The Application of Formal Verification to SPW Designs

Behzad Akbarpour and Sofiène Tahar
Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8, Canada
Email:{behzad, tahar}@ece.concordia.ca

Abstract

The Signal Processing WorkSystem (SPW) of Cadence is an integrated framework for developing DSP and communications products. Formal verification is a complementary technique to simulation based on mathematical logic. The HOL system is an environment for interactive theorem proving in a higher-order logic. It has an open user-extensible architecture which makes it suitable for providing proof support for embedded languages. In this paper, we propose an approach to model SPW descriptions at different abstraction levels in HOL based on the shallow embedding technique. This will enable the formal verification of SPW designs which in the past could only be verified partially using conventional simulation techniques. We illustrate this novel application through a simple case study of a Notch filter.

1 Introduction

Simulation-based methods are currently used by the industrial community for system-level verification, since it can handle the entire design at a time. Simulation, however, cannot provide a high coverage ratio due to the exponential number of test cases to be developed and verified. Therefore, new methods are needed for the economical and reliable verification of digital systems. Formal verification [6] has recently paved a path, showing the utility of finding bugs early in the design cycle. Formal verification techniques are usually classified in two categories [6]: interactive theorem proving and automatic decision diagram based model checking and equivalence checking. In model checking, one checks if the design satisfies some properties (formal specification). With equivalence checking, we check if two designs exhibit the same behavior. The latter techniques have been successfully applied to real industrial design. However, since most tools are based on Binary Decision Diagrams (BDDs), they require the design to be described at the Boolean level. In practice, they often fail to verify a large-scale design because of the so-called state space explosion [6].

The overall aim for this paper is to propose the application of formal methods for modeling SPW (Signal Processing WorkSystem) descriptions at different abstraction levels to enable the formal verification of SPW designs in the theorem proving environment HOL. We propose a shallow embedding for SPW descriptions in which we translate the intended meaning of SPW design blocks into HOL and then complete the formal proof in the theorem proving environment.

The HOL theorem prover is an interactive proof assistant for higher order logic, developed by Gordon [3]. It was explicitly designed for the formal verification of hardware, though it has also been applied to other areas including software verification and formalization of pure mathematics. HOL implements a small set of primitive inference rules, and all theorems must be derived using only these rules. This guards against the assertion of false "theorems". However, it is possible to automate the translation of higher-level proof techniques into the low-level primitives. In this way, HOL users can call on an extensive selection of automated tools or write special-purpose inference rules for a given application domain.

In the present work, several features of HOL are particularly significant. The higher-order logic allows circuit modules to be expressed simply as predicates over inputs and outputs, allowing a very natural and direct mapping from the gate level and RTL (Register Transfer Level) descriptions into the logic. In addition, the extensive infrastructure of real analysis is essential to verify (or even state) the highest level of specification [4]. Finally, the adherence to a small set of primitive rules, gives us a high confidence that the final result is indeed valid.

The rest of this paper is organized as follows: Section 2 describes the SPW tool and its different modules and design development steps. Section 3 introduces our modeling and verification methodology. Section 4 is a case study containing the details of the verification of a Notch filter. Finally, Section 5 concludes the paper.

2 The SPW Tool

The Signal Processing WorkSystem (SPW) of Cadence [9] is an integrated framework for developing DSP and communications products. It graphically represents a system as a network of functional blocks and comes with a vast library of DSP blocks and users can also add their own blocks or build IP (Intellectual Property) blocks by composition of primitive blocks. SPW provides all the tools needed to interactively capture, simulate, test, and implement a broad range of DSP designs. Typical design applications include digital communication systems, image processing, multimedia, radar systems, control systems, digital audio, and high-definition television. SPW can be used to evaluate various architectural approaches to a design and to develop, simulate, and fine-tune algorithms.

