Formal Error Analysis and Verification of the Frequency Domain Equalizer

Anis Souari[1], Amjad Gawanmeh[2], and Sofiène Tahar[1]

[1]Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
[2]Department of Electrical and Computer Eningeering
Khalifa University of Science Technology and Research, Sharjah, UAE
Email: {asouari, amjad, tahar}@ece.concordia.ca

February 2012

Contents

List of Figures

List of Tables

Abstract

In this work we provide formal verification of a frequency domain equalizer using theorem proving techniques in Higher Order Logic framework. We perform formal error analysis to verify an implementation of the equalizer based on the Fast LMS algorithm. The formal error analysis is performed at the floating-point, fixed-point, and real numbers domains. The expressiveness of HOL allows us to model the equalizer in all the three number domains. The errors generated by approximating the floating-point and the fixed-point designs to the real domain were used to complete the error analysis of the frequency domain equalizer by deducting the error between the floating-point and fixed-point formalizations of the design. This application shows the efficiency of formal methods in verifying complex system such as the frequency domain equalizer.

1 Introduction and Motivation

With the recent technological growth, electronic devices have invaded all aspects of our life. These devices are getting more and more compact and consequently more complex. The price of this complexity is the challenge of delivering error-free devices, which requires thorough testing and verification at all stages of the design flow. On the other hand, a faulty design can lead to delays for time-to-market. Therefore, design verification is necessary to avoid such situations and is considered a bottleneck in the design process. In order to verify that an implementation meets its specification, simulation is the most widely used technique in the industry, because it is straightforward and does not need any expertise. It is a functional verification based on test patterns generation, and therefore, it does not provide full coverage for the system under test. On the other hand, formal verification techniques [15] are considered complementary to simulation as they can provide full coverage for the system under test, and in addition, they can catch corner cases bugs in the design.

Equalization is one of the applications of adaptive filtering. Its role consists of eliminating the inter-symbol interference caused by the noise in the transmission environment. To get an output matching as much as possible to the desired response, uncountable adaptive algorithms are used to regulate the filter or the equalizer coefficients. To decrease the filtering complexity, the equalizer can be implemented in the frequency domain where time convolution is replaced by frequency multiplication, this method offers low complexity growth in comparison with the time domain approach. This technique requires the use of the Fast Fourier Transform (FFT) modules. One possible implementation for the equalizer is based on the Fast LMS algorithm.

However, data processing and filtering in the frequency domain requires dealing with data at different domains: real numbers, floating point numbers, and fixed point numbers domains. This conversion generates and accumulates errors because of the different level of accuracy provided by each number domain. Therefore, a frequency domain multiplication based system must be tested thoroughly and error analysis must be conducted to be sure about the correctness of its operation.

Verifying the correctness of the equalizer is very challenging because of many reasons. First the equalizer implementation is based on an iterative algorithm, second, it contains multiple FFT and IFFT blocks, Finally, it contains multiple mathematical operator using several number domains. For these reasons, errors are naturally generated during data conversion between different domains, and can accumulate while performing various algorithmic iterations, FFT, and IFFT operations. Therefore, an implementation of such system must be verified in order to be sure that error accumulation is within acceptable limits. Traditionally, this is achieved using simulation. Simulink framework can be used in order to develop and simulate a model for the system under test. The generated error is estimated in in every step of the simulation for certain number of iterations. To achieve a certain level of assurance, specifically, in safety critical applications, simulation becomes inadequate, since the method is based on estimating the error. In addition, simulation time becomes tremendous. To overcome these verification problems in simulation based verification, we will use theorem proving based verification in order to provide formal error analysis of the Fast LMS algorithm.

Higher-order-logic (HOL) theorem proving [8] is a formal method that is used to conduct precise analysis of various systems. It is based on a system of deduction with a precise semantics and is expressive enough to be used for the precise specification of systems such as the frequency domain equalizer.

In this work we will use HOL theorem proving technique in order to provide error analysis for an implementation of the Frequency domain equalizer based on Fast LMS algorithm. This analysis is required to show that the error generated in the implementation of the Fast LMS

algorithm conforms with the required accuracy of conversion in the Equalizer design to operate properly. The formal analysis of the algorithm intends to show that, when converting from one number domain to another, the algorithm produces the same values with an accepted error margin caused by the round-off error accumulation. We will use the DSP verification methodology developed by Akbarpour [2].

1.1 Related Work

In this section we go through literature work on formalizing error analysis using formal methods. John Harisson [1] leads this research topic as he was using the HOL-Light theorem prover to approximate floating-point algorithms to their mathematical counterparts. He mainly proved that the floating-point exponential function has a correct overflow behavior and when this overflow is absent, the result is linked to a precise error value. The error analysis done by Harisson is similar to the work performing error analysis for DSP algorithms where statistical methods and square error analysis for DSP algorithms are used. In this latter type of error analysis, the error, represented as an independent random variable, is calculated depending on the arithmetic type and the rounding mode. Finally, the mean square error is given after performing the error analysis.

Akbarpour [2] continued the work of [1] and proposed an error analysis framework based on theorem proving and dedicated specially to DSP algorithms. The methodology is based on the idea of representing the system in the three domains; the real, the floating-point and the fixed-point. Then, he calculated the error in the transitions from real to floating-point and real to fixed-point. Finally, the error in the transition from floating-point to fixed-point is driven by doing a subtraction between the two types of error calculated before. To valorize his work and to show the feasibility of his methodology, Akbarpour applied his technique on digital filters [3] as well as on a 16 point radix 2 FFT [4]. Abu Nasser [5] adopted the methodology of Akbarpour to study the error analysis of an FFT-IFFT which is a combination of a 64 point radix 4 FFT and IFFT blocks. Abu Nasser proves that the approach in [2] is scalable and his work is considered as big case study of the work of Akbarpour. Our Work is also considered as an application of the formal verification framework developed in [2] since it is dealing with the error analysis of a frequency domain equalizer.

The application we verify in this work is considered more complex and error prone than the design in [5], where there is only a single combination FFT and IFFT blocks, whereas our system is composed of three FFTs and two IFFTs. In addition, the Equalizer is based on more arithmetic operations that use complex numbers of type real, floating-point and fixed-point numbers. Hence, the formalization of error expressions and error analysis we intend to perform on the design is based on a theorem for complex numbers of the above different types. Finally, error analysis for the equalizer requires formalizing input vectors to be able to store various symbols in each iteration of the Fast LMS algorithm.

The rest of the report is organized as follows. Section 2 provides a theoretical description of the FFT and the frequency domain equalizer along with its implementation based on the Fast LMS algorithms. Section 3 introduces HOL theorem proving and definitions of the main theorems leading to perform the error analysis in HOL. In section 4, the error analysis of the frequency domain equalizer is formalized in HOL. Finally, Section 5 concludes the report and presents hints for future work directions.

2 Frequency Domain Equalizer

Data alteration between the frequency domain and the time domain requires the use of some tools ensuring the preservation of data during this transition. The most useful tool enabling representing a signal in the frequency domain is the discrete transform that helps to decrease the computational complexity related to signal processing just like convolution. One of the most efficient transforms allowing the transition between the two domains is the Fast Fourier Transform (FFT).

2.1 Discrete Fourier Transform

The Discrete Fourier Transform is one of the structures of the Fourier Transforms allowing the conversion of a discrete signal from the time domain to the frequency domain. The expression below represents the equation calculating the Discrete Fourier Transform.

$$X(k) = \sum_{n=0}^{N-1} x(nT)e^{-jkwnT} \tag{1}$$

It is obvious that this expression is similar to the equation calculating the Fourier Transform for continuous signals in time domain. The DFT has two important properties which are:

- **Summetry**

That means that the two elements *X(k)* and *X(k+N)* resulting by applying the Discrete Fourier Transform on the signal x are the same. As a result, it can be deduced that there is a periodicity with period *N*.

