closest number, based on a given property like *even.* This can be done in HOL as follows:

```
⊢def  closest v p s x =
        @a. ((is_closest v s x a) ∧
             ((∃b. (is_closest v s x b) ∧ (p b)) ⟹  (p a)))
```

Finally, we define the actual fixed-point rounding function for an arbitrary output attributes.

```
⊢def  fxp_round X o_mode q_mode n_bits x =
        if (x > MAX X ∨ x < MIN X) then
              ((fxp_overflow X o_mode n_bits x),overflow)
        else ((fxp_quantize X q_mode x),no_except)
```

where *fxp_overflow* is the fixed-point overflow function as defined in the previous section and supports all overflow modes, and *fxp_quantize* is the fixed-point quantization function that supports all quantization modes. The fixed-point rounding function takes as argument a real number, an output attributes, the quantization and overflow modes, and the number of saturated bits. It returns a fixed-point number and an exception flag. The function first checks for overflow, and in case of overflow returns the result based on the overflow mode, and sets the exception flag to *overflow.* Otherwise, it performs the quantization based on the quantization mode, and sets the exception flag to *no_except.*

2.4.6 Fixed-Point Arithmetic Operations

Fixed-point arithmetic operations such as addition or multiplication take two fixed-point input operands and store the result into a third. The attributes of the inputs and output need not match one another. Both unsigned and two's complement inputs and output are allowed. The result is formatted into the output as specified by the output attributes and by the overflow and loss of precision mode parameters. In our formalization, we first deal with exceptional cases such as invalid operation and loss of sign. If any of the input numbers is invalid, then the result is *Null* and the exception flag *invalid* is raised. If the result is negative but the output is unsigned then zero is returned and the exception flag *loss_sign* is raised. Also in the case of division by zero, the output value is forced to zero and the *invalid* flag is raised. Otherwise, we take the real value of the input arguments, perform the operation as infinite precision, then quantize the result according to the desired quantization and overflow modes. Formally, the operations for addition, subtraction, multiplication, and division are defined as follows:

```
⊢def fxpAdd X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a + value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a + value b)
```

```
⊢def fxpSub X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a − value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a − value b)
```

```
⊢def fxpMul X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a * value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a * value b)
```

```
⊢def  fxpDiv X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value b = 0) then
           (fxp (WORD (REPLICATE (wordlength X) F),X),invalid)
      else if (value a / value b < 0 ∧ is_unsigned X) then
           (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a / value b)
```

2.5 Verification of Fixed-Point Operations

According to the discussion in Section 2.4.3, each fixed-point number has a corresponding real number value. The correctness of a fixed-point operation can be specified by comparing its output with the true mathematical result, using the valuation function *value* that converts a fixed-point to an infinitely precise number. For example, the correctness of a fixed-point adder *fxpAdd* is specified by comparing it with its ideal counterpart +. That is, for each pair of fixed-point numbers *(a,b)*, we compare *value (a)* + *value (b)* and *value (fxpAdd (a,b))*. In other words, we check if the diagram in Figure 2.3 commutes.

$$\begin{tikzcd}
\text{value (a) , value (b)} \arrow[r, "+"] & \begin{array}{c}\text{value (a) + value (b)} \\ \approx ? \\ \text{value (fxpAdd (a,b))}\end{array} \\
\text{a , b} \arrow[u, "\text{value}"] \arrow[r, "\text{fxpAdd}"'] & \text{fxpAdd (a,b)} \arrow[u, "\text{value}"']
\end{tikzcd}$$

Figure 2.3: Correctness criteria for fixed-point addition

For this purpose we define the error resulting from quantizing a real number to a fixed-point value as follows:

```
⊢def  fxperror X o_mode q_mode n_bits x =
      value (FST (fxp_round X o_mode q_mode n_bits x)) − x
```

and then establish the correctness theorems for all four fixed-point arithmetic operations.

```
Theorem 1: FXP_ADD_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpAdd (X) o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpAdd (X) o_mode q_mode n_bits a b)) =
   value (a) + value (b) +
   (fxperror (X) o_mode q_mode n_bits (value (a) + value (b))))
```

```
Theorem 2: FXP_SUB_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpSub X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpSub X o_mode q_mode n_bits a b)) =
   value (a) − value (b) +
   (fxperror X o_mode q_mode n_bits (value a − value b)))
```

```
Theorem 3: FXP_MUL_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpMul X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpMul X o_mode q_mode n_bits a b)) =
   (value a * value b) +
   (fxperror X o_mode q_mode n_bits (value a * value b)))
```

```
Theorem 4: FXP_DIV_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpDiv X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpDiv X o_mode q_mode n_bits a b)) =
   (value a / value b) +
   (fxperror X o_mode q_mode n_bits (value a / value b)))
```

The theorems are composed of two parts. The first part is about the validity of the fixed-point arithmetic operation output and states that if the input fixed-point numbers and the output attributes are valid then the result of the fixed-point operation is valid. The second part of the theorem relates the result of the fixed-point arithmetic operations to the real result based on the corresponding error function. To prove these main theorems, a number of lemmas have been established. We first proved lemmas concerning the approximation of a real number with a fixed-point number. We proved that in a finite non-empty set of fixed-point numbers, we can find the best approximation to a real number based on a given valuation function (*Lemma 1*).

```
Lemma 1: FXP_IS_CLOSEST_EXISTS
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  ∃ (a: fxp). is_closest v s x a
```

Then, we proved that the chosen best approximation to a real number satisfying a property *p* from a finite and non-empty set of fixed-point numbers is unique (*Lemma 2*), and is itself a member of the set (*Lemma 3*), and is itself the best approximation of the real number (*Lemma 4*).

```
Lemma 2: FXP_CLOSEST_IS_EVERYTHING
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹
   is_closest v s x (closest v p s x) ∧
   ((∃b. is_closest v s x b ∧ p b) ⟹  p (closest v p s x))
```

```
Lemma 3: FXP_CLOSEST_IN_SET
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  (closest v p s x) IN s
```

```
Lemma 4: FXP_CLOSEST_IS_CLOSEST
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  is_closest v s x (closest v p s x)
```

Finally, we proved that the chosen best approximation to a real number satisfying a property *p* from the set of all valid fixed-point numbers with a given attributes is itself a valid fixed-point number (*Lemma 5*).

```
Lemma 5: IS_VALID_CLOSEST
⊢  (validAttr X) ⟹
   Isvalid (closest v p {a | Isvalid a ∧ ((Attrib a) = X)} x)
```

Besides, we proved that the set of all valid fixed-point numbers with a given attributes is finite (*Lemma 6*).

```
Lemma 6: FINITE_VALID_ATTRIB
⊢  FINITE {a | Isvalid a ∧ (Attrib a = X)}
```

The proof of this lemma is a bit complicated. For this purpose we made use of some built-in theorems about finite sets in the HOL *pred_sets* library [51]. Among these are the two fundamental theorems *FINITE_EMPTY* and *FINITE_INSERT*, which state that the empty set is indeed finite and the insertion of an element to a finite set constructs a finite set. Other theorems state that the union of two finite sets (*FINITE_UNION*), the image of a function on a finite set (*IMAGE_FINITE*), a singleton set[2] (*FINITE_SING*), the cross combination of two finite sets (*FINITE_CROSS*), and any subset of a finite set (*SUBSET_FINITE*) is itself a finite set. Using these theorems together with the definition of a valid fixed-point number helped us to break down the proof of the finiteness of all valid fixed-point numbers to the proof of finiteness of the set of all Boolean words with a given word length (*WORD_FINITE*) and the set of all natural numbers less than a given value (*FINITE_COUNT*). The last lemmas are proved by induction on the word length of the Boolean word and the maximum limit of the natural numbers, respectively.

We also proved that the set of all valid fixed-point numbers is nonempty (*Lemma 7*).

```
Lemma 7: IS_VALID_NONEMPTY
⊢  (validAttr X) ⟹  ¬({a | Isvalid a ∧ (Attrib a = X)} = EMPTY)
```

Finally, we proved that the result of quantizing a real number, which is in the range representable by a given valid attributes, is a valid fixed-point number (*Lemma 8*).

```
Lemma 8: IS_VALID_QUANTIZATION
⊢  (validAttr X) ⟹  Isvalid (FST (fxp_round X o_mode q_mode n_bits x))
```

The validity of the quantization directly implies validity of the fixed-point operation output, and this completes the proof of the first parts of the theorems. The second parts of the theorems are proved using the properties of the real arithmetic in HOL and rewriting with the definitions of the *fxpAdd, fxpSub, fxpMul, fxpDiv,* and *fxperror* functions.

The second main theorem on fixed-point error analysis concerns bounding the quantization error. The error can be absolutely quantified as follows:

[2] a set that contains precisely one element.

```
Theorem 5: FXP_ERROR_BOUND_THM
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬ (x < MIN (X)) ⟹
   abs (fxperror X o_mode q_mode n_bits x) ≤ inv (&2 pow fracbits X)
```

According to this theorem, the error in quantizing a real number which is in the range representable by a given set of attributes X is less than the quantity $1 \ / \ 2^{fracbits\ (X)}$. This theorem is valid for all fixed-point quantization modes. However, for *RND, RND_ZERO, RND_MIN_INF, RND_INF*, and *RND_CONV* modes, which quantize to the nearest representable value, the error can be bounded to $1 \ / \ 2^{(fracbits\ (X)+1)}$ by extending the theorem.

To explain the theorem, we consider the following fact that relates the definition of the fixed-point numbers to the rationals.

An N-bit binary word, when interpreted as an unsigned fixed-point number, can take on values from a subset P of the non-negative rationals given by

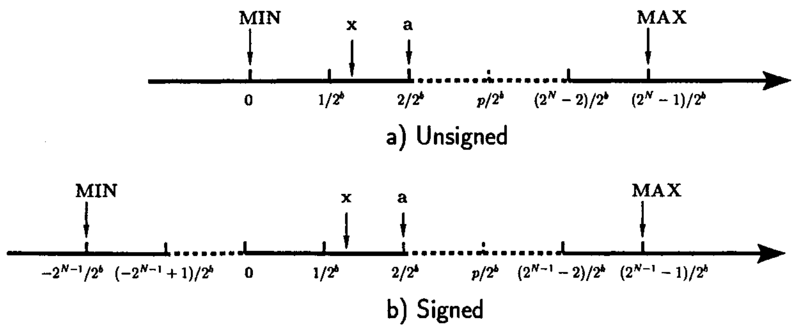
$$P \ = \ \{p/2^b \ | \ 0 \ \leq \ p \ \leq \ 2^N \ - \ 1, \ p \ \in \ \mathbb{Z}\} \tag{2.6}$$

Similarly, for signed two's complement representation, we have

$$P \ = \ \{p/2^b \ | \ -2^{N-1} \ \leq \ p \ \leq \ 2^{N-1} \ - \ 1, \ p \ \in \ \mathbb{Z}\} \tag{2.7}$$

Note that P contains 2^N elements and b represents the fractional bits in each case.

Based on this fact, we can depict the range of values covered for each case as shown in Figure 2.4.



a) Unsigned

b) Signed

Figure 2.4: Fixed-point values on the real axis

Thereafter, the representable range of fixed-point numbers is divided into 2^N equispaced quantization steps with the distance between two representable steps equal to $1 \ / \ 2^b$. Suppose that $x \in \mathbb{R}$ is approximated by a fixed-point number a. The position of these values are labeled in Figure 2.4. The error $| \ x - a \ |$ is hence less than the length of one interval, or $1 \ / \ 2^b$, as mentioned in the second theorem.

In HOL, we first proved that the quantization result is the nearest value to a real number and the corresponding error is minimum compared to the other fixed-point numbers (*Lemma 9*).

```
Lemma 9: FXP_ERROR_AT_WORST_LEMMA
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ∧
  (Isvalid a) ∧ (Attrib a = X) ⟹
  abs (fxperror X o_mode q_mode n_bits x) ≤ abs (value a − x)
```

Then we proved that each representable real value x can be surrounded by two representable rational numbers (*Lemma 10*).

```
Lemma 10: FXP_ERROR_BOUND_LEMMA1
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
  ∃k. (k < 2 EXP wordlength X) ∧ (&k / (&2 pow fracbits X) ≤ x) ∧
  (x < (&(SUC k) / (&2 pow fracbits (X))))
```

Also we proved that the difference between the real number and the surrounding rationals is less than $1 \ / \ 2^{fracbits \ (X)}$ (*Lemma 11*).

```
Lemma 11: FXP_ERROR_BOUND_LEMMA2
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
  ∃k. (k ≤ 2 EXP wordlength X) ∧
  abs (x − &k / (&2 pow (fracbits (X)))) ≤ inv (&2 pow (fracbits (X)))
```

Finally, we proved that for each real value we can find a fixed-point number with the required error characteristics (*Lemma 12*).

```
Lemma 12: FXP_ERROR_BOUND_LEMMA3
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹ ∃(w: bool word).
  abs (value (fxp (w,X)) − x) ≤ inv (&2 pow (fracbits X)) ∧
  (WORDLEN w = wordlength X)
```

Since the quantization produces the minimum error as stated in *Lemma 9*, the proof of the second main theorem (*Theorem 5*) is a direct consequence of *Lemma 12*. In these proofs, we have treated the case of signed and unsigned numbers separately since they have different definitions for *MAX*, *MIN*, and *value* functions. For signed numbers a special attention needs also to be paid to deal with negative numbers.

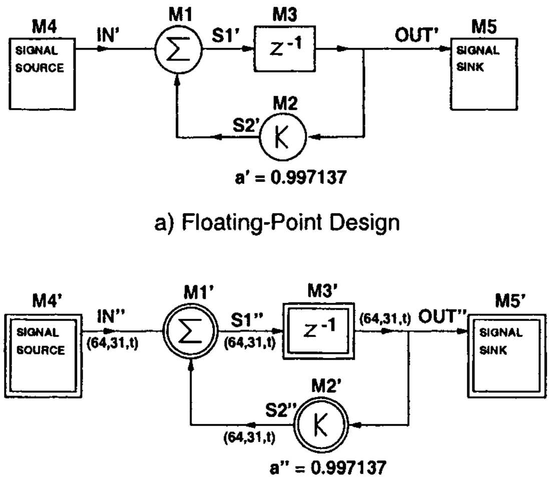
2.6 Application with SPW

In this section we demonstrate how to apply the formalization of fixed-point arithmetic presented in the previous sections for the verification of the transition from floating-point to fixed-point algorithmic levels. We have chosen SPW as application tool and the case of an *Integrator* as an example circuit. A digital integrator is a discrete time system that transforms a sequence of input numbers into another sequence of output, by means of a specific computational algorithm. To describe the general functionality of a digital integrator, let $\{x_t\}$, $\{w_t\}$, and a denote the input sequence, output sequence, and constant coefficient of the integrator, respectively. Then the integrator can be specified by the difference equation:

$$w_t = x_{t-1} + a\ w_{t-1} \tag{2.8}$$

Thereafter, the output sequence at time t is equal to the input sequence at time *t - 1*, added to the output at time *t - 1* multiplied by the integrator coefficient.

Figure 2.5 shows the SPW design of an integrator. The integrator is first designed and simulated using the SPW predefined floating-point blocks and parameters (Figure 2.5 (a)). The design is composed of an adder (*M1*), a multiplier by constant (*M2*), and a delay (*M3*) block, together with signal source (*M4*) and sink (*M5*) elements. The input signal, the output signal, and the output of the adder and multiplier blocks are labeled by *IN', OUT', S1',* and *S2'*, respectively. Figure 2.5 (b) shows the converted fixed-point design in which each block is replaced with the corresponding fixed-point block (*M1', M2', M3', M4', M5'*). Fixed-point blocks are shown by double circles and squares to distinguish them from the floating-point blocks. The attributes of all fixed-point block outputs are set to $(64, 31, t)$ to ensure that overflow and quantization do not affect the



a) Floating-Point Design

b) Fixed-Point Design

Figure 2.5: SPW design of an integrator

system operation. The corresponding fixed-point signals are labeled by *IN"*, *OUT"*, *S1"*, and *S2"*.

In HOL, we first model the design at each level as predicates in higher-order logic. The predicates corresponding to the floating-point design are as follows:

```
⊢def  Float_Gain_Block a′ b′ c′ = (∀t. c′ t = a′ t float_mul b′)
⊢def  Float_Delay_Block a′ b′ = (∀t. b′ t = a′ (t − 1))
⊢def  Float_Add_Block a′ b′ c′ = (∀t. c′ t = a′ t float_add b′ t)

⊢def  Float_Integrator_Imp X a′ IN′ OUT′ =
      ∃ S1′ S2′.
      Float_Add_Block IN′ S2′ S1′ ∧
      Float_Delay_Block S1′ OUT′ ∧
      Float_Gain_Block OUT′ a′ S2′
```

where X is the floating-point format. In these definitions, we have used available formalization of floating-point arithmetic in HOL [29]. Floating-point data types are stored in SPW in the standard IEEE 64 bit double precision format.

The HOL description of the fixed-point implementation is as follows:

```
⊢def  Fxp_Gain_Block a'' b'' c'' = (∀t. c'' t = a'' t fxp_mul b'')
⊢def  Fxp_Delay_Block a'' b'' = (∀t. b'' t = a'' (t - 1))
⊢def  Fxp_Add_Block a'' b'' c'' = (∀t. c'' t = a'' t fxp_add b'' t)
```

```
⊢def  Fxp_Integrator_Imp X' o_mode q_mode n_bits a'' IN'' OUT'' =
      ∃ S1'' S2''.
      Fxp_Add_Block IN'' S2'' S1'' ∧
      Fxp_Delay_Block S1'' OUT'' ∧
      Fxp_Gain_Block OUT'' a'' S2''
```

where *X'* is the fixed-point format, and the functions *fxp_add* and *fxp_mul* are defined as follows:

```
⊢def  a'' fxp_add b'' = FST (fxpAdd X' o_mode q_mode n_bits a'' b'')
⊢def  a'' fxp_mul b'' = FST (fxpMul X' o_mode q_mode n_bits a'' b'')
```

In the next step, we describe each design as a difference equation relating the input and output samples according to the equation (4.4).

```
⊢def  FLOAT_Integrator_Spec X a' IN' OUT' =
      ∀t. OUT' t = (IN' (t - 1) float_add (a' float_mul OUT' (t - 1)))
```

```
⊢def  FXP_Integrator_Spec X' o_mode q_mode n_bits a'' IN'' OUT'' =
      ∀t. OUT'' t = (IN'' (t - 1) fxp_add (a'' fxp_mul OUT'' (t - 1)))
```

The following lemmas ensure that the implementation at each level satisfies the corresponding specification.

```
Lemma 13: FLOAT_INTEGRATOR_IMP_SPEC
⊢  Float_Integrator_Imp X a' IN' OUT' ⟹
   Float_Integrator_Spec X a' IN' OUT'
```

```
Lemma 14: FXP_INTEGRATOR_IMP_SPEC
⊢  Fxp_Integrator_Imp X' o_mode q_mode n_bits a'' IN'' OUT'' ⟹
   Fxp_Integrator_Spec X' o_mode q_mode n_bits a'' IN'' OUT''
```

Now we assume that the floating-point and fixed-point input sequences are the rounded versions of an infinite precision ideal input *IN*, and we have

```
⊢def  IN' t = round X To_nearest (IN t)
⊢def  IN'' t = FST (fxp_round X' o_mode q_mode n_bits (IN t))
```

where *round* is the floating-point rounding function, and *To_nearest* is the corresponding mode for rounding to nearest floating-point number [29]. We also make some other assumptions on finiteness and validity of floating-point and fixed-point inputs, coefficients, and intermediate results, in order to have finite and valid final outputs. Using these assumptions and based on the theorems *FXP_ADD_THM* and *FXP_MUL_THM* (Section 2.5) and the corresponding ones in floating-point theory [29], we prove the following theorem concerning the error between the real values of the floating-point and fixed-point precision integrator output samples.

```
Theorem 6: INTEGRATOR_THM
⊢  Float_Integrator_Imp X a' IN' OUT' ∧
   Fxp_Integrator_Imp X' o_mode q_mode n_bits a'' IN'' OUT''
   ⟹
   Val (OUT' t) - value (OUT'' t) =
   Val a' * Val (OUT' (t - 1)) -
   value a'' * value (OUT'' (t - 1)) +
   error (IN (t - 1)) +
   error (Val a' * Val (OUT' (t - 1))) +
   error (Val (IN' (t - 1)) + Val (a' float_mul OUT' (t - 1))) -
   fxperror X' o_mode q_mode n_bits (IN (t - 1)) -
   fxperror X' o_mode q_mode n_bits
   (value a'' * value (OUT'' (t - 1))) -
   fxperror X' o_mode q_mode n_bits
   (value (IN'' (t - 1)) + value (a'' fxp_mul OUT'' (t - 1)))
```

where *Val* is the floating-point valuation function, and *error* is the floating-point rounding error function [29]. According to *Theorem 6*, for a valid and finite set of input and output sequences at time *(t - 1)* to the integrator design at the floating-point and fixed-point levels, we can have finite and valid outputs at time t, and the difference in the real values corresponding to these output samples can be expressed as the difference in input and output values multiplied by the corresponding coefficients, taking into account the effects of finite precision in coefficients and arithmetic operations. To find a constant upper bound for the difference between the outputs, we use *Theorem 5* on the fixed-point error quantification. Similarly, for the floating-point error bound analysis we proved the following lemma:

```
Lemma 15: ERROR_BOUND_NORM_STRONG_NORMALIZE
⊢ normalizes X x ⟹
   ∃ j. abs (error x) ≤ (2 pow j / 2 pow (bias X + fracwidth X))
```

where *normalizes* defines the criteria for an arbitrary real number to be in the range of normalized floating-point numbers, *bias* defines the exponent bias in the floating-point format which is a constant used to make the exponent's range non-negative, and *fracwidth* extracts the fraction width parameter from the floating-point format. According to *Lemma 15*, if the absolute value of a real number is in the representable range of the normalized floating-point numbers with the format X and located in the j'th binade (the floating-point numbers between two adjacent powers of 2), then the absolute value of the error is less than or equal to $2^j/2^{(bias\ X\ +\ fracwidth\ X)}$. The lemma is proved based on the general floating-point absolute error bound theorem developed in [29].

Finally, we proved the following theorem (*Theorem 7*) that bounds the output error of the integrator design in the transition from the floating-point to fixed-point levels.

```
Theorem 7: INTEGRATOR_FP_TO_FXP_ERROR_BOUND_THM
⊢ Float_Integrator_Imp X a′ IN′ OUT′ ∧
  Fxp_Integrator_Imp X′ o_mode q_mode n_bits a″ IN″ OUT″
  ⟹
  ∃ j1 j2 j3.
  abs (Val (OUT′ t) − value (OUT″ t)) ≤
  2 * abs (a) * M +
  (2 pow j1 + 2 pow j2 + 2 pow j3) / 2 pow (bias X + fracwidth X) +
  3 / (2 pow (fracbits X′))
```

In the proof of this theorem, we have assumed that the real values of the floating-point and fixed-point integrator coefficients are equal (*Val a' = value a"* $= a$), hence ignoring the effects of inaccuracies in the integrator coefficient. We have also assumed that the floating-point and fixed-point output values are bounded to a constant value (M). The parameters *j1, j2,* and *j3* are related to the binades in which the real valued arguments of the three floating-point error expressions in *Theorem 6* are located.

2.7 Conclusion

In this chapter, we established the formalization of fixed-point arithmetic in the HOL theorem prover. The formalization presented in this chapter can be considered as a complement to the floating-point formalizations which are widely available in the literature. Based on the proposed fixed-point formalization, in the next chapters we will focus on the verification of the error analysis between the real numbers and the floating-point and fixed-point algorithmic levels for digital filters and FFT algorithms. We also discuss the transitions from the floating-point and fixed-point algorithmic levels to hardware implementations for FFT algorithms.