SPW consists of several modules. The main modules are the File Manager, Block Diagram Editor (BDE), Signal Calculator, and Signal Simulator. The File Manager is a unified tool that lets users create and manage SPW libraries, access all types of SPW data files, and invoke the various SPW tools. The BDE graphically represents a system as a set of functional blocks connected by wires. Each block is a symbol that represents an operation, and the interconnecting wires symbolize the flow of signals between blocks. The Signal Calculator creates input signals for simulation and analyzes the output signals from a design. It can also perform signal calculations, fixed-point and floating-point operations, signal filtering, and analysis functions such as Fast Fourier Transform (FFT) and cross correlation. The Signal Simulator is a tool that simulates the operation of a signal flow system designed with BDE. Given the BDE block diagram and a set of input signals, the simulator determines the output signals of the system over a specified interval. It writes the results into a set of signal files that can be displayed and analyzed in the Signal Calculator.

The Hardware Design System (HDS) is an optional SPW software package which adds fixed-point simulation and hardware synthesis to SPW. The combination of SPW and HDS is a system-level design tool that bridges the gap between system specification and hardware implementation. Using HDS, the designer can accurately model the behavior of a fixed-point, bit-limited digital signal-processing system or logic design and directly generate an optimized VHDL or Verilog HDL (Hardware Description Language) of the design for simulation and synthesis. HDS provides a main library which is a comprehensive set of blocks used for various fixed-point signal processing functions: bit manipulation, clocking, signal flow control, logic functions, mathematical functions, simulation I/O, and vector processing. Using this set of blocks, one can build, simulate, and synthesize a wide range of fixed-point signal-processing algorithms and architectures.

A design project in SPW typically consists of the following steps (Figure 1):

- **Floating-Point Algorithm:** Design and build a high-level system using SPW's standard floating-point library blocks to specify the signal processing algorithm. Then verify the integrity of the algorithm by simulating it as a floating-point system in SPW.
- **Fixed-Point Algorithm:** When the algorithm works as intended, replace the floating-point blocks such as adders, multipliers, and delays with their fixed-point counterparts from the standard HDS block library (either manually or using the Float-to-Fixed-Point conversion utility). Determine the optimum fixed-point attributes and word lengths appropriate for the system. Then run the simulation again to see the quantization and overflow effects.
- **Hardware Architecture:** Create a block diagram using HDS architectural blocks such as shift registers, flip-flops, buffers, multiplexers, and memory elements to specify the architecture of the system. Once again, check the functionality of the design with the SPW simulator.
- **Generation of VHDL Code:** Using the same architectural design that we already simulated, generate a design description in a standard HDL using the Verilog Link or VHDL Link tool. An HDL simulator can be used to verify the integrity of the hardware description and to determine timing effects not modeled by the SPW simulator. The HDL simulation can be directly run from the SPW environment, and the results of both SPW and HDL simulations can be viewed in the Signal Calculator. From the verified HDL description, we can use any of several synthesis tools for implementing the design in hardware, such as data path compilers and logic synthesis tools available from Cadence, Synopsys, and other vendors. The result is a netlist from which we can generate a chip layout.

3 Formal Verification Methodology

We propose a methodology for the specification and verification of the SPW designs at different abstraction levels using higher-order logic (Figure 2). For this purpose, we first specify the design at different abstraction levels such as floating-point (FP), fixed-point (FXP), register transfer (RT), and gate levels, as predicates in higher-order logic. The process of specifying a hardware description language in higher-order logic is commonly known as semantic embedding. There are two main approaches [2]: deep embedding and shallow embedding. In deep embedding, the abstract syntax of a design description is represented by terms,