- **Convolution**

The circular convolution as well as the linear convolution can be deduced by the use of the Discrete Fourier Transform. The time convolution theorem declares that a convolution in time domain is transformed into a simple multiplication in the frequency domain. This is summarized by the following expression.

$$x(n) = x_1(m) * x_2(n) = F^{-1}X_1(k)X_2(k) \tag{2}$$

where x, x_1 and x_2 are finite periodic signals having the same length and * expresses circular convolution. F^{-1} expresses the Inverse Discrete Fourier Transform. This property of the Discrete Fourier Transform has great importance for this project because instead of using convolution in time domain which can increase the computational cost, we can use the multiplication in the frequency domain and as a result we can get the same results with less complexity by just applying the DFT.

2.2 Fast Fourier Transform

One of the most useful and efficient algorithms used to put in work the Discrete Fourier Transform is the Fast Fourier Transform (FFT). Its role consists in transforming a vector x expressed in time domain to its equivalent X in the frequency domain. To reduce the computational cost and to make the algorithm more efficient, The FFT is based on the principle of the built in redundancy used by the DFT. Besides the FFT permitting the transformation of a signal from the time domain to the frequency domain, another algorithm allowing the conversion of a signal

in the other direction that means from the frequency domain to the time domain; it is the Inverse Fourier Transform (IFFT).

The computational complexity for the Discrete Fourier Transform is equal to N2 complex multiplies, but when we are talking about the FFT this number is decreased and its complexity is equal to (N/2) log2 (2N) complex multiplies + N complex adds. To get much better results, the block length N must be an integer power of 2. By doing this, we enhance the performance of the FFT algorithm. The FFT block length must be the same as the input signal block length.

2.3 The Fast LMS Algorithm

The implementation of the frequency domain equalizer [6, 7] is based on an adaptive frequency domain algorithm called Fast LMS. This version of Fast LMS algorithm is based on the overlap-save convolution algorithm. Updating the equalizer coefficients in the frequency domain using the Fast LMS algorithm is similar to the process in the time domain using the LMS algorithm [6]. One difference between the two procedures in terms of updating coefficients consists in that the Fast LMS algorithm updates the coefficients block by block not sample by sample. Figure 1 below illustrates the principle of the Fast LMS algorithm.

The Fast LMS algorithm is described as follows:

1. The FFT is applied to a 2N input block obtained from the input signal.

$$U(k) = FFT\{u(n)\} \tag{3}$$

2. Multiplying U(k) by the filter coefficients gives the equalizer output in the frequency domain. The coefficients are adjusted before.

$$Y(k) = U(k).W(k) \tag{4}$$

   Then an IFFT is applied to Y(K) to get the result in the time domain.

$$y(n) = IFFT\{Y(k)\} \tag{5}$$

   Because of the circular convolution, only the last N samples must be kept and they will represent the output of the equalizer.

$$y(n) = y(N + 1 \rightarrow 2N) \tag{6}$$

3. Next, a subtraction between the desired signal and the current equalizer output must be calculated to calculate the error signal.

$$e(n) = d(n) - y(n) \tag{7}$$

   where e(n) is the error and d(n) is the desired signal. After that the error must be transformed to the frequency domain, that is why an FFT is applied to e(n) after adding N zeros to its start.

$$E(k) = FFT\{zeros, e(n)\} \tag{8}$$

4. After calculating the conjugate of the U(k), it is multiplied by the error in the frequency domain. Then, an IFFT is applied to the result. Only the first N samples of this result are kept because of the circular convolution.

$$g(n) = IFFT\{E(k).U'(k)\} \tag{9}$$

$$g(n) = g(1 \rightarrow N) \tag{10}$$

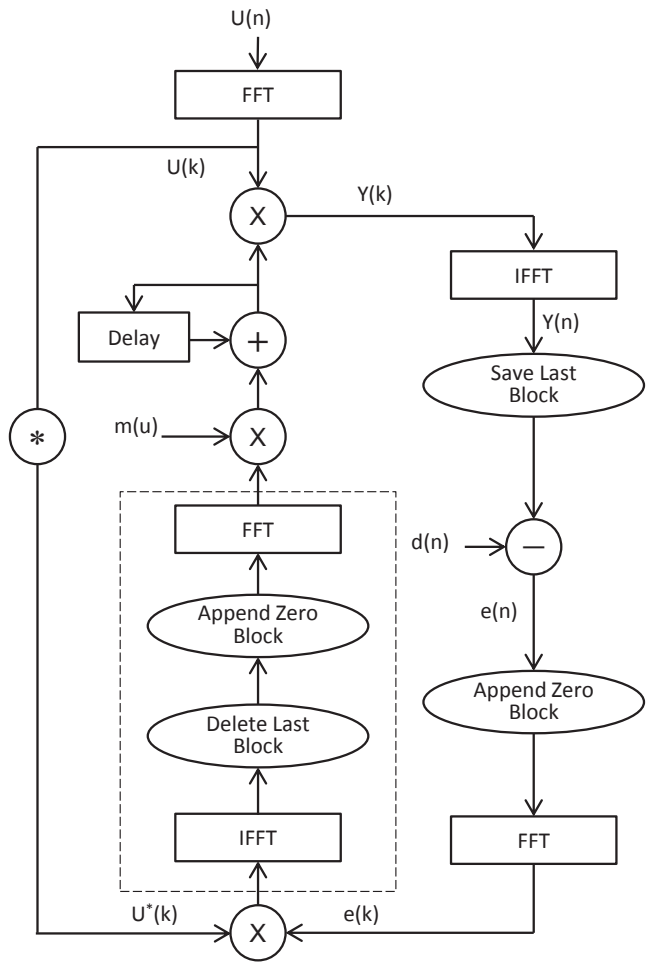

Figure 1: Frequency Domain Equalizer Design using the Fast LMS algorithm [6]

5. A *2N* point FFT is now applied on g(n) after adding N zeros to its end then the result is multiplied by the step size parameter μ.

$$g(n) = g(n)\ followed\ by\ N\ zeros \tag{11}$$

$$W_1(k) = \mu.FFT\{g(n)\} \tag{12}$$

The obtained result consists in the update factor for the equalizer coefficients, that is why it is added to the previous value of the filter coefficients.

$$W_1(k+1) = W(k) + W_1(k) \tag{13}$$

6. The updated equalizer coefficients are set and ready to be used with the next input block. From one iteration to another the error is decreasing since the coefficients are updated progressively.

The implementation of the Equalizer based on the Fast LMS algorithm was tested using simulation in Simulink environment [14]. The simulation was based on error estimation for the 4-tap frequency domain equalizer converges after almost 200 symbols to reach the value of -40 dB. On the other hand, an FPGA based implementation is simulated for a 2-tap equalizer on the one million gate Spartan 3 FPGA board. The results obtained from the Simulink model were better than those obtained from the System Generator model because symbol were described using floating point in Simulink fixed point in System Generator. Figure 2 below shows the Simulink model for the Fast LMS algorithm.

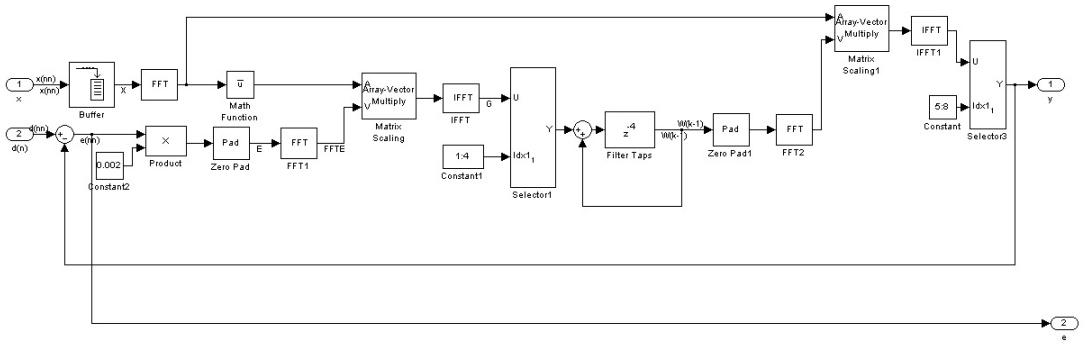

Figure 2: Fast LMS algorithm in Simulink [14]

Testing based verification for the Fast LMS algorithm in the above cases was based on estimating the error generated in every step. The accuracy of the verification process was affected thoughtfully by the method used in the framework to model numbers, be it floating point or fixed point. In addition, the verification process was based on applying a specific number of iterations, therefore in order to get more assurance about generated error, more simulation is required. For a certain level of assurance, specifically, in safety critical applications, this becomes infeasible, since simulation time becomes tremendous.