Chapter 3

Error Analysis of Digital Filters in HOL

3.1 Introduction

Digital filters are a particularly important class of DSP (Digital Signal Processing) systems. A digital filter is a discrete time system that transforms a sequence of input numbers into another sequence of output, by means of a computational algorithm [39]. Digital filters are used in a wide variety of signal processing applications, such as spectrum analysis, digital image and speech processing, and pattern recognition. Due to their well-known advantages, digital filters are often replacing classical analog filters. The three distinct and most outstanding advantages of the digital filters are their flexibility, reliability, and modularity. Excellent methods have been developed to design these filters with desired characteristics. The design of a filter is the process of determination of a transfer function from a set of specifications given either in the frequency domain, or in the time domain, or for some applications, in both. The design of a digital filter starts from an ideal real specification. In a theoretical analysis of the digital filters, we generally assume that signal values and system coefficients are represented in the real number system and are expressed to an infinite precision. When implemented as a special-purpose digital hardware or as a computer algorithm, we must represent the signals and coefficients in some digital number system that must always be of a finite precision. Therefore, arithmetic operations must be carried out with an accuracy limited by this finite word length. There is a variety of types of arithmetic used in the implementation of digital systems. Among the most common are the floating-point and fixed-point. Here, all operands are represented by a special format or assigned a fixed word length and a fixed exponent, while the control structure and the operations of the ideal program remain unchanged. The transformation from the real to the floating-point and fixed-point forms is quite tedious and error-prone. On the implementation side, the fixed-point model of the algorithm has to be transformed into the best suited target description, either using a hardware description or a programming language.

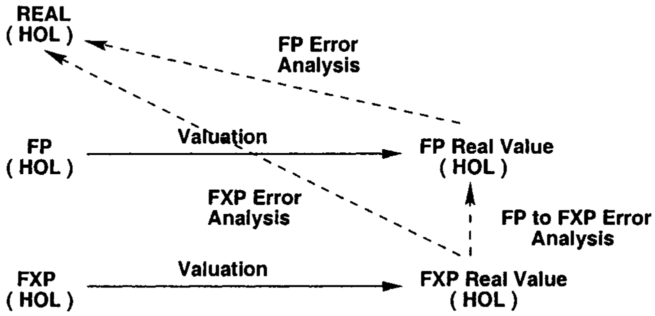


Figure 3.1: Error Analysis Approach

In this chapter we describe the error analysis of digital filters using the HOL theorem proving environment [23] based on the commutating diagram shown in Figure 3.1. Thereafter, we first model the ideal real filter specification and the corresponding floating-point and fixed-point implementations as predicates in higher-order logic. For this, we make use of existing theories in HOL on the construction of real numbers [27], the formalization of IEEE-754 standard based floating-point arithmetic [28, 29], and the formalization of fixed-point arithmetic described in Chapter 2. We use valuation functions to find the real values of the floating-point and fixed-point filter outputs and define the errors as the differences between these values and the corresponding output of the ideal real specification. Then we establish fundamental lemmas on the error analysis of the floating-point and fixed-point roundings and arithmetic operations against their abstract mathematical counterparts. Finally, we use these lemmas as a model to derive expressions for the accumulation of the roundoff error in parametric Lth-order digital filters, for each of the three basic forms of realization: direct, parallel, and cascade [59]. Using these forms, our verification methodology can be scaled up to any larger-order filter, either directly or by decomposing the design into a combination of internal sub-blocks. While the theoretical work on computing the errors due to finite precision effects has been extensively studied since the late sixties [47], it is for the first time in this thesis, that a formalization and proof of this analysis for digital filters is done using a mechanical theorem prover, here the HOL. Our results are found to be in a good agreement with the theoretical ones.

The rest of this chapter is organized as follows: Section 3.2 introduces the fundamental lemmas in HOL for the error analysis of the floating-point and fixed-point rounding and arithmetic operations. Section 3.3 describes the details of the error analysis in HOL of the class of linear difference equation digital filters implemented in the three basic forms of realization. Finally, Section 3.4 concludes the chapter.

3.2 Error Analysis Models

In this section we introduce the fundamental error analysis theorems [74, 20], and the corresponding lemmas in HOL for the floating-point [28, 29] and fixed-point arithmetics. These theorems are then used in the next sections as a model for the analysis of the roundoff error in digital filters.

3.2.1 Floating-Point Error Model

In analyzing the effects of floating-point roundoff, the effects of rounding will be represented multiplicatively. The following theorem is the most fundamental in the floating-point rounding-error theory [74, 20].

Theorem 1: If the real number x located within the floating-point range, is rounded to the closest floating-point number x_R, then

$$x_R = x(1+\delta), \text{ where } |\delta| \leq 2^{-p} \tag{3.1}$$

and p is the precision of the floating-point format.

In HOL, we proved this theorem in the IEEE single precision floating-point format for the case of rounding to nearest as follows:

```
Lemma 1: FLOAT_ROUND_RELATIVE_ERROR
⊢ normalizes x ⟹ ∃ e. abs (e) < (1 / 2 pow ((fracwidth X) + 1)) ∧
   (Val (float (round X To_nearest x)) = x * (1 + e))
```

where the function *normalizes* defines the criteria for an arbitrary real number to be in the normalized range of floating-point numbers [28], *fracwidth* extracts the fraction width parameter from the floating-point format X, *Val* is the floating-point valuation function, *float* is the bijection function that converts a triple of natural numbers into the floating-point type, and *round* is the floating-point rounding function [29].

To prove this theorem [20], we first proved the following lemma which locates a real number in a binade (the floating-point numbers between two adjacent powers of 2):

```
Lemma 2: REAL_IN_BINADE
⊢ normalizes x ⟹ ∃ j. j ≤ ((emax X) - 2) ∧
   (2 pow (j + 1) / 2 pow (bias X)) ≤ abs x ∧
   abs x < (2 pow (j + 2) / 2 pow (bias X))
```

where the function *emax* defines the maximum exponent in a given floating-point format, and *bias* defines the exponent bias in the floating-point format which is a constant used to make the exponent's range nonnegative. Using this lemma we can rewrite the general floating-point absolute error bound theorem (ERROR_BOUND_NORM_STRONG) developed in [29] as follows:

```
Lemma 3: ERROR_BOUND_NORM_STRONG_NORMALIZE
⊢ normalizes x ⟹
   ∃ j. abs (error x) ≤ (2 pow j / 2 pow (bias X + fracwidth X))
```

which states that if the absolute value of a real number is in the representable range of the normalized floating-point numbers, then the absolute value of the error is less than or equal to $2^j/2^{(bias\ X\ +\ fracwidth\ X)}$. The function *error*, defines the error resulting from rounding a real number to a floating-point value which is defined as follows [29]:

```
⊢def  error x = (Val (float (round X To_nearest x)) − x)
```

Since $(2^{(j+1)}\ /\ 2^{(bias\ X)}) \leq |x|$ for the real numbers in the normalized region as proved in Lemma 2, we have $(|error\ x|\ /\ |x|) \leq (2^j\ /\ 2^{(bias\ X\ +\ fracwidth\ X)})\ /$ $(2^{(j+1)}\ /\ 2^{(bias\ X)})$ or $(|error\ x|\ /\ |x|) \leq (1\ /\ 2^{((fracwidth\ X)\ +\ 1)})$. Finally, defining $e = (error\ x\ /\ x)$ will complete the proof of the floating-point relative error bound theorem as described in Lemma 1.

Next, we apply the floating-point relative rounding error analysis theorem (Theorem 1) to the verification of the arithmetic operations. The goal is to prove the following theorem in which floating-point arithmetic operations such as addition, subtraction, multiplication, and division are related to their abstract mathematical counterparts according to the corresponding errors.

Theorem 2: Let $*$ denote any of the floating-point operations +, -, × , /. Then

$$fl\ (x\ *\ y)\ =\ (x\ *\ y)(1\ +\ \delta),\quad \text{where } |\delta| \leq 2^{-p} \tag{3.2}$$

and p is the precision of the floating-point format. The notation *fl (.)* is used to denote that the operation is performed using the floating-point arithmetic.

To prove this theorem in HOL, we start from the already proved lemmas on absolute analysis of rounding error in floating-point arithmetic operations (`FLOAT_ADD ,FLOAT_SUB,FLOAT_MUL,FLOAT_DIV`) developed in [29]. We have converted these lemmas to the following relative error analysis version, using the relative error bound analysis of floating-point rounding (Lemma 1):

Lemma 4: FLOAT_ADD_RELATIVE

```
⊢ Finite a ∧ Finite b ∧ normalizes (Val a + Val b)
    ⟹ Finite (a + b) ∧ ∃ e. abs e ≤ (1 / 2 pow ((fracwidth X) + 1)) ∧
        (Val (a + b) = (Val a + Val b) * (1 + e))
```

Lemma 5: FLOAT_SUB_RELATIVE

```
⊢ Finite a ∧ Finite b ∧ normalizes (Val a − Val b)
    ⟹ Finite (a − b) ∧ ∃ e. abs e ≤ (1 / 2 pow ((fracwidth X) + 1)) ∧
        (Val (a − b) = (Val a − Val b) * (1 + e))
```

Lemma 6: FLOAT_MUL_RELATIVE

```
⊢ Finite a ∧ Finite b ∧ normalizes (Val a * Val b)
    ⟹ Finite (a * b) ∧ ∃ e. abs e ≤ (1 / 2 pow ((fracwidth X) + 1)) ∧
        (Val (a * b) = (Val a * Val b) * (1 + e))
```

Lemma 7: FLOAT_DIV_RELATIVE

```
⊢ Finite a ∧ Finite b ∧ ¬ Iszero b ∧ normalizes (Val a / Val b)
    ⟹ Finite (a / b) ∧ ∃ e. abs e ≤ (1 / 2 pow ((fracwidth X) + 1)) ∧
        (Val (a / b) = (Val a / Val b) * (1 + e))
```

where the function *Finite* defines the finiteness criteria for the floating-point numbers, and the function *Iszero* checks if a given floating-point number is equal to zero. Note that we use the conventional symbols for arithmetic operations on floating-point numbers using the operator overloading feature of HOL. The lemmas are composed of two parts. The first part is about the finiteness of the floating-point operation output. It states that for each pair of finite floating-point numbers, if the real result is in the representable range of normalized floating-point numbers, then the output result is also finite. For floating-point division, the second operand should be nonzero to avoid the division by zero. The second part of the lemmas states that the result of a floating-point operation is the exact result, perturbed by a relative error of bounded magnitude.

3.2.2 Fixed-Point Error Model

While the rounding error for the floating-point arithmetic enters into the system multiplicatively, it is an additive component for the fixed-point arithmetic. In this case the fundamental error analysis theorem can be stated as follows [74].

Theorem 3: If the real number x located in the range of the fixed-point numbers with format *X'*, is rounded to the closest fixed-point number x'_R, then

$$x'_R \; = \; x \; + \; \epsilon, \quad \text{where } |\epsilon| \; \leq \; 2^{-fracbits\;(X')} \tag{3.3}$$

and *fracbits* is a function that extracts the number of bits that are to the right of the binary point in the given fixed-point format.

This theorem is proved in HOL as follows:

```
Lemma 5: FXP_ROUND_ABSOLUTE_ERROR_BOUND
⊢  (validAttr X′) ∧ (representable X′ x) ⟹
   abs (Fxp_error X′ x) ≤ (1 / 2 pow (fracbits X′))
```

where the function *validAttr* defines the validity of the fixed-point format, *representable* defines the criteria for a real number to be in the representable range of the fixed-point format, and *Fxp_error* defines the fixed-point rounding error.

The verification of the fixed-point arithmetic operations using the *absolute* error analysis of the fixed-point rounding (Theorem 3) can be stated as in the following theorem in which the fixed-point arithmetic operations are related to their abstract mathematical counterparts according to the corresponding errors.

Theorem 4: Let $*$ denote any of the fixed-point operations +, -, × , /, with a given format *X'*. Then

$$fxp\,(x \; * \; y) \; = \; (x \; * \; y) \; + \; \epsilon, \quad \text{where } |\epsilon| \; \leq \; 2^{-fracbits\;(X')} \tag{3.4}$$

and the notation *fxp (.)* is used to denote that the operation is performed using the fixed-point arithmetic. This theorem is proved in HOL using the following lemmas:

Lemma 9: FXP_ADD_ABSOLUTE

```
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X′) ∧
   representable X′ (value a + value b) ⟹ (Isvalid (FxpAdd X′ a b)) ∧
   ∃ e. abs e ≤ inv (2 pow (fracbits X′)) ∧
   value (FxpAdd X′ a b) = (value a + value b) + e
```

Lemma 10: FXP_SUB_ABSOLUTE

```
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X′) ∧
   representable X′ (value a − value b) ⟹ (Isvalid (FxpSub X′ a b)) ∧
   ∃ e. abs e ≤ inv (2 pow (fracbits X′)) ∧
   value (FxpSub X′ a b) = (value a − value b) + e
```

Lemma 11: FXP_MUL_ABSOLUTE

```
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X′) ∧
   representable X′ (value a * value b) ⟹ (Isvalid (FxpMul X′ a b)) ∧
   ∃ e. abs e ≤ inv (2 pow (fracbits X′)) ∧
   value (FxpMul X′ a b) = (value a * value b) + e
```

Lemma 12: FXP_DIV_ABSOLUTE

```
⊢ (Isvalid a) ∧ (Isvalid b) ∧ ¬ (value b = 0) ∧ validAttr (X′) ∧
   representable X′ (value a / value b) ⟹ (Isvalid (FxpDiv X′ a b)) ∧
   ∃ e. abs e ≤ inv (2 pow (fracbits X′)) ∧
   value (FxpDiv X′ a b) = (value a / value b) + e
```

where the function *Isvalid* defines the validity of a fixed-point number, *value* is the fixed-point valuation function, and *FxpAdd, FxpSub, FxpMul,* and *FxpDiv* are the corresponding functions for fixed-point addition, subtraction, multiplication, and division operations, respectively. According to these lemmas, if the input fixed-point numbers and the output attributes are valid, then the result of fixed-point operations is valid. For fixed-point division, the second operand should be nonzero to avoid the division by zero. The result of the fixed-point operations is the exact result, perturbed by an absolute error of bounded magnitude.

3.3 Error Analysis of Digital Filters using HOL

In this section, the principal results for roundoff accumulation in digital filters using theorem proving are derived and summarized. We shall employ the models for floating- and fixed-point roundoff errors in HOL presented in the previous section. To illustrate our approach, we first considered the case of first- and second-order digital filters. Then, we extended this analysis to the general case of the direct form realization of a parametric L^{th}-order filter of which the first- and second-order filters are special cases. Finally, we applied our approach to the parallel and cascade forms. Using these forms, larger-order filters can be treated as a combination of first- and second-order filters. Then, the total error is computed by accumulating the error in all internal sub-filters. In the following, we will first describe in details the theory behind the analysis and then explain how each step of this analysis is performed in HOL.

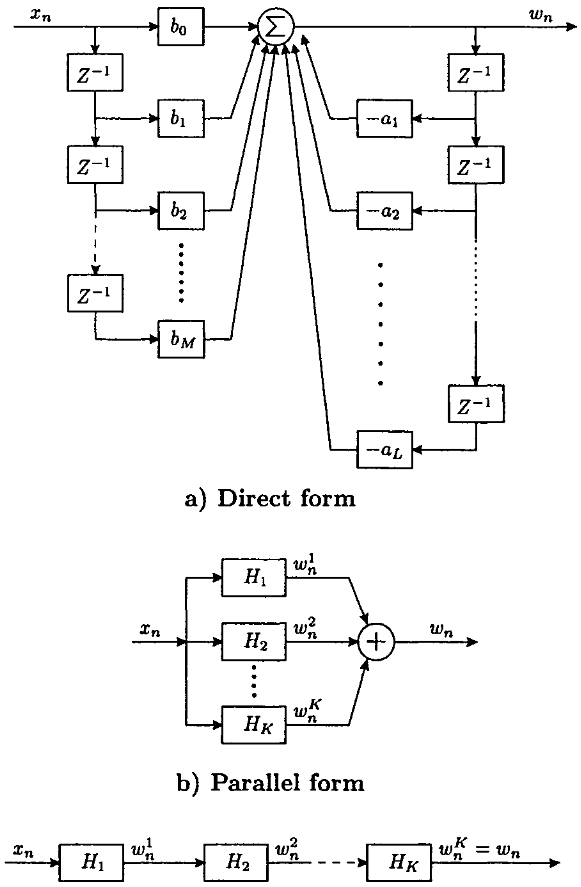
The class of digital filters considered in this paper is that of linear constant coefficient filters specified by the difference equation:

$$w_n = \sum_{i=0}^{M} b_i \, x_{n-i} - \sum_{i=1}^{L} a_i \, w_{n-i} \tag{3.5}$$

where $\{x_n\}$ is the input sequence and $\{w_n\}$ is the output sequence. L is the order of the filter, and M can be any positive number less than L. There are three basic forms of realizing a digital filter, namely the direct, parallel, and cascade forms (Figure 3.2) [59].

If the output sequence is calculated by using the equation (3.5), the digital filter is said to be realized in the direct form. Figure 3.2 (a) illustrates the direct form realization of the filter using the corresponding blocks for the addition, multiplication by a constant operations, and the delay element.

The implementation of a digital filter in the parallel form is shown in Figure 3.2 (b) in which the entire filter is visualized as the parallel connection of the simpler filters H_i of a lower order. In this case, K intermediate outputs $\{w_n^i\}$, $i = 1,2,\ldots,K$ are first calculated and then summed to form the total output $\{w_n\}$. Therefore, for the input sequence $\{x_n\}$ we have:



a) Direct form

b) Parallel form

c) Cascade form

Figure 3.2: Basic forms of digital filter realizations

$$w_n^i = f_i x_n + g_i x_{n-1} - c_i w_{n-1}^i - d_i w_{n-2}^i \tag{3.6}$$

where the parameters f_i, g_i, c_i, and d_i are obtained from the parameters a_i and b_i in equation (3.5) using the parallel expansion. The output of the entire filter w_n, is then related to w_n^i by:

$$w_n = w_n^1 + w_n^2 + \cdots + w_n^K \tag{3.7}$$

The implementation of a digital filter in the cascade form is shown in Figure 3.2 (c) in which the filter is visualized as a cascade of lower filters. From the input $\{x_n\}$, the intermediate output $\{w_n^1\}$ is first calculated, and then this is the input to the second filter. Continuing in this manner, the final output $w_n^K = w_n$ is calculated. Since the output of the ith section (w_n^i) is the input of the *(i+1)*th section, the following equation holds:

$$w_n^{i+1} = w_n^i + k_i w_{n-1}^i + l_i w_{n-2}^i - c_i w_{n-1}^{i+1} - d_i w_{n-2}^{i+1} \tag{3.8}$$

where the parameters k_i, l_i, c_i, and d_i are obtained from the parameters a_i and b_i in equation (3.5) using the serial expansion.

There are three common sources of errors associated with the filter of the equation (3.5), namely [47]:

1. **input quantization:** caused by the quantization of the input signal $\{x_n\}$ into a set of discrete levels.

2. **coefficient inaccuracy:** caused by the representation of the filter coefficients $\{a_k\}$ and $\{b_k\}$ by a finite word length.

3. **round-off accumulation:** caused by the accumulation of roundoff errors at arithmetic operations.

Our concern in this thesis is round-off accumulation effect only. However, the results can be extended by minor modification to consider other sources of error. Therefore, for the digital filter of the equation (3.5) the actual computed output reference is in general different from $\{w_n\}$. We denote the actual floating-point and fixed-point outputs by $\{y_n\}$ and $\{v_n\}$, respectively. Then, we define the corresponding errors at the nth output sample as:

$$e_n = y_n - w_n \tag{3.9}$$

$$e_n' = v_n - w_n \tag{3.10}$$

$$e_n'' = v_n - y_n \tag{3.11}$$

where e_n and e_n' are defined as the errors between the actual floating-point and fixed-point implementations and the ideal real specification, respectively. e_n'' is the error in the transition from the floating-point to fixed-point levels.

3.3.1 First-Order Filter

To illustrate our approach for the analysis of roundoff errors with floating- and fixed-point arithmetic, let us consider a first-order filter. Let x_n, w_n, and a denote the ideal real input signal, output response, and the coefficient of the filter, that is, the filter parameters with no roundoff noise and x'_n, y_n, a' and x''_n, v_n, a'' denote the corresponding actual floating-point and fixed-point filter parameters in the presence of roundoff noise, respectively. Then we can write:

$$w_n = aw_{n-1} + x_n \tag{3.12}$$

The corresponding computed floating- and fixed-point outputs are:

$$y_n = fl\ [a'y_{n-1} + x'_n] \tag{3.13}$$

$$v_n = fxp\ [a''v_{n-1} + x''_n] \tag{3.14}$$

The notations fl (.) and fxp (.) are used to denote that the operations are performed using the floating- and fixed-point arithmetics, respectively. In HOL, we specified the first-order digital filter in real, floating-, and fixed-point abstraction levels, as predicates in higher-order logic. The corresponding codes are as follows.

```
⊢def First_Order_Filter_Ideal_Spec a x w =
     ∀n. w n = a * w (n - 1) + x n

⊢def First_Order_Filter_Float_Imp a′ x′ y =
     ∀n. y n = a′ * y (n - 1) + x′ n

⊢def First_Order_Filter_Fxp_Imp X a″ x″ v =
     ∀n. v n = (FxpAdd X (FxpMul X a″ (v (n - 1))) (x″ n))
```

The calculation of Equation (3.13) is to be performed in the following manner. First the product $a'y_{n-1}$ is calculated separately. Then it is added to x'_n to obtain y_n. Similar discussion can be applied for the calculation of the fixed-point output v_n according to the Equation (3.14). Following Sandberg [66], a flowgraph as given in Figure 3.3 may be drawn by using the fundamental error analysis theorems on floating-point and fixed-point arithmetic operations introduced in Section 3.2 as given in Equations (3.2) and (3.4).

Formally, a flowgraph is a network of directed branches that connect at nodes. Associated with each node is a variable or node value. Each branch has an input signal and an output signal with a direction indicated by an arrowhead on it. In a linear flowgraph, the output of a branch is a linear transformation of the input to the branch. The simplest examples are constant multipliers and adders, i.e., when the output of the branch is simply a multiplication or an addition of the input to the branch with a constant value, which are the only classes we consider in this paper. The linear operation represented by the branch is typically indicated next to the arrowhead showing the direction of the branch. For the case of a constant multiplier and adder, the constant is simply shown next to the arrowhead. When an explicit indication of the branch operation is omitted, this indicates a branch transmittance of unity, or identity transformation. By definition, the value at each node in a flowgraph is the sum of the outputs of all the branches entering the node. To complete the definition of the flowgraph notation, we define two special types of nodes. *(1) Source nodes* that have no entering branches. They are used to represent the injection of the external inputs or signal sources into a flowgraph. *(2) Sink nodes* that have only entering branches. They are used to extract the outputs from a flowgraph [59]. Note that we have used one flowgraph to represent both the floating-point and fixed-point cases, simultaneously. For floating-point errors, the branch operations are interpreted as constant multiplications, while for fixed-point errors the branch operations are interpreted as constant additions.

The quantities ϵ_n and ξ_n are errors caused by roundoff at each floating-point arithmetic step. The corresponding error quantities for fixed-point roundoff are ϵ'_n and ξ'_n.

Therefore the actual y_n is seen to be given explicitly by

$$y_n = [ay_{n-1}(1 + \epsilon_n) + x_n](1 + \xi_n) \tag{3.15}$$

In HOL, we established the following lemma to compute the real value of the floating-point output for the first-order filter according to the Equation (3.15).