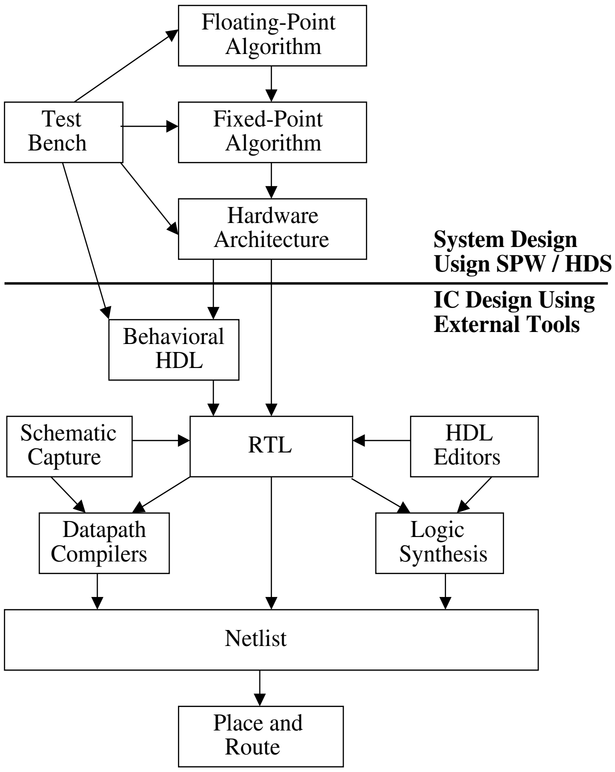

Figure 1. SPW Design Flow

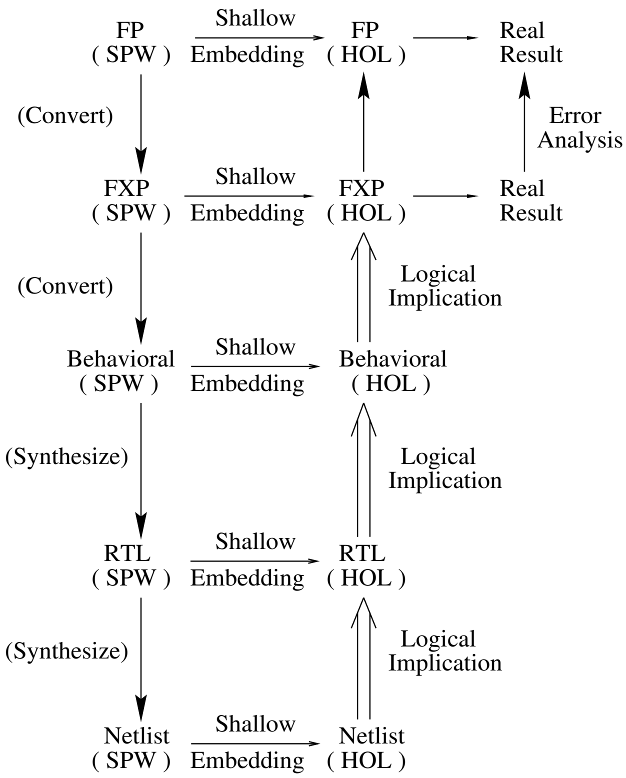

Figure 2. Verification Methodology

which are then interpreted by semantic functions defined in the logic that assign meaning to the design. With this method, it is possible to reason about classes of designs, since one can quantify over the syntactic structures. However, setting up HOL types of abstract syntax and semantic functions can be very tedious. In a shallow embedding on the other hand, the design is modeled directly by a formal specification of its functional behavior. This eliminates the effort of defining abstract syntax and semantic functions, but it also limits the proofs to functional properties. In this project, since our main concern is to check the correctness of designs based on their functionality, we propose shallow embedding for SPW descriptions: translate the intended meaning of SPW design blocks into HOL and then complete the formal proof in HOL theorem prover. The embedding is done once and for all, not on a per design basis because often similar operations are involved in the implementation of different systems.

In the SPW design flow, we first focus on the conversion between floating-point and fixed-point algorithmic levels. All blocks are embedded in HOL through representing the behavior of the blocks by HOL predicates which can be manipulated to verify the correctness of the conversion with respect to a requirement specification in HOL. Floating-point data types are stored in SPW as standard IEEE 64 bit double precision format. In the definition of the floating-point blocks, we used the formalization of the IEEE standard developed in [4]. For fixed-point blocks, we use the formalization of SPW fixed-point arithmetic developed in [1]. For the verification of the transition from floating-point to fixed-point levels, the best approach is to establish an error analysis. When digital signal processing operations are implemented on a computer or with special-purpose hardware, errors and constraints due to finite word length are unavoidable. The main categories of finite register length effects are errors due to quantization of input samples, errors due to roundoff in the arithmetic, constraints on signal levels imposed by the need to prevent overflow, and quantization of system coefficients. These error sources can be considered separately and quantized in higher-order logic.

