Verification of the Fast Fourier Transform using HOL Theorem Proving

Behzad Akbarpour, Sofiéne Tahar

Electrical and Computer Engineering Department

Concordia University, Montreal, Canada

Email: {behzad,tahar}@ece.concordia.ca

Technical Report

March 2004

Abstract

The fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform (DFT) with a substantial time saving over conventional methods. Two basic classes of FFT algorithms are the decimation-in-time (DIT) and decimation-in-frequency (DIF). In this report we describe the formal verification of the FFT algorithms at different abstraction levels based on the HOL theorem prover. We make use of existing theories in HOL on real and complex real, IEEE standard based floating-point, and fixed-point arithmetic to model the FFT algorithms in HOL. Then, we establish fundamental analysis lemmas as our models to derive expressions for the accumulation of roundoff error in floating- and fixed-point FFT algorithms implementation in comparison to the corresponding ideal real specification. Based on the extensively studied related work on computing the errors due to finite precision effects in the realization of FFT algorithms with floating- and fixed-point arithmetics using theoretical paper-and-pencil proofs and simulation techniques since the late sixties, we performed a similar error analysis using the HOL theorem proving environment. The HOL formalization and proofs are found to be in good agreement with the theoretical counterparts. Finally, we use a classical hierarchical proof approach in HOL to prove that the FFT implementation at behavioral, register transfer (RT), and netlist gate levels implies the high level fixed-point algorithmic specification.

Keywords: Formal Verification, Fast Fourier Transform, Theorem Proving, Fixed-point, Floating-point, HOL

Contents

1 Introduction

The discrete Fourier transform (DFT) plays an important role in the analysis, design, and implementation of discrete-time signal processing algorithms and systems [26]. The fast Fourier transform (FFT) [5, 8] is a highly efficient method for computing the discrete Fourier coefficients of a finite sequence of complex data. Because of the substantial time saving over the 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 sequence of length N into successively smaller discrete Fourier transforms. The manner in which this principle is implemented leads to a a variety of different algorithms, all with comparable improvements in computational speed. There are two canonical forms and 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 [9], called decimation-in-time (DIT), derives its name from the fact that in process of arranging the computation into smaller transformations, the input sequence $x\ [n]$ (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 [12], the sequence of discrete Fourier transform coefficients $X\ [k]$ is decomposed into smaller subsequences, hence its name, decimation-in-frequency (DIF). In a theoretical analysis of the fast Fourier transform, we generally assume that signal values and system coefficients are represented in the real number system and expressed to infinite precision. When implemented as a special-purpose digital hardware or as a computer algorithm, we must represent signals and coefficients in some digital number system that must always be of finite precision. There is an inherent accuracy problem in calculating the Fourier coefficients, since the signals are represented by a finite number of bits and the arithmetic operations must be carried out with an accuracy limited by this finite word length. There are a variety of types of arithmetic used in the implementation of digital systems. Among the most common are floating- 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 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 or a programming language. This design process can be aided by a number of specialized CAD (Computer Aided Design) tools such as SPW (Cadence) [29], CoCentric (Synopsys) [7], and Matlab-Simulink (Mathworks) [23].

In this report we describe the formal verification of the fast Fourier transform algorithms using HOL theorem proving environment [13] based on the commutating diagram shown in Figure 1. In the FFT design flow, we first focus on the transition from real to floating- and fixed-point levels. Thereafter, we first model the ideal real specification of the FFT algorithms and the corresponding floating- and fixed-point implementations as predicates in higher-order logic. For this, we make use of existing theories in HOL on construction of real [14] and complex real [17] numbers , formalization of IEEE-754 standard [18] based floating-point arithmetic [15, 16], and formalization of fixed-point arithmetic [1, 2]. 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 roundings and arithmetic operations against their abstract mathematical counterparts. Finally, we use these lemmas as a model 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, for each of the two canonical forms of realization: decimation-in-time (DIT) and decimation-in-frequency (DIF). 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 [20], it is for the first time in this report, that a formalization and proof of this analysis is done 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 to floating- and fixed-point levels, we turn on to the HDL representation. At this point, we used well known formal techniques to model the FFT design at the behavioral and/or RT levels and the corresponding synthesized gate level netlist within the HOL environment. The next step is to verify these different levels using a classical hierarchical proof approach in HOL [24]. In this approach, we hierarchically proved that the gate level implementation implies the more abstracted RTL (Register Transfer Level). Then this RTL is related by a formal proof to the behavioral specification. The latter is used to imply 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.

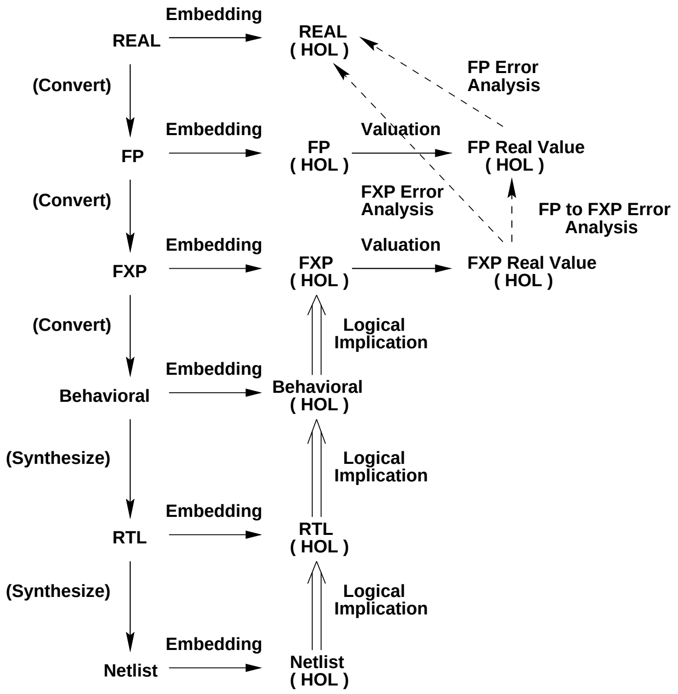

Figure 1: FFT specification and verification methodology

The rest of this report is organized as follows: Section 2 reviews some related work. Section 3 describes the details of the error analysis in HOL of the FFT algorithms at the real, floating-, and fixed-point levels. Section 4 describes the verification of the FFT algorithms in the transition from fixed-point to register transfer levels. Finally, Section 5 concludes the report.

2 Related Work

Work on the analysis of errors due to finite precision effects in the realization of the FFT algorithms has always existed since their early days, however, using theoretical 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 [32] presented an analysis for the fixed-point accuracy of the radix-2 decimation-in-time FFT algorithm. This analysis leads to approximate upper and lower bounds on the root-mean-square error. The paper also presents simulation results that correlate well with the analytical results. Tran-Thong and Liu [30] presented an error analysis of the fixed-point radix-2 FFT algorithms based on a statistical model for the roundoff noise. Both decimation-in-time and decimation-in-frequency are treated assuming either rounding or two's complement chopping. The extension of the analysis to mixed radix was also discussed. Both an analytical approach and a numerical simulation approach to the problem of the computation of the error means were presented. Knight and Kaiser [21] derived worst case error bounds for the computation of the fast Fourier transform in fixed-point arithmetic for any arithmetic number base and any prime factorization of the data array length. 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 [12] who presented an upper bound on the mean-squared error for floating-point FFT algorithm. Weinstein [31] presented a statistical model for roundoff errors to calculate the mean-squared error of the floating-point fast Fourier transform for the special case of white-noise input. The results of the analysis have been verified with excellent agreement using experimental simulation. Kaneko and Liu [20] presented a detailed analysis of the roundoff error in FFT decimation-in-time algorithm using floating-point arithmetic. Both rounding and chopping arithmetics are considered. Some special cases are included as examples that illustrate the general results. This analysis is later extended by the same authors for the case of FFT decimation-in-time algorithm [22]. Oppenheim and Weinstein [27] discussed in some detail the effects of finite register length on implementations of linear recursive difference equation digital filters, and fast Fourier transform (FFT) algorithm. For these algorithms, the differing quantization effects of floating- and fixed-point arithmetic are examined and compared. Ramos [28] presented an analysis of roundoff errors occurring in floating-point computation of the fast Fourier transform. The paper derives upper bounds for the ratios of the root-mean-square (RMS) and maximum roundoff errors in the output data to the RMS value of the output data for both single and multidimensional transformations. These bounds are compared experimentally with actual roundoff errors. James [19] presented models for the coefficient quantization and arithmetic quantization in the floating-point FFT. The accuracy and applicability of these models are treated by computer simulations on white noise and narrow-band signals.

In order to validate the error analysis, most of the above work compare the theoretical results with experimental simulation. In this report, we show how the above error analysis for the FFT algorithms can be mechanically performed using the HOL theorem prover. Our focus will be on the process of translating the hand proofs into equivalent proofs in HOL. We propose a formal verification methodology based on the shallow embedding [4] of the intended meaning of the FFT algorithm designs at the real, floating- and fixed-point levels in higher-order logic. Thereafter, we complete the formal proof in the HOL theorem proving environment hence providing a superior approach to validation by simulation. The analysis we propose is mostly inspired by the work done by Kaneko and Liu [20] who proposed a general approach to the error analysis problem of the decimation in frequency FFT algorithms using floating-point arithmetic. Following similar approach, we have extended this analysis for the decimation in time and fixed-point FFT algorithms. In both cases, good agreement between formal and theoretical results are obtained.

Previous work on the formalization and mechanical verification of the FFT algorithm was done by Gamboa [11] using the ACL2 theorem prover. The author formalized the FFT as a recursive data-parallel algorithm, using the data structure of powerlists. 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 [25] into a mechanical proof in ACL2. In the same line, Capretta [6] presented the formalization of the FFT using the type theory proof tool Coq. To facilitate the definition of the transform by structural recursion, he 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). While [11] and [6] prove simple theorems to check the correctness of the FFT algorithm against the original definition of the discrete Fourier transform (DFT), our work formally specifies and verifies the FFT algorithm realizations at different levels of abstraction based on different data types. Besides, the definition used for the FFT in [11, 6] is based on the radix-2 decimation-in-time algorithm. In our work, we cover both decimation-in-time and decimation-in-frequency algorithms, and radixes other than 2. The methodology presented in this paper is to the best of our knowledge, the first project of its kind in which we cover 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.