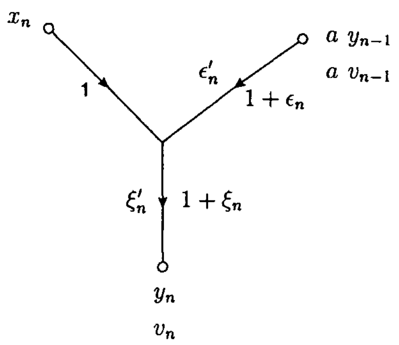


Figure 3.3: Error flowgraph for the first-order filter

```
Lemma 13: FIRST_ORDER_FILTER_FLOAT_OUTPUT_VALUE
⊢  First_Order_Filter_Float_Imp a′ x′ y ⟹
   ∃ e1 e2. abs e1 ≤ (1 / 2 pow 24) ∧ abs e2 ≤ (1 / 2 pow 24) ∧
   (Val (y n) = (Val (a′) * Val (y (n − 1)) * (1 + e1) +
   Val (x′ n)) * (1 + e2))
```

Similarly, the actual fixed-point output v_n is given explicitly by

$$v_n = [av_{n-1} + \epsilon'_n + x_n] + \xi'_n \tag{3.16}$$

and the corresponding lemma is established in HOL as follows:

```
Lemma 14: FIRST_ORDER_FILTER_FXP_OUTPUT_VALUE
⊢  First_Order_Filter_Fxp_Imp X a″ x″ v ⟹
   ∃ e1 e2. abs e1 ≤ inv (2 pow (fracbits X)) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   value (FxpAdd X (FxpMul X a″ (v (n − 1))) (x″ n)) =
   ((value (a″) * value (v (n − 1)) + e1) + value (x″ n)) + e2
```

For error analysis, we need to calculate the y_n and v_n sequences from Equations (3.15) and (3.16), and compare them with the ideal output sequence w_n specified by the Equation (3.12) to obtain the corresponding errors e_n, e'_n, and e''_n, according to the Equations (3.9), (3.10), and (3.11). Therefore, the difference equations for the errors between different levels showing the accumulation of roundoff error are derived as follows:

1) Floating-Point Error Analysis:

$$e_n - a\ e_{n-1} = a\ y_{n-1}(\epsilon_n + \xi_n + \epsilon_n \xi_n) + x_n \xi_n \tag{3.17}$$

To prove this theorem in HOL, we first defined the error as the difference between the output of the real filter specification, and the corresponding real value of the floating-point filter implementation (*Float_Error*). Then, we established the following lemma for the accumulation of round-off error in floating-point realization of the first-order filter, according to the Equation (3.17).

```
Lemma 15: FIRST_ORDER_FILTER_FLOAT_TO_REAL_THM
⊢  First_Order_Filter_Ideal_Spec a x w ∧
   First_Order_Filter_Float_Imp a′ x′ y ⟹
   ∃ e1 e2. abs e1 ≤ (1 / 2 pow 24) ∧ abs e2 ≤ (1 / 2 pow 24) ∧
   (Float_Error n − a * Float_Error (n − 1) =
   a * Val (y (n − 1)) * (e1 + e2 + e1 * e2) + (x n) * e2)
```

2) Fixed-Point Error Analysis:

$$e'_n - a\ e'_{n-1} = \epsilon'_n + \xi'_n \tag{3.18}$$

To prove this theorem in HOL, we first defined the error as the difference between the output of the real filter specification, and the corresponding real value of the fixed-point filter implementation (*Fxp_Error*). Then, we established the following lemma for the accumulation of round-off error in fixed-point realization of the first-order filter, according to the Equation (3.18).

```
Lemma 16: FIRST_ORDER_FILTER_FXP_TO_REAL_THM
⊢  First_Order_Filter_Ideal_Spec a x w ∧
   First_Order_Filter_Fxp_Imp X a″ x″ v ⟹
   ∃ e1 e2. abs e1 ≤ inv (2 pow (fracbits X)) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   (Fxp_Error n − a * Fxp_Error (n − 1) = e1 + e2)
```

3) Floating- to Fixed-Point Error Analysis:

$$e''_n - a\ e''_{n-1} = \epsilon'_n + \xi'_n - a\ y_{n-1}(\epsilon_n + \xi_n + \epsilon_n \xi_n) - x_n \xi_n \tag{3.19}$$

To prove this theorem in HOL, we first defined the error as the difference between the real value of the output of the fixed-point filter implementation, and the corresponding real value of the floating-point filter implementation (*Float_Fxp_Error*). Then, we established the following lemma for the accumulation of round-off error in transition from floating-point to fixed-point levels of the first-order filter, according to the Equation (3.19).

```
Lemma 17: FIRST_ORDER_FILTER_FXP_TO_FLOAT_THM
⊢  First_Order_Filter_Ideal_Spec a x w ∧
   First_Order_Filter_Float_Imp a′ x′ y ∧
   First_Order_Filter_Fxp_Imp X a″ x″ v ⟹
   ∃ e1 e2 e3 e4. abs e1 ≤ inv (2 pow (fracbits X)) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   abs e3 ≤ (1 / 2 pow 24) ∧ abs e4 ≤ (1 / 2 pow 24) ∧
   (Float_Fxp_Error n − a * Float_Fxp_Error (n − 1) =
   e1 + e2 − a * Val (y (n − 1)) * (e3 + e4 + e3 * e4) − (x n) * e4)
```

We proved these lemmas using the fundamental error analysis lemmas (Lemmas 4,5, and 6 for floating-point, and Lemmas 9,10, and 11 for fixed-point), based on the error models presented in Section 3.2.

3.3.2 Second-Order Filter

A second-order filter is specified by

$$w_n = b_0 x_n - (a_1 w_{n-1} + a_2 w_{n-2}) \tag{3.20}$$

The corresponding computed floating- and fixed-point outputs are

$$y_n = fl\ [b_0 x_n - (a_1 y_{n-1} + a_2 y_{n-2})] \tag{3.21}$$

$$v_n = fxp\ [b_0 x_n - (a_1 v_{n-1} + a_2 v_{n-2})] \tag{3.22}$$

In HOL, we specified the second-order digital filter in real, floating-, and fixed-point abstraction levels, as predicates in higher-order logic. The corresponding codes are as follows.

```
⊢def Second_Order_Filter_Ideal_Spec a b x w =
      ∀n. w n = (b 0 * x n − (a 1 * w (n − 1) + a 2 * w (n − 2)))
```

```
⊢def Second_Order_Filter_Float_Imp a′ b′ x′ y =
      ∀n. y n = (b′ 0 * x′ n − (a′ 1 * y (n − 1) + a′ 2 * y (n − 2)))
```

```
⊢def Second_Order_Filter_Fxp_Imp X a″ b″ x″ v =
      ∀n. v n = (FxpSub X (FxpMul X (b″ 0) (x″ n))
                  (FxpAdd X (FxpMul X (a″ 1) (v (n − 1)))
                  (FxpMul X (a″ 2) (v (n − 2)))))
```

The calculation of Equation (3.21) is performed in the following manner. First, the products $a_1 y_{n-1}$, $a_2 y_{n-2}$, and $b_0 x_n$ are calculated separately. Then $a_1 y_{n-1}$ and $a_2 y_{n-2}$ are added. Finally, this sum is subtracted from $b_0 x_n$ to obtain y_n. Similar discussion can be applied for the calculation of the fixed-point output v_n according to the Equation (3.22). A flowgraph for the error of this case is drawn in Figure 3.4. The quantities $\delta_{n,0}$, $\epsilon_{n,1}$, $\epsilon_{n,2}$, η_n, ξ_n are errors caused by floating-point roundoff at each arithmetic step. The corresponding error quantities for fixed-point roundoff are $\delta'_{n,0}$, $\epsilon'_{n,1}$, $\epsilon'_{n,2}$, η'_n, ξ'_n.

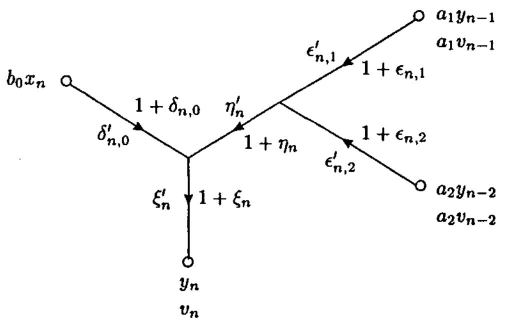


Figure 3.4: Error flowgraph for the second-order filter

Therefore the actual y_n is seen to be given explicitly by

$$y_n = b_0 \; \theta_{n,0} \; x_n \; - \sum_{k=1}^{2} a_k \; \phi_{n,k} \; y_{n-k} \tag{3.23}$$

where

$$\theta_{n,0} = (1 + \delta_{n,0})(1 + \xi_n)$$
$$\phi_{n,1} = (1 + \epsilon_{n,1})(1 + \eta_n)(1 + \xi_n)$$
$$\phi_{n,2} = (1 + \epsilon_{n,2})(1 + \eta_n)(1 + \xi_n)$$

In HOL, we established the following lemma to compute the real value of the floating-point output for the second-order filter according to the Equation (3.23).

```
Lemma 18: SECOND_ORDER_FILTER_FLOAT_OUTPUT_VALUE
⊢  Second_Order_Filter_Float_Imp a′ b′ x′ y ⟹
   ∃ t f. Val (y n) = Val (b′ 0) * (t 0) * Val (x′ n) −
   sum (1,2) (λ i. Val (a′ i) * (f i) * Val (y (n − i))) ∧
   ∃ e1 e2 e3 e4 e5. abs e1 ≤ (1 / 2 pow 24) ∧
   abs e2 ≤ (1 / 2 pow 24) ∧ abs e3 ≤ (1 / 2 pow 24) ∧
   abs e4 ≤ (1 / 2 pow 24) ∧ abs e5 ≤ (1 / 2 pow 24) ∧
   t 0 = (1 + e1) * (1 + e5) ∧
   f 1 = (1 + e2) * (1 + e4) * (1 + e5) ∧
   f 2 = (1 + e3) * (1 + e4) * (1 + e5)
```

Similarly, the actual fixed-point output v_n is given explicitly by

$$v_n = [b_0 x_n - (a_1 v_{n-1} + a_2 v_{n-2})] + \delta'_{n,0} + \epsilon'_{n,1} + \epsilon'_{n,2} + \eta'_n + \xi'_n \tag{3.24}$$

and the corresponding lemma is established in HOL as follows:

```
Lemma 19: SECOND_ORDER_FILTER_FXP_OUTPUT_VALUE
⊢  Second_Order_Filter_Fxp_Imp X a″ b″ x″ v ⟹
   ∃ e1 e2 e3 e4 e5. abs e1 ≤ inv (2 pow (fracbits X)) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   abs e3 ≤ inv (2 pow (fracbits X)) ∧
   abs e4 ≤ inv (2 pow (fracbits X)) ∧
   abs e5 ≤ inv (2 pow (fracbits X)) ∧
   value (v n) = value (b″ 0) * value (x″ n) −
   (value (a″ 1) * value (v (n − 1)) +
   value (a″ 2) * value (v (n − 2))) +
   e1 + e2 + e3 + e4 + e5
```

For error analysis, we need to calculate the y_n and v_n sequences from Equations (3.23) and (3.24), and compare them with the ideal output sequence w_n specified by the Equation (3.20) to obtain the corresponding errors e_n, e'_n, and e''_n, according to the Equations (3.9), (3.10), and (3.11). Therefore, the difference equations for the errors between different levels showing the accumulation of roundoff error are derived as follows:

1) Floating-Point Error Analysis:

$$e_n \ + \ a_1 \ e_{n-1} \ + \ a_2 \ e_{n-2} \ = b_0 x_n(\theta_{n,0} - 1) \ - [a_1 y_{n-1} \ (\phi_{n,1} - 1) + a_2 y_{n-2} \ (\phi_{n,2} - 1)] \tag{3.25}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the second-order filter, according to the Equation (3.25).

```
Lemma 20: SECOND_ORDER_FILTER_FLOAT_TO_REAL_THM
⊢  Second_Order_Filter_Ideal_Spec a b x w ∧
   Second_Order_Filter_Float_Imp a' b' x' y ⟹
   ∃ t f. Float_Error n + a 1 * Float_Error (n - 1) +
   a 2 * Float_Error (n - 2) = b 0 * x n * (t 0 - 1) -
   (a 1 * Val (y (n - 1)) * (f 1 - 1) +
   a 2 * Val (y (n - 2)) * (f 2 - 1)) ∧
   ∃ e1 e2 e3 e4 e5. abs e1 ≤ (1 / 2 pow 24) ∧
   abs e2 ≤ (1 / 2 pow 24) ∧ abs e3 ≤ (1 / 2 pow 24) ∧
   abs e4 ≤ (1 / 2 pow 24) ∧ abs e5 ≤ (1 / 2 pow 24) ∧
   t 0 = (1 + e1) * (1 + e5) ∧
   f 1 = (1 + e2) * (1 + e4) * (1 + e5) ∧
   f 2 = (1 + e3) * (1 + e4) * (1 + e5)
```

2) Fixed-Point Error Analysis:

$$e'_n \ + \ a_1 \ e'_{n-1} \ + \ a_2 \ e'_{n-2} \ = \ \delta'_{n,0} \ + \ \epsilon'_{n,1} \ + \ \epsilon'_{n,2} \ + \ \eta'_n \ + \ \xi'_n \tag{3.26}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in fixed-point realization of the second-order filter, according to the Equation (3.26).

```
Lemma 21: SECOND_ORDER_FILTER_FXP_TO_REAL_THM
⊢  Second_Order_Filter_Ideal_Spec a b x w ∧
   Second_Order_Filter_Fxp_Imp X a'' b'' x'' v ⟹
   ∃ e1 e2 e3 e4 e5. abs e1 ≤ (inv (&2 pow (fracbits X))) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   abs e3 ≤ inv (2 pow (fracbits X)) ∧
   abs e4 ≤ inv (2 pow (fracbits X)) ∧
   abs e5 ≤ inv (2 pow (fracbits X)) ∧
   Fxp_Error n + a 1 * Fxp_Error (n - 1) + a 2 * Fxp_Error (n - 2) =
   e1 + e2 + e3 + e4 + e5
```

3) Floating- to Fixed-Point Error Analysis:

$$e''_n + a_1 e''_{n-1} + a_2 e''_{n-2} = \delta'_{n,0} + \epsilon'_{n,1} + \epsilon'_{n,2} + \eta'_n + \xi'_n - \quad (3.27)$$

$$b_0 x_n(\theta_{n,0} - 1) + a_1 y_{n-1} (\phi_{n,1} - 1) + a_2 y_{n-2} (\phi_{n,2} - 1)$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in transition from floating-point to fixed-point levels of the second-order filter, according to the Equation (3.27).

```
Lemma 22: SECOND_ORDER_FILTER_FXP_TO_FLOAT_THM
⊢  Second_Order_Filter_Ideal_Spec a b x w ∧
   Second_Order_Filter_Float_Imp a′ b′ x′ y ∧
   Second_Order_Filter_Fxp_Imp X a″ b″ x″ v ⟹
   ∃ t f e1′ e2′ e3′ e4′ e5′.
   abs e1 ≤ inv (2 pow (fracbits X)) ∧
   abs e2 ≤ inv (2 pow (fracbits X)) ∧
   abs e3 ≤ inv (2 pow (fracbits X)) ∧
   abs e4 ≤ inv (2 pow (fracbits X)) ∧
   abs e5 ≤ inv (2 pow (fracbits X)) ∧
   (Float_Fxp_Error n + a 1 * Float_Fxp_Error (n − 1) +
   a 2 * Float_Fxp_Error (n − 2) =
   e1′ + e2′ + e3′ + e4′ + e5′ − b 0 * x n * (t 0 − 1) +
   (a 1 * Val (y (n − 1)) * (f 1 − 1) +
   a 2 * Val (y (n − 2)) * (f 2 − 1))) ∧
   ∃ e1 e2 e3 e4 e5. abs e1 ≤ (1 / 2 pow 24) ∧
   abs e2 ≤ (1 / 2 pow 24) ∧ abs e3 ≤ (1 / 2 pow 24) ∧
   abs e4 ≤ (1 / 2 pow 24) ∧ abs e5 ≤ (1 / 2 pow 24) ∧
   t 0 = (1 + e1) * (1 + e5) ∧
   f 1 = (1 + e2) * (1 + e4) * (1 + e5) ∧
   f 2 = (1 + e3) * (1 + e4) * (1 + e5)
```

We proved these lemmas using the fundamental error analysis lemmas (Lemmas 4,5, and 6 for floating-point, and Lemmas 9,10, and 11 for fixed-point), based on the error models presented in Section 3.2.

3.3.3 Lth-Order Filter (Direct Form)

The direct form realization of a parametric Lth-order filter is specified by Equation (3.5). The corresponding computed floating- and fixed-point outputs are

$$y_n \;=\; fl\,(\sum_{k=0}^{M} b_k\; x_{n-k} \;-\; \sum_{k=1}^{L} a_k\; y_{n-k}) \tag{3.28}$$

and

$$v_n \; = \; fxp\,(\sum_{k=0}^{M} b_k \; x_{n-k} \; - \; \sum_{k=1}^{L} a_k \; v_{n-k}) \tag{3.29}$$

In HOL, we specified the direct form realization of a parametric Lth-order digital filter in real, floating-, and fixed-point abstraction levels, as predicates in higher-order logic. The real specification is defined in HOL using Equation (3.5). For this we used the expression *sum* (m, n) f denoting $\sum_{i\,=\,m}^{m+n-1} f(i)$, which is a function available in the HOL real library [27] and defines the finite summation on the real numbers. The floating- and fixed-point specifications are defined in HOL according to the Equations (3.28) and (3.29). For these cases, we defined similar functions for finite summation on the floating-point (*float_sum*) and fixed-point (*fxp_sum*) numbers, using recursive definition in HOL. The corresponding codes in HOL are as follows.

```
⊢def  L_Order_Filter_Direct_Form_Ideal_Spec a b x w M L =
      ∀n. w n = sum (0,SUC M) (λ i. b i * x (n - i)) -
                  sum (1,L) (λ i. a i * w (n - i))

⊢def  ∀ f n m. (float_sum (n,0) f = float (0,0,0)) ∧
                (float_sum (n,SUC m) f = float_sum (n,m) f + f (n + m))

⊢def  L_Order_Filter_Direct_Form_Float_Imp a′ b′ x′ y M L =
      ∀ n. y n = float_sum (0,SUC M) (λ i. b′ i * x′ (n - i)) -
                  float_sum (1,L) (λ i. a′ i * y (n - i))


⊢def  ∀ X f n m. (fxp_sum (n,0) X f =
                  (fxp (WORD (REPLICATE (streamlength X) F),X))) ∧
                  (fxp_sum (n,SUC m) X f =
                  FxpAdd X (fxp_sum (n,m) X f) f (n+m))

⊢def  L_Order_Filter_Direct_Form_Fxp_Imp X a″ b″ v M L =
      ∀ n. v n = FxpSub X
                   (fxp_sum (0,SUC M) X (λ i. FxpMul X b″ i x″ (n - i)))
                   (fxp_sum (1,L) X (λ i. FxpMul X a i y (n - i)))
```

The calculation of Equation (3.28) is to be performed in the following manner. First, the output products $a_k\ y_{n-k}$, $k = 1, 2, ..., L$ are calculated separately and then summed. Next, the same is done for the input products $b_k\ x_{n-k}$, $k = 0, 1, ..., M$. Finally, the output summation is subtracted from the input one to obtain the main floating-point output y_n. Similar discussion can be applied for the calculation of the fixed-point output v_n according to the Equation (3.29). The corresponding flowgraph showing the effect of roundoff error using the fundamental error analysis theorems (Theorems 2 and 4) according to the Equations (3.2) and (3.4), is given by Figure 3.5 which also indicates the order of the calculation. The quantities $\delta_{n,k}$, $k = 0, 1, ..., M$, $\epsilon_{n,k}$, $k = 1, 2, ..., L$, $\zeta_{n,k}$, $k = 1, 2, ..., M$, $\eta_{n,k}$, $k = 2, 3, ..., L$, and ξ_n are errors caused by floating-point roundoff at each arithmetic step. The corresponding error quantities for fixed-point roundoff are $\delta'_{n,k}$, $k = 0, 1, ..., M$, $\epsilon'_{n,k}$, $k = 1, 2, ..., L$, $\zeta'_{n,k}$, $k = 1, 2, ..., M$, $\eta'_{n,k}$, $k = 2, 3, ..., L$, and ξ'_n.

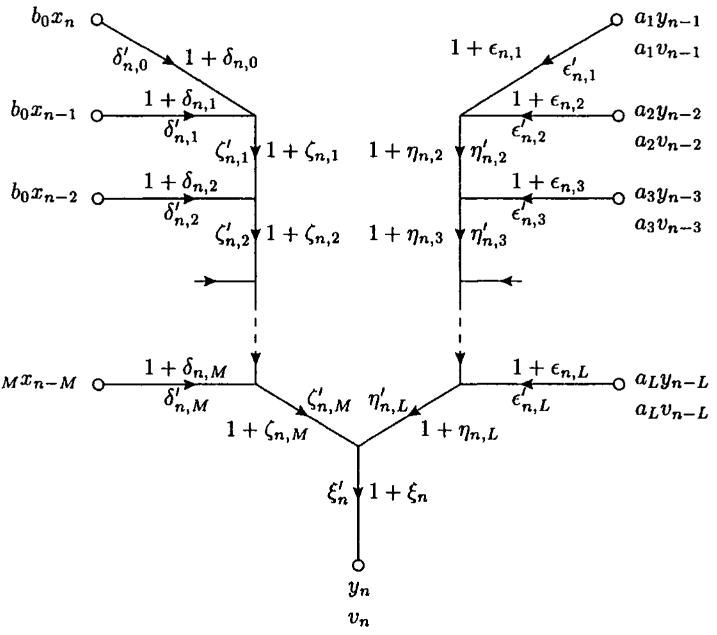


Figure 3.5: Error flowgraph for Lth-order filter (Direct form)

Therefore, the actual floating-point output y_n is seen to be given explicitly by:

$$y_n = \sum_{k=0}^{M} b_k\ \theta_{n,k}\ x_{n-k}\ -\ \sum_{k=1}^{L} a_k\ \phi_{n,k}\ y_{n-k} \tag{3.30}$$

where

$$\begin{aligned}
\theta_{n,0} &= (1+\xi_n)(1+\delta_{n,0})\prod_{i=1}^{M}(1+\zeta_{n,i}) \\
\theta_{n,j} &= (1+\xi_n)(1+\delta_{n,j})\prod_{i=j}^{M}(1+\zeta_{n,i}) \quad j \;=\; 1,2,...,M \\
\phi_{n,1} &= (1+\xi_n)(1+\epsilon_{n,1})\prod_{i=2}^{L}(1+\eta_{n,i}) \\
\phi_{n,j} &= (1+\xi_n)(1+\epsilon_{n,j})\prod_{i=j}^{L}(1+\eta_{n,i}) \quad j \;=\; 2,3,...,L
\end{aligned}$$

In HOL, we first defined finite multiplication on the real numbers recursively as the expression *mul* (m, n) f denoting $\prod_{i\,=\,m}^{m+n-1} f(i)$ as follows:

```
⊢def  ∀ f n m. (mul (n,0) f = 1) ∧
               (mul (n,SUC m) f = mul (n,m) f * f (n + m))
```

Then, we established the following lemma to compute the real value of the floating-point filter output for the direct form of realization according to the Equations (3.30).

```
Lemma 23: L_ORDER_FILTER_DIRECT_FORM_FLOAT_OUTPUT_VALUE
⊢  L_Order_Filter_Direct_Form_Float_Imp a' b' x' y M L ⟹
   ∃ t f. (Val (y n) = (if L = 0 then
   sum (0,SUC M) (λ i. (Val (b' i) * t i * Val (x' (n - i))))
   else sum (0,SUC M) (λ i. (Val (b' i) * t i * Val (x' (n - i)))) -
   sum (1,L) (λ i. (Val (a' i) * f i * Val (y (n - i)))))) ∧

   ∃ k d p e z. abs k ≤ (1 / 2 pow 24) ∧
   (∀ i. (i ≤ M) ⟹  (abs (d i) ≤ (1 / 2 pow 24))) ∧
   (∀ i. (i ≤ M) ⟹  (abs (p i) ≤ (1 / 2 pow 24))) ∧
   (∀ i. (i ≤ L) ⟹  (abs (e i) ≤ (1 / 2 pow 24))) ∧
   (∀ i. (i ≤ L) ⟹  (abs (z i) ≤ (1 / 2 pow 24))) ∧
   (t 0 = (1 + k) * (1 + d 0) * (mul (1,M) (λ i. (1 + p i)))) ∧
   (∀ j. (1 ≤ j ∧ j ≤ M) ⟹
   (t j = (1 + k) * (1 + d j) *
   (mul (j,(M - (j - 1))) (λ j. (1 + p j))))) ∧
   (f 1 = (1 + k) * (1 + e 1) * (mul (2,(L - 1)) (λ i. (1 + z i)))) ∧
   (∀ j. (2 ≤ j ∧ j ≤ L) ⟹
   (f j = (1 + k) * (1 + e j) * (mul (j,(L - j + 1)) (λ j. (1 + z j)))))
```