For example, consider a digital filter specified by the input-output relationship:

$$w_n \;=\; \sum_{k=0}^{M} b_k \, x_{n-k} \;-\; \sum_{k=1}^{N} a_k \, w_{n-k} \qquad (1)$$

where $\{x_n\}$ is the input sequence and $\{w_n\}$ is the output sequence. There are three common sources of errors associated with the filter in (1) namely [8]:

1) Input quantization — caused by the quantization of the input signal into a set of discrete levels.

2) Coefficient accuracy — caused by the fact that the coefficients $\{a_k\}$ and $\{b_k\}$ are realized with finite word length.

3) Round-off accumulation — caused by the accumulation of errors committed at each arithmetic operation because these operations are carried out with only finite bit accuracy.

Therefore, for the digital filter of (1) the actual computed output reference is in general different from $\{w_n\}$. We denote the actual output by $\{y_n\}$ and define:

$$e_n \; = \; y_n \; - \; w_n \qquad (2)$$

as the output error at the n^{th} sample. It is important for the designer and the user of a digital filter to be able to determine some measure of the error e_n. A real number can be represented using a finite number of bits in either the fixed- or the floating-point form [10]. The error introduced in such a representation is different in each case. Consider first the fixed-point format. Suppose a number v which has been normalized so that $\mid v \mid \leq \; 1$ has the binary expansion in two's complement representation as :

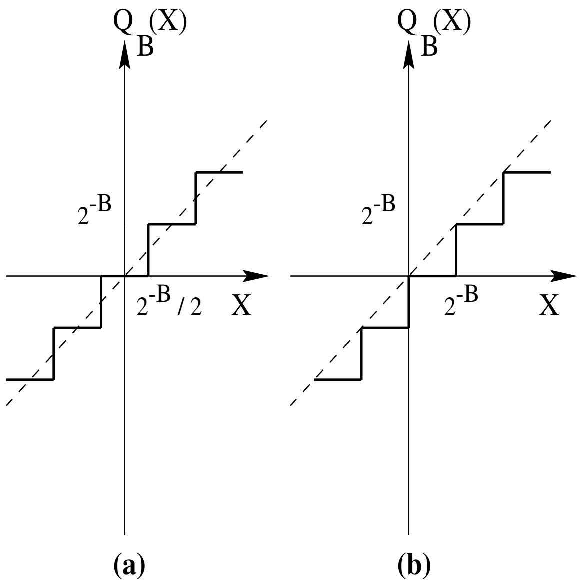

Figure 3. Transfer Characteristics for Rounding and Truncation

$$v \; = \; - \, v_0 \; + \; \sum_{k=1}^{\infty} v_k 2^{-k}, \quad v_k \; = \; 1 \; or \; 0 \qquad (3)$$

The quantity v_0 is referred to as the `sign bit`. If $v_0 \; = \; 0$, then $0 \; \leq \; v \; \leq 1$, and if $v_0 \; = \; 1$ then $-1 \; \leq \; v \; < 0$. An arbitrary real number v would require an infinite number of bits for its exact binary representation. If we use only a finite number of bits `(B + 1)`, then the representation of equation (3) must be modified to:

$$Q_B[v] \; = \; - \, v_0 \; + \; \sum_{k=1}^{B} v_k 2^{-k} \qquad (4)$$

The resulting binary representation is quantized so that the smallest difference between numbers is:

$$\triangle \; = \; 2^{-B} \qquad (5)$$

The operation of quantizing a number to `(B + 1)` bits can be implemented by rounding or by truncation, but in either case quantization is a nonlinear memoryless operation. Figures 3(a) and 3(b) show the input/output relation for two's complement rounding and truncation, respectively.