To overcome these verification problems in simulation based verification, we will use theorem proving based verification in order to provide formal error analysis of the Fast LMS algorithm.

3 Error Analysis In HOL Theorem Prover

3.1 HOL general description

The HOL [9] theorem prover is an interactive tool dedicated to conduct proofs in higher-order logic. The most common notations used in HOL are presented in Table 1. There are only four types of terms in HOL: variables, constants, function application and lambda terms. The main core of HOL consists of five axioms and eight inference rules. All the theories existing in HOL are built on the top of them. All the theories should be proved before they are added to HOL inference system.

3.2 Preliminaries

To formalize the error due to the floating-point rounding in HOL, we refer to the following theorem. This latter is considered as the most fundamental theorem dealing with the floating-point rounding error.

Table 1: HOL Notation

Standard Notation	HOL Notation	Description
$\top$	T	True
$\bot$	F	False
And	$\wedge$	Logical And
Or	$\vee$	Logical Or
$\Longrightarrow$	==>	Implies That
$\forall$ x	! x	Forall x
$\exists$ x	? x	There Exists x

Theorem 1: *If x is a real number within the floating-point range, then* $x_R = x(1 + \delta), where|\delta| \leq 2^{-p}$

where p is the precision of the floating-point format. Looking at the theorem above, we notice that in the floating-point domain the rounding error is introduced multiplicatively [10, 11]. Applying this theorem on arithmetic operations (addition, subtraction, multiplication and division), leads to the following expression, where * refers to any type of arithmetic operation.

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

Any floating-point operation is linked to its abstract mathematical counterpart by above the theorem according to a precise error value. Concerning the fixed-point domain, the rounding effect is presented additively. As it is shown in the fundamental error analysis theorem below, the floating-point value x_R is given by the addition of the error to the real number x.

Theorem 2: *If x is a real number within the fixed-point range, then* $x_R = (x + \epsilon), where|\epsilon| \leq 2^{-fracbits(X)}$

Similar to the floating-point domain, the application of the above theorem on the fixed-point arithmetic operations is summarized in the following expression:

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

where * refers to all common operations.

3.3 Methodology

Our methodology, as depicted in Figure 3, is based on a formal model for numbers in three different domains: fixed point, real, and floating point domain and a valuation procedure for numbers conversion from the fixed point domain to real domain, and also from floating point domain to the real domain, all for FFT operations. Based on this conversion, error analysis is performed between the actual real values obtained from FFT and the converted ones from both floating point and fixed point domains. Finally further analysis is performed to show the error analysis between fixed point and floating point.

The rectangles in Figure 3 refer to the Fast LMS algorithm in a specific domain; real, floating-point or fixed point. The hexagons are dedicated to represent the results of the different error analysis that we have done.

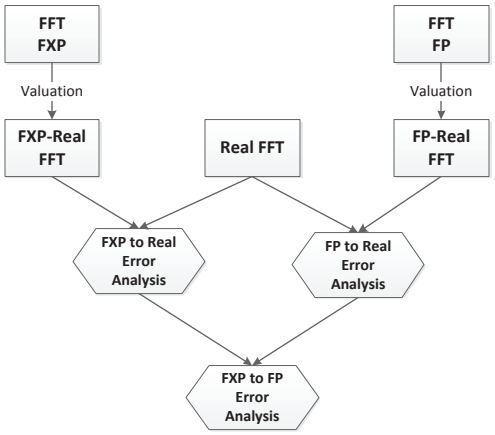

Figure 3: Error Analysis Verification Methodology in HOL

In this verification methodology, the Fast LMS algorithm should be formalized in the three number domains exactly the same way as it is defined in section 2.3. Next, the valuation functions should be used in order to return real approximations of the floating-point and fixed-point algorithms. The error analysis consists, first, in extracting the rounding error by performing the FXP to Real and FP to Real error analysis which are defined respectively as the difference between the approximations of the fixed-point and the floating-point algorithms and the real specification. Finally, we perform the FXP to FP error analysis by deducing the round-off error between floating-point and fixed-point.

To perform the error analysis of the Fast LMS algorithm, we used the existing theories in HOL and built on top of them the necessary theories to reason about error generation and accumulation in the equalizer. First, the construction of complex numbers was necessary for the implementation of the specification. Regarding the floating-point and the fixed-point modeling of the design, we used, respectively, the formalization of IEEE 754 standard based floating-point arithmetic [3] the fixed-point arithmetic formalization developed by Akbarpour *et. al.* [12].

3.4 Formalizing the Fast LMS Algorithm for the Frequency Domain Equalizer in HOL

In order to model the error analysis of the frequency domain equalizer, many existing HOL theories developed by Abdullah [5] were used. We also constructed complex numbers and many other required functions like the complex sum in all three number domains. The error analysis is finally formalized using the theorems and lemmas that we have developed.

The definitions that we used in this section, except those of the FFT and IFFT, were defined in [5]. We modified most of them to be coherent with the theorems that we developed.

We defined an HOL theorem for every building block of the design, then, we defined one comprehensive theorem for the whole design to show the validity of rounding and error accumulation. These theorems theorems were all proved in HOL framework. In this section, we will discuss the major theorems we defined for the design. The structure and relationship between these algorithms is shown in Figure 4.

3.4.1 Real Number Domain Modeling

Modeling the frequency domain equalizer using HOL in real domain requires formalizing complex numbers. We defined complex as a new datatype based on a pair of real numbers as shown below:
`complex =` $\vdash_{def}$ `complex of (real # real)`

The real and imaginary part of a complex number are also formalized in HOL.
`Re_def =` $\vdash_{def}$ `Re (complex (a,b)) = a`

`Im_def =` $\vdash_{def}$ `Im (complex (a,b)) = b`

We also define properties on complex numbers that are needed to model the equalizer in HOL. For instance, the conjugation which is formalized using this definition:
`CNJ =` $\vdash_{def}$ $\forall$ `z. CNJ z = complex (Re z,¬ Im z)`

Arithmetic operations on complex numbers such as addition, subtraction and multiplication were also defined in HOL and properties about these operations were proved, such as complex multiplication and complex addition commutativity.

The principal n-roots of unity, was defined in HOL using Euler's identity as follows:

```
principal_root =
⊢def ∀ n k. principal_root n k =
complex (cos n * k * Π/¬2,sin n * k * Π/¬ 2)
```

In addition, a required complex constant was defined as
`complex_4_def =` $\vdash_{def}$ `complex_4 = complex (1 / 4,0)`

One of the most important functions is complex summation; used specially to define FFT and IFFT. It is defined recursively as follows:

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

where n and 0 are upper and lower indices, respectively, and f is a function. Based on the above necessary definitions, we can define both FFT and IFFT in HOL as follows:

```
real_FFT_def =
⊢def ∀ x k. real_FFT x k =
 rec_sum (0,3) (\n. principal_root n k * EL n x)

real_IFFT_def =
⊢def ∀ L k. real_FFT x k =
 complex_4 * rec_sum (0,3) (\n. principal_root n k * EL n L)
```

These definitions are based on complex summation and the principal n-roots functions, that were defined above. In addition, we adopted the function *EL* from the *pairTheory* in order to extract the *nth* element of a complex list L. The term `\n` in the *real_FFT_def* and *real_IFFT_def* theorems is a *lambda-abstraction* which means that the sum is a function of n.