To formally specify and verify the FFT algorithms in higher-order logic, the need for a formalization of real and complex real, floating-, and fixed-point arithmetics is unavoidable. For real, complex real, and floating-point arithmetics, we have used the work done by Harrison who constructed the real numbers library [14] and formalized the complex real numbers [17] in HOL. He also developed a generic floating-point theory [15] to define and prove the most fundamental terms and lemmas about the IEEE-754 standard [18], and to formalize and verify floating-point algorithms such as exponential function [16] against their behavioral specification[1]. For fixed-point arithmetic, we developed in [1] and [2] a theory in HOL on similar properties of fixed-point arithmetic as described in the Cadence SPW tool and the SystemC language, respectively. Using these formalizations, we were able to describe the fast Fourier transform algorithms at different levels in HOL and analyze the error in transition from one level to the other one.

[1]These theories were developed using HOL Light. As part of this project, we have ported most of them to HOL4 Kananaskis 2.

3 Error Analysis of the Fast Fourier Transform

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 details 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 [26]

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

where $W_N = e^{-j2\pi/N}$ and $j = \sqrt{-1}$. For simplicity, our discussion is restricted to the radix-2 FFT algorithm where the number of points N to be Fourier transformed satisfy the relationship $N = 2^m$, m integer valued. 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

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

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

$$p^* \;=\; 2^{m-1}p_{m-1} + 2^{m-2}p_{m-2} + \cdots + 2p_1 + p_0 \tag{3}$$

- **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 [20]

$$A_{k+1}(p) = \begin{cases} A_k(p) + A_k(p + 2^{m-1-k}) & \text{if } \ p_k = 0 \\ [A_k(p - 2^{m-1-k}) - A_k(p)]\ w_k(p) & \text{if } \ p_k = 1 \end{cases} \tag{4}$$

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

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

  Equation (4) is carried out for $k = 0,1,2,\ldots,m-1$, with $A_0(p) = x(p)$. It can be shown [12] 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 (2) and (3), respectively. Figure 2 shows the signal flowgraph [2] of the actual computation for the case $N = 2^4$.

[2]For readers not familiar with the notion of flowgraph, it consists of nodes and directed branches. Each node represents a variable which is the weighted sum of the variables at the originating nodes of all the branches that terminate on that node. The weights, if other than unity, are shown for each branch.

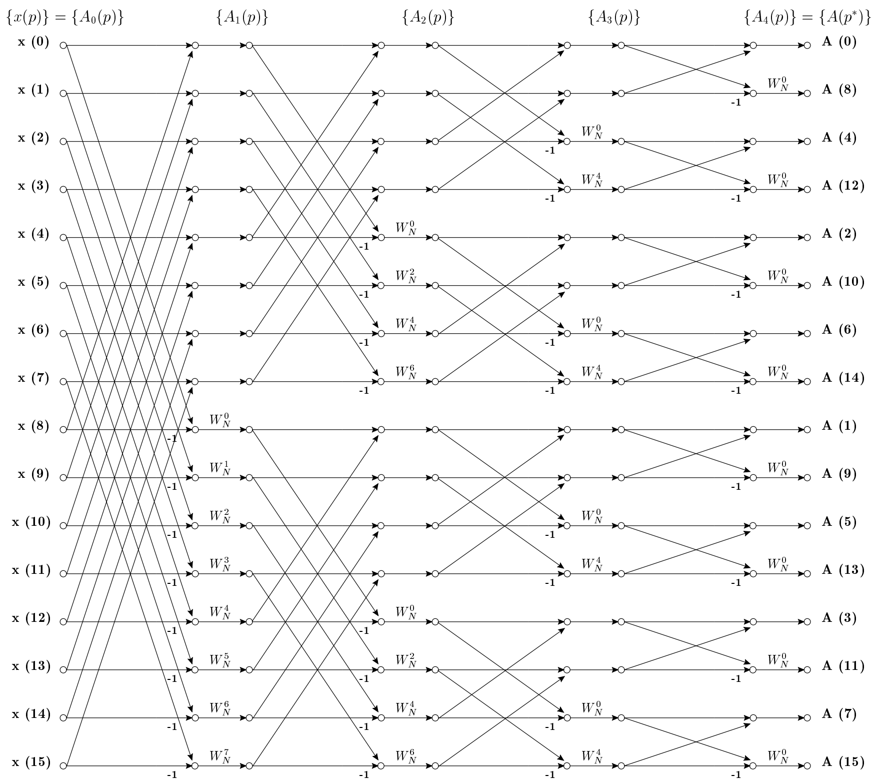

Figure 2: Signal flowgraph of decimation-in-frequency FFT, $N = 2^4$

- **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 [20]

$$A_{k+1}(p) = \begin{cases} A_k(p) + w_k(p)\ A_k(p+2^k) & \text{if } \ p_{m-1-k} = 0 \\ A_k(p-2^k) - w_k(p)\ A_k(p) & \text{if } \ p_{m-1-k} = 1 \end{cases} \tag{5}$$

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

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

  Equation (5) 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 (2) and (3), respectively. It can be shown [9] 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 3 shows the signal flowgraph of the actual computation for the case $N = 2^4$.

There are three common sources of errors associated with the FFT algorithms, namely [20]:

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

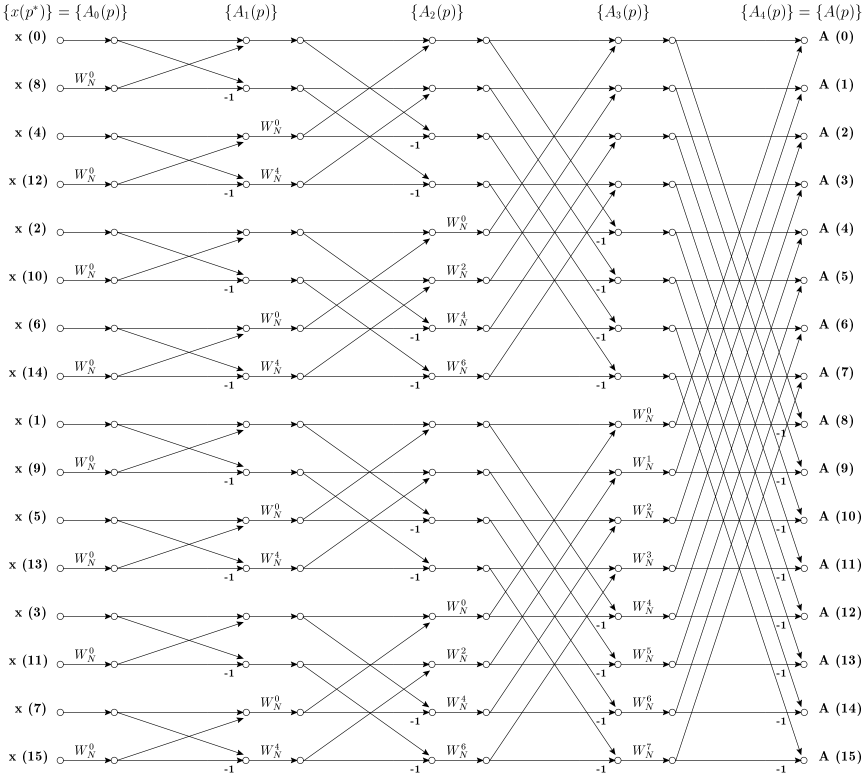

Figure 3: Signal flowgraph of decimation-in-time FFT, $N = 2^4$

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) and (5) 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

$$e_k(p) = A'_k(p) - A_k(p) \tag{6}$$
$$e'_k(p) = A''_k(p) - A_k(p) \tag{7}$$
$$e''_k(p) = A''_k(p) - A'_k(p) \tag{8}$$

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. Then

$$e(p) = e_m(p^*) \tag{9}$$
$$e'(p) = e'_m(p^*) \tag{10}$$
$$e''(p) = e''_m(p^*) \tag{11}$$

are the corresponding errors committed in the computation of the Fourier coefficients $\{A(p)\}_{p=0}^{N-1}$ by using the FFT method.

It is clear from the above discussion that for the fast Fourier transform algorithm of Equation (4), we have

$$A'_{k+1}(p) = \begin{cases} fl(A'_k(p) + A'_k(p + 2^{m-1-k})) & \text{if } p_k = 0 \\ fl([A'_k(p - 2^{m-1-k}) - A'_k(p)]\ w_k(p)) & \text{if } p_k = 1 \end{cases} \tag{12}$$

and

$$A''_{k+1}(p) = \begin{cases} fxp(A''_k(p) + A''_k(p + 2^{m-1-k})) & \text{if } p_k = 0 \\ fxp([A''_k(p - 2^{m-1-k}) - A''_k(p)]\ w_k(p)) & \text{if } p_k = 1 \end{cases} \tag{13}$$

where the notations $fl(.)$ and $fxp(.)$ are used to denote that the operations are performed using the floating- and fixed-point arithmetics, respectively.

In analyzing the effect of floating-point roundoff, the effect of rounding will be represented multiplicatively. Let $*$ denote any of the arithmetic operations +, -, $\times$, /, it is known [10, 33] that, if p represents the precision of the floating-point format, then

$$fl\ (x\ *\ y)\ =\ (x\ *\ y)(1\ +\ \delta),\quad \text{where}\ |\delta|\ \leq\ 2^{-p} \tag{14}$$

The notation fl (.) is used to denote that the operation is performed using the 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 can be stated as

$$fxp\ (x\ *\ y)\ =\ (x\ *\ y)\ +\ \epsilon,\quad \text{where}\ |\epsilon|\ \leq\ 2^{-fracbits\ (X)} \tag{15}$$

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 the fixed-point arithmetic. We have proved equations (14) and (15) as theorems in higher-order logic within HOL. The corresponding details can be found in [3].