Considering the effects of quantization, we often define the `quantization error` as:

$$e \; = \; Q_B[v] \; - \; v \qquad (6)$$

for the case of two's-complement rounding, $-\triangle/2 < \; e \; \leq \triangle/2$, and for two's complement truncation, $-\triangle \; < \; e \; \leq \; 0$. In [1], we have proved a theorem in higher-order logic for bounding the error in fixed-point arithmetic operations against their abstract mathematical counterparts. An alternative to fixed-point arithmetic is a floating-point representation. A floating-point number is written in the form $F \; = \; (sgn) \; 2^c.M$, where M, the mantissa, is a fraction between $1/2$ and 1, and c, the characteristic, can be either positive or negative. For the case of floating-point arithmetic, the effect of truncation or rounding is reflected only in the mantissa. It is convenient in the floating-point case to describe the error in a multiplicative rather than in an additive sense as is done in fixed-point arithmetic. In other words, for a floating-point word, if x represents the value before truncation or rounding and `Q(x)` represents the value after, then we express `Q(x)` as equal to $x \; (1 \; + \; \varepsilon)$, and the relative error ε is bounded by $-2^{-p} \; \leq \; \varepsilon \; \leq \; 2^{-p}$ where p is the precision of the floating-point format. This essentially describes the "$1 \; + \; \varepsilon$" property in the error analysis of floating-point arithmetic [5].

DSP algorithms are often tuned by experimenting, e.g., using C-level implementations with IEEE floats. Therefore, for the original verification goal between floating-point and fixed-point levels, we may consider the IEEE floats as the reference for fixed-point implementations and then compute the output error as given in equation (2). A simpler approach is to consider the infinitely precise domain as the reference for both floating-point and fixed-point levels, and then compare the relative errors in each case. With this approach we can treat the uni-directional data flow which usually occurs in the implementation of finite impulse response (FIR) filters. Besides, the method is applicable to bidirectional data flows in infinite impulse response (IIR) structures with at most a fixed number of loops. Larger designs can be treated as a cascade or parallel combination of first and second order terms. Then the total error is computed by accumulating the error in all internal sub-blocks.

After handling the transition from floating-point to fixed-point levels, we turn on to the HDL representation. Using the HDS tool, we generate the corresponding description of the design at the behavioral and/or RT levels and then synthesize the code using, e.g., Synopsys tools to reach the logic gate level netlist. At this point, we can use well known formal techniques to model the design in each of these levels in higher-order logic within the HOL environment. The next step is to verify these different levels using a classical hierarchical proof approach in HOL [7].

Let `X`, `Y` and `Z` be the set of input and output signals and constant parametes corresponding to a typical design. Then, our final goal is to prove the following theorem in HOL:

```
∀ X Y Z ∃ Error. GATE_IMP X Y Z ⟹
FLOAT_SPEC Float (X) Float (Y) Float (Z) ∧
Error (X,Y,Z)
```

`Error` is a general expression to cover all sources of error due to the finite precision effects in the implementation of the design as discussed before. This goal cannot be reached directly, due to the very high abstraction gap between the gate and floating-point algorithmic levels. The proof scheme hence needs to be changed to hierarchically prove that the gate level implies the more abstract RTL. Then this RTL was related, by a formal proof to a modular 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 through the error analysis. This can be formalized as follows in HOL, where `Float` and `Fxp` are data abstraction functions which map binary words to floating-point and fixed-point numbers, respectively:

```
∀ X Y Z. GATE_IMP X Y Z ⟹  RTL_IMP X Y Z

∀ X Y Z. RTL_IMP X Y Z ⟹  BEHAVIORAL_IMP X Y Z

∀ X Y Z. BEHAVIORAL_IMP X Y Z ⟹
         FIXED_IMP  Fxp (X) Fxp (Y) Fxp (Z)

∀ X Y Z ∃ Error. FIXED_IMP Fxp (X) Fxp (Y) Fxp (Z)
⟹  FLOAT_SPEC Float (X) Float (Y) Float (Z) ∧
Error (X,Y,Z)
```

There is usually a high level of regularity and modularity in DSP designs so that primitive blocks such as adders, multipliers, and delays can be used to build larger and more complicated designs such as filters. So, each design can be modeled in HOL as a conjunction of lower level blocks. This will help us in the reuse of the developed models and theories in building the higher levels of specification. Also, the verification goals of large circuits can be broken to the verification proofs of sub-level modules. These proofs are then composed to yield the original goals.