Similarly, the actual fixed-point output v_n is given explicitly by

$$v_n = \sum_{k=0}^{M} b_k\ x_{n-k} - \sum_{k=1}^{L} a_k\ v_{n-k} + \sum_{k=0}^{M} \delta'_{n,k} + \sum_{k=1}^{M} \zeta'_{n,k} + \sum_{k=1}^{L} \epsilon'_{n,k} + \sum_{k=2}^{L} \eta'_{n,k} + \xi'_n \tag{3.31}$$

and the corresponding lemma is established in HOL as follows:

```
Lemma 24: L_ORDER_FILTER_DIRECT_FORM_FXP_OUTPUT_VALUE_EXPAND
⊢ L_Order_Filter_Direct_Form_Fxp_Imp X a″ b″ x″ v M L ⟹
   ∃ k d p e z. abs k ≤ (inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ M) ⟹ abs (d i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ M) ⟹ abs (p i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ L) ⟹ abs (e i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ L) ⟹ abs (z i) ≤ inv (2 pow (fracbits X))) ∧
   (value (v n) = if (L = 0) then
   sum (0,SUC M) (λ i. value (b″ i) * value (x″ (n - i))) +
   sum (0,SUC M) (λ i. d i) + sum (1,M) (λ j. p j) + k else
   sum (0,SUC M) (λ i. value (b″ i) * value (x″ (n - i))) -
   sum (1,L) (λ i. value (a″ i) * value (v (n - i))) +
   sum (0,SUC M) (λ i. d i) + sum (1,M) (λ j. p j) +
   sum (1,L) (λ i. e i) + sum (2,(L - 1)) (λ j. z j) + k)
```

For error analysis, we need to calculate the y_n and v_n sequences from Equations (3.30) and (3.31), and compare them with the ideal output sequence w_n specified by the Equation (3.5) to obtain the corresponding errors e_n, e'_n, and e''_n, according to the Equations (3.9), (3.10), and (3.11). Therefore, the difference equations for the errors between different levels showing the accumulation of roundoff error are derived as follows:

1) Floating-Point Error Analysis:

$$e_n\ +\ \sum_{k=1}^{L} a_k\ e_{n-k} = \sum_{k=0}^{M} b_k\ (\theta_{n,k} - 1)\ x_{n-k}\ -\ \sum_{k=1}^{L} a_k\ (\phi_{n,k} - 1)\ y_{n-k} \tag{3.32}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the direct form filter, according to the Equation (3.32).

```
Lemma 25: L_ORDER_FILTER_DIRECT_FORM_FLOAT_TO_REAL_THM
⊢ L_Order_Filter_Direct_Form_Ideal_Spec a b x w M L ∧
  L_Order_Filter_Direct_Form_Float_Imp a′ b′ x′ y M L ⟹
  ∃ t f. (if L = 0 then (Float_Error n =
  sum (0,SUC M) (λ i. Val (b′ i) * (t i − 1) * Val (x′ (n − i)))) else
  (((Float_Error n) + sum (1,L) (λ i. a i * (Float_Error (n − i))) =
  sum (0,SUC M) (λ i. Val (b′ i) * (t i − 1) * Val (x′ (n − i))) −
  sum (1,L) (λ i. Val (a′ i) * (f i − 1) * Val (y (n − i)))))) ∧
  ∃ k d p e z. (abs k ≤ (1 / 2 pow 24)) ∧
  (∀ i. (i ≤ M) ⟹  (abs (d i) ≤ (1 / 2 pow 24))) ∧
  (∀ i. (i ≤ M) ⟹  (abs (p i) ≤ (1 / 2 pow 24))) ∧
  (∀ i. (i ≤ L) ⟹  (abs (e i) ≤ (1 / 2 pow 24))) ∧
  (∀ i. (i ≤ L) ⟹  (abs (z i) ≤ (1 / 2 pow 24))) ∧
  (t 0 = (1 + k) * (1 + d 0) * (mul (1,M) (λ i. (1 + p i)))) ∧
  (∀ j. (1 ≤ j ∧ j ≤ M) ⟹
  (t j = (1 + k) * (1 + d j) *
  (mul (j,(M − (j − 1))) (λ j. (1 + p j))))) ∧
  (f 1 = (1 + k) * (1 + e 1) * (mul (2,(L − 1)) (λ i. (1 + z i)))) ∧
  (∀ j. (2 ≤ j ∧ j ≤ L) ⟹
  (f j = (1 + k) * (1 + e j) * (mul (j,(L − j + 1)) (λ j. (1 + z j)))))
```

2) Fixed-Point Error Analysis:

$$e'_n + \sum_{k=1}^{L} a_k\, e'_{n-k} = \sum_{k=0}^{M} \delta'_{n,k} + \sum_{k=1}^{M} \zeta'_{n,k} + \sum_{k=1}^{L} \epsilon'_{n,k} + \sum_{k=2}^{L} \eta'_{n,k} + \xi'_n \qquad (3.33)$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in fixed-point realization of the direct form filter, according to the Equation (3.33).

```
Lemma 26: L_ORDER_FILTER_DIRECT_FORM_FXP_TO_REAL_THM
⊢  L_Order_Filter_Direct_Form_Ideal_Spec a b x w M L ∧
   L_Order_Filter_Direct_Form_Fxp_Imp X a″ b″ x″ v M L ⟹
   ∃ k d p e z. abs k ≤ inv (2 pow (fracbits X)) ∧
   (∀ i. (i ≤ M) ⟹  abs (d i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ M) ⟹  abs (p i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ L) ⟹  abs (e i) ≤ inv (2 pow (fracbits X))) ∧
   (∀ i. (i ≤ L) ⟹  abs (z i) ≤ inv (2 pow (fracbits X))) ∧
   (if (L = 0) then (Fxp_Error n =
   sum (0,SUC M) (λ i. d i) + sum (1,M) (λ j. p j) + k) else
   (Fxp_Error n + sum (1,L) (λ i. a i * Fxp_Error (n - i)) =
   sum (0,SUC M) (λ i. d i) + sum (1,M) (λ j. p j) +
   sum (1,L) (λ i. e i) + sum (2,(L - 1)) (λ j. z j) + k))
```

3) Floating- to Fixed-Point Error Analysis:

$$e''_n + \sum_{k=1}^{L} a_k \, e''_{n-k} = \sum_{k=0}^{M} \delta'_{n,k} + \sum_{k=1}^{M} \zeta'_{n,k} + \sum_{k=1}^{L} \epsilon'_{n,k} + \sum_{k=2}^{L} \eta'_{n,k} + \xi'_n - \sum_{k=0}^{M} b_k \, (\theta_{n,k} - 1) \, x_{n-k} + \sum_{k=1}^{L} a_k \, (\phi_{n,k} - 1) \, y_{n-k} \tag{3.34}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in transition from floating-point to fixed-point levels of the direct form filter, according to the Equation (3.34).

Lemma 27: L_ORDER_FILTER_DIRECT_FORM_FXP_TO_FLOAT_THM

```
⊢ L_Order_Filter_Direct_Form_Ideal_Spec a b x w M L ∧
  L_Order_Filter_Direct_Form_Float_Imp a′ b′ x′ y M L ⟹
  ∃ t f k1 d1 p1 e1 z1.
  (abs k1 ≤ inv (2 pow (fracbits X))) ∧
  (∀ i. (i ≤ M) ⟹  abs (d1 i) ≤ inv (2 pow (fracbits X))) ∧
  (∀ i. (i ≤ M) ⟹  abs (p1 i) ≤ inv (2 pow (fracbits X))) ∧
  (∀ i. (i ≤ L) ⟹  abs (e1 i) ≤ inv (2 pow (fracbits X))) ∧
  (∀ i. (i ≤ L) ⟹  abs (z1 i) ≤ inv (2 pow (fracbits X))) ∧
  (if (L = 0) then
  (Float_Fxp_Error n = sum (0,SUC M) (λ i. d1 i) +
  sum (1,M) (λ j. p1 j) + k1 − (sum (0,SUC M)
  (λ i. Val (b′ i) * (t i − 1) * Val (x′ (n − i))))) else
  (Float_Fxp_Error n + sum (1,L) (λ i. a i * Float_Fxp_Error (n − i)) =
  sum (0,SUC M) (λ i. d1 i) + sum (1,M) (λ j. p1 j) +
  sum (1,L) (λ i. e1 i) + sum (2,(L − 1)) (λ j. z1 j) + k1 −
  sum (0, (SUC M)) (λ i. Val (b′ i) * (t i − 1) * Val (x′ (n − i))) +
  sum (1,L) (λ i. Val (a′ i) * (f i − 1) * Val (y (n − i))))) ∧
  ∃ k2 d2 p2 e2 z2. abs k2 ≤ (1 / 2 pow 24) ∧
  (∀ i. (i ≤ M) ⟹  abs (d2 i) ≤ (1 / 2 pow 24)) ∧
  (∀ i. (i ≤ M) ⟹  abs (p2 i) ≤ (1 / 2 pow 24)) ∧
  (∀ i. (i ≤ L) ⟹  abs (e2 i) ≤ (1 / 2 pow 24)) ∧
  (∀ i. (i ≤ L) ⟹  abs (z2 i) ≤ (1 / 2 pow 24)) ∧
  (t 0 = (1 + k2) * (1 + d2 0) * (mul (1,M) (λ i. (1 + p2 i)))) ∧
  (∀ j. (1 ≤ j ∧ j ≤ M) ⟹  (t j = (1 + k2) * (1 + d2 j) *
  (mul (j,(M − (j − 1))) (λ j. (1 + p2 j))))) ∧
  (f 1 = (1 + k2) * (1 + e2 1) * (mul (2,(L − 1)) (λ i. (1 + z2 i)))) ∧
  (∀ j. (2 ≤ j ∧ j ≤ L) ⟹  (f j = (1 + k2) * (1 + e2 j) *
  (mul (j,(L − j + 1)) (λ j. (1 + z2 j)))))
```

We proved these lemmas using the fundamental error analysis lemmas (Lemmas 4,5, and 6 for floating-point, and Lemmas 9,10, and 11 for fixed-point), based on the error models presented in Section 3.2. The lemmas are proved by induction on parameters L and M for the direct form of realization.

3.3.4 Lth-Order Filter (Parallel Form)

For parallel form of realization, the ith parallel path is described by Equation (3.6). The corresponding computed floating- and fixed-point outputs are

$$y_n^i = fl\ [f_i x_n + g_i x_{n-1} - c_i y_{n-1}^i - d_i y_{n-2}^i] \tag{3.35}$$

$$v_n^i = fxp\ [f_i x_n + g_i x_{n-1} - c_i v_{n-1}^i - d_i v_{n-2}^i] \tag{3.36}$$

The output of the entire parallel form filter is described by Equation (3.7). The corresponding computed floating- and fixed-point outputs are

$$y_n = fl\ [y_n^1 + y_n^2 + ... + y_n^K] \tag{3.37}$$

$$v_n = fxp\ [v_n^1 + v_n^2 + ... + v_n^K] \tag{3.38}$$

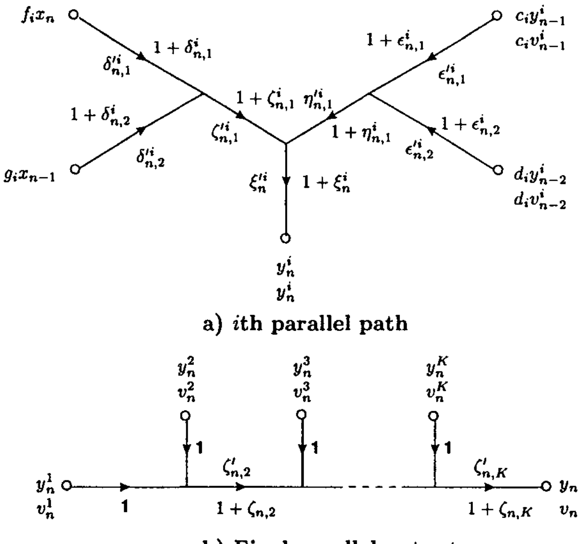
In HOL, we first specified the ith parallel path in real, floating-, and fixed-point abstraction levels, using Equations (3.6), (3.35), and (3.36). Then, we specified the entire output as defined in Equations (3.7), (3.37), and (3.38) using the finite summation functions. The corresponding codes are as follows.

```
⊢def Parallel_Form_Ideal_Spec c d f g x ww w K =
     ∀ n. w n = sum (1, K) (λ i. ww i n) ∧
     ∀ i. ww i n = f i * x n + g i * x (n - 1) -
                      c i * ww i (n - 1) - d i * ww i (n - 2)

⊢def Parallel_Form_Float_Imp c' d' f' g' x' yy y K =
     ∀ n. y n = float_sum (1,K) (λ i. yy i n) ∧
     ∀ i. (i ≥ 1 ∧ i ≤ K) ⟹ yy i n =
     f' i * x' n + g' i * x' (n - 1) -
     (c' i * yy i (n - 1) + d' i * yy i (n - 2))
```

```
⊢def  Parallel_Form_Fxp_Imp X c'' d'' f'' g'' x'' vv v K =
      ∀ n. v n = fxp_sum (1,K) X (λ i. vv i n) ∧
      ∀ i. (i ≥ 1 ∧ i ≤ K) ⟹  vv i n = FxpSub X
      (FxpAdd X (FxpMul X (f'' i) (x'' n))
      (FxpMul X (g'' i) (x'' (n - 1))))
      (FxpAdd X (FxpMul X (c'' i) (vv i (n - 1)))
      (FxpMul X (d'' i) (vv i (n - 2))))
```

Figure 3.6 shows the error flowgraph for the parallel form realization of a parametric Lth-order filter.



a) *i*th parallel path

b) Final parallel output

Figure 3.6: Error flowgraph for Lth-order filter (Parallel form)

The corresponding error flowgraph for the ith parallel path is shown in Figure 3.6 (a). The actual floating-point output sequence for the ith parallel path is therefore given by

$$y^i_n = f_i x_n \theta^i_{n,1} + g_i x_{n-1}\theta^i_{n,2} - c_i y^i_{n-1}\phi^i_{n,1} - d_i y^i_{n-2}\phi^i_{n,2} \tag{3.39}$$

where

$$\theta^i_{n,1} = (1+\delta^i_{n,1})(1+\zeta^i_{n,1})(1+\xi^i_n)$$

$$\theta^i_{n,2} = (1+\delta^i_{n,2})(1+\zeta^i_{n,1})(1+\xi^i_n)$$

$$\phi^i_{n,1} = (1+\epsilon^i_{n,1})(1+\eta^i_{n,1})(1+\xi^i_n)$$

$$\phi^i_{n,2} = (1+\epsilon^i_{n,2})(1+\eta^i_{n,1})(1+\xi^i_n)$$

If the summation of (3.37) is carried out from the left to the right, a corresponding flowgraph can be drawn as given in Figure 3.6 (b). Thus

$$y_n = \sum_{i=1}^{K} \psi_{n,i} y^i_n \tag{3.40}$$

where

$$\psi_{n,i} = \begin{cases} \prod_{j=2}^{K}(1+\zeta_{n,j}), & \text{i = 1} \\ \prod_{j=i}^{K}(1+\zeta_{n,j}), & i \geq 2 \end{cases}$$

In HOL, we established the following lemma to compute the real value of the floating-point output of the parallel form of realization according to the Equations (3.39) and (3.40).

```
Lemma 28: PARALLEL_FORM_FLOAT_OUTPUT_VALUE
⊢  Parallel_Form_Float_Imp c′ d′ f′ g′ x′ yy y K ⟹
   (∃ t f. Val (yy i n) = Val (f′ i) * Val (x′ n) * (t 1) +
   Val (g′ i) * Val (x′ (n - 1)) * (t 2) -
   Val (c′ i) * Val (yy i (n - 1)) * (f 1) -
   Val (d′ i) * Val (yy i (n - 2)) * (f 2)) ∧
   ∃ k d1 d2 p e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
```

```
abs (d1 i) ≤ (1 / 2 pow 24) ∧
abs (d2 i) ≤ (1 / 2 pow 24) ∧
abs (p i) ≤ (1 / 2 pow 24) ∧
abs (e1 i) ≤ (1 / 2 pow 24) ∧
abs (e2 i) ≤ (1 / 2 pow 24) ∧
abs (z i) ≤ (1 / 2 pow 24) ∧
t 1 = (1 + d1 i) * (1 + p i) * (1 + k) ∧
t 2 = (1 + d2 i) * (1 + p i) * (1 + k) ∧
f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
f 2 = (1 + e2 i) * (1 + z i) * (1 + k) ∧
∃ s. (Val (y n) = sum (1,K) (λ i. s i * Val (yy i n)) ∧
∃ k. s i = (if (i = 1) then (mul (2,(K - 1)) (λ i. (1 + k i))) else
(mul (i,(K - i + 1)) (λ i. (1 + k i)))))
```

Similarly, the actual fixed-point outputs of the parallel form of realization v_n^i, and v_n are given explicitly by

$$v_n^i = f_i x_n + g_i x_{n-1} - c_i v_{n-1}^i - d_i v_{n-2}^i + \delta'^i_{n,1} + \delta'^i_{n,2} + \zeta'^i_{n,1} + \epsilon'^i_{n,1} + \epsilon'^i_{n,2} + \eta'^i_{n,1} + \xi'^i_n \tag{3.41}$$

and

$$v_n = \sum_{i=1}^{K} v_n^i + \sum_{i=2}^{K} \zeta'_{n,i} \tag{3.42}$$

and the corresponding lemma is established in HOL as follows:

```
Lemma 29: PARALLEL_FORM_FXP_OUTPUT_VALUE
⊢ Parallel_Form_Fxp_Imp X c'' d'' f'' g'' x'' vv v K ⟹
  ∃ k' d1' d2' p' e1' e2' z'.
  abs (k' i) ≤ inv (2 pow (fracbits X)) ∧
  abs (d1' i) ≤ inv (2 pow (fracbits X)) ∧
  abs (d2' i) ≤ inv (2 pow (fracbits X)) ∧
abs (p′ i) ≤ inv (2 pow (fracbits X)) ∧
abs (e1′ i) ≤ inv (2 pow (fracbits X)) ∧
abs (e2′ i) ≤ inv (2 pow (fracbits X)) ∧
abs (z′ i) ≤ inv (2 pow (fracbits X)) ∧
value (vv i n) = value (f″ i) * value (x″ n) +
value (g″ i) * value (x″ (n − 1)) −
value (c″ i) * value (vv i (n − 1)) −
value (d″ i) * value (vv (n − 2)) +
d1′ i + d2′ i + p′ i + e1′ i + e2′ i + z′ i + k′ i ∧
∃ s′. value (v n) = sum (1,K) (λ i. value (vv i n)) +
sum (2,(K − 1)) (λ j. s′ j)
```

For error analysis of the parallel form, we first define the corresponding errors at the ith parallel path output sample as:

$$e_n^i = y_n^i - w_n^i \tag{3.43}$$

$$e_n'^i = v_n^i - w_n^i \tag{3.44}$$

$$e_n''^i = v_n^i - w_n^i \tag{3.45}$$

Then, we calculate the y_n^i, y_n, v_n^i, and v_n sequences from Equations (3.39), (3.40), (3.41), and (3.42), respectively and compare them with the ideal output sequences w_n^i, and w_n specified by the Equations (3.6), and (3.7) to obtain the corresponding errors e_n^i, $e_n'^i$, $e_n''^i$, e_n, e_n', and e_n'' according to the Equations (3.43), (3.44), (3.45), (3.9), (3.10), and (3.11), respectively. Therefore, the difference equations for the errors between different levels showing the accumulation of roundoff error are derived as follows:

1) Floating-Point Error Analysis:

$$e_n^i + c_i e_{n-1}^i + d_i e_{n-2}^i = f_i x_n(\theta_{n,1}^i - 1) + g_i x_{n-1}(\theta_{n,2}^i - 1) - c_i y_{n-1}^i(\phi_{n,1}^i - 1) - \tag{3.46}$$

$$d_i y^i_{n-2}(\phi^i_{n,2} - 1)$$

and

$$e_n - \sum_{i=1}^{K} e^i_n = \sum_{i=1}^{K} (\psi_{n,i} - 1)\; y^i_n \tag{3.47}$$

To prove these theorems in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the parallel form filter, according to the Equations (3.46), and (3.47).

```
Lemma 30: PARALLEL_FORM_FLOAT_TO_REAL_THM
⊢  Pararllel_Form_Ideal_Spec c d f g x ww w K ∧
   Parallel_Form_Float_Imp c′ d′ f′ g′ x′ yy y K  ⟹
   ∃ t f. Float_Error i n + c i * Float_Error i (n − 1) +
   d i * Float_Error i (n − 2) =
   f i * x n * (t 1 − 1) + g i * x (n − 1) * (t 2 − 1) −
   c i * Val (y (n − 1)) * (f 1 − 1) −
   d i * Val (y (n − 2)) * (f 2 − 1) ∧
   ∃ k d1 d2 p e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
   abs (d1 i) ≤ (1 / 2 pow 24) ∧
   abs (d2 i) ≤ (1 / 2 pow 24) ∧
   abs (p i) ≤ (1 / 2 pow 24) ∧
   abs (e1 i) ≤ (1 / 2 pow 24) ∧
   abs (e2 i) ≤ (1 / 2 pow 24) ∧
   abs (z i) ≤ (1 / 2 pow 24) ∧
   t 1 = (1 + d1 i) * (1 + p i) * (1 + k) ∧
   t 2 = (1 + d2 i) * (1 + p i) * (1 + k) ∧
   f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
   f 2 = (1 + e2 i) * (1 + z i) * (1 + k) ∧
   ∃ s. (Float_Error n = sum (1,K) (λ i. Float_Fxp_Error i n) +
   sum (1, K) (λ i. (((s i) − 1) * Val (yy i n))) ∧
   ∃ k. s i = if (i = 1) then mul (2, (K − 1)) (λ i. (1 + (k i))) else
   mul (i,(K − i + 1)) (λ i. (1 + (k i))))
```

2) Fixed-Point Error Analysis:

$$e'^i_n + c_i e'^i_{n-1} + d_i e'^i_{n-2} = \delta'^i_{n,1} + \delta'^i_{n,2} + \zeta'^i_{n,1} + \epsilon'^i_{n,1} + \epsilon'^i_{n,2} + \eta'^i_{n,1} + \xi'^i_n \tag{3.48}$$

and

$$e'_n - \sum_{i=1}^{K} e'^i_n = \sum_{i=1}^{K} \zeta'_{n,i} \tag{3.49}$$

To prove these theorems in HOL, we established the following lemma for the accumulation of round-off error in fixed-point realization of the parallel form filter, according to the Equations (3.48), and (3.49).

```
Lemma 31: PARALLEL_FORM_FXP_TO_REAL_THM
⊢ Parallel_Form_Ideal_Spec c d f g x ww w K ∧
  Parallel_Form_Fxp_Imp X c″ d″ f″ g″ x″ vv v K ⟹
  ∃ k′ d1′ d2′ p′ e1′ e2′ z′.
  abs (k′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (d1′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (d2′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (p′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (e1′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (e2′ i) ≤ inv (2 pow (fracbits X)) ∧
  abs (z′ i) ≤ inv (2 pow (fracbits X)) ∧
  Fxp_Error i n + c i * Fxp_Error i (n − 1) +
  d i * Fxp_Error i (n − 2) =
  d1′ i + d2′ i + p′ i + e1′ i + e2′ i + z′ i + k′ i ∧
  ∃ s′. Fxp_Error n = sum (1,K) (λ i. Fxp_Error i n) +
  sum (2,(K − 1)) (λ j. s′ j)
```

3) Floating- to Fixed-Point Error Analysis:

$$e''^i_n + c_i e''^i_{n-1} + d_i e''^i_{n-2} = \delta'^i_{n,1} + \delta'^i_{n,2} + \zeta'^i_{n,1} + \epsilon'^i_{n,1} + \epsilon'^i_{n,2} + \eta'^i_{n,1} + \xi'^i_n - \tag{3.50}$$

$$f_i x_n(\theta^i_{n,1} - 1) - g_i x_{n-1}(\theta^i_{n,2} - 1) + c_i y^i_{n-1}(\phi^i_{n,1} - 1) + d_i y^i_{n-2}(\phi^i_{n,2} - 1)$$

and

$$e''_n - \sum_{i=1}^{K} e_n^{\prime\prime i} = \sum_{i=1}^{K} \zeta'_{n,i} - \sum_{i=1}^{K} (\psi_{n,i} - 1)\; y_n^i \tag{3.51}$$

To prove these theorems in HOL, we established the following lemma for the accumulation of round-off error in transition from floating-point to fixed-point realizations of the parallel form filter, according to the Equations (3.50), and (3.51).

```
Lemma 32: PARALLEL_FORM_FXP_TO_FLOAT_THM
⊢  Parallel_Form_Ideal_Spec c d f g x ww w K ∧
   Parallel_Form_Float_Imp c′ d′ f′ g′ x′ yy y K ∧
   Parallel_Form_Fxp_Imp X c″ d″ f″ g″ x″ vv v K ⟹
   ∃ t f k′ d1′ d2′ p′ e1′ e2′ z′.
   abs (k′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (z′ i) ≤ inv (2 pow (fracbits X)) ∧
   Float_Fxp_Error i n + c i * Float_Fxp_Error i (n − 1) +
   d i * Float_Fxp_Error i (n − 2) =
   d1′ i + d2′ i + p′ i + e1′ i + e2′ i + z′ i + k′ i −
   (f i * x n * (t 1 − 1) +
   g i * (x (n − 1)) * (t 2 − 1) −
   (c i * Val (y (n − 1)) * (f 1 − 1) +
   d i * Val (y (n − 2)) * (f 2 − 1))) ∧
   ∃ k d1 d2 p e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
   abs (d1 i) ≤ (1 / 2 pow 24) ∧
   abs (d2 i) ≤ (1 / 2 pow 24) ∧
   abs (p i) ≤ (1 / 2 pow 24) ∧
   abs (e1 i) ≤ (1 / 2 pow 24) ∧
   abs (e2 i) ≤ (1 / 2 pow 24) ∧
   abs (z i) ≤ (1 / 2 pow 24) ∧
```

```
t 1 = (1 + d1 i) * (1 + p i) * (1 + k) ∧
t 2 = (1 + d2 i) * (1 + p i) * (1 + k) ∧
f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
f 2 = (1 + e2 i) * (1 + z i) * (1 + k) ∧
∃ s s'. Float_Fxp_Error n = sum (1,K) (λ i. Float_Fxp_Error i n) +
sum (2,(K - 1)) (λ j. s' j) -
sum (1, K) (λ i. ((s i - 1) * Val (yy i n))) ∧
∃ k. s i = if (i = 1) then mul (2, (K - 1)) (λ i. (1 + k i)) else
mul (i,(K - i + 1)) (λ i. (1 + k i))
```

We proved these lemmas using the fundamental error analysis lemmas (Lemmas 4,5, and 6 for floating-point, and Lemmas 9,10, and 11 for fixed-point), based on the error models presented in Section 3.2. The lemmas are proved by induction on the parameter K which is defined as the number of internal sub-filters connected in parallel form to generate the final output, according to the Equation (3.7).