In equation (4) the $\{A_k(p)\}$ are complex numbers, so their real and imaginary parts are calculated separately. Let

$$\begin{array}{ll} B_k(p) = Re\ [A_k(p)] & C_k(p) = Im\ [A_k(p)] \\ U_k(p) = Re\ [w_k(p)] & V_k(p) = Im\ [w_k(p)] \end{array} \tag{16}$$

where the notations *Re* [.] and *Im* [.] denote, respectively, the real and imaginary parts of the quantity inside bracket [.]. Equation (4) can be rewritten as

$$\left.\begin{array}{l} B_{k+1}(p) = B_k(p) + B_k(q) \\ C_{k+1}(p) = C_k(p) + C_k(q) \end{array}\right\} \quad \text{if } p_k = 0 \tag{17}$$

$$\left.\begin{array}{l} B_{k+1}(p) = [B_k(r) - B_k(p)]\ U_k(p) - [C_k(r) - C_k(p)]\ V_k(p) \\ C_{k+1}(p) = [C_k(r) - C_k(p)]\ U_k(p) + [B_k(r) - B_k(p)]\ V_k(p) \end{array}\right\} \quad \text{if } p_k = 1 \tag{18}$$

where $q\ =\ p + 2^{m-1-k}$ and $r\ =\ p\ -\ 2^{m-1-k}$. On using the prime and double primes to denote the actual floating- and fixed-point calculated results, the real and imaginary parts of $A'_{k+1}(p)$ and $A''_{k+1}(p)$ according to equations (12) and (13) are given by

$$\left.\begin{array}{l} B'_{k+1}(p) = fl(B'_k(p) + B'_k(q)) \\ C'_{k+1}(p) = fl(C'_k(p) + C'_k(q)) \end{array}\right\} \quad \text{if} \ \ p_k = 0 \tag{19}$$

$$\left.\begin{array}{l} B'_{k+1}(p) = fl([B'_k(r) - B'_k(p)]\ U_k(p) - [C'_k(r) - C'_k(p)]\ V_k(p)) \\ C'_{k+1}(p) = fl([C'_k(r) - C'_k(p)]\ U_k(p) + [B'_k(r) - B'_k(p)]\ V_k(p)) \end{array}\right\} \quad \text{if} \ \ p_k = 1 \tag{20}$$

and

$$\left.\begin{array}{l} B''_{k+1}(p) = fxp(B''_k(p) + B''_k(q)) \\ C''_{k+1}(p) = fxp(C''_k(p) + C''_k(q)) \end{array}\right\} \quad \text{if} \ \ p_k = 0 \tag{21}$$

$$\left.\begin{array}{l} B''_{k+1}(p) = fxp([B''_k(r) - B''_k(p)]\ U_k(p) - [C''_k(r) - C''_k(p)]\ V_k(p)) \\ C''_{k+1}(p) = fxp([C''_k(r) - C''_k(p)]\ U_k(p) + [B''_k(r) - B''_k(p)]\ V_k(p)) \end{array}\right\} \quad \text{if} \ \ p_k = 1 \tag{22}$$

The corresponding error flowgraph showing the effect of roundoff error using the fundamental floating- and fixed-point error analysis theorems according to the equations (14) and (15), respectively, is given in Figure 4, which also indicates the order of the calculation.

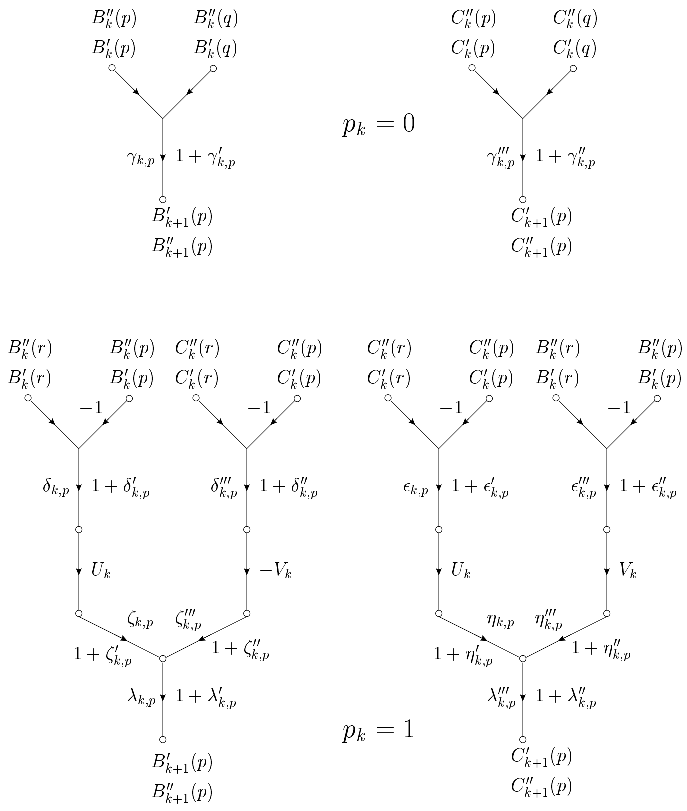

Figure 4: Error flowgraph for decimation-in-frequency FFT

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 $\lambda''_{k,p}$ 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 $\lambda'''_{k,p}$. Therefore, the actual real and imaginary parts of the floating- and fixed-point outputs $A'_{k+1}(p)$ and $A''_{k+1}(p)$ are seen to be given explicitly by

$$\left.\begin{array}{l} B'_{k+1}(p) = [B'_k(p) + B'_k(q)](1+\gamma'_{k,p}) \\ C'_{k+1}(p) = [C'_k(p) + C'_k(q)](1+\gamma''_{k,p}) \end{array}\right\} \quad \text{if } \ p_k = 0 \tag{23}$$

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

and

$$\left.\begin{array}{l} B''_{k+1}(p) = [B''_k(p) + B''_k(q)] + \gamma_{k,p} \\ C''_{k+1}(p) = [C''_k(p) + C''_k(q)] + \gamma'''_{k,p} \end{array}\right\} \quad \text{if } \ p_k = 0 \tag{24}$$

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

The errors $e_k(p)$, $e'_k(p)$, and $e''_k(p)$ defined in equations (6), (7), and (8) are complex and can be rewritten as

$$e_k(p) = B'_k(p) - B_k(p) + j[C'_k(p) - C_k(p)] \tag{25}$$

$$e'_k(p) = B''_k(p) - B_k(p) + j[C''_k(p) - C_k(p)] \tag{26}$$

$$e''_k(p) = B''_k(p) - B'_k(p) + j[C''_k(p) - C'_k(p)] \tag{27}$$

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

with

$$e_0(p) = e'_0(p) = e''_0(p) = 0, \qquad p = 0, 1, \ldots, N-1 \tag{28}$$

From equations (18), (24), (25), (25), (26), and (27), we have

1. *FFT Real to Floating-Point:*

$$e_{k+1}(p) = \begin{cases} e_k(p) + e_k(q) + f_k(p) & \text{if } \ p_k = 0 \\ [e_k(r) - e_k(p)]\ w_k(p) + f_k(p) & \text{if } \ p_k = 1 \end{cases} \tag{29}$$

where $f_k(p)$ is given by

$$f_k(p) = \begin{cases} \gamma'_{k,p}[B'_k(p) + B'_k(q)] + j\gamma''_{k,p}[C'_k(p) + C'_k(q)] \quad \text{if } \ \ p_k = 0 \\ [(1+\delta'_{k,p})(1+\zeta'_{k,p})(1+\lambda'_{k,p}) - 1][B'_k(r) - B'_k(p)]U_k(p) \\ -[(1+\delta''_{k,p})(1+\zeta''_{k,p})(1+\lambda'_{k,p}) - 1][C'_k(r) - C'_k(p)]V_k(p) \\ +j[(1+\epsilon'_{k,p})(1+\eta'_{k,p})(1+\lambda''_{k,p}) - 1][C'_k(r) - C'_k(p)]U_k(p) \\ +j[(1+\epsilon''_{k,p})(1+\eta''_{k,p})(1+\lambda''_{k,p}) - 1][B'_k(r) - B'_k(p)]V_k(p) \\ \hfill \text{if } \ \ p_k = 1 \end{cases} \tag{30}$$

2. *FFT Real to Fixed-Point:*

$$e'_{k+1}(p) = \begin{cases} e'_k(p) + e'_k(q) + f'_k(p) & \text{if } \ p_k = 0 \\ [e'_k(r) - e'_k(p)] \ w_k(p) + f'_k(p) & \text{if } \ p_k = 1 \end{cases} \tag{31}$$

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

$$f'_k(p) = \begin{cases} \gamma_{k,p} + j\gamma'''_{k,p} \quad \text{if } \ p_k = 0 \\ \delta_{k,p}U_k(p) + \zeta_{k,p} - \delta'''_{k,p}V_k(p) - \zeta'''_{k,p} + \lambda_{k,p} + \\ j(\epsilon_{k,p}U_k(p) + \eta_{k,p} + \epsilon'''_{k,p}V_k(p) + \eta'''_{k,p} + \lambda'''_{k,p}) \quad \text{if } \ p_k = 1 \end{cases} \tag{32}$$

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

$$e''_{k+1}(p) = \begin{cases} e''_k(p) + e''_k(q) + f'_k(p) - f_k(p) & \text{if } \ p_k = 0 \\ [e''_k(r) - e''_k(p)] \ w_k(p) + f'_k(p) - f_k(p) & \text{if } \ p_k = 1 \end{cases} \tag{33}$$

where $f_k(p)$ and $f'_k(p)$ are given by equations (30) and (32).