4 Application Case Study

In this section, we demonstrate how the methodology presented in the previous section can be used for the verification of a second order Notch filter (Figure 4) designed in SPW. The filter is first designed and simulated using the floating-point blocks and parameters (Figure 4(a)). The design is composed of `Add`, `Gain` (multiply by a constant), and `Delay` blocks together with signal source and sink elements. Figure 4(b) shows the converted fixed-point design. Fixed-point blocks are shown by double circles and squares to distinguish them from floating-point counterparts. Fixed-point signal values are expressed as a binary stream and a set of attributes. The attributes specify how the binary stream is interpreted. In this example, the attributes of all fixed-point block outputs are set to `<64,31,t>`. This means that we have used sixty four bits to represent the signal values, the numbers are in two's complement format in which the most significant bit is the sign bit, and the binary point is fixed at the thirty first position following the sign bit.

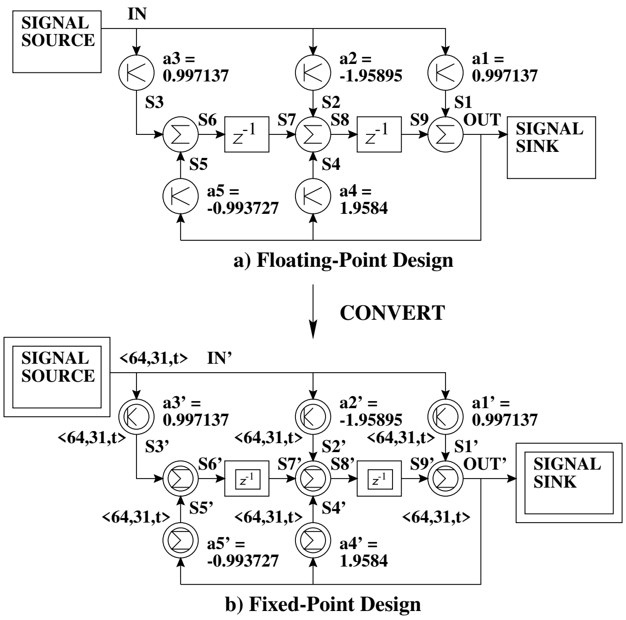

Figure 4. SPW Design of a Second Order Notch Filter

Based on the proposed methodology, we first modeled the design in each level as predicates in higher-order logic.

Primitive blocks are defined using the corresponding functions in floating-point and fixed-point theories in HOL. The whole filter is then implemented as a conjunction of floating-point blocks as follows:

```
⊢def Float_Gain_Block a b c =
     (∀ n. c (n) = a (n) float_mul b)

⊢def Float_Delay_Block a b =
     (∀ n. b (n) = a (n − 1))

⊢def Float_Add_Block a b c =
     (∀ n. c (n) = a (n) float_add b (n))

⊢def Float_3IN_Add_Block a b c d =
     (∀ n. d (n) =
     a (n) float_add b (n) float_add c (n))

⊢def Notch_60_Float_Imp a1 a2 a3 a4 a5 IN OUT =
     ∃ S1 S2 S3 S4 S5 S6 S7 S8 S9.
     (Float_Gain_Block IN a1 S1) ∧
     (Float_Gain_block IN a2 S2) ∧
     (Float_Gain_block IN a3 S3) ∧
     (Float_Add_block S3 S5 S6) ∧
     (Float_Delay_block S6 S7) ∧
     (Float_3IN_Add_block S2 S7 S4 S8) ∧
     (Float_Delay_block S8 S9) ∧
     (Float_Add_block S1 S9 OUT) ∧
     (Float_Gain_block OUT a4 S4) ∧
     (Float_Gain_block OUT a5 S5)
```