3.4.2 Floating-point Domain Modeling

Complex numbers modeling in this domain is similar to the real domain except that here the complex numbers are represented as pair of floats.

```
complex = ⊢def  complex of (float # float)
```

The definitions of real and imaginary parts, *float_Re* and *float_Im* respectively, are defined in the same way as the definitions presented above. The arithmetic operations on complex numbers of type float and the principal n-roots of unity in floating-point domain, *float_principal_root*, are defined in a straightforward way. The definitions are not given here since they are similar to the aforementioned definition. Floating-point complex summation is defined using as follows:

```
float_complex_sum_def  =
⊢def  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)
```

John Harrison tackles in [1] the IEEE floating-point formalization as well as the rounding problem of floating-point numbers. This latter issue consists in linking a real number to its nearest floating-point one. There are three ways to do the rounding which are either towards zero or towards negative or positive infinity. In this work, we defined the function *float_complex_round* which returns the rounding value of a floating-point complex number. This function uses the predefined *round* function which has three parameters; *float_format* dedicated for the floating-point precision, *To_nearest* indicating the rounding format, and finally *Re z* or *Im z* representing the real number that we want to round. The floating-point complex rounding function is given by the definition below:

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

The inverse function of rounding is called valuation, it gives the equivalent real of any floating-point number. It is defined in HOL as *Val*. For the valuation of the complex numbers of type *float* we define the function *float_complex_val*.

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

A detailed definition and description of the valuation function is given by John Harrison in [1]. The functions used to define *Val* such as *valof* and *defloat* can be found in the *floatTheory* [8] of HOL.

The FFT and IFFT blocks in the floating-point domain are formalized as follows:

```
float_FFT_def =
⊢def  ∀ x k. float_FFT x k =
    float_complex_sum (0,3) (\n. float_principal_root n k * EL n x)

float_IFFT_def =
⊢def  ∀ L k. float_IFFT x k =
    float_complex_4 * float_complex_sum (0,3)
    (\n. float_principal_root_1 n k * EL n L)
```

We used the functions *float_principal_root* and *float_principal_root_1* to define *float_FFT* and *float_IFFT*, respectively. These two former functions were the rounding result of the two functions *principal_root* and *principal_root_1* defined in the previous section.

3.4.3 Fixed-point Modeling

The formalization of the Fast LMS algorithm in HOL in the fixed-point domain is different from the formalization in the other two domains. This is due to the use of the primitive parameters for arbitrary attributes related to fixed-point numbers. We used *fxp* to define complex numbers in the fixed point domain. Retrieving the real and imaginary parts of a complex number of type *fxp* is achieved using *fxp_Re* and *fxp_Im*, respectively. Complex addition, complex subtraction and complex multiplication are defined using functions *fxp_complex_add*, *fxp_complex_sub* and *fxp_complex_mul*, respectively. As we mentioned the definition of the complex summation for the fixed-point domain is given as:

```
fxp_complex_sum_def =
⊢def  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_sum (n,m) X f) (f (n + m))
```

The function *Fxp_round* converts a real number into its fixed-point equivalent number. To perform rounding of a complex number of type *fxp*, we defined the function *fxp_complex_round_def* as follows:

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

To obtain the real value of a fixed-point number, we use the function *value* that is defined in the *fxpTheory* in HOL. The function *fxp_complex_value* is defined to perform the valuation of fixed-point complex numbers is defined as:

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

Using the above definitions,we formalize the FFT and IFFT blocks in the fixed-point domain as follows:

```
fxp_FFT_def =
⊢def  ∀ X x k. fxp_FFT X x k =
    fxp_complex_sum (0,3) X (\n. (fxp_complex_mul X
    (fxp_principal_root X n k)  (EL n x) ))

fxp_IFFT_def =
⊢def  ∀  X L k. fxp_IFFT X L k  =
    fxp_complex_mul X (fxp_complex_4 X) (fxp_complex_sum (0,3) X
    (\n.(fxp_complex_mul X (fxp_principal_root_1 X n k)  (EL n L) )))
```

where *fxp_complex_4 X* refers to the term $1/N$ in the IFFT equation. It is a constant complex number which its value is equal to $1/4$ since the number of taps, N, adopted for our design is equal to 4.

3.5 Formal Error Analysis in HOL

After modeling the basic components of the frequency domain equalizer in HOL in the three number domains, we can proceed with error analysis of the equalizer. We formalize errors in the floating-point and fixed-point domains for the designs using the following two lemmas; *float_complex_val* and *fxp_complex_value*, which are defined as follows:

```
float_complex_val =
⊢lemma  ∀ z. float_complex_val z =
    complex (Val (float_Re z),Val (float_Im z))

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

The effect of these functions on the arithmetic operations is inherited from the effect of the function *Val* and *value*. The function *Val* is used to define the rounding error due to the valuation of the floating-point in real number. The effect of the *Val* function on arithmetic operations is defined as:

1. $\forall\ a\ b.\exists\ e.Val(a+b) = (Val\ a + Val\ b) * (1+e)$
2. $\forall\ a\ b.\exists\ e.Val(a-b) = (Val\ a - Val\ b) * (1+e)$
3. $\forall\ a\ b.\exists\ e.Val(a*b) = (Val\ a * Val\ b) * (1+e)$

where a and b be two floating-point numbers and e a real number. e in the above lemmas defines the error caused by the valuation of the fixed-point in real number.

The effect of the *value* function on arithmetic operations is given as:

1. $\forall\ a\ b\ X.\exists\ e.value(FxpAdd\ X\ a\ b) = value\ a + value\ b + e$
2. $\forall\ a\ b\ X.\exists\ e.value(FxpSub\ X\ a\ b) = value\ a - value\ b + e$
3. $\forall\ a\ b\ X.\exists\ e.value(FxpMul\ X\ a\ b) = value\ a * value\ b + e$

These lemmas are dedicated to complete the error analysis for rounding numbers between different domains. In order to perform complete error analysis for the whole design, each block of the Fast LMS algorithm described above in Figure 1 must be formalized in HOL. Figure 4 below shows the structure of HOL theorems that were defined and proved for every block in the equalizer.

For illustrated purposes, we present below the steps of the error analysis applied on one of the blocks of the equalizer which is the FFT. First, the floating-point FFT is valuated using the *float_complex_val* lemma. The *FPReal FFT* is given as:

```
      (((((float_complex_val((float_principal_root 0 k)* (EL 0 x1)) *
    complex ((1 + e3) , 0)) +
   float_complex_val ((float_principal_root 1 k) * (EL 1 x1))) *
    complex ((1 + e2) , 0)) +
   float_complex_val((float_principal_root 2 k) * (EL 2 x1))) *
   complex ((1 + e1) , 0))
```

where *complex((1 + e1) , 0))*, *complex((1 + e2) , 0))* and *complex((1 + e3) , 0))* are the errors generated because of the valuation.

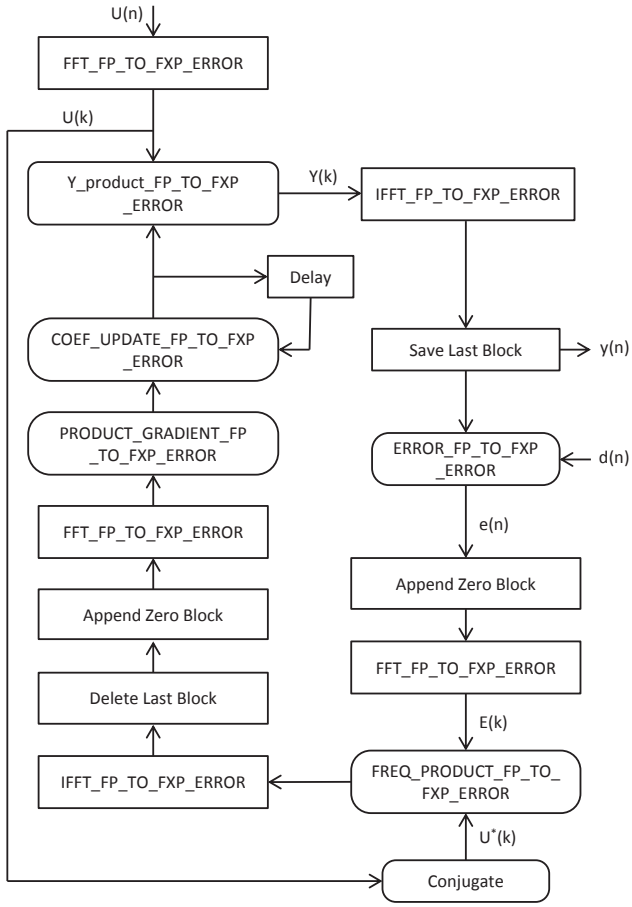

Figure 4: Structure of HOL Theorems for the Equalizer

Next theorem deals with error analysis between the real and the floating-point representations of the FFT block. It consists of a simple subtraction between the *FP-Real FFT* and the *Real FFT*. The theorem is defined as:

```
FFT_REAL_TO_FP_ERROR =
  ⊢thm   ∀ x x1 k. ∃e1 e2 e3.
   FFT_REAL_TO_FP_ERROR x x1 k =
    (((((float_complex_val
   ((float_principal_root 0 k)* (EL 0 x1)) * complex ((1 + e3) , 0))
   + float_complex_val ((float_principal_root 1 k) * (EL 1 x1)))
   * complex ((1 + e2) , 0)) + float_complex_val
   ((float_principal_root 2 k) * (EL 2 x1))) * complex ((1 + e1) , 0))
   - (rec_sum (0,3) (\n.((principal_root n k)* (EL n x)))))
```

The function *fxp_complex_value* is defined in order to perform error analysis between the real and the fixed-point representations of the FFT block. On the other hand, the fixed-point FFT is valuated using the *fxp* complex value and an approximation of the FFT in real domain. The valuated FFT is given by:

```
     (((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
(EL 0 x1))) + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul
X (fxp_principal_root X 1 k) (EL 1 x1))) + complex (e2 , e2))
+ fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
(EL 2 x1))) + complex (e1, e1))
```

where *complex(e1, e1)*, *complex(e2, e2)* and *complex(e3, e3)* are the errors generated because of the valuation.

The real to fixed-point error analysis of the FFT is established by proving that the produced error is is equivalent to subtraction between the valuated fixed-point FFT expression and the real one. The theorem below is used to formalize this error:

```
FFT_REAL_TO_FXP_ERROR =
⊢thm  ∀ x x1 X k.∃e1 e2 e3.
   FFT_REAL_TO_FXP_ERROR x x1 X k =
   (((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
   (EL 0 x1))) + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul
   X (fxp_principal_root X 1 k) (EL 1 x1))) + complex (e2 , e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
   (EL 2 x1))) + complex (e1, e1)) -  (rec_sum (0,3) (\(n:num).
   ((principal_root n k) * (EL n x)))))
```

Once the real to fixed-point error analysis is achieved, the fixed-point to floating-point error analysis of the FFT block is obtained by deducting the results of the real to floating-point and the real to fixed-point error expressions as given in Figure 3 above. The following theorem is defined in order to establish the fixed-point to floating-point error analysis of the FFT:

```
FFT_FP_TO_FXP_ERROR =
⊢thm  ∀  x xp xf X k. ∃  e1 e2 e3 e4 e5 e6.
   FFT_FP_TO_FXP_ERROR x xp xf X k =
   (((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
   (EL 0 xf))) + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul X
   (fxp_principal_root X 1 k)  (EL 1 xf))) + complex (e2, e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
    (EL 2 xf))) + complex (e1, e1)) - (((((float_complex_val
   ((float_principal_root 0 k) * (EL 0 xp)) * complex ((1 + e6) , 0))
   + float_complex_val ((float_principal_root 1 k) * (EL 1 xp)))
   * complex ((1 + e5) , 0)) + float_complex_val((float_principal_root 2 k)
```

Having an HOL theorem defined and proved for every building block of the design, we then need to define one comprehensive theorem for the whole design. This algorithm combines all of the algorithms that defines the block of the design together. Theorem $FAST\_LMS\_FP\_TO\_FXP\_ERROR$, given in the Appendix, is used to obtain the rounding error from each block and accumulate it together with the error produced by the successor block in the design as depicted in Figure 4. Eventually, the rounding error is obtained for the whole design and validated for the algorithm. This theorem is proved in HOL deduction system, which verifies that the rounding and accumulated error produced by steps of the algorithm are within the accepted range given in the specification of the design.

Theorems for error analysis of the blocks of the frequency domain equalizer that were formalized and proved in HOL are presented in the Appendix.

3.6 Discussion

Many existing theories in HOL, e.g., *arithmeticTheory*, *realTheory*, *listTheory*, *pairTheory*, *realLib*, *numLib*, *floatTheory*, *fxpTheory*, *ieeeTheory*, ..., were used to derive the rounding error analysis of the frequency domain equalizer. The definitions of the Fast LMS algorithm were formalized and proved based on these theories. All the definitions we formalized first in the real domain, and then all the arithmetic operators were overload to build the design in the floating-point and fixed-point domains using *floatTheory* and *fxpTheory*, respectively.

In order to complete the error analysis of the Fast LMS algorithm, quantification over functions, variables and objects was required. That is why, all the definitions were defined in higher order logic. Now, coming to the proof steps, HOL provides many tactics and tacticals. In most cases, it was difficult to prove some theorems in one shot, so we split them into many subgoals that were proved one by one and then were combined together.

The equalizer application shows that formal error analysis is applicable on larger scale systems such as the one we have analyzed, which is traditionally analyzed with paper and pencil or simulation based techniques based on estimating the error. Formal analysis proves that the implementation meets its specification with 100% coverage, something that is not feasible in simulation. In addition compared to the classical analytical technique, this method is computerized and has been conducted using an interactive tool, and, the theorems can be efficiently reused to verify other designs that make use of the same algorithm.

Some specific error analysis theorems that were used in this work were defined and proven by Akbarpour [2] and Abdullah [5]. However, we had to build our own theorems on top of these in order to formalize and verify every block of the Fast LMS algorithm, and consequently define one single theorem for the whole design. This shows that relevant theorems that are proven in HOL can be reused efficiently in order to verify complex systems of similar properties. In fact, scalability of HOL theorems is one of its best features, since wee designed proven theorems can be reused efficiently to verify other designs, which reduces time and effort, in particular while using the interactive HOL framework environment.

4 Conclusion and Future Work

In this work we apply theorem proving technique to provide formal error analysis for the frequency domain equalizer. The equalizer can be implemented using the Fast LMS algorithm where a number of mathematical operations are performed on numbers in three different domains: floating-point, fixed-point and real numbers. This requirers data to be converted between these domains, which in turn produces errors that can accumulate during the several iterations of the algorithm. This application is an ideal candidate for the error analysis framework developed by Akbarpour [2], where we can perform formal error analysis between real and floating-point numbers, and also between real and fixed-point numbers, and consequently the error analysis between fixed-point and floating-point is conducted. However, to perform this analysis, basic formal definitions and theorems were required in order to handle iterative nature of the algorithm and the multiple FFT and IFFT blocks in the equalizer.

Formal error analysis is used to show that errors in the equalizer algorithm occurring while converting from one number domain to the another are within the accepted range based on the deign specification of the equalizer. However, this was not a straightforward task and required building on top of existing HOL theorems for error analysis. In addition, derivation of new expressions for the accumulation of round-off error in the algorithm was necessary to obtain the correct formal model for rounding error. We developed an HOL theorem for every building block of the design, then, we defined one comprehensive theorem for the whole design to show

the validity of rounding and error accumulation. These theorems theorems were all proved in HOL framework. This application shows that formal error analysis is applicable on larger scale systems such as the one we have analyzed, which is traditionally analyzed with paper and pencil or simulation based techniques based on estimating the error.

As future work, we plan to extend the current work and perform error analysis using the GAPPA framework developed by Melquiond [13] [1]. GAPPA is a tool dedicated to formally proving programs dealing with floating-point and fixed-point arithmetics. The tool can return a numerical error range when performing error analysis, and therefore, our design will represent a challenge for this new theorem proving tool and it will be a good case study to compare the results returned by GAPPA and HOL in terms of use easiness and efficiency. Another interesting approach is to use first order theorem proving to verify properties about the functional behavior of the equalizer.

[1] The idea was proposed during a discussion with Dr. Marc Daumas

References

[1] J. Harrison. *Floating Point Verification in HOL Light: the Exponential Function*. Technical Report 428, University of Cambridge Computer Laboratory, Cambridge, UK, 1997.

[2] B. Akbarpour. *Formal Verification Methodology of DSP Designs*. PhD thesis, Department of ECE, Concordia University, Montreal, QC, Canada, 2005.

[3] B. Akbarpour and S. Tahar. *Error Analysis of Digital Filters using Theorem Proving*. In Theorem Proving in Higher Order Logics, volume 3223 of Lecture Notes in Computer Science, pages 1-16. Springer-Verlag, 2004.

[4] B. Akbarpour and S. Tahar. *A Methodology for the Formal Verification of FFT Algorithms in HOL*. In Formal Methods in Computer-Aided Design, volume 3312 of Lecture Notes in Computer Science, pages 37-51. Springer-Verlag, 2004.

[5] A. N. M. Abdullah, *Formal Analysis and Verification of an OFDM Modem Design*. MASc Thesis, Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada, February 2006.

[6] M. O. Fril, *Frequency Domain Adaptive Filtering*. MASc Thesis, National University of Ireland, 2005.

[7] P. A. Dmochowski. *Frequency Domain Equalization for High Data Rate Multipath Channels*. MASc Thesis, Queen's University, 2001

[8] M.J.C. Gordon. *Mechanizing Programming Logics in Higher Order Logic*. Technical report. University of Cambridge, Computer Laboratory, 1988.

[9] HOL Sourceforge Project. The HOL System Reference. http://hol.sourceforge.net, September 2011.

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

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

[12] B. Akbarpour, A. Dekdouk, and S. Tahar. *Formalization of Fixed-point Arithmetic in HOL*. Formal Methods in System Design, 27(1-2):173-200, 2005.

[13] G. Melquiond, *De l'Arithmtique d'Intervalles la Certification de Programmes*. PhD thesis, cole Normale Suprieure de Lyon, France, 2006.

[14] A. Souari, *FPGA Implementation of a Frequency Domain Equalizer*. MASc Thesis, cole Nationale dIngnieurs de Sousse, Sousse, Tunisie, December 2010.

[15] T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

A Real to Floating-point Error Analysis