The accumulation of roundoff errors are determined by the recursive equations (29), (30), (31), (32), (33) with initial conditions given by equation (28). In HOL, we first constructed complex numbers on reals similar to [17]. 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 $coords : \mathbb{C} \rightarrow \mathbb{R}^2$. We used convenient abbreviations for the real (*Re*) and imaginary (*Im*) parts of a complex number, and also defined arithmetic operations such as addition, subtraction, and multiplication on complex numbers. We overloaded the usual symbols $(+, -, \times)$ for $\mathbb{C}$ and $\mathbb{R}$. Similarly, we constructed complex numbers on floating- and fixed-point numbers. The type for floating- and fixed-point complex numbers are defined in bijection with $float \times float$ and $fxp \times fxp$, with the bijections written in HOL as $float\_complex : float^2 \rightarrow \mathbb{C}$, $float\_coords : \mathbb{C} \rightarrow float^2$, and $fxp\_complex : fxp^2 \rightarrow \mathbb{C}$ and $fxp\_coords : \mathbb{C} \rightarrow fxp^2$, respectively. The abbreviations for the real and imaginary parts of a floating- and fixed-point complex number are called (*float_Re*), (*float_Im*), and (*fxp_Re*), (*fxp_Im*), respectively. Finally, we defined arithmetic operations on floating- and fixed-point complex numbers. For floating-point complex arithmetic operations we overloaded the usual symbols, and for fixed-point case the symbols *fxp_complex_add, fxp_complex_sub,* and *fxp_complex_mul* are used. Then we defined the principal N-roots on unity ($e^{-j2\pi/N} = cos\ (2\pi n/N) - j\ sin\ (2\pi n/N)$), and its powers (*OMEGA*) as a complex number using the sine and cosine functions available in the transcendental theory of the HOL reals library [14]. We specified expressions in HOL for expansion of a natural number into a binary form in normal and rearranged order according to the equations (2) and (3). The above enables us to specify the FFT algorithms in real (*REAL_FFT*), floating- (*FLOAT_FFT*), and fixed-point (*FXP_FFT*) abstraction levels using recursive definitions in HOL as described in equations (4), (12), and (13). Then we defined 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 (16). Later, we proved in separate lemmas that the real and imaginary parts of the FFT algorithm in real, floating-, and fixed-point levels can be expanded as in equations (18), (20), and (22). Then we proved lemmas to introduce an error in each of the arithmetic steps in real and imaginary parts of the floating- and fixed-point FFT algorithms according to the equations (24), and

(25). We proved these lemmas using the fundamental error analysis lemmas for basic arithmetic operations [3] according to the equations (14) and (15). Then we defined in HOL the error of the pth element of the floating- ($REAL\_TO\_FLOAT\_FFT\_ERROR$) and fixed-point ($REAL\_TO\_FXP\_FFT\_ERROR$) 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 (6), (7), and (8). Thereafter, we proved lemmas to rewrite the errors as complex numbers using the real and imaginary parts according to the equations (25), (26), and (27). Finally, we proved 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 (28), (29), (30), (31), (32), and (33). The corresponding HOL codes are listed in Appendix A.

4 FFT Fixed-Point to Netlist Verification

In this section, we describe the verification of the transition from fixed-point specification to RTL implementation for FFT algorithms. 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 [34]. Figure 5 shows the overall block diagram of the design. The basic elements are memory, delay, multiplexer, and dragonflies. 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 real and imaginary components of complex input and output samples and the phase factors are represented as 16-bit 2's complement numbers. In general, the 16-point pipelined FFT requires the calculation of two radix-4 dragonfly ranks. The unscrambling operation is performed using the output bit-reversing buffer.

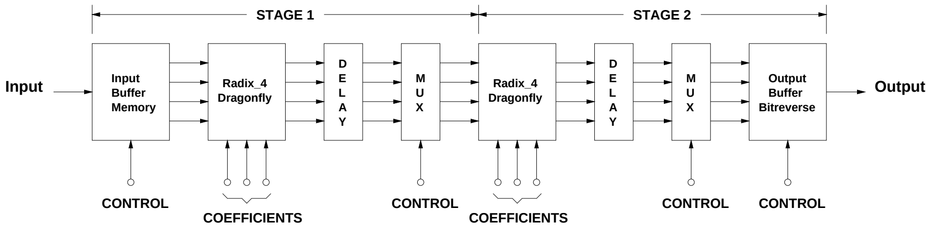

Figure 5: Radix-4 16-point pipelined FFT implementation

To define the radix-4 FFT algorithm [5, 26], we represent the indices p and n in equation (1) in a base 4 (quaternary number system) as

$$p = 4p_1 + p_0, \qquad p_1, p_0 \; = \; 0, 1, 2, 3 \tag{34}$$

$$n = 4n_1 + n_0, \qquad n_1, n_0 \; = \; 0, 1, 2, 3 \tag{35}$$

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

$$A_1(p_0, n_0) \;=\; \sum_{n_1=0}^{3} x(n_1, n_0)\ (W_{16})^{4p_0n_1} \tag{36}$$

$$A_2(p_0, p_1) \;=\; \sum_{n_0=0}^{3} A_1(p_0, n_0)\ (W_{16})^{(4p_1+p_0)n_0} \tag{37}$$

The final result can be written as

$$A(p_1, p_0) = A_2(p_0, p_1) \tag{38}$$

Thus, as in the radix-2 algorithm, the results are in reversed order. Based on equations (36), (37), and (38) we can develop a signal flowgraph for radix-4 16-point FFT algorithm as shown in Figure 6. The graph is composed of two successive radix-4 dragonfly stages. Each radix-4 dragonfly is a successive combination of a radix-4 butterfly with four twiddle factor multipliers. 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 6 also illustrates the unscrambling procedure for the radix-4 algorithm.

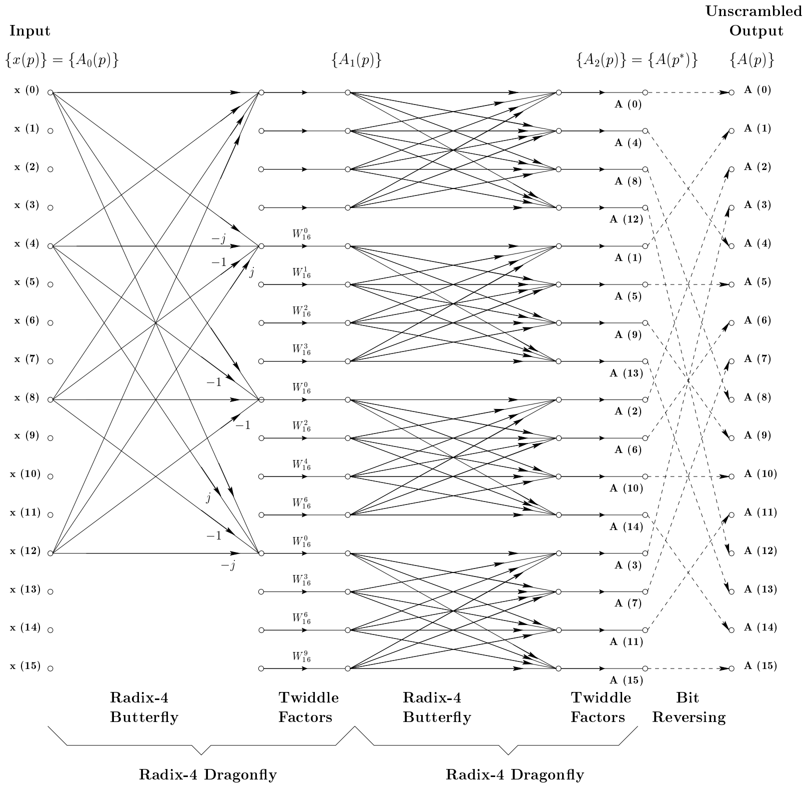

Figure 6: Signal flowgraph of radix-4 16-point FFT

In HOL, we first modeled the RTL description of a radix-4 butterfly as a predicate in higher-order logic. The block takes a vector of four complex input data and performs the operations as depicted in the flowgraph of Figure 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, subtraction, and multiplication 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 (*DRAGONFLY_RTL*) as a conjunction of a radix-4 butterfly and four 16-bit twiddle factor complex multipliers as shown in Figure 6. Finally, we modeled the complete RTL description of the radix-4 16-point FFT structure (*DIF_FFT_RTL*) in HOL. The FFT block is defined as a conjunction of 8 instantiations of radix-4 dragonfly blocks according to Figure 6, by applying the proper time instances of the input and output signals to each block. Following similar steps, we described a fixed-point radix-4 16-point FFT structure (*DIF_FFT_FXP*) in HOL using complex fixed-point data types and arithmetic operations.

For the formal verification, we first proved that the FFT RTL description implies the corresponding fixed-point model.

$$\begin{aligned} \vdash_{thm}\ & \forall\ N\ X_r\ X_i\ W_r\ W_i\ A_r\ A_i. \\ & DIF\_FFT\_RTL\ N\ X_r\ X_i\ W_r\ W_i\ A_r\ A_i\ \Longrightarrow \\ & DIF\_FFT\_FXP\ N\ FXP\_VECT\_COMPLEX\ (X_r, X_i) \\ & \qquad FXP\_VECT\_COMPLEX\ (W_r, W_i) \\ & \qquad FXP\_VECT\_COMPLEX\ (A_r, A_i) \end{aligned} \tag{39}$$

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.

$$\begin{aligned} \vdash_{thm}\ & \forall\ N\ A_r\ A_i\ W_r\ W_i\ Q_r\ Q_i. \\ & DRAGONFLY\_RTL\ N\ A_r\ A_i\ W_r\ W_i\ Q_r\ Q_i\ \Longrightarrow \\ & \qquad DRAGONFLY\_FXP\ N\ FXP\ (A_r)\ FXP\ (A_i) \\ & \qquad FXP\ (W_r)\ FXP\ (W_i)\ FXP\ (Q_r)\ FXP\ (Q_i) \end{aligned} \tag{40}$$

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. We have also described the radix-4 16-point fixed-point FFT algorithm (*FXP_FFT_ALGORITHM*) using the defining equations (36), (37), and (38). Then we proved that the expanded fixed-point description based on the flowgraph of Figure 6 implies the corresponding closed form fixed-point algorithmic description.

$$\begin{aligned} \vdash_{thm}\ & \forall\ N\ X\ W\ A. \\ & DIF\_FFT\_FXP\ N\ X\ W\ A\ \Longrightarrow \\ & \forall\ p.\ A\ (p)\ =\ FXP\_FFT\_ALGORITHM\ N\ X\ W\ p \end{aligned} \tag{41}$$

This way we completed the verification of the RTL implementation to fixed-point algorithmic specification of a radix-4 16-point FFT algorithm. The corresponding HOL codes are listed in Appendix A.