A similar description using fixed-point blocks is given as:

```
⊢def FXP_Gain_Block a′ b′ c′ =
     (∀n. c′ (n) = a′ (n) fxp_mul b′)

⊢def FXP_Delay_Block a′ b′ =
     (∀ n. b′ (n) = a′ (n − 1))

⊢def FXP_Add_Block a′ b′ c′ =
     (∀ n. c′ (n) = a′ (n) fxp_add b′ (n))

⊢def FXP_3IN_Add_Block a′ b′ c′ d′ =
     (∀ n. d′ (n) =
     a′ (n) fxp_add b′ (n) fxp_add c′ (n))

⊢def Notch_60_Fxp_Imp a1′ a2′ a3′ a4′ a5′ IN′ OUT′ =
     ∃ S1′ S2′ S3′ S4′ S5′ S6′ S7′ S8′ S9′.
     (Fxp_Gain_Block IN′ a1′ S3′) ∧
     (Fxp_Gain_block IN′ a2′ S2′) ∧
     (Fxp_Gain_block IN′ a3′ S1′) ∧
     (Fxp_Add_block S1′ S5′ S6′) ∧
     (Fxp_Delay_block S6′ S7′) ∧
     (Fxp_3IN_Add_block S2′ S7′ S4′ S8′) ∧
     (Fxp_Delay_block S8′ S9′) ∧
     (Fxp_Add_block S3′ S9′ OUT′) ∧
     (Fxp_Gain_block OUT′ a4′ S4′) ∧
     (Fxp_Gain_block OUT′ a5′ S5′)
```

Separately and independently from the actual implementations, we described the designs as a difference equation relating the input and output samples.

```
⊢def Notch_60_Float_Spec a1 a2 a3 a4 a5 IN OUT =
     ∀n. OUT (n) =
     (IN (n) float_mul a1) float_add
     (IN (n − 1) float_mul a2) float_add
     (IN (n − 2) float_mul a3) float_add
     (OUT (n − 1) float_mul a4) float_add
     (OUT (n − 2) float_mul a5)

⊢def Notch_60_FXP_Spec a1′ a2′ a3′ a4′ a5′ IN′ OUT′ =
     ∀n. OUT′ (n) =
     (IN′ (n) fxp_mul a1′) fxp_add
     (IN′ (n − 1) fxp_mul a2′) fxp_add
     (IN′ (n − 2) fxp_mul a3′) fxp_add
     (OUT′ (n − 1) fxp_mul a4′) fxp_add
     (OUT′ (n − 2) fxp_mul a5′)
```

Next, we proved that the implementation in each case implies the corresponding specification (Theorems 1 and 2).

```
Theorem 1: Notch_60_FLOAT_IMP_TO_SPEC_THM

⊢ Notch_60_Float_Imp a1 a2 a3 a4 a5 IN OUT ⟹
  Notch_60_Float_Spec a1 a2 a3 a4 a5 IN OUT

Theorem 2: Notch_60_FXP_IMP_TO_SPEC_THM

⊢ Notch_60_FXP_Imp a1′ a2′ a3′ a4′ a5′ IN′ OUT′ ⟹
  Notch_60_FXP_Spec a1′ a2′ a3′ a4′ a5′ IN′ OUT′
```

For the error analysis of floating-point and fixed-point arithmetic, we proved the following theorem (Theorem 3) which states the error between the real values of the floating- and fixed-point precision output samples.

```
Theorem 3: NOTCH_60_FXP_TO_FLOAT_THM

⊢ Notch_60_Float_Imp a1 a2 a3 a4 a5 IN OUT ∧
  NOTCH_60_Fxp_Imp a1′ a2′ a3′ a4′ a5′ IN′ OUT′ ∧
  Isvalid IN′ (n) ∧ Isvalid IN′ (n − 1) ∧
  Isvalid IN′ (n − 2) ∧ Isvalid OUT′ (n − 1) ∧
  Isvalid OUT′ (n − 2) ∧ Finite IN (n) ∧
  Finite IN (n − 1) ∧ Finite IN (n − 2) ∧
  Finite OUT (n − 1) ∧ Finite OUT (n − 2)
  ⟹
  Finite OUT (n) ∧ Isvalid OUT′ (n) ∧
  Val (OUT (n)) − value (OUT′ (n)) =
  (Val (IN (n)) * Val (a1) −
  value (IN′ (n)) * value (a1′)) +
  (Val (IN (n − 1)) * Val (a2) −
  value (IN′ (n − 1)) * value (a2′)) +
  (Val (IN (n − 2)) * Val (a3) −
  value (IN′ (n − 2)) * value (a3′)) +
  (Val (OUT (n − 1)) * Val (a4) −
  value (OUT′ (n − 1)) * value (a4′)) +
  (Val (OUT (n − 2)) * Val (a5) −
  value (OUT′ (n − 2)) * value (a5′)) +
  Floaterror (n,a1,a2,,a3,a4,a5,IN,OUT) +
  Fxperror (n,,a1′,a2′,a3′,a4′,,a5′,IN′,OUT′)
```