```
FFT_REAL_TO_FP_ERROR =
 ⊢thm  ∀ x x1 k. ∃e1 e2 e3.
   FFT_REAL_TO_FP_ERROR x x1 k =
    (((((float_complex_val
   ((float_principal_root 0 k)* (EL 0 x1)) * complex ((1 + e3) , 0))
   + float_complex_val ((float_principal_root 1 k) * (EL 1 x1)))
   * complex ((1 + e2) , 0)) + float_complex_val
   ((float_principal_root 2 k) * (EL 2 x1))) * complex ((1 + e1) , 0))
   - (rec_sum (0,3) (\n.((principal_root n k)* (EL n x)))))

IFFT_REAL_TO_FP_ERROR =
⊢thm  ∀ L L1 k.∃e1 e2 e3.
   IFFT_REAL_TO_FP_ERROR L L1 k =
   float_complex_val float_complex_4 * ((((((float_complex_val
   ((float_principal_root_1 0 k) * (EL 0 L1)) * complex ((1 + e3) , 0))
   + float_complex_val ((float_principal_root_1 1 k) * (EL 1 L1)))
   * complex ((1 + e2) , 0)) + float_complex_val
   ((float_principal_root_1 2 k) * (EL 2 L1))) * complex ((1 + e1) , 0)))
   * complex (1 + e,0) - (complex_4 * rec_sum (0,3)
   (\n. ((principal_root_1 n k) * ((EL n L)))))

Y_product_REAL_TO_FP_ERROR =
⊢thm  ∀ W1 u W1_1 u_1 Y1.∃e1.
   Y_product_REAL_TO_FP_ERROR W1 u W1_1 u_1 Y1 =
   MAP (\k. (float_complex_val (float_CNJ (float_FFT u_1 k)))
   * (float_complex_val (EL k W1_1)) * complex (1 + e1,0)
   - (CNJ (real_FFT u k) * EL k W1)) Y1

ERROR_REAL_TO_FP_ERROR =
⊢thm  ∀ d d1 y y1 e.∃e1.
   ERROR_REAL_TO_FP_ERROR d d1 y y1 e =
   MAP (\k. (float_complex_val (EL k d1)
   - float_complex_val (EL k y1)) * complex (1 + e1,0)
   - (EL k d - EL k y)) e

FREQ_PRODUCT_REAL_TO_FP_ERROR  =
⊢thm  ∀ u e E u1 e1 E1 P.∃ e2.
   FREQ_PRODUCT_REAL_TO_FP_ERROR  u e E u1 e1 E1 P =
   MAP (\k. (float_complex_val (float_CNJ (float_FFT u1 k)))
   * (float_complex_val (EL k (float_error_FFT e1 E1)))
   * complex (1 + e2,0) - (CNJ (real_FFT u k) * ( EL k (error_FFT e E)))) P

PRODUCT_GRADIENT_REAL_TO_FP_ERROR =
⊢thm  ∀ W2 W2_1 W3.∃e1.
   PRODUCT_GRADIENT_REAL_TO_FP_ERROR W2 W2_1 W3 =
   MAP (\k. (float_complex_val float_complex_2)
   * (float_complex_val ( EL k W2_1)) * complex (1 + e1,0)
```

```
   - (complex_2 * ( EL k W2))) W3

COEF_UPDATE_REAL_TO_FP_ERROR =
⊢thm  ∀ W1 W3 W1_1 W3_1 W2.∃e1.
   COEF_UPDATE_REAL_TO_FP_ERROR W1 W3 W1_1 W3_1 W2 =
   MAP (\k. ((float_complex_val (EL k W1_1))
   + (float_complex_val (EL k W3_1))) * complex (1 + e1,0)
   - (EL k W1 + EL k W3)) W2
```

B Real to Fixed-point Error Analysis

```
FFT_REAL_TO_FXP_ERROR =
⊢thm  ∀ x x1 X k.∃e1 e2 e3.
   FFT_REAL_TO_FXP_ERROR x x1 X k =
   (((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
   (EL 0 x1))) + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul
   X (fxp_principal_root X 1 k) (EL 1 x1))) + complex (e2 , e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
   (EL 2 x1))) + complex (e1, e1)) -  (rec_sum (0,3) (\(n:num).
   ((principal_root n k) * (EL n x)))))

IFFT_REAL_TO_FXP_ERROR =
⊢thm  ∀ L L1 k X.∃ e e1 e2 e3.
   IFFT_REAL_TO_FXP_ERROR L L1 k X =
   ((fxp_complex_value (fxp_complex_4 X)) * (((((fxp_complex_value
   ((fxp_complex_mul X (fxp_principal_root_1 X 0 k)  (EL 0 L1)))
   + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul X
   (fxp_principal_root_1 X 1 k)  (EL 1 L1))) + complex (e2 , e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root_1 X 2 k)
   (EL 2 L1))) + complex (e1, e1))) + complex (e , e)
   - (complex_4 * rec_sum (0,3) (\n. ((principal_root_1 n k) * ((EL n L))))))