3.3.5 Lth-Order Filter (Cascade Form)

The cascade form realization of a parametric Lth-order filter is specified by Equation (3.8). The corresponding computed floating- and fixed-point outputs are

$$y_n^{i+1} = fl[y_n^i + k_i y_{n-1}^i + l_i y_{n-2}^i - c_i y_{n-1}^{i+1} - d_i y_{n-2}^{i+1}] \tag{3.52}$$

$$v_n^{i+1} = fxp[v_n^i + k_i v_{n-1}^i + l_i v_{n-2}^i - c_i v_{n-1}^{i+1} - d_i v_{n-2}^{i+1}] \tag{3.53}$$

In HOL, we specified the second-order digital filter in real, floating-, and fixed-point abstraction levels, using recursive definitions as predicates in higher-order logic. The corresponding codes are as follows.

```
⊢def Cascade_Form_Ideal_Spec c d k l x w =
     ∀ n. (w 0 n = x n ∧ ∀ i.
     w (SUC i) n = ((((w i n + k i * w i (n - 1)) +
     l i * w i (n - 2)) - c i * w (SUC i) (n - 1)) -
     d i * w (SUC i) (n - 2)))
```

```
⊢def Cascade_Form_Float_Imp c' d' k' l' x' y =
     ∀ n. (y 0 n = x' n ∧ ∀ i.
     y (SUC i) n = ((((y i n + (k' i * y i (n - 1))) +
     (l' i * y i (n - 2))) - (c' i * y (SUC i) (n - 1))) -
     (d' i * y (SUC i) (n - 2))))
```

```
⊢def Cascade_Form_Fxp_Imp X c'' d'' k'' l'' x'' v =
     ∀ n. (v 0 n = x'' n ∧ ∀ i. v (SUC i) n =
     FxpSub X (FxpSub X (FxpAdd X (FxpAdd X v i n
     (FxpMul X (k'' i) v i (n - 1)))
     (FxpMul X (l'' i) v i (n - 2)))
     (FxpMul X (c'' i) v (i + 1) (n - 1)))
     (FxpMul X (d'' i) v (i + 1) (n - 2)))
```

Figure 3.6 shows the error flowgraph for the cascade form realization of a parametric Lth-order filter.

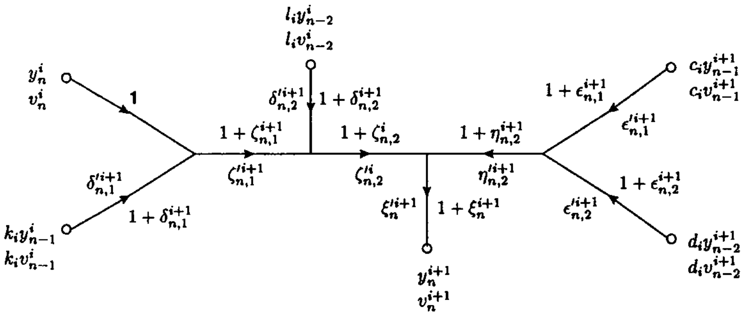


Figure 3.7: Error flowgraph for Lth-order filter (Cascade form)

The actual floating-point output sequence for the cascade form is therefore given by

$$y_n^{i+1} = y_n^i \theta_{n,0}^{i+1} + f_i y_{n-1}^i \theta_{n,1}^{i+1} + g_i y_{n-2}^i \theta_{n,2}^{i+1} - c_i y_{n-1}^i \phi_{n,1}^{i+1} - d_i y_{n-2}^i \phi_{n,2}^{i+1} \qquad (3.54)$$

where

$$\theta_{n,0}^{i+1} = (1 + \zeta_{n,1}^{i+1})(1 + \zeta_{n,2}^{i+1})(1 + \xi_{n}^{i+1})$$

$$\theta_{n,1}^{i+1} = (1 + \delta_{n,1}^{i+1})(1 + \zeta_{n,1}^{i+1})(1 + \zeta_{n,2}^{i+1})(1 + \xi_{n}^{i+1})$$

$$\theta_{n,2}^{i+1} = (1 + \delta_{n,2}^{i+1})(1 + \eta_{n,1}^{i+1})(1 + \xi_{n}^{i+1})$$

$$\phi_{n,1}^{i+1} = (1 + \epsilon_{n,1}^{i+1})(1 + \eta_{n,1}^{i+1})(1 + \xi_{n}^{i+1})$$

$$\phi_{n,2}^{i+1} = (1 + \epsilon_{n,2}^{i+1})(1 + \eta_{n,1}^{i+1})(1 + \xi_{n}^{i+1})$$

In HOL, we established the following lemma to compute the real value of the floating-point output of the parallel form of realization according to the Equation (3.54).

```
Lemma 33: CASCADE_FORM_FLOAT_OUTPUT_VALUE
⊢  Cascade_Form_Float_Imp c′ d′ k′ l′ x′ y ⟹
   ∃ t f. Val (y (SUC i) n) = Val (y i n) * (t 0) +
   Val (k′ i) * Val (y i (n − 1)) * (t 1) +
   Val (l′ i) * Val (y i (n − 2)) * (t 2) −
   Val (c′ i) * Val (y i (n − 1)) * (f 1) −
   Val (d′ i) * Val (y i (n − 2)) * (f 2) ∧
   ∃ k d1 d2 p1 p2 e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
   abs (d1 i) ≤ (1 / 2 pow 24) ∧
   abs (d2 i) ≤ (1 / 2 pow 24) ∧
   abs (p1 i) ≤ (1 / 2 pow 24) ∧
   abs (p2 i) ≤ (1 / 2 pow 24) ∧
   abs (e1 i) ≤ (1 / 2 pow 24) ∧
   abs (e2 i) ≤ (1 / 2 pow 24) ∧
   abs (z i) ≤ (1 / 2 pow 24) ∧
```

```
t 0 = (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
t 1 = (1 + d1 i) * (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
t 2 = (1 + d2 i) * (1 + p2 i) * (1 + k) ∧
f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
f 2 = (1 + e2 i) * (1 + z i) * (1 + k)
```

Similarly, the actual fixed-point outputs of the cascade form of realization is given explicitly by

$$v_n^{i+1} = v_n^i + f_i v_{n-1}^i + g_i v_{n-2}^i - c_i v_{n-1}^{i+1} - d_i v_{n-2}^{i+1} + \delta'^{i+1}_{n,1} + \zeta'^{i+1}_{n,1} + \delta'^{i+1}_{n,2} + \zeta'^{i}_{n,2} + \epsilon'^{i+1}_{n,1} + \epsilon'^{i+1}_{n,2} + \eta'^{i+1}_{n,2} + \xi'^{i+1}_{n} \tag{3.55}$$

and the corresponding lemma is established in HOL as follows:

```
Lemma 34: CASCADE_FORM_FXP_OUTPUT_VALUE
⊢  Cascade_Form_Fxp_Imp X c″ d″ k″ l″ x″ v ⟹
   ∃ k′ d1′ d2′ p1′ p2′ e1′ e2′ z′.
   abs (k′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (z′ i) ≤ inv (2 pow (fracbits X)) ∧
   value (v (SUC i) n) = value (v i n) +
   value (k″ i) * value (v i (n - 1)) +
   value (l″ i) * value (v i (n - 2)) +
   value (l″ i) * value (v i (n - 2)) +
   value (c″ i) * value (v (SUC i) (n - 1)) +
   value (d″ i) * value (v (SUC i) (n - 2)) +
   d1′ i + p1′ i + d2′ i + p2′ i + e1′ i + e2′ i + z′ i + k′ i
```

For error analysis of the cascade form, we first define the corresponding errors as:

$$e_n^{i+1} = y_n^{i+1} - w_n^{i+1} \tag{3.56}$$

$$e_n'^{i+1} = v_n^{i+1} - w_n^{i+1} \tag{3.57}$$

$$e_n''^{i+1} = v_n^{i+1} - w_n^{i+1} \tag{3.58}$$

Then, we calculate the y_n^{i+1}, and v_n^{i+1} sequences from Equations (3.54), (3.55), respectively and compare them with the ideal output sequences w_n^{i+1} specified by the Equations (3.8) to obtain the corresponding errors e_n^{i+1}, $e_n'^{i+1}$, and $e_n''^{i+1}$ according to the Equations (3.56), (3.57), (3.58), respectively. Therefore, the difference equations for the errors between different levels showing the accumulation of roundoff error are derived as follows:

1) Floating-Point Error Analysis:

$$\begin{aligned} e_n^{i+1} = {} & e_n^i + f_i e_{n-1}^i + g_i e_{n-2}^i - c_i e_{n-1}^{i+1} - d_i e_{n-2}^{i+1} + y_n^i(\theta_{n,0}^{i+1} - 1) + f_i y_{n-1}^i(\theta_{n,1}^{i+1} - 1) + \\ & g_i y_{n-2}^i(\theta_{n,2}^{i+1} - 1) - c_i y_{n-1}^i(\phi_{n,1}^{i+1} - 1) - d_i y_{n-2}^i(\phi_{n,2}^{i+1} - 1) \end{aligned} \tag{3.59}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the cascade form filter, according to the Equation (3.59).

```
Lemma 35: CASCADE_FORM_FLOAT_TO_REAL_THM
⊢  Cascade_Form_Ideal_Spec c d k l x w ∧
   Cascade_Form_Float_Imp c′ d′ k′ l′ x′ y ⟹
   ∃ t f. Float_Error (SUC i) n = Float_Error i n +
   k i * Float_Error i (n − 1) + l i * Float_Error i (n − 2) −
   c i * Float_Error (SUC i) (n − 1) −
   d i * Float_Error (SUC i) (n − 2) +
   Val (y i n) * (t 0 − 1) +
   (k i) * Val (y i (n − 1) (t 1 − 1) +
   (l i) * Val (y i (n − 2)) (t 2 − 1) −
   (c i) * Val (y i (n − 1)) (f 1 − 1) −
   (d i) * Val (y i (n − 2)) (f 2 − 1) ∧
   ∃ k d1 d2 p1 p2 e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
   abs (d1 i) ≤ (1 / 2 pow 24) ∧
   abs (d2 i) ≤ (1 / 2 pow 24) ∧
   abs (p1 i) ≤ (1 / 2 pow 24) ∧
   abs (p2 i) ≤ (1 / 2 pow 24) ∧
   abs (e1 i) ≤ (1 / 2 pow 24) ∧
   abs (e2 i) ≤ (1 / 2 pow 24) ∧
   abs (z i) ≤ (1 / 2 pow 24) ∧
   t 0 = (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
   t 1 = (1 + d1 i) * (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
   t 2 = (1 + d2 i) * (1 + p2 i) * (1 + k) ∧
   f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
   f 2 = (1 + e2 i) * (1 + z i) * (1 + k)
```

2) Fixed-Point Error Analysis:

$$\begin{aligned} e_n^{\prime\prime i+1} &= e_n^{\prime\prime i} + f_i e_{n-1}^{\prime\prime i} + g_i e_{n-2}^{\prime\prime i} - c_i e_{n-1}^{\prime\prime i+1} - d_i e_{n-2}^{\prime\prime i+1} + \\ & \delta_{n,1}^{\prime\prime i+1} + \zeta_{n,1}^{\prime\prime i+1} + \delta_{n,2}^{\prime\prime i+1} + \zeta_{n,2}^{\prime\prime i+1} + \epsilon_{n,1}^{\prime\prime i+1} + \epsilon_{n,2}^{\prime\prime i+1} + \eta_{n,2}^{\prime\prime i+1} + \xi_n^{\prime\prime i+1} \end{aligned} \tag{3.60}$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the cascade form filter, according to the Equation (3.60).

```
Lemma 36: CASCADE_FORM_FXP_TO_REAL_THM
⊢  Cascade_Form_Ideal_Spec c d k l x w ∧
   Cascade_Form_Fxp_Imp X c'' d'' k'' l'' x'' v ⟹
   ∃ k' d1' d2' p1' p2' e1' e2' z'.
   abs (k' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d1' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d2' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p1' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p2' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e1' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e2' i) ≤ inv (2 pow (fracbits X)) ∧
   abs (z' i) ≤ inv (2 pow (fracbits X)) ∧
   Fxp_Error (SUC i) n = Fxp_Error i n + k i * Fxp_Error i (n - 1) +
   l i * Fxp_Error i (n - 2) - c i * Fxp_Error (SUC i) (n - 1) -
   d i * Fxp_Error (SUC i) (n - 2) +
   d1' i + p1' i + d2' i + p2' i + e1' i + e2' i + z' i + k' i
```

3) Floating- to Fixed-Point Error Analysis:

$$e''^{i+1}_n = e''^{i}_n + f_i e''^{i}_{n-1} + g_i e''^{i}_{n-2} - c_i e''^{i+1}_{n-1} - d_i e''^{i+1}_{n-2} + \delta'^{i+1}_{n,1} + \zeta'^{i+1}_{n,1} + \delta'^{i+1}_{n,2} + \zeta'^{i+1}_{n,2} + \epsilon'^{i+1}_{n,1} + \epsilon'^{i+1}_{n,2} + \eta'^{i+1}_{n,2} + \xi'^{i+1}_{n} - y^i_n(\theta^{i+1}_{n,0} - 1) - f_i y^i_{n-1}(\theta^{i+1}_{n,1} - 1) - g_i y^i_{n-2}(\theta^{i+1}_{n,2} - 1) + c_i y^i_{n-1}(\phi^{i+1}_{n,1} - 1) + d_i y^i_{n-2}(\phi^{i+1}_{n,2} - 1) \quad (3.61)$$

To prove this theorem in HOL, we established the following lemma for the accumulation of round-off error in floating-point realization of the cascade form filter, according to the Equation (3.61).

Lemma 37: CASCADE_FORM_FXP_TO_FLOAT_THM

```
⊢  Cascade_Form_Ideal_Spec c d k l x w ∧
   Cascade_Form_Float_Imp c′ d′ k′ l′ x′ y ∧
   Cascade_Form_Fxp_Imp X c″ d″ k″ l″ x″ v ⟹
   ∃ t f k′ d1′ d2′ p1′ p2′ e1′ e2′ z′.
   abs (k′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (d2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (p2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e1′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (e2′ i) ≤ inv (2 pow (fracbits X)) ∧
   abs (z′ i) ≤ inv (2 pow (fracbits X)) ∧
   Float_Fxp_Error (SUC i) n = Float_Fxp_Error i n +
   k i * Float_Fxp_Error i (n − 1) +
   l i * Float_Fxp_Error i (n − 2) −
   c i * Float_Fxp_Error (SUC i) (n − 1) −
   d i * Fxp_Error (SUC i) (n − 2) +
   d1′ i + p1′ i + d2′ i + p2′ i + e1′ i + e2′ i + z′ i + k′ i −
   Val (y i n) * (t 0 − 1) −
   (k i) * Val (y i (n − 1) (t 1 − 1) −
   (l i) * Val (y i (n − 2)) (t 2 − 1) +
   (c i) * Val (y i (n − 1)) (f 1 − 1) +
   (d i) * Val (y i (n − 2)) (f 2 − 1) ∧
   ∃ k d1 d2 p1 p2 e1 e2 z. abs (k i) ≤ (1 / 2 pow 24) ∧
   abs (d1 i) ≤ (1 / 2 pow 24) ∧
   abs (d2 i) ≤ (1 / 2 pow 24) ∧
   abs (p1 i) ≤ (1 / 2 pow 24) ∧
   abs (p2 i) ≤ (1 / 2 pow 24) ∧
   abs (e1 i) ≤ (1 / 2 pow 24) ∧
   abs (e2 i) ≤ (1 / 2 pow 24) ∧
   abs (z i) ≤ (1 / 2 pow 24) ∧
   t 0 = (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
   t 1 = (1 + d1 i) * (1 + p1 i) * (1 + p2 i) * (1 + k) ∧
   t 2 = (1 + d2 i) * (1 + p2 i) * (1 + k) ∧
```

```
f 1 = (1 + e1 i) * (1 + z i) * (1 + k) ∧
f 2 = (1 + e2 i) * (1 + z i) * (1 + k)
```

We proved these lemmas using the fundamental error analysis lemmas (Lemmas 4,5, and 6 for floating-point, and Lemmas 9,10, and 11 for fixed-point), based on the error models presented in Section 3.2.

3.4 Conclusion

In this chapter, we described our comprehensive methodology for the error analysis of generic digital filters using the HOL theorem prover. We believe this is the first time a complete formal framework is considered using mechanical proofs in HOL for the error analysis of digital filters. In the next chapter, we describe the formal verification of FFT algorithms. We perform a similar error analysis between the real numbers and the floating-point and fixed-point algorithmic levels. We also perform the verification for the transition from the floating-point and fixed-point algorithmic levels to hardware implementations for FFT algorithms.

Chapter 4

Verification of FFT Algorithms in HOL

4.1 Introduction

The fast Fourier transform (FFT) [9, 16] is a highly efficient method for computing the discrete Fourier transform (DFT) coefficients of a finite sequence of complex data. Because of the substantial time saving over conventional methods, the fast Fourier transform has found important applications in a number of diverse fields such as spectrum analysis, speech and optical signal processing, and digital filter design. FFT algorithms are based on the fundamental principle of decomposing the computation of the discrete Fourier transform of a finite-length sequence of length N into successively smaller discrete Fourier transforms. The manner in which this principle is implemented leads to a variety of different algorithms, all with comparable improvements in computational speed. There are two basic classes of FFT algorithms for which the number of arithmetic multiplications and additions as a measure of computational complexity is proportional to $N \, log \, N$ rather than N^2 as in the conventional methods. The first proposed by Cooley and Tukey [18], called decimation-in-time (DIT), derives its name from the fact that in the process of arranging the computation into smaller transformations, the input sequence (generally thought of as a time sequence) is decomposed into successively smaller subsequences. In the second general class of algorithms proposed by Gentleman and Sande [22], the sequence of discrete Fourier transform coefficients is decomposed into smaller subsequences, hence its name, decimation-in-frequency (DIF).

In Chapter 1, Figure 1.1 illustrates a generic DSP (digital signal processing) design flow as used in leading industrial projects for the design of FFT algorithms. Thereafter, the design process starts from an ideal real specification used for the theoretical analysis of the fast Fourier transform. Here signal values and system coefficients are represented with real numbers expressed to infinite precision. When implemented as a special-purpose digital hardware or as a computer algorithm, these must be represented in some digital number system of finite precision. There is hence an inherent accuracy problem in calculating the Fourier coefficients, since the arithmetic operations must be carried out with an accuracy limited by the finite word length of signals. Among the most common types of arithmetic used in the implementation of FFT systems are floating- and fixed-point [59]. Here, all operands are represented by a special format or assigned a fixed word length and a fixed exponent, while the control structure and the operations of the ideal program remain unchanged. The transformation from real to floating- and fixed-point is quite tedious and error-prone. On the implementation side, the fixed-point model of the algorithm has to be transformed into the best suited target description, either using a hardware description languages (HDL) or a software programming language.

The conformance of the fixed-point implementation with respect to the descriptions in floating-point or real algorithm on one hand, and the RT (Register Transfer) and gate levels on the other hand is verified by simulation techniques. Simulation is, however, known to provide partial verification as it cannot cover all design errors, especially for large systems. In this chapter, we are proposing the use of formal methods for the modeling and verification of FFT algorithms. Adopting formal verification generally means using methods of mathematical proof to ensure the quality of the design, to improve the robustness of a design and to speed up the overall system design and development cycles.

The proposed verification approach is depicted in the commutating diagram shown in Figure 4.1, where we model the ideal real specification of the FFT algorithms and the corresponding floating- and fixed-point representations as well as the RT and gate level

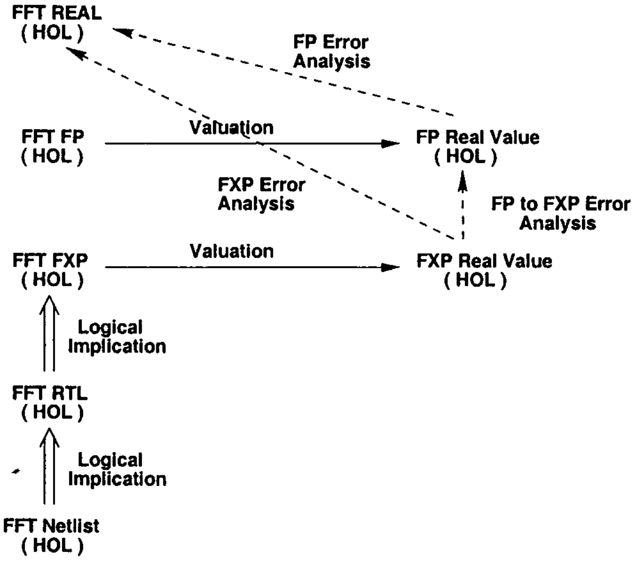


Figure 4.1: Proposed FFT specification and verification approach

implementations as predicates in higher-order logic. The overall methodology for the formal specification and verification of FFT algorithms will be based on the idea of shallow embedding of languages [4] using the HOL theorem proving environment [23]. In the proposed approach, we first focus on the transition from real to floating- and fixed-point levels. For this, we make use of existing theories in HOL on the construction of real [27] and complex [32] numbers, the formalization of IEEE-754 standard based floating-point arithmetic [28, 29], and the formalization of fixed-point arithmetic. We use valuation functions to find the real values of the floating- and fixed-point FFT outputs and define the error as the difference between these values and the corresponding output of the ideal real specification. Then we establish fundamental lemmas on the error analysis of floating- and fixed-point rounding and arithmetic operations against their abstract mathematical counterparts. Finally, based on these lemmas, we derive, for each of the two canonical forms of realization, expressions for the accumulation of roundoff error in floating- and fixed-point FFT algorithms using recursive definitions and initial conditions. While theoretical work on computing the errors due to finite precision effects in the realization of FFT algorithms with floating- and fixed-point arithmetics has been extensively studied since the late sixties [41], this thesis contains the first formalization and proof of this analysis using a mechanical theorem prover, here HOL. The formal results are found to be in good agreement with the theoretical ones.