5 Conclusions

In this paper, we described a comprehensive methodology for the verification of generic fast Fourier transform algorithms using the HOL theorem prover. The proposed approach covers the two canonical forms (decimation-in-time, and decimation-in-frequency) of realization of the FFT algorithms at the real, floating- and fixed-point specification, down to RT implementation levels of abstraction, entirely specified in HOL. We proved lemmas to derive expressions for the accumulation of roundoff error in floating- and fixed-point implementations both compared to the ideal real specification. Then we proved that the FFT RTL implementation implies the corresponding specification at 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 on the analysis of errors due to finite precision effects, and applied them on the realization of the FFT algorithms. Related work did exist since the late sixties using theoretical paper-and-pencil proofs and simulation techniques. We believe this is the first time a complete formal framework is considered using mechanized proofs in HOL for the verification of the fast Fourier transform algorithms at different levels of abstraction. The methodology presented in this paper 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. We are currently investigating the verification of complex wired and wireless communication systems, whose building blocks, heavily make use of several instances of the FFT algorithms.

References

[1] B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of Cadence SPW Fixed-Point Arithmetic in HOL," In Integrated Formal Methods, LNCS 2335, Springer-Verlag, pp. 185-204, 2002.

[2] B. Akbarpour, S. Tahar, "Modeling SystemC Fixed-Point Arithmetic in HOL," In Formal Methods and Software Engineering, LNCS 2885, Springer-Verlag, pp. 206-225, 2003.

[3] B. Akbarpour, S. Tahar, "Error Analysis of Digital Filters using HOL Theorem Proving," Technical Report, Concordia University, Department of Electrical and Computer Engineering, February 2004. This report is available on-line at http://ece.concordia.ca/∼behzad/Tech-DFT.ps.

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

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

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

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

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

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

[11] 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.

[12] W. M. Gentleman, 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.

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

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

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

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

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

[18] IEEE, Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.

[19] D. V. James, "Quantization Errors in the Fast Fourier Transform," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-23, No. 3, pp. 277-283, June 1975.

[20] 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.

[21] W. R. Knight, R. Kaiser, "A Simple Fixed-Point Error Bound for the Fast Fourier Transform," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-27, No. 6, pp. 615-620, December 1979.

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

[23] Mathworks, Inc., "Fixed-Point Blockset User's Guide (ver. 2.0)," 1999.

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

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

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

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

[28] G. U. Ramos, "Roundoff Error Analysis of the Fast Fourier Transform," Mathematics of Computation, Vol. 25, No. 116, pp. 757-768, October 1971.

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

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

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

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

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

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

A List of Definitions and Lemmas in HOL

In this section we provide a complete list of the definitions and lemmas in HOL for specification and verification of the FFT algorithms at different levels of abstraction.

1. **Specification of an *N* order radix-2 decimation-in-frequency FFT using real and complex numbers:**

   - **Constructing complex numbers:**

   - The mutual inverse type bijections:

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

   - Abbreviations for the real and imaginary parts:

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

- Imaginary unit:

```
ii = ⊢def  ii = complex (0,1)
```

- Definitions of the arithmetic operations:

```
compadd = ⊢def  ∀ a b. compadd a b = (FST a + FST b,SND a + SND b)
compsub = ⊢def  ∀ a b. compsub a b = (FST a - FST b,SND a - SND b)
compmul =
    ⊢def  ∀ a b.
         compmul a b =
         (FST a * FST b - SND a * SND b,FST a * SND b - SND a * FST b)
```

- Arithmetic operations on complex numbers:

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

complex_sub =
    ⊢def  ∀ w z. w complex_sub z = complex (compsub (coords w) (coords z))

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

- Definition of finite sums on complex numbers:

```
complex_sumc =
    ⊢def  (∀ n f. complex_sumc n 0 f = complex (0,0)) ∧
       ∀ n m f. complex_sumc n (SUC m) f = complex_sumc n m f + f (n + m)

COMPLEX_SUM_DEF = ⊢def  ∀ m n f. complex_sum (m,n) f = complex_sumc m n f

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

- Principal N-roots on unity:

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

- Discrete Fourier transform:

```
DFT =
    ⊢def  ∀ x N.
          DFT x N =
          (λ p. complex_sum (0,N) (λ i. x i * principal_root_1 (i * p) N))
```

- Expansion of a natural number into a binary form:

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

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

- Logarithm of base two:

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

- Definition of finite sums on natural numbers:

```
num_sumc =
    ⊢def  (∀ n f. num_sumc n 0 f = 0) ∧
       ∀ n m f. num_sumc n (SUC m) f = num_sumc n m f + f (n + m)

NUM_SUM_DEF = ⊢def  ∀ m n f. num_sum (m,n) f = num_sumc m n f

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

- Powers of principal N-roots on unity:

```
Z =
    ⊢def  ∀ k p N. 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
```

- Rearranged order binary form of a natural number:

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

- N order fast Fourier transform algorithm in ideal real domain:

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

- Calculation of the real and imaginary parts of FFT algorithm:

```
FFT_REAL = ⊢def  ∀ x N k p. FFT_REAL x N k p = Re (FFT x N k p)

FFT_IMAGE = ⊢def  ∀ x N k p. FFT_IMAGE x N k p = Im (FFT x N k p)
```

- Real and imaginary parts of powers of principal N-roots on unity:

```
OMEGA_REAL = ⊢def  ∀ k p N. OMEGA_REAL k p N = Re (OMEGA k p N)

OMEGA_IMAGE = ⊢def  ∀ k p N. OMEGA_IMAGE k p N = Im (OMEGA k p N)
```

- Rewriting the FFT algorithm using real and imaginary parts:

```
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))
```

2. **FFT real to floating-point error analysis:**

- Constructing the complex numbers on floating-point numbers:

```
float_complex_tybij =
    ⊢def  (∀ a. float_complex (float_coords a) = a) ∧
       ∀ r. K T r = (float_coords (float_complex r) = r)
```

- Real and complex parts of a floating-point complex number:

```
float_Re = ⊢def  ∀ z. float_Re z = FST (float_coords z)

float_Im = ⊢def  ∀ z. float_Im z = SND (float_coords z)
```

- Arithmetic operations on floating-point complex numbers:

```
float_compadd =
    ⊢def  ∀ a b. float_compadd a b = (FST a + FST b,SND a + SND b)

float_compsub =
    ⊢def  ∀ a b. float_compsub a b = (FST a - FST b,SND a - SND b)

float_compmul =
    ⊢def  ∀ a b.
           float_compmul a b =
           (FST a * FST b - SND a * SND b,FST a * SND b - SND a * FST b)

float_complex_add =
    ⊢def  ∀ w z.
           w float_complex_add z =
           float_complex (float_compadd (float_coords w) (float_coords z))

float_complex_sub =
    ⊢def  ∀ w z.
           w float_complex_sub z =
           float_complex (float_compsub (float_coords w) (float_coords z))

float_complex_mul =
    ⊢def  ∀ w z.
           w float_complex_mul z =
           float_complex (float_compmul (float_coords w) (float_coords z))
```

- Definition of finite sums on floating-point complex numbers:

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

FLOAT_COMPLEX_SUM_DEF =
    ⊢def  ∀ m n f. float_complex_sum (m,n) f = float_complex_sumc m n f

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

- Rounding to complex floating-point formats:

```
float_complex_round =
    ⊢def  ∀ z.
           float_complex_round z =
           float_complex
             (float (round float_format To_nearest (Re z)),
              float (round float_format To_nearest (Im z)))
```

- Floating-point complex number valuations:

```
float_complex_Val =
    ⊢def  ∀z. float_complex_Val z =
           complex (Val (float_Re z),Val (float_Im z))
```

- Floating-point complex powers of principal N-roots on unity:

```
FLOAT_OMEGA =
    ⊢def  ∀ k p N. FLOAT_OMEGA k p N = float_complex_round (OMEGA k p N)
```

- N order fast Fourier transform algorithm in actual floating-point domain:

```
FLOAT_FFT =
    ⊢def  (∀ x N. FLOAT_FFT x N 0 = (λ p. float_complex_round (x p))) ∧
       ∀ x N k.
         FLOAT_FFT x N (SUC k) =
         (λ p.
            (if DIG k p = 0 then
               FLOAT_FFT x N k p +
               FLOAT_FFT x N k (p + 2 ** (Log_2 N - 1 - k))
             else
               (FLOAT_FFT x N k (p - 2 ** (Log_2 N - 1 - k)) -
                FLOAT_FFT x N k p) * FLOAT_OMEGA k p N))
```

- Computation error of the pth element of floating-point FFT at step k:

```
FLOAT_FFT_ERROR =
    ⊢def  ∀x N k p.
         FLOAT_FFT_ERROR x N k p =
         float_complex_Val (FLOAT_FFT x N k p) - FFT x N k p
```

- Error in computation of Fourier coefficient using floating-point FFT:

```
FLOAT_FFT_FINAL_ERROR =
    ⊢def  ∀ x N p.
         FLOAT_FFT_FINAL_ERROR x N p =
         FLOAT_FFT_ERROR x N (Log_2 N) (p_star p (Log_2 N))
```

- Real and imaginary parts of floating-point FFT algorithm:

```
FLOAT_FFT_REAL =
  ⊢def  ∀ x N k p. FLOAT_FFT_REAL x N k p = float_Re (FLOAT_FFT x N k p)

FLOAT_FFT_IMAGE =
  ⊢def  ∀ x N k p. FLOAT_FFT_IMAGE x N k p = float_Im (FLOAT_FFT x N k p)
```