Y_product_REAL_TO_FXP_ERROR =
⊢thm  ∀ W1_2 u W1_1 u_1 X Y1. ∃ e1.
   Y_product_REAL_TO_FXP_ERROR W1_2 u W1_1 u_1 X Y1 =
   MAP (\k. (fxp_complex_value (fxp_CNJ X (fxp_FFT X u_1 k)))
   * (fxp_complex_value (EL k W1_1)) + complex (e1 , e1)
   - (CNJ (real_FFT u k) * EL k W1_2)) Y1

ERROR_REAL_TO_FXP_ERROR =
⊢thm  ∀ d d1 y y1 X e. ∃e1.
   ERROR_REAL_TO_FXP_ERROR d d1 y y1 X e  =
   MAP (\k. (fxp_complex_value (EL k d1) - fxp_complex_value (EL k y1))
   + complex (e1,e1) - (EL k d - EL k y)) e

FREQ_PRODUCT_REAL_TO_FXP_ERROR  =
⊢thm  ∀ u e E u1 e1 E1 X P. ∃ e2.
   FREQ_PRODUCT_REAL_TO_FXP_ERROR u e E u1 e1 E1 X P =
```

```
   MAP (\k.(fxp_complex_value (fxp_CNJ X (fxp_FFT X u1 k)))
   * (fxp_complex_value ( EL k (fxp_error_FFT X e1 E1))) + complex (e2, e2)
    - (CNJ (real_FFT u k) * ( EL k (error_FFT e E)))) P

PRODUCT_GRADIENT_REAL_TO_FXP_ERROR =
⊢thm  ∀ W2 W2_1 X W3. ∃ e1.
   PRODUCT_GRADIENT_REAL_TO_FXP_ERROR W2 W2_1 X W3 =
   MAP (\k. (fxp_complex_value (fxp_complex_2 X)) * (fxp_complex_value
   ( EL k W2_1)) + complex (e1, e1) - (complex_2 * ( EL k W2))) W3

COEF_UPDATE_REAL_TO_FXP_ERROR =
⊢thm  ∀ W11 W3 W1_1 W3_1  W2. ∃ e1.
   COEF_UPDATE_REAL_TO_FXP_ERROR W11 W3 W1_1 W3_1 X W2 =
   MAP (\k. (fxp_complex_value (EL k W1_1)) + (fxp_complex_value
   (EL k W3_1)) + complex (e1, e1) - (EL k W11 + EL k W3)) W2
```

C Floating-point to Fixed-point Error Analysis

```
FFT_FP_TO_FXP_ERROR =
⊢thm  ∀  x xp xf X k. ∃  e1 e2 e3 e4 e5 e6.
   FFT_FP_TO_FXP_ERROR x xp xf X k =
   (((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
   (EL 0 xf))) + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul X
   (fxp_principal_root X 1 k)  (EL 1 xf))) + complex (e2, e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
    (EL 2 xf))) + complex (e1, e1)) - (((((float_complex_val
   ((float_principal_root 0 k) * (EL 0 xp)) * complex ((1 + e6) , 0))
   + float_complex_val ((float_principal_root 1 k) * (EL 1 xp)))
   * complex ((1 + e5) , 0)) + float_complex_val((float_principal_root 2 k)
   * (EL 2 xp))) * complex ((1 + e4) , 0)))

IFFT_FP_TO_FXP_ERROR =
⊢thm  ∀  L Lp Lf X k. ∃ e1 e2 e3 e4 e5 e6 e7.
   IFFT_FP_TO_FXP_ERROR L Lp Lf X k =
   ((fxp_complex_value (fxp_complex_4 X)) * (((((fxp_complex_value
   ((fxp_complex_mul X (fxp_principal_root_1 X 0 k)  (EL 0 Lp)))
   + complex (e3 , e3)) + fxp_complex_value (fxp_complex_mul X
   (fxp_principal_root_1 X 1 k)  (EL 1 Lp)))+ complex (e2 , e2))
   + fxp_complex_value (fxp_complex_mul X (fxp_principal_root_1 X 2 k)
   (EL 2 Lp))) + complex (e1, e1))) + complex (e , e)
   - (float_complex_val float_complex_4 * ((((((float_complex_val
   ((float_principal_root_1 0 k) * (EL 0 Lf)) * complex ((1 + e7) , 0))
   + float_complex_val ((float_principal_root_1 1 k) * (EL 1 Lf)))
   * complex ((1 + e6) , 0)) + float_complex_val ((float_principal_root_1 2 k)
   * (EL 2 Lf))) * complex ((1 + e5) , 0))) * complex (1 + e4,0)))

Y_product_FP_TO_FXP_ERROR =
⊢thm  ∀  Wr u W1_1 W1_2 u_2 u_1 X. ∃ e1 e2.
```

```
   Y_product_FP_TO_FXP_ERROR  Wr u W1_1 W1_2 u_2 u_1 X =
   MAP (\n. (fxp_complex_value (fxp_CNJ X (fxp_FFT X u_1 n)))
   * (fxp_complex_value (EL n W1_1)) + complex (e1 , e1)
   - (float_complex_val (float_CNJ (float_FFT u_2 n)))
   * (float_complex_val (EL n W1_2)) * complex (1 + e2,0)) [0;1;2;3]

ERROR_FP_TO_FXP_ERROR =
⊢thm  ∀  d d1 d2 y y1 y2 X. ∃ e1 e2.
   ERROR_FP_TO_FXP_ERROR d d1 d2 y y1 y2 X =
   MAP (\n. (fxp_complex_value (EL n d1) - fxp_complex_value (EL n y1))
   + complex (e1,e1) - (float_complex_val (EL n d2) -
 float_complex_val (EL n y2))  * complex (1 + e2,0)) [0;1;2;3]

FREQ_PRODUCT_FP_TO_FXP_ERROR =
⊢thm  ∀  u u1 u2 e6 e4 e5 E E1 E2 X. ∃  e1 e2.
   FREQ_PRODUCT_FP_TO_FXP_ERROR u u1 u2 e6 e4 e5 E E1 E2 X =
   MAP (\n. (fxp_complex_value (fxp_CNJ X (fxp_FFT X u1 n)))
   * (fxp_complex_value (EL n (fxp_error_FFT X e4 E1)))
   + complex (e1, e1) - (float_complex_val (float_CNJ (float_FFT u2 n)))
   * (float_complex_val (EL n (float_error_FFT e5 E2)))
   * complex (1 + e2,0)) [0;1;2;3]

PRODUCT_GRADIENT_FP_TO_FXP_ERROR =
⊢thm  ∀  W2 W2_1 W2_2 X. ∃  e1 e2.
   PRODUCT_GRADIENT_FP_TO_FXP_ERROR W2 W2_1 W2_2 X =
   MAP (\n. (fxp_complex_value (fxp_complex_2 X)) * (fxp_complex_value
   ( EL n W2_1)) + complex (e1, e1) - (float_complex_val float_complex_2)
   * (float_complex_val (EL n W2_2))* complex (1 + e2,0)) [0;1;2;3]

COEF_UPDATE_FP_TO_FXP_ERROR =
⊢thm  ∀  W11 W3 W1_1 W3_1 W1_2 W3_2 X. ∃ e1 e2.
   COEF_UPDATE_FP_TO_FXP_ERROR W11 W3 W1_1 W3_1 W1_2 W3_2 X =
   MAP (\n. (fxp_complex_value (EL n W1_1)) + (fxp_complex_value (EL n W3_1))
   + complex (e1, e1) - ((float_complex_val (EL n W1_2))
   + (float_complex_val (EL n W3_2))) *  complex (1+ e2,0)) [0;1;2;3]
```

D Single Theorem for the whole Frequency Domain Equalizer Error Analysis