After handling the transition from real to floating- and fixed-point levels, we turn to the HDL representation. At this point, we use well known techniques to model the FFT design at the RTL level within the HOL environment. The last step is to verify this level using a classical hierarchical proof approach in HOL [52]. In this way, we hierarchically prove that the FFT RTL implementation implies the high level fixed-point algorithmic specification, which has already been related to the floating-point description and the ideal real specification through the error analysis. The verification can be extended, following similar manner, down to gate level netlist either in HOL or using other commercial verification tools as depicted in Figure 4.1, which is not covered in this paper.

The rest of the chapter is organized as follows: Section 4.2 describes the details of the error analysis in HOL of the FFT algorithms at the real, floating-point and fixed-point levels. Section 4.3 describes the verification of the FFT algorithms in the transition to the RTL and gate level netlist for a radix-4 16-point FFT implementation. Finally, Section 4.4 concludes the chapter.

4.2 Error Analysis of FFT Algorithms in HOL

In this section, the principal results for roundoff accumulation in FFT algorithms using HOL theorem proving are derived and summarized. For the most part, the following discussion is phrased in terms of the decimation-in-frequency form of radix-2 algorithm. The results, however, are applicable with only minor modification to the decimation-in-time form. Furthermore, most of the ideas employed in the error analysis of the radix-2 algorithms can be utilized in the analysis of other algorithms. In the following, we will first describe in detail the theory behind the analysis and then explain how this analysis is performed in HOL.

The discrete Fourier transform of a sequence $\{x(n)\}_{n=0}^{N-1}$ is defined as [59]

$$A(p) \;=\; \sum_{n=0}^{N-1} x(n)\;(W_N)^{np}, \qquad p \;=\; 0,1,2,\ldots,N-1 \tag{4.1}$$

where $W_N = e^{-j2\pi/N}$ and $j = \sqrt{-1}$. The multiplicative factors $(W_N)^{np}$ are called twiddle factors. For simplicity, our discussion is restricted to the radix-2 FFT algorithm, in which the number of points N to be Fourier transformed satisfy the relationship $N = 2^m$, where m is an integer value. The results can be extended to radices other than 2. By using the FFT method, the Fourier coefficients $\{A(p)\}_{p=0}^{N-1}$ can be calculated in $m = \log_2 N$ iterative steps. At each step, an array of N complex numbers is generated by using only the numbers in the previous array. To explain the FFT algorithm, let each integer p, $p = 0, 1, 2, \ldots, N-1$, be expanded into a binary form as

$$p = 2^{m-1}p_0 + 2^{m-2}p_1 + \cdots + 2p_{m-2} + p_{m-1}, \qquad p_k = 0 \text{ or } 1 \tag{4.2}$$

and let p^* denote the number corresponding to the reverse bit sequences of p, i.e.,

$$p^* = 2^{m-1}p_{m-1} + 2^{m-2}p_{m-2} + \cdots + 2p_1 + p_0 \tag{4.3}$$

- **Decimation-in-Frequency (DIF) FFT Algorithm:**

Let $\{A_k(p)\}_{p=0}^{N-1}$ denote the N complex numbers calculated at the kth step. Then the decimation in frequency (DIF) FFT algorithm can be expressed as [41]

$$A_{k+1}(p) = \begin{cases} A_k(p) + A_k(p + 2^{m-1-k}) & \text{if } p_k = 0 \\ [A_k(p - 2^{m-1-k}) - A_k(p)]\, w_k(p) & \text{if } p_k = 1 \end{cases} \tag{4.4}$$

where $w_k(p)$ is a power of W_N given by $w_k(p) = (W_N)^{z_k(p)}$, where

$$z_k(p) = 2^k\,(2^{m-1-k}p_k + 2^{m-2-k}p_{k+1} + \cdots + 2p_{m-2} + p_{m-1}) - 2^{m-1}p_k \tag{4.5}$$

Equation (4.4) is carried out for $k = 0, 1, 2, \ldots, m-1$, with $A_0(p) = x(p)$. It can be shown [22] that at the last step $\{A_m(p)\}_{p=0}^{N-1}$ are the discrete Fourier coefficients in rearranged order. Specifically, $A_m(p) = A(p^*)$ with p and p^* expanded and defined as in Equations (4.2) and (4.3), respectively. Figure 4.2 shows the signal flowgraph of the actual computation for the case $N = 2^4$.

Formally, a flowgraph consists of nodes and directed branches. Each branch has an input signal and an output signal with a direction indicated by an arrowhead on it. Each node represents a variable which is the weighted sum of the variables at the originating nodes of the branches that terminate on that node. The weights, if other than unity, are shown for each branch. Source nodes have no entering branches. They are used to represent the injection of the external inputs or signal sources into the flowgraph. Sink nodes have only entering branches. They are used to extract the outputs from the flowgraph [59].

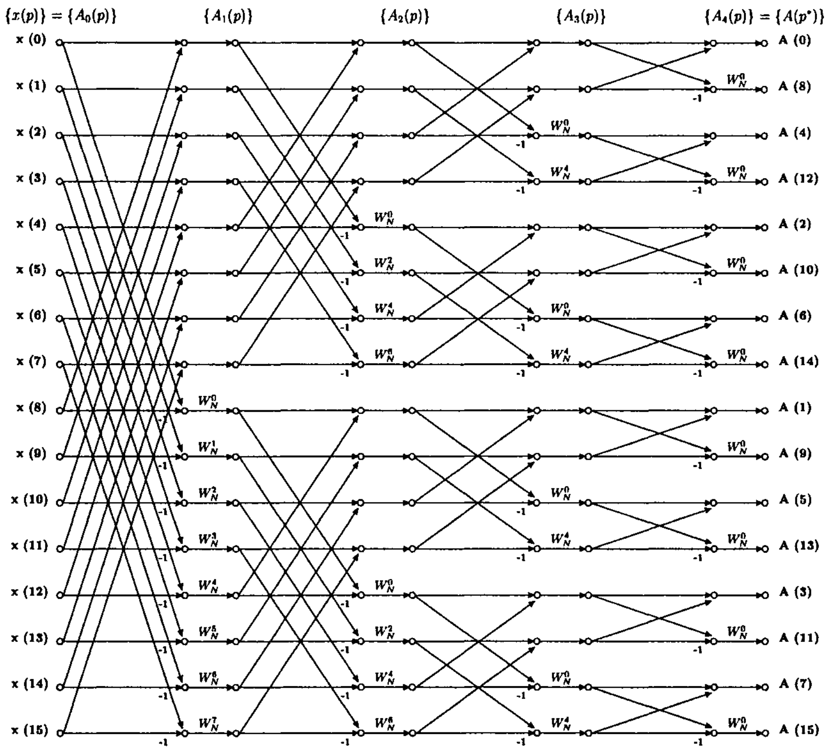


Figure 4.2: Signal flowgraph of decimation-in-frequency FFT, $N = 2^4$

- **Decimation-in-Time (DIT) FFT Algorithm:**

Let $\{A_k(p)\}_{p=0}^{N-1}$ denote the N complex numbers calculated at the kth step. Then the decimation in time (DIT) FFT algorithm can be expressed as [48]

$$A_{k+1}(p) = \begin{cases} A_k(p) + w_k(p)\ A_k(p+2^k) & \text{if } p_{m-1-k} = 0 \\ A_k(p-2^k) - w_k(p)\ A_k(p) & \text{if } p_{m-1-k} = 1 \end{cases} \tag{4.6}$$

where $w_k(p)$ is a power of W_N given by $w_k(p) = (W_N)^{z_k(p)}$, where

$$z_k(p) = 2^{m-1-k}\ (2^k p_{m-1-k} + 2^{k-1} p_{m-k} + \cdots + 2p_{m-2} + p_{m-1}) - 2^{m-1} p_{m-1-k} \tag{4.7}$$

Equation (4.6) is carried out for $k = 0, 1, 2, \ldots, m-1$, with $A_0(p) = x(p^*)$ with p and p^* expanded and defined as in Equations (4.2) and (4.3), respectively. It can be shown [18] that at the last step $\{A_m(p)\}_{p=0}^{N-1}$ are the discrete Fourier coefficients in the normal order. Specifically, $A_m(p) = A(p)$. Figure 4.3 shows the signal flowgraph of the actual computation for the case $N = 2^4$.

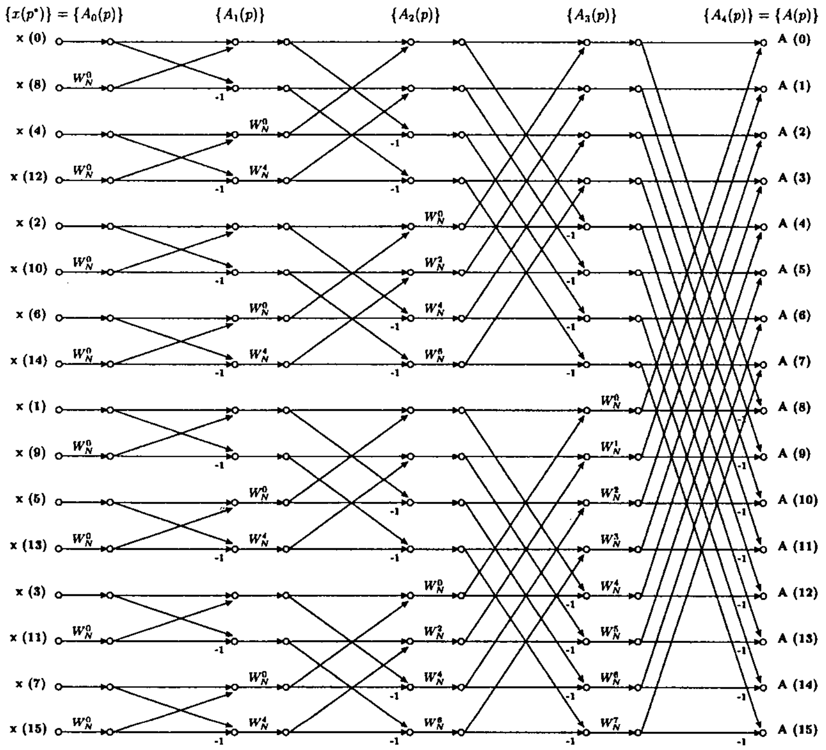


Figure 4.3: Signal flowgraph of decimation-in-time FFT, $N = 2^4$

There are three common sources of errors associated with the FFT algorithms, namely [41]:

1. **Input Quantization:** caused by the quantization of the input signal $\{x_n\}$ into a set of discrete levels.

2. **Coefficient Accuracy:** caused by the representation of the coefficients $\{w_k(p)\}$ by a finite word length.

3. **Round-Off Accumulation:** caused by the accumulation of roundoff errors at arithmetic operations.

Therefore, the actual array computed by using equations (4.4) and (4.6) are in general different from $\{A_k(p)\}_{p=0}^{N-1}$. We denote the actual floating- and fixed-point computed arrays by $\{A'_k(p)\}_{p=0}^{N-1}$ and $\{A''_k(p)\}_{p=0}^{N-1}$, respectively. Then, we define the corresponding errors of the pth element at step k as

$$e_k(p) = A'_k(p) - A_k(p) \tag{4.8}$$

$$e'_k(p) = A''_k(p) - A_k(p) \tag{4.9}$$

$$e''_k(p) = A''_k(p) - A'_k(p) \tag{4.10}$$

where $e_k(p)$ and $e'_k(p)$ are defined as the error between the actual floating- and fixed-point implementations and the ideal real specification, respectively. $e''_k(p)$ is the error in transition from floating- to fixed-point levels.

In analyzing the effect of floating-point roundoff, the effect of rounding will be represented multiplicatively. Letting $*$ denote any of the arithmetic operations +, -, × , /, as proved in Section 3.2, if p represents the precision of the floating-point format, then

$$fl\ (x\ *\ y)\ =\ (x\ *\ y)(1\ +\ \delta),\quad \text{where}\ |\delta|\ \leq\ 2^{-p} \tag{4.11}$$

The notation $fl\ (.)$ is used to denote that the operation is performed using floating-point arithmetic. The theorem relates the floating-point arithmetic operations such as addition, subtraction, multiplication, and division to their abstract mathematical counterparts according to the corresponding errors.

While the rounding error for floating-point arithmetic enters into the system multiplicatively, it is an additive component for fixed-point arithmetic. In this case the fundamental error analysis theorem for fixed-point arithmetic operations against their abstract mathematical counterparts as shown in Section 3.2 can be stated as

$$fxp\ (x\ *\ y)\ =\ (x\ *\ y)\ +\ \epsilon, \quad \text{where } |\epsilon|\ \leq\ 2^{-fracbits\ (X)} \tag{4.12}$$

and *fracbits* is the number of bits that are to the right of the binary point in the given fixed-point format X. The notation fxp (.) is used to denote that the operation is performed using fixed-point arithmetic. We have proved equations (4.11) and (4.12) as theorems in higher-order logic within HOL. The theorems are proved under the assumption that there is no overflow or underflow in the operation result. This means that the input values are scaled so that the real value of the result is located in the ranges defined by the maximum and minimum representable values of the given floating-point and fixed-point formats.

In equation (4.4) the $\{A_k(p)\}$ are complex numbers, so their real and imaginary parts are calculated separately. Let

$$\begin{array}{ll} B_k(p) = Re\ [A_k(p)] & C_k(p) = Im\ [A_k(p)] \\ U_k(p) = Re\ [w_k(p)] & V_k(p) = Im\ [w_k(p)] \end{array} \tag{4.13}$$

where the notations Re [.] and Im [.] denote, respectively, the real and imaginary parts of the quantity inside the bracket [.]. Equation (4.4) can be rewritten as

$$\left.\begin{array}{l} B_{k+1}(p) = B_k(p) + B_k(q) \\ C_{k+1}(p) = C_k(p) + C_k(q) \end{array}\right\} \quad \text{if } p_k = 0 \tag{4.14}$$

$$\left.\begin{array}{l} B_{k+1}(p) = [B_k(r) - B_k(p)]\ U_k(p) - [C_k(r) - C_k(p)]\ V_k(p) \\ C_{k+1}(p) = [C_k(r) - C_k(p)]\ U_k(p) + [B_k(r) - B_k(p)]\ V_k(p) \end{array}\right\} \quad \text{if } p_k = 1$$

where $q = p + 2^{m-1-k}$ and $r = p - 2^{m-1-k}$. Similarly, we can express the real and imaginary parts of $A'_{k+1}(p)$, $B'_{k+1}(p)$ and $C'_{k+1}(p)$, and $A''_{k+1}(p)$, $B''_{k+1}(p)$ and $C''_{k+1}(p)$, using the floating- and fixed-point operations, respectively. The corresponding error flowgraph showing the effect of roundoff error using the fundamental floating- and fixed-point error analysis theorems according to the equations (4.11) and (4.12), respectively, is given in

Figure 4.4, which also indicates the order of the calculation.

The quantities $\gamma'_{k,p}$, $\gamma''_{k,p}$, $\delta'_{k,p}$, $\delta''_{k,p}$, $\epsilon'_{k,p}$, $\epsilon''_{k,p}$, $\zeta'_{k,p}$, $\zeta''_{k,p}$, $\eta'_{k,p}$, $\eta''_{k,p}$, $\lambda'_{k,p}$, and $\lambda''_{k,p}$ in Figure 4.4 are errors caused by floating-point roundoff at each arithmetic step. The corresponding error quantities for fixed-point roundoff are $\gamma_{k,p}$, $\gamma'''_{k,p}$, $\delta_{k,p}$, $\delta'''_{k,p}$, $\epsilon_{k,p}$, $\epsilon'''_{k,p}$, $\zeta_{k,p}$, $\zeta'''_{k,p}$, $\eta_{k,p}$, $\eta'''_{k,p}$, $\lambda_{k,p}$, and $\lambda'''_{k,p}$. Thereafter, the actual real and imaginary parts of the floating- and fixed-point outputs $A'_{k+1}(p)$ and $A''_{k+1}(p)$, respectively are seen to be given explicitly by

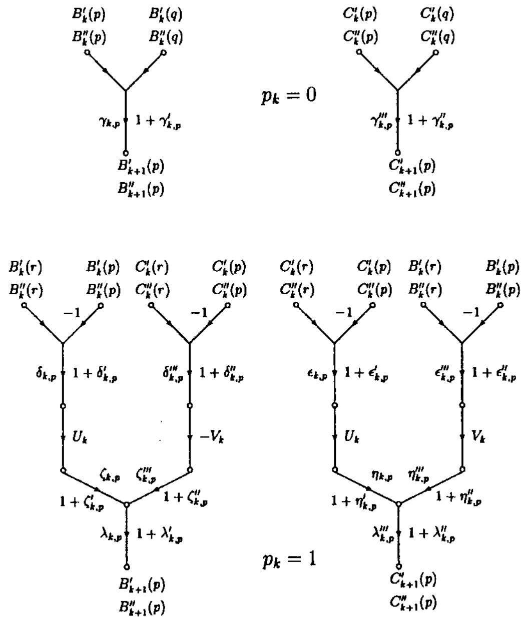


Figure 4.4: Error flowgraph for decimation-in-frequency FFT

$$\left.\begin{aligned} B'_{k+1}(p) &= [B'_k(p) + B'_k(q)](1+\gamma'_{k,p}) \\ C'_{k+1}(p) &= [C'_k(p) + C'_k(q)](1+\gamma''_{k,p}) \end{aligned}\right\} \quad \text{if } p_k = 0 \tag{4.15}$$

$$\left.\begin{aligned} &B'_{k+1}(p) = [B'_k(r) - B'_k(p)]\ U_k(p)(1+\delta'_{k,p})(1+\zeta'_{k,p})(1+\lambda'_{k,p}) \\ &-\ [C'_k(r) - C'_k(p)]\ V_k(p)(1+\delta''_{k,p})(1+\zeta''_{k,p})(1+\lambda'_{k,p}) \\ &C'_{k+1}(p) = [C'_k(r) - C'_k(p)]\ U_k(p)(1+\epsilon'_{k,p})(1+\eta'_{k,p})(1+\lambda''_{k,p}) \\ &+\ [B'_k(r) - B'_k(p)]\ V_k(p)(1+\epsilon''_{k,p})(1+\eta''_{k,p})(1+\lambda''_{k,p}) \end{aligned}\right\} \quad \text{if } p_k = 1$$

and

$$\left.\begin{array}{l} B''_{k+1}(p) = [B''_k(p) + B''_k(q)] + \gamma_{k,p} \\ C''_{k+1}(p) = [C''_k(p) + C''_k(q)] + \gamma'''_{k,p} \end{array}\right\} \quad \text{if } \ p_k = 0 \tag{4.16}$$

$$\left.\begin{array}{l} B''_{k+1}(p) = [B''_k(r) - B''_k(p) + \delta_{k,p}]\ U_k(p) + \zeta_{k,p} - \\ ([C''_k(r) - C''_k(p) + \delta'''_{k,p}]\ V_k(p) + \zeta'''_{k,p}) + \lambda_{k,p} \\ C''_{k+1}(p) = [C''_k(r) - C''_k(p) + \epsilon_{k,p}]\ U_k(p) + \eta_{k,p} + \\ ([B''_k(r) - B''_k(p) + \epsilon'''_{k,p}]\ V_k(p) + \eta'''_{k,p}) + \lambda'''_{k,p} \end{array}\right\} \quad \text{if } \ p_k = 1$$

The errors $e_k(p)$, $e'_k(p)$, and $e''_k(p)$ defined in equations (4.8), (4.9), and (4.10) are complex and can be rewritten as

$$e_k(p) = B'_k(p) - B_k(p) + j[C'_k(p) - C_k(p)] \tag{4.17}$$

$$e'_k(p) = B''_k(p) - B_k(p) + j[C''_k(p) - C_k(p)] \tag{4.18}$$

$$e''_k(p) = B''_k(p) - B'_k(p) + j[C''_k(p) - C'_k(p)] \tag{4.19}$$

$$k = 1, 2, \ldots, m, \ p = 0, 1, \ldots, N-1$$

with

$$e_0(p) = e'_0(p) = e''_0(p) = 0, \qquad p = 0, 1, \ldots, N-1 \tag{4.20}$$

From equations (4.14), (4.15), (4.16), (4.17), (4.18), and (4.19), we derive the following error analysis cases:

1. **FFT Real to Floating-Point:**

$$e_{k+1}(p) = \begin{cases} e_k(p) + e_k(q) + f_k(p) & \text{if } \ p_k = 0 \\ [e_k(r) - e_k(p)]\ w_k(p) + f_k(p) & \text{if } \ p_k = 1 \end{cases} \tag{4.21}$$

where $f_k(p)$ is given by

$$f_k(p) = \begin{cases} \gamma'_{k,p}[B'_k(p) + B'_k(q)] + j\gamma''_{k,p}[C'_k(p) + C'_k(q)] \quad \text{if } \ p_k = 0 \\ [(1+\delta'_{k,p})(1+\zeta'_{k,p})(1+\lambda'_{k,p}) - 1][B'_k(r) - B'_k(p)]U_k(p) \\ -[(1+\delta''_{k,p})(1+\zeta''_{k,p})(1+\lambda'_{k,p}) - 1][C'_k(r) - C'_k(p)]V_k(p) \\ +j[(1+\epsilon'_{k,p})(1+\eta'_{k,p})(1+\lambda''_{k,p}) - 1][C'_k(r) - C'_k(p)]U_k(p) \\ +j[(1+\epsilon''_{k,p})(1+\eta''_{k,p})(1+\lambda''_{k,p}) - 1][B'_k(r) - B'_k(p)]V_k(p) \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \ p_k = 1 \end{cases} \tag{4.22}$$

2. **FFT Real to Fixed-Point:**

$$e'_{k+1}(p) = \begin{cases} e'_k(p) + e'_k(q) + f'_k(p) & \text{if } p_k = 0 \\ [e'_k(r) - e'_k(p)]\ w_k(p) + f'_k(p) & \text{if } p_k = 1 \end{cases} \tag{4.23}$$

where $f'_k(p)$ is given by

$$f'_k(p) = \begin{cases} \gamma_{k,p} + j\gamma'''_{k,p} \quad \text{if } p_k = 0 \\ \delta_{k,p}U_k(p) + \zeta_{k,p} - \delta'''_{k,p}V_k(p) - \zeta'''_{k,p} + \lambda_{k,p} + \\ j(\epsilon_{k,p}U_k(p) + \eta_{k,p} + \epsilon'''_{k,p}V_k(p) + \eta'''_{k,p} + \lambda'''_{k,p}) \quad \text{if } p_k = 1 \end{cases} \tag{4.24}$$

3. **FFT Floating- to Fixed-Point:**

$$e''_{k+1}(p) = \begin{cases} e''_k(p) + e''_k(q) + f'_k(p) - f_k(p) & \text{if } p_k = 0 \\ [e''_k(r) - e''_k(p)]\ w_k(p) + f'_k(p) - f_k(p) & \text{if } p_k = 1 \end{cases} \tag{4.25}$$

where $f_k(p)$ and $f'_k(p)$ are given by equations (4.22) and (4.24).

The accumulation of roundoff error is determined by the recursive equations (4.21), (4.22), (4.23), (4.24), and (4.25), with initial conditions given by equation (4.20).

In HOL, we first constructed complex numbers on reals similar to [32]. We defined in HOL a new type for complex numbers, to be in bijection with $\mathbb{R} \times \mathbb{R}$. The bijections are written in HOL as *complex* : $\mathbb{R}^2 \rightarrow \mathbb{C}$ and *coords* : $\mathbb{C} \rightarrow \mathbb{R}^2$.

```
⊢def  (∀ a. complex (coords a) = a) ∧ (∀ r. coords (complex r) = r)
```

We used convenient abbreviations for the real (*Re*) and imaginary (*Im*) parts of a complex number.

```
⊢def  Re z = FST (coords z)
⊢def  Im z = SND (coords z)
```

and also defined arithmetic operations such as addition, subtraction, and multiplication on complex numbers. We overloaded the usual symbols $(+, -, \times)$ for $\mathbb{C}$ and $\mathbb{R}$.

```
⊢def  compadd a b = (FST a + FST b,SND a + SND b)
```

```
⊢def  compsub a b = (FST a - FST b,SND a - SND b)
```

```
⊢def  compmul a b = (FST a * FST b - SND a * SND b,
                      FST a * SND b - SND a * FST b)
```

```
⊢def  w complex_add z = complex (compadd (coords w) (coords z))
```

```
⊢def  w complex_sub z = complex (compsub (coords w) (coords z))
```

```
⊢def  w complex_mul z = complex (compmul (coords w) (coords z))
```

Furthermore, we defined using recursive definition in HOL expressions for the finite summation on complex numbers.

```
⊢def  (complex_sum (n,0) f = complex (0,0)) ∧
       (complex_sum (n,SUC m) f = complex_sum (n,m) f + f (n + m))
```

Similarly, we constructed complex numbers on floating-point numbers (*float_complex, float_coords, float_Re, float_Im, float_complex_add, float_complex_sub, float_complex_mul, float_complex_sum*) and fixed-point numbers (*fxp_complex, fxp_coords, fxp_Re, fxp_Im, fxp_complex_add, fxp_complex_sub, fxp_complex_mul, fxp_complex_sum*). We also defined rounding and valuation functions for floating-point (*float_complex_round, float_complex_Val*) and fixed-point (*fxp_complex_round, fxp_complex_value*) complex numbers.