- Real and imaginary parts of floating-point complex powers of principal N-roots on unity:

```
FLOAT_OMEGA_REAL =
    ⊢def  ∀ k p N. FLOAT_OMEGA_REAL k p N = float_Re (FLOAT_OMEGA k p N)

FLOAT_OMEGA_IMAGE =
    ⊢def  ∀ k p N. FLOAT_OMEGA_IMAGE k p N = float_Im (FLOAT_OMEGA k p N)
```

- Rewriting the floating-point FFT algorithm using real and imaginary parts:

```
Lemma 2:
        ∀ x N k p.
          (if DIG k p = 0 then
             (FLOAT_FFT_REAL x N (SUC k) p =
              FLOAT_FFT_REAL x N k p +
              FLOAT_FFT_REAL x N k (p + 2 ** (Log_2 N − 1 − k))) ∧
             (FLOAT_FFT_IMAGE x N (SUC k) p =
              FLOAT_FFT_IMAGE x N k p +
              FLOAT_FFT_IMAGE x N k (p + 2 ** (Log_2 N − 1 − k)))
           else
             (FLOAT_FFT_REAL x N (SUC k) p =
              (FLOAT_FFT_REAL x N k (p − 2 ** (Log_2 N − 1 − k)) −
               FLOAT_FFT_REAL x N k p) * FLOAT_OMEGA_REAL k p N −
              (FLOAT_FFT_IMAGE x N k (p − 2 ** (Log_2 N − 1 − k)) −
               FLOAT_FFT_IMAGE x N k p) * FLOAT_OMEGA_IMAGE k p N) ∧
             (FLOAT_FFT_IMAGE x N (SUC k) p =
              (FLOAT_FFT_IMAGE x N k (p − 2 ** (Log_2 N − 1 − k)) −
               FLOAT_FFT_IMAGE x N k p) * FLOAT_OMEGA_REAL k p N +
              (FLOAT_FFT_REAL x N k (p − 2 ** (Log_2 N − 1 − k)) −
               FLOAT_FFT_REAL x N k p) * FLOAT_OMEGA_IMAGE k p N))
```

- Introducing an error for each of the arithmetic steps in real and imaginary parts of floating-point FFT algorithm:

```
Lemma 3:
        ∀ 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)))
```

- Rewriting the error of the pth element of floating-point FFT at step k as a complex number:

```
Lemma 4:
        ∀ x N k p.
          FLOAT_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)
```

- Accumulation of round-off error in floating-point FFT:

```
Lemma 5:
        ∀x N k p.
          (FLOAT_FFT_ERROR x N 0 p = complex (0,0)) ∧
          ∃f.
            (FLOAT_FFT_ERROR x N (SUC k) p =
             (if DIG k p = 0 then
                FLOAT_FFT_ERROR x N k p +
                FLOAT_FFT_ERROR x N k (p + 2 ** (Log_2 N − 1 − k)) +
                f x N k p
              else
                (FLOAT_FFT_ERROR x N k (p − 2 ** (Log_2 N − 1 − k)) −
                 FLOAT_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)))
```

3. **FFT real to fixed-point error analysis:**

- Constructing the complex numbers on fixed-point numbers:

```
fxp_complex_tybij =
    ⊢def (∀ a. fxp_complex (fxp_coords a) = a) ∧
       ∀r. K T r = (fxp_coords (fxp_complex r) = r)
```

- Real and imaginary parts of a fixed-point complex number:

```
fxp_Re = ⊢def  ∀ z. fxp_Re z = FST (fxp_coords z)

fxp_Im = ⊢def  ∀ z. fxp_Im z = SND (fxp_coords z)
```

- Arithmetic operations on fixed-point complex numbers:

```
FxpAdd =
    ⊢def  ∀ X a b. FxpAdd X a b = FST (fxpAdd X Convergent Clip a b)

FxpSub =
    ⊢def  ∀ X a b. FxpSub X a b = FST (fxpSub X Convergent Clip a b)

FxpMul =
    ⊢def  ∀ X a b. FxpMul X a b = FST (fxpMul X Convergent Clip a b)
```

```
fxp_compadd =
    ⊢def  ∀ X a b.
          fxp_compadd X a b =
          (FxpAdd X (FST a) (FST b),FxpAdd X (SND a) (SND b))

fxp_compsub =
    ⊢def  ∀ X a b.
          fxp_compsub X a b =
          (FxpSub X (FST a) (FST b),FxpSub X (SND a) (SND b))

fxp_compmul =
    ⊢def  ∀ X a b.
          fxp_compmul X a b =
          (FxpSub X (FxpMul X (FST a) (FST b)) (FxpMul X (SND a) (SND b)),
           FxpSub X (FxpMul X (FST a) (SND b)) (FxpMul X (SND a) (FST b)))

fxp_complex_add =
    ⊢def  ∀ X w z.
          fxp_complex_add X w z =
          fxp_complex (fxp_compadd X (fxp_coords w) (fxp_coords z))

fxp_complex_sub =
    ⊢def  ∀ X w z.
          fxp_complex_sub X w z =
          fxp_complex (fxp_compsub X (fxp_coords w) (fxp_coords z))

float_complex_mul =
    ⊢def  ∀ X w z.
          fxp_complex_mul X w z =
          fxp_complex (fxp_compmul X (fxp_coords w) (fxp_coords z))
```

- Definition of finite sums on fixed-point complex numbers:

```
fxp_complex_sumc =
    ⊢def  (∀ n X f.
           fxp_complex_sumc n 0 X f =
           fxp_complex
             (fxp (WORD (REPLICATE (streamlength X) F),X),
              fxp (WORD (REPLICATE (streamlength X) F),X))) ∧
        ∀ n m X f.
          fxp_complex_sumc n (SUC m) X f =
          fxp_complex_add X (fxp_complex_sumc n m X f) (f (n + m))

FXP_COMPLEX_SUM_DEF =
    ⊢def  ∀ m n X f. fxp_complex_sum (m,n) X f = fxp_complex_sumc m n X f

fxp_complex_sum =
    ⊢def  ∀ X f n m.
          (fxp_complex_sum (n,0) X f =
           fxp_complex
             (fxp (WORD (REPLICATE (streamlength X) F),X),
              fxp (WORD (REPLICATE (streamlength X) F),X))) ∧
          (fxp_complex_sum (n,SUC m) X f =
           fxp_complex_add X (fxp_complex_sumc n m X f) (f (n + m)))
```

- Rounding to complex fixed-point formats:

```
Fxp_round =
    ⊢def  ∀ X x. Fxp_round X x = FST (fxp_round X Convergent x Clip)


fxp_complex_round =
    ⊢def  ∀ X z.
         fxp_complex_round X z =
         fxp_complex (Fxp_round X (Re z),Fxp_round X (Im z))
```

- Fixed-point imaginary unit:

```
fxp_ii = ⊢def  ∀ X. fxp_ii X = fxp_complex_round X ii
```

- Fixed-point complex number valuations:

```
fxp_complex_value =
    ⊢def  ∀ z. fxp_complex_value z =
          complex (value (fxp_Re z),value (fxp_Im z))
```

- Fixed-point complex powers of principal N-roots on unity:

```
FXP_OMEGA =
    ⊢def  ∀ X k p N. FXP_OMEGA X k p N =
          fxp_complex_round X (OMEGA k p N)
```

- N order fast Fourier transform algorithm in actual fixed-point domain:

```
FXP_FFT =
    ⊢def  (∀ X x N. FXP_FFT X x N 0 = (λ p. fxp_complex_round X (x p))) ∧
       ∀ X x N k.
         FXP_FFT X x N (SUC k) =
         (λ p.
            (if DIG k p = 0 then
               fxp_complex_add X (FXP_FFT X x N k p)
                 (FXP_FFT X x N k (p + 2 ** (Log_2 N - 1 - k)))
             else
               fxp_complex_sub X
                 (fxp_complex_mul X
                    (FXP_FFT X x N k (p - 2 ** (Log_2 N - 1 - k)))
                    (FXP_OMEGA X k p N))
                 (fxp_complex_mul X (FXP_FFT X x N k p)
                    (FXP_OMEGA X k p N))))
```

- Computation error of the pth element of fixed-point FFT at step k:

```
FXP_FFT_ERROR =
    ⊢def  ∀ X x N k p.
         FXP_FFT_ERROR X x N k p =
         fxp_complex_value (FXP_FFT X x N k p) - FFT x N k p
```

- Error in computation of Fourier coefficient using fixed-point FFT:

```
FXP_FFT_FINAL_ERROR =
    ⊢def  ∀ X x N p.
          FXP_FFT_FINAL_ERROR X x N p =
          FXP_FFT_ERROR X x N (Log_2 N) (p_star p (Log_2 N))
```

- Calculation of the real and imaginary parts of fixed-point FFT algorithm:

```
FXP_FFT_REAL =
  ⊢def  ∀ X x N k p. FXP_FFT_REAL X x N k p = fxp_Re (FXP_FFT X x N k p)

FXP_FFT_IMAGE =
  ⊢def  ∀ X x N k p. FXP_FFT_IMAGE X x N k p = fxp_Im (FXP_FFT X x N k p)
```

- Real and imaginary parts of fixed-point complex powers of principal N-roots on unity:

```
FXP_OMEGA_REAL =
    ⊢def  ∀ X k p N. FXP_OMEGA_REAL X k p N = fxp_Re (FXP_OMEGA X k p N)

FXP_OMEGA_IMAGE =
    ⊢def  ∀ X k p N. FXP_OMEGA_IMAGE X k p N = fxp_Im (FXP_OMEGA X k p N)
```