```
FAST_LMS_FP_TO_FXP_ERROR =
⊢thm  ∀  ! x xp xf Wr u W1_1 W1_2 u_2 u_1 L Lp Lf d d1
   d2 y y1 y2 x_1 xp_1 xf_1 u_0 u1 u2 e_1 e1 e2 E E1 E2 L_1 Lp_1 Lf_1 x_2 xp_2
   xf_2 k W2 W2_1 W2_2 W11 W3 W1_11 W3_1 W1_21 W3_2 d d1 d2 y y1 y2 W11 W3 W1_11
   W3_1 W1_21 W3_2 W2 W2_1 W2_2 k X.
∃  ? e46 e47 e44 e45 e16 e17 e18 e19 e20 e21 e30 e31 e32 e33 e34 e35 e36
   e37  e42 e43 e10 e11 e12 e13 e14 e15 e40 e41 e22 e23 e24 e25 e26 e27 e28 e29
   e38 e39 e7 e8 e9 e4 e5 e6.
  (FAST_LMS_FP_TO_FXP_ERROR x xp xf Wr u W1_1 W1_2 u_2 u_1 L Lp Lf x_1 xp_1 xf_1
```

```
 u_0 u1 u2 e_1 e1 e2 E E1 E2 L_1 Lp_1 Lf_1 x_2 xp_2 xf_2 W11 W3 W1_11 W3_1 W1_21
 W3_2 W2 W2_1 W2_2 d d1 d2 y y1 y2 k X) =
((MAP (\n. (fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 0 k)
(EL 0 xf)) + complex (e9,e9) + fxp_complex_value (fxp_complex_mul X
(fxp_principal_root X 1 k) (EL 1 xf)) + complex (e8,e8) + fxp_complex_value
(fxp_complex_mul X (fxp_principal_root X 2 k) (EL 2 xf)) + complex (e7,e7) -
((float_complex_val (float_principal_root 0 k * EL 0 xp) * complex (1 + e6,0)
+ float_complex_val (float_principal_root 1 k * EL 1 xp)) * complex (1 + e5,0)
+ float_complex_val (float_principal_root 2 k * EL 2 xp)) * complex (1 + e4,0)))
[(0: num); 1; 2; 3]) +
(MAP (\n. (fxp_complex_value (fxp_CNJ X (fxp_FFT X u_1 (&n)))) *
(fxp_complex_value (EL n (W1_1: fxp_complex list))) + complex (e38 , e38)
 - (float_complex_val (float_CNJ (float_FFT u_2 (&n)))) * (float_complex_val
(EL n (W1_2: float_complex list))) * complex (1 + e39,0)) [0;1;2;3]) +
 (MAP (\n. (fxp_complex_value (fxp_complex_4 X) *
 (fxp_complex_value (fxp_complex_mul X (fxp_principal_root_1 X 0 k)
 (EL 0 Lp)) + complex (e25,e25) + fxp_complex_value (fxp_complex_mul X
 (fxp_principal_root_1 X 1 k) (EL 1 Lp)) + complex (e24,e24) + fxp_complex_value
 (fxp_complex_mul X (fxp_principal_root_1 X 2 k) (EL 2 Lp)) + complex (e23,e23))
 + complex (e22,e22) - float_complex_val float_complex_4 *
 (((float_complex_val (float_principal_root_1 0 k * EL 0 Lf) * complex (1 + e29,0) +
 float_complex_val (float_principal_root_1 1 k * EL 1 Lf)) * complex (1 + e28,0) +
 float_complex_val (float_principal_root_1 2 k * EL 2 Lf)) *
 complex (1 + e27,0)) * complex (1 + e26,0))) [(0: num); 1; 2; 3]) +
(MAP (\n. fxp_complex_value (EL n d1) - fxp_complex_value (EL n y1) +
complex (e40,e40) - (float_complex_val (EL n d2) -float_complex_val (EL n y2))
* complex (1 + e41,0))[0; 1; 2; 3])  +
(MAP (\n. ((((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
(EL 0 (xf_1:fxp_complex list)))) + complex (e12 , e12)) + fxp_complex_value
(fxp_complex_mul X (fxp_principal_root X 1 k)  (EL 1 (xf_1:fxp_complex list))))
+ complex (e11, e11)) + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
(EL 2 (xf_1:fxp_complex list)))) + complex (e10, e10)) - (((((float_complex_val
((float_principal_root (&0) k) * (EL 0 (xp_1:float_complex list))) * complex
(((1:real) + e15) , 0)) + float_complex_val ((float_principal_root (&1) k) *
(EL 1(xp_1:float_complex list)))) * complex (((1:real) + e14) , 0)) +
float_complex_val((float_principal_root (&2) k) * (EL 2 (xp_1:float_complex list))))
* complex (((1:real) + e13) , 0)))) [(0: num); 1; 2; 3]) +
(MAP (\n. fxp_complex_value (fxp_CNJ X (fxp_FFT X u1 (& n))) *
fxp_complex_value (EL n (fxp_error_FFT X e1 E1)) + complex (e42,e42) -
float_complex_val (float_CNJ (float_FFT u2 (& n))) *
float_complex_val (EL n (float_error_FFT e2 E2)) * complex (1 + e43,0)) [0; 1; 2; 3]) +
(MAP (\n. (fxp_complex_value (fxp_complex_4 X) * (fxp_complex_value
(fxp_complex_mul X (fxp_principal_root_1 X 0 k) (EL 0 Lp_1)) + complex (e33,e33) +
fxp_complex_value (fxp_complex_mul X (fxp_principal_root_1 X 1 k) (EL 1 Lp_1)) +
complex (e32,e32) + fxp_complex_value (fxp_complex_mul X
(fxp_principal_root_1 X 2 k) (EL 2 Lp_1))  + complex (e31,e31)) + complex (e30,e30) -
(float_principal_root_1 0 k * EL 0 Lf_1) * complex (1 + e37,0) + float_complex_val
(float_principal_root_1 1 k * EL 1 Lf_1)) * complex (1 + e36,0) +
float_complex_val (float_principal_root_1 2 k * EL 2 Lf_1)) *
```

```
complex (1 + e35,0)) * complex (1 + e34,0))) [(0: num); 1; 2; 3]) +
(MAP (\n. ((((((fxp_complex_value ((fxp_complex_mul X (fxp_principal_root X 0 k)
(EL 0 (xf_1:fxp_complex list)))) + complex (e18 , e18)) + fxp_complex_value
(fxp_complex_mul X (fxp_principal_root X 1 k)  (EL 1 (xf_2:fxp_complex list))))
+ complex (e17, e17)) + fxp_complex_value (fxp_complex_mul X (fxp_principal_root X 2 k)
(EL 2 (xf_2:fxp_complex list)))) + complex (e16, e16)) - (((((float_complex_val
((float_principal_root (&0) k) * (EL 0 (xp_2:float_complex list))) *
complex (((1:real) + e21) , 0)) + float_complex_val ((float_principal_root (&1) k)
* (EL 1(xp_2:float_complex list)))) * complex (((1:real) + e20) , 0)) +
float_complex_val((float_principal_root (&2) k) * (EL 2 (xp_2:float_complex list))))
* complex (((1:real) + e19) , 0))) ) [(0: num); 1; 2; 3]) +
( MAP (\n. fxp_complex_value (fxp_complex_2 X) * fxp_complex_value (EL n W2_1) +
complex (e44,e44) - float_complex_val float_complex_2 * float_complex_val (EL n W2_2)
* complex (1 + e45,0)) [0; 1; 2; 3]) +( MAP (\n. fxp_complex_value (EL n W1_11) +
fxp_complex_value (EL n W3_1) + complex (e46,e46) - (float_complex_val (EL n W1_21) +
float_complex_val (EL n W3_2)) * complex (1 + e47,0)) [0; 1; 2; 3]))
```