Then we defined the principal N-roots on unity ($e^{-j2\pi n/N} = cos\ (2\pi n/N) - j\ sin\ (2\pi n/N)$), and its powers (*OMEGA*) as a complex number using the sine and cosine functions available in the transcendental theory of the HOL reals library [27].

```
⊢def  principal_root_1 n N =
       complex (cos ¬2 * pi * & n / & N , sin ¬2 * pi * & n / & N)
```

We specified expressions in HOL for expansion of a natural number into a binary form in normal and rearranged order according to the equations (4.2), (4.3), and (4.5).

```
⊢def  DIG n m = (m DIV 2 ** n) MOD 2
```

```
⊢def  Binary_Form p m = (λ k. DIG (m − 1 − k) p)
```

```
⊢def  Log_2 p = @k. p = 2 ** k
```

```
⊢def  (num_sum (n,0) f = 0) ∧
       (num_sum (n,SUC m) f = num_sum (n,m) f + f (n + m))
```

```
⊢def  Z k p N = 2 ** k * num_sum (k,Log_2 N − k)
      (λ i. 2 **(Log_2 N − 1 − i) * DIG i p) −
      2 ** (Log_2 N − 1) * DIG k p
```

```
⊢def  p_star p m = num_sum (0,m) (λ i. 2 ** m * DIG i p)
```

The above enables us to specify the FFT algorithms in real (*FFT*), floating- (*FLOAT_FFT*), and fixed-point (*FXP_FFT*) abstraction levels using recursive definitions in HOL as described in equation (4.4).

```
⊢def  (FFT x N 0 = (λ p. x p)) ∧
         FFT x N (SUC k) =

     (λ p.
            (if DIG k p = 0 then
               FFT x N k p + FFT x N k (p + 2 ** (Log_2 N − 1 − k))
             else
               (FFT x N k (p − 2 ** (Log_2 N − 1 − k)) − FFT x N k p) *
               OMEGA k p N))
```

Then we define the real and imaginary parts of the FFT algorithm (*FFT_REAL*, *FFT_IMAGE*) and powers of the principal N-roots on unity (*OMEGA_REAL*, *OMEGA_IMAGE*) according to the equation (4.13).

$\vdash_{def}$ FFT_REAL x N k p = Re (FFT x N k p)

$\vdash_{def}$ FFT_IMAGE x N k p = Im (FFT x N k p)

$\vdash_{def}$ OMEGA_REAL k p N = Re (OMEGA k p N)

$\vdash_{def}$ OMEGA_IMAGE k p N = Im (OMEGA k p N)

Later, we prove in separate lemmas that the real and imaginary parts of the FFT algorithm in real, floating-point, and fixed-point levels can be expanded as in equation (4.14). In following, we show the HOL expansion theorem (*Lemma 1*) for real numbers. Similar lemmas have been derived for the floating- and fixed-point levels.

```
Lemma 1:
        ∀ x N k p.
          (if DIG k p = 0 then
             (FFT_REAL x N (SUC k) p =
              FFT_REAL x N k p +
              FFT_REAL x N k (p + 2 ** (Log_2 N - 1 - k))) ∧
             (FFT_IMAGE x N (SUC k) p =
              FFT_IMAGE x N k p +
              FFT_IMAGE x N k (p + 2 ** (Log_2 N - 1 - k)))
           else
             (FFT_REAL x N (SUC k) p =
              (FFT_REAL x N k (p - 2 ** (Log_2 N - 1 - k)) -
               FFT_REAL x N k p) * OMEGA_REAL k p N -
              (FFT_IMAGE x N k (p - 2 ** (Log_2 N - 1 - k)) -
               FFT_IMAGE x N k p) * OMEGA_IMAGE k p N) ∧

       (FFT_IMAGE x N (SUC k) p =
              (FFT_IMAGE x N k (p - 2 ** (Log_2 N - 1 - k)) -
               FFT_IMAGE x N k p) * OMEGA_REAL k p N +
              (FFT_REAL x N k (p - 2 ** (Log_2 N - 1 - k)) -
               FFT_REAL x N k p) * OMEGA_IMAGE k p N))
```

Then we prove lemmas to introduce an error in each of the arithmetic steps in real and imaginary parts of the floating-point and fixed-point FFT algorithms according to the equations (4.15), and (4.16). In following, we show the HOL theorem (*Lemma 2*) corresponding to the error analysis of the transition from real to floating-point. Similar theorems have been proven for the transitions from, respectively, real and floating-point to the fixed-point level.

```
Lemma 2:
        ∀ x N k p.
          ∃ e.
            ∀ i.
              1 ≤ i ∧ i ≤ 12 ⟹
              e i ≤ 1 / 2 pow 24 ∧
              (if DIG k p = 0 then
                 (Val (FLOAT_FFT_REAL x N (SUC k) p) =
                  (Val (FLOAT_FFT_REAL x N k p) +
                   Val
                 (FLOAT_FFT_REAL x N k (p + 2 ** (Log_2 N - 1 - k)))) *
                  (1 + e 1)) ∧
                 (Val (FLOAT_FFT_IMAGE x N (SUC k) p) =
                  (Val (FLOAT_FFT_IMAGE x N k p) +
                   Val
                     (FLOAT_FFT_IMAGE x N k
                        (p + 2 ** (Log_2 N - 1 - k)))) * (1 + e 2))
else
  (Val (FLOAT_FFT_REAL x N (SUC k) p) =
   (Val
  (FLOAT_FFT_REAL x N k (p - 2 ** (Log_2 N - 1 - k))) -
    Val (FLOAT_FFT_REAL x N k p)) *
   Val (FLOAT_OMEGA_REAL k p N) * (1 + e 3) * (1 + e 4) *
   (1 + e 5) -
   (Val
 (FLOAT_FFT_IMAGE x N k (p - 2 ** (Log_2 N - 1 - k))) -
    Val (FLOAT_FFT_IMAGE x N k p)) *
  Val (FLOAT_OMEGA_IMAGE k p N) * (1 + e 6) * (1 + e 7) *
   (1 + e 5)) ∧
  (Val (FLOAT_FFT_IMAGE x N (SUC k) p) =
   (Val
      (FLOAT_FFT_IMAGE x N k (p - 2 ** (Log_2 N - 1 - k))) -
    Val (FLOAT_FFT_IMAGE x N k p)) *
   Val (FLOAT_OMEGA_REAL k p N) * (1 + e 8) * (1 + e 9) *
   (1 + e 10) +
   (Val
      (FLOAT_FFT_REAL x N k (p - 2 ** (Log_2 N - 1 - k))) -
    Val (FLOAT_FFT_REAL x N k p)) *
   Val (FLOAT_OMEGA_IMAGE k p N) * (1 + e 11) * (1 + e 12) *
   (1 + e 10)))
```

We prove these lemmas using the fundamental error analysis lemmas for basic arithmetic operations according to the equations (4.11) and (4.12). Then we defined in HOL the error of the pth element of the floating- (*FLOAT_TO_REAL_FFT_ERROR*) and fixed-point (*FXP_TO_REAL_FFT_ERROR*) FFT algorithms at step k, and the corresponding error in transition from floating- to fixed-point (*FLOAT_TO_FXP_FFT_ERROR*), according to the equations (4.8), (4.9), and (4.10).

```
⊢def FLOAT_TO_REAL_FFT_ERROR x N k p =
       float_complex_Val (FLOAT_FFT x N k p) - FFT x N k p
```

Thereafter, we prove lemmas to rewrite the errors as complex numbers using the real and imaginary parts according to the equations (4.17), (4.18), and (4.19), respectively. The HOL theorem (*Lemma 3*) for the real numbers to floating-point transition is given below.

```
Lemma 3:
        ∀ x N k p.
          FLOAT_TO_REAL_FFT_ERROR x N k p =
          complex
            (Val (FLOAT_FFT_REAL x N k p) - FFT_REAL x N k p,
             Val (FLOAT_FFT_IMAGE x N k p) - FFT_IMAGE x N k p)
```

Finally, we prove a set of lemmas to determine the accumulation of roundoff error in floating- and fixed-point FFT algorithms by recursive equations and initial conditions according to the equations (4.20), (4.21), (4.22), (4.23), (4.24), and (4.25). *Lemma 4* represents the HOL theorem for the transition from real numbers to floating-point.

```
Lemma 4:
        ∀x N k p.
          (FLOAT_TO_REAL_FFT_ERROR x N 0 p = complex (0,0)) ∧
          ∃f.
            (FLOAT_TO_REAL_FFT_ERROR x N (SUC k) p =
             (if DIG k p = 0 then
                FLOAT_TO_REAL_FFT_ERROR x N k p +
                FLOAT_TO_REAL_FFT_ERROR x N k (p + 2 ** (Log_2 N - 1 - k)) +
                f x N k p
              else
                (FLOAT_TO_REAL_FFT_ERROR x N k (p - 2 ** (Log_2 N - 1 - k)) -
                 FLOAT_TO_REAL_FFT_ERROR x N k p) * OMEGA k p N + f x N k p)) ∧
            ∃e.
              ∀i.
                1 ≤ i ∧ i ≤ 12 ⟹
                e i ≤ 1 / 2 pow 24 ∧
                (f x N k p =
                 (if DIG k p = 0 then
  complex
    (e 1 *
     (Val (FLOAT_FFT_REAL x N k p) +
      Val
        (FLOAT_FFT_REAL x N k
           (p + 2 ** (Log_2 N - 1 - k)))),
     e 2 *
     (Val (FLOAT_FFT_IMAGE x N k p) +
      Val
        (FLOAT_FFT_IMAGE x N k
           (p + 2 ** (Log_2 N - 1 - k)))))
else
  complex
    (((1 + e 3) * (1 + e 4) * (1 + e 5) - 1) *
     (Val
        (FLOAT_FFT_REAL x N k
           (p - 2 ** (Log_2 N - 1 - k))) -
  Val (FLOAT_FFT_REAL x N k p)) * OMEGA_REAL k p N -
     ((1 + e 6) * (1 + e 7) * (1 + e 5) - 1) *
     (Val
        (FLOAT_FFT_IMAGE x N k
           (p - 2 ** (Log_2 N - 1 - k))) -
      FFT_IMAGE x N k p) * OMEGA_IMAGE k p N,
     ((1 + e 8) * (1 + e 9) * (1 + e 10) - 1) *
     (Val
        (FLOAT_FFT_IMAGE x N k
           (p - 2 ** (Log_2 N - 1 - k))) -
  Val (FLOAT_FFT_IMAGE x N k p)) * OMEGA_REAL k p N -
     ((1 + e 11) * (1 + e 12) * (1 + e 10) - 1) *
     (Val
        (FLOAT_FFT_REAL x N k
           (p - 2 ** (Log_2 N - 1 - k))) -
      Val (FLOAT_FFT_REAL x N k p)) *
     OMEGA_IMAGE k p N))
```

4.3 FFT Design Implementation Verification

In this section, we describe the application of the proposed approach for the verification in HOL of the transition from real, floating- and fixed-point specifications to RTL and gate level netlist implementations of an FFT algorithm. We have chosen the case study of a radix-4 pipelined 16-point complex FFT core available as a VHDL RTL model in the Xilinx Coregen library [76]. We have also used Synopsys tools to generate the gate level netlist of the design. All proofs have been conducted in HOL, hence establishing a correctness of the FFT design implementation with respect to its high level algorithmic specifications.

Figure 4.5 shows the overall block diagram of the Radix-4 16-point pipelined FFT design. The basic elements are memories, delays, multiplexers, and dragonflies. In general, the 16-point pipelined FFT requires the calculation of two radix-4 dragonfly ranks. Each radix-4 dragonfly is a successive combination of a radix-4 butterfly with four twiddle factor multipliers. The FFT core accepts naturally ordered data on the input buses in a continuous stream, performs a complex FFT, and streams out the DFT samples on the output buses in a natural order. These buses are respectively the real and imaginary components of the input and output sequences. An internal input data memory controller orders the data into blocks to be presented to the FFT processor. The twiddle factors are stored in coefficient memories. The real and imaginary components of complex input and output samples and the phase factors are represented as 16-bit 2's complement numbers. The unscrambling operation is performed using the output bit-reversing buffer.

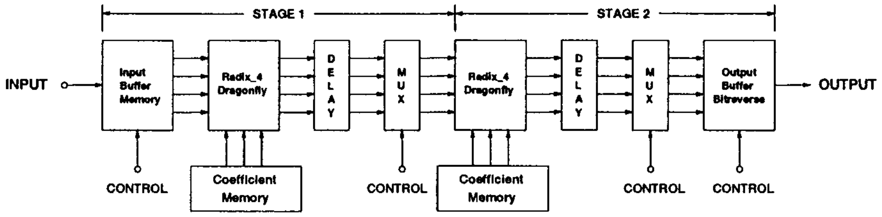


Figure 4.5: Radix-4 16-point pipelined FFT implementation

To define the radix-4 FFT algorithm [9, 59], we represent the indices p and n in equation (4.1) in a base 4 (quaternary number system) as

$$p = 4p_1 + p_0, \qquad p_1, p_0 = 0, 1, 2, 3 \tag{4.26}$$

$$n = 4n_1 + n_0, \qquad n_1, n_0 = 0, 1, 2, 3 \tag{4.27}$$

It is easy to verify that as n_0 and n_1 take on all possible values in the range indicated, n goes through all possible values from 0 to 15 with no values repeated. This is also true for the frequency index p. Using these index mappings, we can express the radix-4 16-point FFT algorithm recursively as

$$A_1(p_0, n_0) = \sum_{n_1=0}^{3} x(n_1, n_0)\ (W_{16})^{4p_0 n_1} \tag{4.28}$$

$$A_2(p_0, p_1) = \sum_{n_0=0}^{3} A_1(p_0, n_0)\ (W_{16})^{(4p_1+p_0)n_0} \tag{4.29}$$

The final result can be written as

$$A(p_1, p_0) = A_2(p_0, p_1) \tag{4.30}$$

Thus, as in the radix-2 algorithm, the results are in reversed order. Based on equations (4.28), (4.29), and (4.30) we can develop a signal flowgraph for the radix-4 16-point FFT algorithm as shown in Figure 4.6, which is an expanded version of the pipelined implementation of Figure 4.5. The graph is composed of two successive radix-4 dragonfly stages. To alleviate confusion in this graph we have shown only one of the radix-4 butterflies in the first stage. Also, we have not shown the multipliers for the radix-4 butterflies in the second stage since they are similar to the representative butterfly of the first stage. Figure 4.6 also illustrates the unscrambling procedure for the radix-4 algorithm.

In HOL, we first modeled the RTL description of a radix-4 butterfly as a predicate in higher-order logic (*radix_4_butterfly_RTL*).

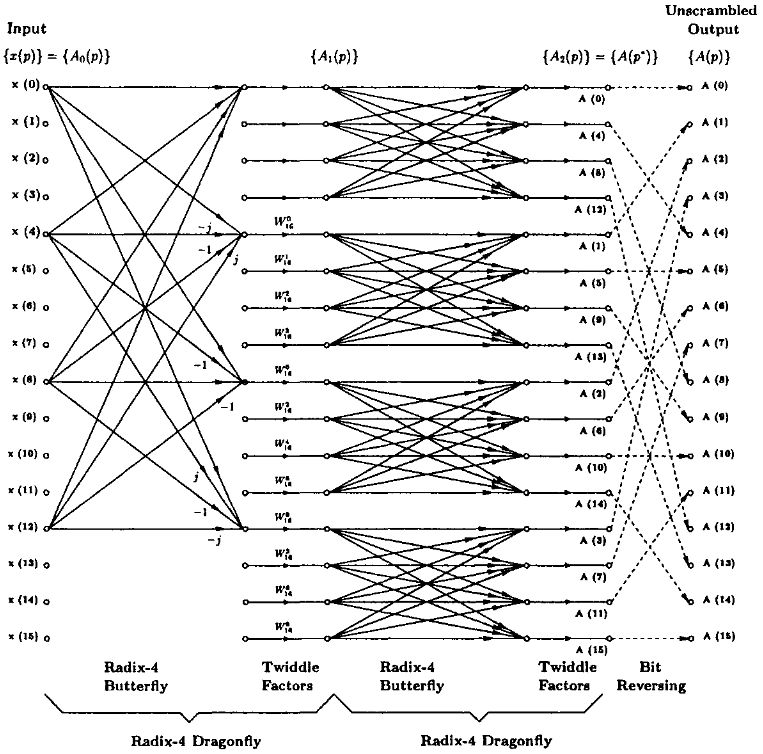


Figure 4.6: Signal flowgraph of radix-4 16-point FFT

```
⊢def radix_4_butterfly_RTL N ar ai br bi cr ci dr di q1r q1i q2r q2i q3r
          q3i q4r q4i =
        (∃ y1r y1i y2r y2i.
          N_complex_add_RTL N ar ai br bi y1r y1i ∧
          N_complex_add_RTL N cr ci dr di y2r y2i ∧
          N_complex_add_RTL N y1r y1i y2r y2i q1r q1i) ∧
          (∃ y3r y3i y4r y4i y5r y5i y6r y6i.
N_complex_mul_two_comp_RTL  N br bi (NBWORD N 0) (NBWORD N 1) y3r y3i ∧
N_complex_sub_RTL N ar ai y3r y3i y4r y4i ∧
N_complex_mul_two_comp_RTL N dr di (NBWORD N 0) (NBWORD N 1) y5r y5i ∧
N_complex_sub_RTL N y5r y5i cr ci y6r y6i ∧
N_complex_add_RTL N y4r y4i y6r y6i q2r q2i) ∧
(∃ y7r y7i y8r y8i.
N_complex_sub_RTL N ar ai br bi y7r y7i ∧
N_complex_sub_RTL N cr ci dr di y8r y8i ∧
N_complex_add_RTL N y7r y7i y8r y8i q3r q3i) ∧
```

```
∃ y9r y9i y10r y10i y11r y11i y12r y12i.
N_complex_mul_two_comp_RTL N br bi (NBWORD N 0) (NBWORD N 1) y9r y9i ∧
N_complex_add_RTL N ar ai y9r y9i y10r y10i ∧
N_complex_mul_two_comp_RTL N dr di (NBWORD N 0) (NBWORD N 1) y11r y11i ∧
N_complex_add_RTL N y11r y11i cr ci y12r y12i ∧
N_complex_sub_RTL N y10r y10i y12r y12i q4r q4i
```

The block takes a vector of four complex input data and performs the operations as depicted in the flowgraph of Figure 4.6, to generate a vector of four complex output signals. The real and imaginary parts of the input and output signals are represented as 16-bit Boolean words. We defined separate functions in HOL for arithmetic operations such as addition (*N_complex_add_RTL*), subtraction (*N_complex_sub_RTL*), and multiplication (*N_complex_mul_two_comp_RTL*) on complex two's complement 16-bit Boolean words. Then, we built the complete butterfly structure using a proper combination of these primitive operations.

Thereafter, we described a radix-4 dragonfly block (*radix_4_dragonfly_RTL*) as a conjunction of a radix-4 butterfly and four 16-bit twiddle factor complex multipliers as shown in Figure 4.6.

```
⊢def radix_4_dragonfly_RTL N ar ai br bi cr ci dr di wr wi
       q1r q1i q2r q2i q3r q3i q4r q4i =
        ∃ s1 s2 s3 s4 s5 s6 s7 s8.
          radix_4_butterfly_RTL N ar ai br bi cr ci dr di
          s1 s2 s3 s4 s5 s6 s7 s8 ∧
          N_complex_mul_two_comp_RTL N s1 s2 (wr 1) (wi 1) q1r q1i ∧
          N_complex_mul_two_comp_RTL N s3 s4 (wr 2) (wi 2) q2r q2i ∧
          N_complex_mul_two_comp_RTL N s5 s6 (wr 3) (wi 3) q3r q3i ∧
          N_complex_mul_two_comp_RTL N s7 s8 (wr 4) (wi 4) q4r q4i
```

Finally, we modeled the complete RTL description of the radix-4 16-point FFT structure (*radix_4_16_point_DIF_FFT_RTL*) in HOL. The FFT block is defined as a conjunction of 8 instantiations of radix-4 dragonfly blocks according to Figure 4.6, by applying the proper time instances of the input and output signals to each block.

```
⊢def radix_4_16_point_DIF_FFT_RTL  xr xi ar ai wr wi =
       ∃ a1r a1i.
       radix_4_dragonfly_RTL N (xr 0) (xi 0) (xr 4) (xi 4) (xr 8) (xi 8)
         (xr 12) (xi 12) (wr 0) (wi 0) (a1r 0) (a1i 0) (a1r 4) (a1i 4)
         (a1r 8) (a1i 8) (a1r 12) (a1i 12) ∧
       radix_4_dragonfly_RTL N (xr 1) (xi 1) (xr 5) (xi 5) (xr 9) (xi 9)
         (xr 13) (xi 13) (wr 1) (wi 1) (a1r 1) (a1i 1) (a1r 5) (a1i 5)
         (a1r 9) (a1i 9) (a1r 13) (a1i 13) ∧
       radix_4_dragonfly_RTL N (xr 2) (xi 2) (xr 6) (xi 6) (xr 10)
         (xi 10) (xr 14) (xi 14) (wr 2) (wi 2) (a1r 2) (a1i 2) (a1r 6)
         (a1i 6) (a1r 10) (a1i 10) (a1r 14) (a1i 14) ∧
       radix_4_dragonfly_RTL N (xr 3) (xi 3) (xr 7) (xi 7) (xr 11)
         (xi 11) (xr 15) (xi 15) (wr 3) (wi 3) (a1r 3) (a1i 3) (a1r 7)
         (a1i 7) (a1r 11) (a1i 11) (a1r 15) (a1i 15) ∧
       radix_4_dragonfly_RTL N (a1r 0) (a1i 0) (a1r 1) (a1i 1) (a1r 2)
         (a1i 2) (a1r 3) (a1i 3) (wr 4) (wi 4) (ar 0) (ai 0) (ar 4)
         (ai 4) (ar 8) (ai 8) (ar 12) (ai 12) ∧
       radix_4_dragonfly_RTL N (a1r 4) (a1i 4) (a1r 5) (a1i 5) (a1r 6)
         (a1i 6) (a1r 7) (a1i 7) (wr 5) (wi 5) (ar 1) (ai 1) (ar 5)
         (ai 5) (ar 9) (ai 9) (ar 13) (ai 13) ∧
       radix_4_dragonfly_RTL N (a1r 8) (a1i 8) (a1r 9) (a1i 9) (a1r 10)
         (a1i 10) (a1r 11) (a1i 11) (wr 6) (wi 6) (ar 2) (ai 2) (ar 6)
         (ai 6) (ar 10) (ai 10) (ar 14) (ai 14) ∧
       radix_4_dragonfly_RTL N (a1r 12) (a1i 12) (a1r 13) (a1i 13)
         (a1r 14) (a1i 14) (a1r 15) (a1i 15) (wr 7) (wi 7) (ar 3) (ai 3)
         (ar 7) (ai 7) (ar 11) (ai 11) (ar 15) (ai 15)
```

Following similar steps, we described a radix-4 16-point FFT structure as fixed-point (*radix_4_16_point_DIF_FFT_fxp*), floating-point (*radix_4_16_point_DIF_FFT_float*), and real (*radix_4_16_point_DIF_FFT_real*) domains in HOL using the corresponding complex data types and arithmetic operations.