- Rewriting the fixed-point FFT algorithm using real and imaginary parts:

```
Lemma 6:
        ∀ X x N k p.
          (if DIG k p = 0 then
             (FXP_FFT_REAL X x N (SUC k) p =
              FxpAdd X (FXP_FFT_REAL X x N k p)
                (FXP_FFT_REAL X x N k (p + 2 ** (Log_2 N - 1 - k)))) ∧
             (FXP_FFT_IMAGE X x N (SUC k) p =
              FxpAdd X (FXP_FFT_IMAGE X x N k p)
                (FXP_FFT_IMAGE X x N k (p + 2 ** (Log_2 N - 1 - k))))
           else
             (FXP_FFT_REAL X x N (SUC k) p =
              FxpSub X
                (FxpSub X
                   (FxpMul X
                      (FXP_FFT_REAL X x N k (p - 2 ** (Log_2 N - 1 - k)))
                      (FXP_OMEGA_REAL X k p N))
                   (FxpMul X
                      (FXP_FFT_IMAGE X x N k (p - 2 ** (Log_2 N - 1 - k)))
                      (FXP_OMEGA_IMAGE X k p N)))
                (FxpSub X
                   (FxpMul X (FXP_FFT_REAL X x N k p)
                      (FXP_OMEGA_REAL X k p N))
                   (FxpMul X (FXP_FFT_IMAGE X x N k p)
                      (FXP_OMEGA_IMAGE X k p N)))) ∧
```

```
(FXP_FFT_IMAGE X x N (SUC k) p =
 FxpSub X
   (FxpAdd X
      (FxpMul X
         (FXP_FFT_IMAGE X x N k (p − 2 ** (Log_2 N − 1 − k)))
         (FXP_OMEGA_REAL X k p N))
      (FxpMul X
         (FXP_FFT_REAL X x N k (p − 2 ** (Log_2 N − 1 − k)))
         (FXP_OMEGA_IMAGE X k p N)))
   (FxpAdd X
      (FxpMul X (FXP_FFT_IMAGE X x N k p)
         (FXP_OMEGA_REAL X k p N))
      (FxpMul X (FXP_FFT_REAL X x N k p)
         (FXP_OMEGA_IMAGE X k p N)))))
```

- Introducing an error for each of the arithmetic steps in real and imaginary parts of fixed-point FFT algorithm:

```
Lemma 7:
        ∀ X x N k p.
          ∃ e′.
            ∀ i.
              1 ≤ i ∧ i ≤ 12 ⟹
              e′ i ≤ 1 / 2 pow fracbits X ∧
              (if DIG k p = 0 then
                 (value (FXP_FFT_REAL X x N (SUC k) p) =
                  value (FXP_FFT_REAL X x N k p) +
                  value
                    (FXP_FFT_REAL X x N k (p + 2 ** (Log_2 N − 1 − k))) +
                  e′ 1) ∧
                 (value (FXP_FFT_IMAGE X x N (SUC k) p) =
                  value (FXP_FFT_IMAGE X x N k p) +
                  value
                    (FXP_FFT_IMAGE X x N k (p + 2 ** (Log_2 N − 1 − k))) +
                  e′ 2)
               else
                 (value (FXP_FFT_REAL X x N (SUC k) p) =
                  (value
                     (FXP_FFT_REAL X x N k (p − 2 ** (Log_2 N − 1 − k))) −
                   value (FXP_FFT_REAL X x N k p) + e′ 3) *
                  value (FXP_OMEGA_REAL X k p N) + e′ 4 −
                  ((value
                       (FXP_FFT_IMAGE X x N k
                          (p − 2 ** (Log_2 N − 1 − k))) −
                    value (FXP_FFT_IMAGE X x N k p) + e′ 5) *
                   value (FXP_OMEGA_IMAGE X k p N) + e′ 6) + e′ 7) ∧
```

```
(value (FXP_FFT_IMAGE X x N (SUC k) p) =
 (value
    (FXP_FFT_IMAGE X x N k (p − 2 ** (Log_2 N − 1 − k))) −
  value (FXP_FFT_IMAGE X x N k p) + e′ 8) *
 value (FXP_OMEGA_REAL X k p N) + e′ 9 +
 ((value
     (FXP_FFT_REAL X x N k (p − 2 ** (Log_2 N − 1 − k))) −
   value (FXP_FFT_REAL X x N k p) + e′ 10) *
  value (FXP_OMEGA_IMAGE X k p N) + e′ 11) + e′ 12))
```

- Rewriting the error of the pth element of fixed-point FFT at step k as a complex number:

```
Lemma 8:
        ∀ X x N k p.
          FXP_FFT_ERROR X x N k p =
          complex
            (value (FXP_FFT_REAL X x N k p) − FFT_REAL x N k p,
             value (FXP_FFT_IMAGE X x N k p) − FFT_IMAGE x N k p)
```

- Accumulation of round-off error in fixed-point FFT:

```
Lemma 9:
        ∀ X x N k p.
          (FXP_FFT_ERROR X x N 0 p = complex (0,0)) ∧
          ∃ f′.
            (FXP_FFT_ERROR X x N (SUC k) p =
             (if DIG k p = 0 then
                FXP_FFT_ERROR X x N k p +
                FXP_FFT_ERROR X x N k (p + 2 ** (LOG_2 N − 1 − k)) +
                f′ X x N k p
              else
                (FXP_FFT_ERROR X x N k (p − 2 ** (LOG_2 N − 1 − k)) −
                 FXP_FFT_ERROR X x N k p) * OMEGA k p N + f′ X x N k p)) ∧
            ∃ e′.
              ∀ i.
                1 ≤ i ∧ i ≤ 12 ⟹
                e′ i ≤ 1 / 2 pow fracbits X ∧
                (f′ X x N k p =
                 (if DIG k p = 0 then
                    complex (e′ 1,e′ 2)
                  else
                    complex
                      (e′ 3 * OMEGA_REAL k p N + e′ 4 −
                       e′ 5 * OMEGA_IMAGE k p N − e′ 6 + e′ 7,
                       e′ 8 * OMEGA_REAL k p N + e′ 9 +
                       e′ 10 * OMEGA_IMAGE k p N + e′ 11 + e′ 12)))
```

- Floating-point to fixed-point FFT error of the pth element at step k:

```
FLOAT_TO_FXP_FFT_ERROR =
    ⊢def  ∀ X x N k p.
         FLOAT_TO_FFT_ERROR X x N k p =
         fxp_complex_value (FXP_FFT X x N k p) −
         float_complex_Val (FLOAT_FFT x N k p)
```

- Accumulation of round-off error in transition from floating-point to fixed-point FFT:

```
 Lemma 10:
        ∀ X x N k p.
          (FLOAT_TO_FXP_FFT_ERROR X x N 0 p = complex (0,0)) ∧
          ∃ f f′.
            FLOAT_TO_FXP_FFT_ERROR X x N (SUC k) p =
            (if DIG k p = 0 then
               FLOAT_TO_FXP_FFT_ERROR X x N k p +
               FLOAT_TO_FXP_FFT_ERROR X x N k
               (p + 2 ** (LOG_2 N − 1 − k)) + f′ X x N k p − f x N k p
             else
               (FLOAT_TO_FXP_FFT_ERROR X x N k
                  (p − 2 ** (LOG_2 N − 1 − k)) −
                FLOAT_TO_FXP_FFT_ERROR X x N k p) * OMEGA k p N +
               f′ X x N k p − f x N k p)
```

4. **FFT RTL to fixed-point verification (16 point radix-4 DIF FFT):**

- **RTL description of arithmetic operations:**

- N-bit two's complement adder in RTL level:

```
xor = ⊢def  ∀ a b. a xor b = ¬a ∧ b ∨ a ∧ ¬b

carry =
    ⊢def  (∀ a b. carry 0 a b = F) ∧
       ∀ i a b.
         carry (SUC i) a b =
         BIT i a ∧ BIT i b ∨ BIT i a ∧ carry i a b ∨
         BIT i b ∧ carry i a b

N_add_RTL =
    ⊢def  ∀ N a b q.
         N_add_RTL N a b q =
         (∀ i.
            i < N ∧ 0 ≤ i ⟹  BIT i q =
            BIT i a xor (BIT i b xor carry i a b)) ∧
         (BIT N q = carry N a b)
```

- Verification of the RTL to fixed-point level of N-bit addition:

```
FXP = ⊢def  ∀ N a. FXP N a = fxp (a,N,N − 1,1)

FXP_VECT = ⊢def  ∀ N a. FXP_VECT N a = @ b. ∀ i. b i = FXP N (a i)

FXP_VECT_COMPLEX =
    ⊢def  ∀ N a b.
         FXP_VECT_COMPLEX N a b =
         @ c. ∀ i. c i = fxp_complex (FXP N (a i),FXP N (b i))

FXP_VECT_COMPLEX_NUM =
    ⊢def  ∀ N a b.
         FXP_VECT_COMPLEX_NUM N a b =
         @ c. ∀i. c i = FXP_VECT_COMPLEX N (a i) (b i)

 Lemma 11:
         ∀ N a b q.
           N_add_RTL N a b q ⟹
           (FXP N q = FxpAdd (N,N − 1,1) (FXP N a) (FXP N b))
```

- Complex addition in RTL level:

```
N_complex_add_RTL =
    ⊢def  ∀ N ar ai br bi qr qi.
         N_complex_add_RTL N ar ai br bi qr qi =
         N_add_RTL N ar br qr ∧ N_add_RTL N ai bi qi
```

- Verification of the RTL to fixed-point level of N-bit complex adder:

```
 Lemma 12:
         ∀ N ar ai br bi qr qi.
           N_complex_add_RTL N ar ai br bi qr qi ⟹
           (fxp_complex (FXP N qr,FXP N qi) =
            fxp_complex_add (N,N − 1,1) (fxp_complex (FXP N ar,FXP N ai))
              (fxp_complex (FXP N br,FXP N bi)))
```

- N-bit two's complement subtracter in RTL level:

```
borrow =
    ⊢def  (∀ a b. borrow 0 a b = T) ∧
       ∀ i a b.
         borrow (SUC i) a b =
         BIT i a ∧ ¬ BIT i b ∨ BIT i a ∧ borrow i a b ∨
         ¬ BIT i b ∧ borrow i a b

N_sub_RTL =
    ⊢def  ∀ N a b q.
         N_sub_RTL N a b q =
         (∀ i.
            i < N ∧ 0 ≤ i ⟹  BIT i q =
            BIT i a xor ¬ BIT i b xor borrow i a b)) ∧
         (BIT N q = borrow N a b)
```