For the formal verification of the case study of the radix-4 decimation in frequency FFT algorithm based on the commutating diagram of Figure 1.2, we proved that the FFT

RTL description implies the corresponding fixed-point model (*Lemma 5*).

```
Lemma 5:
       ∀ N xr xi ar ai wr wi.
         radix_4_16_point_DIF_FFT_RTL N xr xi ar ai wr wi ⟹
         radix_4_16_point_DIF_FFT_FXP N (FXP_VECT_COMPLEX N xr xi)
           (FXP_VECT_COMPLEX N ar ai) (FXP_VECT_COMPLEX N wr wi)
```

The proof of the FFT block is then broken down into the corresponding proof of the dragonfly block, which itself is broken down to the proof of butterfly and primitive arithmetic operations.

```
Lemma 6:
     ∀ N ar ai br bi cr ci dr di q1r q1i q2r q2i q3r q3i q4r q4i wr wi.
      radix_4_dragonfly_RTL  ar ai br bi cr ci dr di wr wi q1r q1i q2r
                                  q2i q3r q3i q4r q4i ⟹
      radix_4_dragonfly_FXP (N,N - 1,1) (fxp_complex (FXP N ar,FXP N ai))
            (fxp_complex (FXP N br,FXP N bi)) (fxp_complex (FXP N cr,FXP N ci))
            (fxp_complex (FXP N dr,FXP N di)) (FXP_VECT_COMPLEX N wr wi)
            (fxp_complex (FXP N q1r,FXP N q1i)) (fxp_complex (FXP N q2r,FXP N q2i))
            (fxp_complex (FXP N q3r,FXP N q3i)) (fxp_complex (FXP N q4r,FXP N q4i))
```

We used the data abstraction functions *FXP* and *FXP_VECT_COMPLEX* to convert a complex vector of 16-bit two's complement Boolean words into the corresponding fixed-point vector.

For the error analysis of the radix-4 decimation in frequency FFT algorithm and following the discussions in Section 4.2, we proved the theorems below, which state the error between the real values of, respectively, the floating-point (*Lemma 7*) and fixed-point (*Lemma 8*) precision output samples and the corresponding ideal real specification. We also proved a theorem (*Lemma 9*) on the error from the transition from floating-point to fixed-point specifications.

```
Lemma 7:
        ∀ N xr xi ar ai wr wi.
          radix_4_16_point_DIF_FFT_FLOAT N (FLOAT_VECT_COMPLEX N xr xi)
          (FLOAT_VECT_COMPLEX N ar ai) (FLOAT_VECT_COMPLEX N wr wi) ⟹
          radix_4_16_point_DIF_FFT_REAL N (REAL_VECT_COMPLEX N xr xi)
            (REAL_VECT_COMPLEX N ar ai) (REAL_VECT_COMPLEX N wr wi) ∧
          FLOAT_TO_REAL_FFT_ERROR N xr xi ar ai wr wi

Lemma 8:
        ∀ N xr xi ar ai wr wi.
          radix_4_16_point_DIF_FFT_FXP N (FXP_VECT_COMPLEX N xr xi)
            (FXP_VECT_COMPLEX N ar ai) (FXP_VECT_COMPLEX N wr wi)  ⟹
          radix_4_16_point_DIF_FFT_REAL N (REAL_VECT_COMPLEX N xr xi)
            (REAL_VECT_COMPLEX N ar ai) (REAL_VECT_COMPLEX N wr wi) ∧
          FXP_TO_REAL_FFT_ERROR N xr xi ar ai wr wi

Lemma 9:
        ∀ N xr xi ar ai wr wi.
          radix_4_16_point_DIF_FFT_FXP N (FXP_VECT_COMPLEX N xr xi)
            (FXP_VECT_COMPLEX N ar ai) (FXP_VECT_COMPLEX N wr wi)  ⟹
          radix_4_16_point_DIF_FFT_FLOAT N (FLOAT_VECT_COMPLEX N xr xi)
            (FLOAT_VECT_COMPLEX N ar ai) (FLOAT_VECT_COMPLEX N wr wi) ∧
          FLOAT_TO_FXP_FFT_ERROR N xr xi ar ai wr wi
```

According to these theorems, the floating-point and fixed-point implementations and the real specification of a radix-4 decimation in frequency FFT algorithm are related to each other based on the corresponding data abstraction (*FLOAT_VECT_COMPLEX, FXP_VECT_COMPLEX, REAL_VECT_COMPLEX*), and error analysis (*FLOAT_TO_REAL_FFT_ERROR, FXP_TO_REAL_FFT_ERROR, FLOAT_TO_FXP_FFT_ERROR*) functions. These errors are already quantified using the theorems mentioned in Section 4.2.

Finally, using the obtained theorems (*Lemma 5, Lemma 8*), we can easily deduce our ultimate theorem (*Lemma 10*) proving the correctness of the real specification from the RTL implementation, taking into account the error analysis computed beforehand.

```
Lemma 10:
        ∀ N xr xi ar ai wr wi.
          radix_4_16_point_DIF_FFT_RTL N xr xi ar ai wr wi ==>
          radix_4_16_point_DIF_FFT_REAL N (REAL_VECT_COMPLEX N xr xi)
            (REAL_VECT_COMPLEX N ar ai) (REAL_VECT_COMPLEX N wr wi) ∧
          FXP_TO_REAL_FFT_ERROR N xr xi ar ai wr wi
```

4.4 Conclusion

In this chapter, we described a comprehensive methodology for the verification of generic fast Fourier transform algorithms using the HOL theorem prover. We believe this is the first time a complete formal framework has been proposed for the specification and verification of the fast Fourier transform algorithms at different levels of abstraction. The methodology presented in this chapter opens new avenues in using formal methods for the verification of digital signal processing (DSP) systems as complement to traditional theoretical (analytical) and simulation techniques.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we first established the formalization of fixed-point arithmetic in the HOL theorem prover. Unlike floating-point arithmetic, there is no standard for the fixed-point counterpart. We hence defined in this thesis a complete common set of the fixed-point arithmetic supported by most DSP tools, in particular SPW and SystemC. We started first by encoding the fixed-point arithmetic in HOL considering different quantization and overflow modes, as well as exception handling. We then proved two main theorems stating that the operations on fixed-point numbers are closely related to the corresponding operations on infinitely precise values, considering some error. The error is bounded to a certain absolute value which is a function of the output precision. We have also shown by an example how these theorems can be used as a basis for analysis of the quantization errors in the design of fixed-point DSP subsystems. The formalization presented in this thesis can be considered as a complement to the floating-point formalizations which are widely available in the literature. The developed theories have been accepted by the HOL developers to be included in the new public release of HOL.

Based on the developed fixed-point theories, we proposed a comprehensive methodology for the error analysis of generic digital filters using the HOL theorem prover. The proposed approach covers the three basic forms (direct, parallel and cascade) of realization entirely specified in HOL. We made use of existing theories in HOL on real, IEEE standard based floating-point, and fixed-point arithmetic to model the ideal filter specification and the corresponding implementations in higher-order logic. We used valuation functions to define the errors as the differences between the real values of the floating-point and fixed-point filter implementation outputs and the corresponding output of the ideal real filter specification. Finally, we established fundamental analysis lemmas as our model to derive expressions for the accumulation of the roundoff error in digital filters. Related work did exist since the late sixties using theoretical paper-and-pencil proofs and simulation techniques. The authors believe this is the first time a complete formal framework is considered using mechanical proofs in HOL for the error analysis of digital filters.

Furthermore, we established a more elaborated methodology for the verification of generic fast Fourier transform algorithms using the HOL theorem prover. The approach covers the two canonical forms (decimation-in-time, and decimation-in-frequency) of realization of the FFT algorithm using real, floating-, and fixed-point arithmetic as well as their RT implementations, each entirely specified in HOL. We proved lemmas to derive expressions for the accumulation of roundoff error in floating- and fixed-point designs compared to the ideal real specification. Then we proved that the FFT RTL implementation implies the corresponding specification at the fixed-point level using classical hierarchical verification in HOL, hence bridging the gap between hardware implementation and high levels of mathematical specification. In this work we also have contributed to the upgrade and application of established real, complex real, floating- and fixed-point theories in HOL to the analysis of errors due to finite precision effects, and applied them on the realization of the FFT algorithms. Error analyses using theoretical paper-and-pencil proofs do exist since the late sixties while design verification is exclusively done by simulation techniques. We believe this is the first time a complete formal framework has been proposed for the specification and verification of the fast Fourier transform algorithms at different levels of abstraction.

5.2 Future Work

The methodology presented in this thesis opens new avenues in using formal methods for the verification of DSP systems as a complement to the traditional theoretical (analytical) and simulation techniques. There are many opportunities for further work to improve our approach on verifying DSP systems.

- Extend the error analysis lemmas to analyse the worst-case, average, and variance errors.
- Develop a mechanized theory on the properties of random variables and processes for statistical error analysis in HOL.
- Link HOL with computer algebra systems (Maple [13], Mathematica [75]) to create a sound, reliable, and powerful system for the verification of DSP systems.
- Prove the correctness of automatic transitions from floating-point to fixed-point levels.
- Investigate the verification of complex wired and wireless communication systems, whose building blocks, heavily make use of several instances of the FFT algorithms such as OFDM (Orthogonal Frequency Division Multiplexing) modems [58].

Bibliography

[1] M. D. Aagaard and C. -J. H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," In Proceedings International Conference on Computer Aided Design, pp. 7-10, San Jose, California, USA, November 1995.

[2] G. Barrett, "Formal Methods Applied to a Floating Point Number System," IEEE Transactions on Software Engineering, SE-15 (5): 611-621, May 1989.

[3] C. Berg and C. Jacobi, "Formal Verification of the VAMP Floating Point Unit," In Correct Hardware Design and Verification Methods, LNCS 2144, pp. 325-339, Springer-Verlag, 2001.

[4] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel, "Experience with Embedding Hardware Description Languages in HOL," In Theorem Provers in Circuit Design, pp. 129-156, North-Holland, 1992.

[5] P. Bjesse, "Automatic Verification of Combinational and Pipelined FFT Circuits," In Computer Aided Verification, LNCS 1633, pp. 380-393, Springer-Verlag, 1999.

[6] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP," In Correct Hardware Design and Verification Methods, LNCS 2860, pp. 51-65, Springer-Verlag, 2003.

[7] S. Boldo and M. Daumas, "Properties of Two's Complement Floating Point Notations," Software Tools for Technology Transfer, 5 (2-3): 237-246, March 2004.

[8] S. Boldo, M. Daumas, and L. Thèry, "Formal Proofs and Computations in Finite Precision Arithmetic," In Proceedings of the 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, pp. 101-111, Rome, Italy, September 2003.

[9] E. O. Brigham, "The Fast Fourier Transform," Prentice Hall, 1974.

[10] V. Capretta, "Certifying the Fast Fourier Transform with Coq," In Theorem Proving in Higher Order Logics, LNCS 2152, pp. 154-168, Springer-Verlag, 2001.

[11] V. A. Carreno, "Interpretation of IEEE-854 Floating-Point Standard and Definition in the HOL System," NASA TM-110189, September 1995.

[12] Cadence Design Systems, Inc., "Signal Processing WorkSystem (SPW) User's Guide," USA, July 1999.

[13] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt, "A Tutorial Introduction to Maple V," Springer-Verlag, 1992.

[14] Y. -A. Chen and R. E. Bryant, "Verification of Floating Point Adders," In Computer Aided Verification, LNCS 1427, pp. 488-499, Springer-Verlag, 1998.

[15] Synopsys, Inc., "CoCentric^{TM} System Studio User's Guide," USA, Aug. 2001.

[16] W. T. Cochran et. al., "What is the Fast Fourier Transform," IEEE Transactions on Audio and Electroacoustics, AU-15: 45-55, Jun. 1967.

[17] M. Cornea-Hasegan, "Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," Intel Technology Journal, Q2: 1-11, 1998.

[18] J. W. Cooley and J. W. Tukey, "An Algorithm for Machine Calculation of Complex Fourier Series," Mathematics of Computation, 19: 297-301, Apr. 1965.

[19] M. Daumas, L. Rideau, and L. Thèry, "A Generic Library for Floating-Point Numbers and Its Application to Exact Computing," In Theorem Proving in Higher Order Logics, LNCS 2152, pp. 169-184, Springer-Verlag, 2001.

[20] G. Forsythe and C. B. Moler, "Computer Solution of Linear Algebraic Systems," Prentice-Hall, 1967.

[21] R. A. Gamboa, "The Correctness of the Fast Fourier Transform: A Structural Proof in ACL2," Formal Methods in System Design, Special Issue on UNITY, Jan. 2002.

[22] W. M. Gentleman and G. Sande, "Fast Fourier Transforms - For Fun and Profit," In AFIPS Fall Joint Computer Conference, Vol. 29, pp. 563-578, Spartan Books, Washington, DC, 1966.

[23] M. J. C. Gordon and T. F. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic," Cambridge University Press, 1993.

[24] M.J.C. Gordon, R. Milner, and C. P. Wadsworth, "Edinburgh LCF: A Mechanised Logic of Computation," Lecture Notes in Computer Science, vol. 78, Springer-Verlag, 1979.

[25] B. Gold, and C. M. Radar, "Effects of Quantization Noise in Digital Filters," In Proceedings AFIPS Spring Joint Computer Conference, vol. 28, pp. 213-219, 1966.

[26] M. J. C. Gordon and T. F. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic," Cambridge University Press, 1993.

[27] J. R. Harrison, "Constructing the Real Numbers in HOL," Formal Methods in System Design, 5 (1/2): 35-59, 1994.

[28] J. R. Harrison, "A Machine-Checked Theory of Floating-Point Arithmetic," In Theorem Proving in Higher Order Logics, LNCS 1690, pp. 113-130, Springer-Verlag, 1999.

[29] J. R. Harrison, "Floating-Point Verification in HOL Light: The Exponential Function," Formal Methods in System Design, 16 (3): 271-305, 2000.

[30] J. R. Harrison, "Formal Verification of Floating Point Trigonometric Functions," In Formal Methods in Computer-Aided Design, LNCS 1954, pp. 217-233, Springer-Verlag, 2000.

[31] J. R. Harrison, "Formal Verification of IA-64 Division Algorithms," In Theorem Proving in Higher Order Logics, LNCS 1869, pp. 234-251, Springer-Verlag, 2000.

[32] J. R. Harrison, "Complex Quantifier Elimination in HOL," In Supplemental Proceedings of the International Conference on Theorem Proving in Higher Order Logics, pp. 159-174, Edinburgh, Scotland, UK, Sep. 2001.

[33] J. R. Harrison and L. Théry, "A Skeptic's Approach to Combining Hol and Maple," Journal of Automated Reasoning, 21: 279-294, 1998.

[34] M. Huhn, K. Schneider, T. Kropf, and G. Logothetis, "Verifying Imprecisely Working Arithmetic Circuits," In Proceedings Design Automation and Test in Europe Conference, pp. 65-69, Munich, Germany, March 1999.

[35] The Institute of Electrical and Electronic Engineers, Inc., "IEEE, Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754, USA, 1985.

[36] The Institute of Electrical and Electronic Engineers, Inc., "IEEE, Standard for Radix-Independent Floating-Point Arithmetic," ANSI/IEEE Std 854, USA, 1987.

[37] L. B. Jackson, "Roundoff-Noise Analysis for Fixed-Point Digital Filters Realized in Cascade or Parallel Form," IEEE Transactions on Audio and Electroacoustics, AU-18: 107-122, June 1970.

[38] R. Kaivola and M. D. Aagaard, "Divider Circuit Verification with Model Checking and Theorem Proving," In Theorem Proving in Higher Order Logics, LNCS 1869, pp. 338-355, Springer-Verlag, 2000.

[39] J. F. Kaiser, "Digital Filters," In System Analysis by Digital Computer, F. F. Kuo and J. F. Kaiser, Eds., pp. 218-285, Wiley, 1966.

[40] R. Kaivola and K. R. Kohatsu, "Proof Engineering in the Large: Formal Verification of Pentium® 4 Floating-Point Divider," Software Tools for Technology Transfer, 4 (3): 323-334, 2003.

[41] T. Kaneko and B. Liu, "Accumulation of Round-Off Error in Fast Fourier Transforms," Journal of Association for Computing Machinery, 17 (4): 637-654, Oct. 1970.

[42] R. Kaivola and N. Narasimhan, "Formal Verification of the Pentium® 4 Floating-Point Multiplier," In Proceedings Design Automation and Test in Europe Conference, pp. 20-27, Paris, France, March 2002.

[43] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey," ACM Transactions on Design Automation of Electonic Systems, 4: 123-193, April 1999.

[44] J. B. Knowles and R. Edwards, "Effects of a Finite-Word-Length Computer in a Sampled-Data Feedback System," IEE Proceedings, 112: 1197-1207, June 1965.

[45] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design and Simulation Environment," In Proceedings Design Automation and Test in Europe Conference, pp. 429-435, Paris, France, February 1998.

[46] M. Leeser and J. O'Leary, "Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation," In Proceedings International Conference on Computer Design, pp. 526-531, Austin, Texas, USA, October 1995.

[47] B. Liu and T. Kaneko, "Error Analysis of Digital Filters Realized with Floating-Point Arithmetic," Proceedings of the IEEE, 57: 1735-1747, October 1969.

[48] B. Liu and T. Kaneko, "Roundoff Error in Fast Fourier Transforms (Decimation in Time)," Proceedings of the IEEE (Proceedings Letters), 991-992, Jun. 1975.

[49] Mathworks, Inc., "Simulink Reference Manual," USA, 1996.

[50] Mathworks, Inc., "Fixed-Point Blockset, For Use with Simulink, User's Guide," USA, 2004.

[51] T. F. Melham, "The HOL pred_sets Library," University of Cambridge, Computer Laboratory, February 1992.

[52] T. Melham, "Higher Order Logic and Hardware Verification," Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993.

[53] P. S. Miner and J. F. Leathrum, "Verification of IEEE Compliant Subtractive Division Algorithms," In Formal Methods in Computer-Aided Design, LNCS 1166, pp. 64-78, Springer-Verlag, 1996.

[54] P. S. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS," NASA TM-110167, June 1995.

[55] J. Misra, "Powerlists: A Structure for Parallel Recursion," In ACM Transactions on Programming Languages and Systems, 16 (6): 1737-1767, Nov. 1994.

[56] J. S. Moore, T. Lynch, and M. Kaufmann, "A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm," IEEE Transactions on Computers, 47 (9): 913-926, 1998.

[57] S. M. Mueller and W. J. Paul, "Computer Architecture. Complexity and Correctness," Springer-Verlag, 2000.

[58] R. V. Nee and R. Prasad, "OFDM for Wireless Multimedia Communications," Artech House, Boston, 2000.

[59] A. V. Oppenheim and R. W. Schafer, "Discrete-Time Signal Processing," Prentice-Hall, 1989.

[60] A. V. Oppenheim and C. J. Weinstein, "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform," Proceedings of the IEEE, 60 (8): 957-976, August 1972.

[61] Open SystemC Initiative, "SystemC Language Reference Manual," USA, 2004.

[62] J. O' Leary, X. Zhao, R. Gerth, and C.-J.H. Seger, "Formally Verifying IEEE Compliance of Floating-Point Hardware," Intel Technology Journal, Q1: 1-14, 1999.

[63] L.C. Paulson, "ML for the Working Programmer," Cambridge University Press, U.K., 2nd edition, 1996.

[64] D. M. Russinoff, "A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating-Point Adder of the AMD Athlon Processor," In Formal Methods in Computer-Aided Design, LNCS 1954, pp. 3-36, Springer-Verlag, 2000.

[65] J. Sawada and R. Gamboa, "Mechanical Verification of a Square Root Algorithm using Taylor's Theorem," In Formal Methods in Computer-Aided Design, LNCS 2517, pp. 274-291, Springer-Verlag, 2002.

[66] I. W. Sandberg, "Floating-Point-Roundoff Accumulation in Digital Filter Realization," The Bell System Technical Journal, 46: 1775-1791, October 1967.

[67] C.J.Seger, "An Introduction to Formal Hardware Verification," Technical Report 92-13, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., Canada, June 1992.

[68] T. Thong and B. Liu, "Fixed-Point Fast Fourier Transform Error Analysis," IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP 24 (6): 563-573, Dec. 1976.

[69] University of Cambridge, "The HOL System Reference," Computer Laboratory, Cambridge, UK, March 2004.

[70] W. Wong, "Modeling Bit Vectors in HOL: The Word Library," In Higher Order Logic and Its Applications, LNCS 780, pp. 371-384, Springer-Verlag, 1994.

[71] C. Weinstein and A. V. Oppenheim, "A Comparison of Roundoff Noise in Floating Point and Fixed Point Digital Filter Realizations," Proceedings of the IEEE (Proceedings Letters), 57: 1181-1183, June 1969.

[72] C. J. Weinstein, "Roundoff Noise in Floating Point Fast Fourier Transform Computation," IEEE Transactions on Audio and Electroacoustics, AU-17 (3): 209-215, Sep. 1969.

[73] P. D. Welch, "A Fixed-Point Fast Fourier Transform Error Analysis," IEEE Transactions on Audio and Electroacoustics, AU-17 (2): 151-157, Jun. 1969.

[74] J. H. Wilkinson, "Rounding Errors in Algebraic Processes," Prentice-Hall, 1963.

[75] S. Wolfram, "Mathematica, A System for Doing Mathematics by Computer," Addison-Wesley, 1988.

[76] Xilinx, Inc., "High-Performance 16-Point Complex FFT/IFFT V1.0.5, Product Specification," USA, Jul. 2000, http://xilinx.com/ipcenter.

Biography

• Education

- **Concordia University:** Montreal, Quebec, Canada
  Ph.D candidate, in Electrical Engineering, 5/00-present
- **Sharif University of Technology:** Tehran, Iran
  M.Sc., in Electrical Engineering, 9/94 - 3/97
- **Shiraz University:** Shiraz, Iran
  B.Sc., in Electrical Engineering, 3/88 - 9/93

• Work Experience

- **Research Assistant:** 5/00-present
  Hardware Verification Group (HVG), Concordia University
- **Design Engineer:** 9/98-4/00
  Emad Semicon. Co. Ltd., Tehran, Iran

• Publications

– Journal Papers

1. B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of Fixed-Point Arithmetic in HOL," To appear in Formal Methods in Systems Design, Springer-Verlag. [33 pages]
2. B. Akbarpour and S. Tahar, "Error Analysis of Digital Filters using HOL Theorem Proving," Submitted to IEEE Transactions on Circuits and Systems I. [35 pages]
3. B. Akbarpour and S. Tahar, "An Approach for the Formal Verification of FFT Algorithms using Theorem Proving," Submitted to IEEE Transactions on CAD of Integrated Circuits and Systems. [24 pages]

– Conference Papers

1. B. Akbarpour and S. Tahar, "A Methodology for the Formal Verification of FFT Algorithms in HOL," In Formal Methods in Computer-Aided Design, LNCS 3312, pp. 37-51, Springer-Verlag, 2004.
2. B. Akbarpour and S. Tahar, "Error Analysis of Digital Filters using Theorem Proving," In Theorem Proving in Higher Order Logics, LNCS 3223, pp. 1-16, Springer-Verlag, 2004.
3. B. Akbarpour and S. Tahar, "Modeling SystemC Fixed-Point Arithmetic in HOL," In Formal Methods and Software Engineering, LNCS 2885, pp. 206-225, Springer-Verlag, 2003.
4. B. Akbarpour and S. Tahar, "The Application of Formal Verification to SPW Designs," In Proceedings Euromicro Symposium on Digital System Design, IEEE Computer Society Press, pp. 325 -332, Belek, Turkey, September 2003.
5. B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of Cadence SPW Fixed-Point Arithmetic in HOL," In Integrated Formal Methods, LNCS 2335, pp. 185-204, Springer-Verlag, 2002.

– Technical Reports

1. B. Akbarpour and S. Tahar, "Verification of the Fast Fourier Transform using HOL Theorem Proving;," Technical Report, Concordia University, Department of Electrical and Computer Engineering, March 2004. [40 pages]
2. B. Akbarpour and S. Tahar, "Error Analysis of Digital Filters using HOL Theorem Proving," Technical Report, Concordia University, Department of Electrical and Computer Engineering, February 2004. [36 pages]
3. B. Akbarpour, S. Tahar, "Formalization of Fixed-Point Arithmetic in HOL," Technical Report, Concordia University, Department of Electrical and Computer Engineering, September 2002. [21 pages]