- Verification of the RTL to fixed-point level of N-bit subtracter:

```
Lemma 13:
        ∀ N a b q.
          N_sub_RTL N a b q ⟹
          (FXP N q = FxpSub (N,N − 1,1) (FXP N a) (FXP N b))
```

- Complex subtraction in RTL level:

```
N_complex_sub_RTL =
    ⊢def  ∀ N ar ai br bi qr qi.
         N_complex_sub_RTL N ar ai br bi qr qi =
         N_sub_RTL N ar br qr ∧ N_sub_RTL N ai bi qi
```

- Verification of the RTL to fixed-point level of N-bit complex subtracter:

```
Lemma 14:
        ∀ N ar ai br bi qr qi.
          N_complex_sub_RTL N ar ai br bi qr qi ⟹
          (fxp_complex (FXP N qr,FXP N qi) =
           fxp_complex_sub (N,N − 1,1) (fxp_complex (FXP N ar,FXP N ai))
             (fxp_complex (FXP N br,FXP N bi)))
```

- N-bit two's complement multiplier in RTL level:

```
N_mul_unsigned =
    ⊢def  ∀ N a b q.
         N_mul_unsigned N a b q = (q = BWORD (BNVAL a * BNVAL b) (2 * N))

TWO_COMP = ⊢def  ∀ N a. TWO_COMP N a = NBWORD N (SUC (BNVAL (WNOT a)))

UN_SIGNED = ⊢def  ∀ N a. UN_SIGNED N a = WSEG (N − 1) 0 a

CONVERT =
    ⊢def  ∀ N a.
         CONVERT N a =
         (if MSB a then WSEG (N − 1) 0 (TWO_COMP N a) else UN_SIGNED N a)

N_mul_two_comp_RTL =
    ⊢def  ∀ N a b q.
         N_mul_two_comp_RTL N a b q =
         (q =
          WCAT
            (WORD MSB a xor MSB b],
             NBWORD (2 * N − 2)
               (BNVAL (CONVERT N a) * BNVAL (CONVERT N b) +
                BV (MSB a xor MSB b))))
```

- Verification of the RTL to fixed-point level of N-bit multiplier:

```
Lemma 15:
        ∀ N a b q.
          N_mul_two_comp_RTL N a b q ⟹
          (FXP N q = FxpMul (N,N − 1,1) (FXP N a) (FXP N b))
```

- Complex multiplication in RTL level:

```
N_complex_mul_two_comp_RTL =
    ⊢def  ∀ N ar ai br bi qr qi.
         N_complex_mul_two_comp_RTL N ar ai br bi qr qi =
         ∃ s1 s2 s3 s4 s5 s6.
         N_sub_RTL N ar bi s1 ∧ N_sub_RTL N br bi s2 ∧
         N_add_RTL N br bi s3 ∧ N_mul_two_comp_RTL N s1 bi s4 ∧
         N_mul_two_comp_RTL N s2 ar s5 ∧ N_mul_two_comp_RTL N s3 ai s6 ∧
         N_add_RTL N s5 s4 qr ∧ N_add_RTL N s6 s4 qi
```

- Verification of the RTL to fixed-point level of N-bit complex multiplication:

```
 Lemma 16:
         ∀ N ar ai br bi qr qi.
           N_complex_mul_two_comp_RTL N ar ai br bi qr qi ⟹
           (fxp_complex (FXP N qr,FXP N qi) =
            fxp_complex_mul (N,N - 1,1) (fxp_complex (FXP N ar,FXP N ai))
              (fxp_complex (FXP N br,FXP N bi)))
```

- Radix-4 butterfly in fixed-point level:

```
radix_4_butterfly_fxp =
    ⊢def  ∀ X a b c d e f g h.
         radix_4_butterfly_fxp X a b c d e f g h =
         (e =
          fxp_complex_add X (fxp_complex_add X a b)
            (fxp_complex_add X c d)) ∧
         (f =
          fxp_complex_add X
            (fxp_complex_sub X a (fxp_complex_mul X b (fxp_ii X)))
            (fxp_complex_sub X (fxp_complex_mul X d (fxp_ii X)) c)) ∧
         (g =
          fxp_complex_add X (fxp_complex_sub X a b)
            (fxp_complex_sub X c d)) ∧
         (h =
          fxp_complex_sub X
            (fxp_complex_sub X a (fxp_complex_mul X b (fxp_ii X)))
            (fxp_complex_add X c (fxp_complex_mul X d (fxp_ii X))))
```

- Radix-4 butterfly in RTL level:

```
radix_4_butterfly_RTL =
    ⊢def  ∀ N ar ai br bi cr ci dr di q1r q1i q2r q2i q3r q3i q4r q4i.
     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
```

- Verification of the RTL to fixed-point level of a radix-4 butterfly:

```
Lemma 17:
        ∀ N ar ai br bi cr ci dr di q1r q1i q2r q2i q3r q3i q4r q4i.
        radix_4_butterfly_RTL N ar ai br bi cr ci dr di q1r q1i q2r q2i
            q3r q3i q4r q4i ⟹
          radix_4_butterfly_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_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))
```

- Radix-4 dragonfly in fixed-point level:

```
radix_4_dragonfly_fxp =
    ⊢def  ∀ X a b c d w e f g h.
        radix_4_dragonfly_fxp X a b c d w e f g h =
        ∃ y1 y2 y3 y4.
          radix_4_butterfly_fxp X a b c d y1 y2 y3 y4 ∧
          (e = fxp_complex_mul X (w 1) y1) ∧
          (f = fxp_complex_mul X (w 2) y2) ∧
          (g = fxp_complex_mul X (w 3) y3) ∧
          (h = fxp_complex_mul X (w 4) y4)
```

- Radix-4 dragonfly in RTL level:

```
radix_4_dragonfly_RTL =
 ⊢def  ∀ N ar ai br bi cr ci dr di wr wi q1r q1i q2r q2i q3r q3i q4r q4i.
       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
```

- Verification of the RTL to fixed-point level of a radix-4 dragonfly:

```
 Lemma 18:
     ∀ N ar ai br bi cr ci dr di q1r q1i q2r q2i q3r q3i q4r q4i wr wi.
      radix_4_dragonfly_RTL N 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))
```

- Radix-4 16-point decimation-in-frequency FFT in fixed-point level:

```
radix_4_16_point_DIF_FFT_fxp =
    ⊢def  ∀ N x A w.
        radix_4_16_point_DIF_FFT_fxp N x A w =
        ∃ A1.
        radix_4_dragonfly_fxp (N,N − 1,1) (x 0) (x 4) (x 8) (x 12) (w 0)
            (A1 0) (A1 4) (A1 8) (A1 12) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (x 1) (x 5) (x 9) (x 13) (w 1)
            (A1 1) (A1 5) (A1 9) (A1 13) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (x 2) (x 6) (x 10) (x 14) (w 2)
            (A1 2) (A1 6) (A1 10) (A1 14) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (x 3) (x 7) (x 11) (x 15) (w 3)
            (A1 3) (A1 7) (A1 11) (A1 15) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (A1 0) (A1 1) (A1 2) (A1 3)
            (w 4) (A 0) (A 4) (A 8) (A 12) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (A1 4) (A1 5) (A1 6) (A1 7)
            (w 5) (A 1) (A 5) (A 9) (A 13) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (A1 8) (A1 9) (A1 10) (A1 11)
            (w 6) (A 2) (A 6) (A 10) (A 14) ∧
        radix_4_dragonfly_fxp (N,N − 1,1) (A1 12) (A1 13) (A1 14) (A1 15)
            (w 7) (A 3) (A 7) (A 11) (A 15)
```

- Radix-4 16-point decimation-in-frequency FFT in RTL level:

```
radix_4_16_point_DIF_FFT_RTL =
    ⊢def  ∀ N xr xi ar ai wr wi.
          radix_4_16_point_DIF_FFT_RTL N 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)
```

- Verification of the RTL to fixed-point level of radix-4 16-point decimation-in-frequency FFT:

```
Lemma 19:
          ∀ 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_NUM N wr wi)
```

- Radix-4 16-point decimation-in-frequency FFT algorithm in ideal real domain:

```
radix_4_16_point_FFT_real_algorithm =
    ⊢def  ∀ x w.
          radix_4_16_point_FFT_real_algorithm x w =
          (λ p.
             complex_sum (0,4)
               (λ i.
                  complex_sum (0,4)
                  (λ j. x (4 * j + i) * w (4 * p MOD 4 * j)) * w (p * i)))
```

- Radix-4 16-point decimation-in-frequency FFT algorithm in floating-point domain:

```
radix_4_16_point_FFT_floating_point_algorithm =
    ⊢def  ∀ x' w'.
         radix_4_16_point_FFT_floating_point_algorithm x' w' =
         (λ p.
            float_complex_sum (0,4)
              (λ i.
                 float_complex_sum (0,4)
                   (λ j. x' (4 * j + i) * w' (4 * p MOD 4 * j)) *
                 w' (p * i)))
```

- Radix-4 16-point decimation-in-frequency FFT algorithm in fixed-point domain:

```
radix_4_16_point_FFT_fixed_point_algorithm =
    ⊢def  ∀ X x'' w''.
         radix_4_16_point_FFT_fixed_point_algorithm X x'' w'' =
         (λ p.
            fxp_complex_sum (0,4) X
              (λ i.
                 fxp_complex_mul X
                   (fxp_complex_sum (0,4) X
                      (λ j.
                         fxp_complex_mul X (x'' (4 * j + i))
                           (w'' (4 * p MOD 4 * j)))) (w'' (p * i))))
```

- Verification of the expanded fixed-point to closed form fixed-point algorithmic level of radix-4 16-point decimation-in-frequency FFT:

```
 Lemma 20:
         ∀ N x w A.
           radix_4_16_point_DIF_FFT_fxp N x A w ⟹
           ∀ p.
             A p =
             radix_4_16_point_FFT_fixed_point_algorithm (N,N − 1,1) x
               (w (p DIV 4)) p
```