According to this theorem, for a valid and finite set of input and output sequences at times `(n-1)` and `(n-2)`, we can have finite and valid outputs at time `n`. The difference between the output real values at time `n` is expressed as the difference in input and output values at previous time instances multiplied by the corresponding coefficients, taking into account the effects of finite precision in coefficients and arithmetic operations. The finiteness of floating-point numbers and the validity of fixed-point numbers are checked using the predicates `Finite` [4] and `Isvalid` [1], respectively. The functions `Val` [4] and `value` [1] return the real number values corresponding to floating-point and fixed-point numbers, respectively. The functions `Floaterror` and `Fxperror` represent the errors resulting from rounding the real operation results to a fixed-point and floating-point number, respectively. These errors are already quantified using the theorems mentioned in [1] for fixed-point arithmetic, and the corresponding theorems for error analysis in the floating-point case [4].

Next, we generated with SPW the VHDL code corresponding to the Filter design, and used Synopsys to synthesize the gate level netlist. The resulting codes are then translated into HOL notation and the corresponding correctness theorems established as follows (Theorems 4 and 5):

```
Theorem 4: NOTCH_60_Netlist_TO_RTL_THM

⊢  NOTCH_60_Netlist_Imp
   a1′′ a2′′ a3′′ a4′′ a5′′ IN′′ OUT′′
   ⟹  NOTCH_60_RTL_Imp
   a1′′ a2′′ a3′′ a4′′ a5′′ IN′′ OUT′′

Theorem 5: NOTCH_60_RTL_TO_FXP_THM

⊢  NOTCH_60_RTL_Imp
   a1′′ a2′′ a3′′ a4′′ a5′′ IN′′ OUT′′
   ⟹  NOTCH_60_FXP_Imp
   Fxp (a1′′,64,31,1) Fxp (a2′′,64,31,1)
   Fxp (a3′′,64,31,1) Fxp (a4′′,64,31,1)
   Fxp (a5′′,64,31,1) Fxp (IN′′,64,31,1)
   Fxp (OUT′′,64,31,1)
```

Here the input and output signals `IN''` and `OUT''` are Boolean words. To relate them to the corresponding specifications in fixed- and floating-point, we make use of the bijection functions `Fxp` [1] and `Float` [4], respectively. The fixed-point attributes are set to `<64,31,t>`. In the proof of these theorems we used the modular behavior of the circuit, so that we proved separate lemmas for different modules such as adder, multiplier, and delay and then used these lemmas in the proof of the original theorems.

Finally, using the obtained Theorems 1 to 5, we can easily deduce our ultimate theorem (Theorem 6) proving the correctness of the floating-point specification from the gate level implementation, taking into account the error analysis computed beforehand.

```
Theorem 6: NOTCH_60_Netlist_TO_FLOAT_THM

⊢  NOTCH_60_Netlist_Imp
   a1′′ a2′′ a3′′ a4′′ a5′′ IN′′ OUT′′ ∧
   Isvalid Fxp (IN′′ (n),64,31,1) ∧
   Isvalid Fxp (IN′′ (n − 1),64,31,1) ∧
   Isvalid Fxp (IN′′ (n − 2),64,31,1) ∧
   Isvalid Fxp (OUT′′ (n − 1),64,31,1) ∧
   Isvalid Fxp (OUT′′ (n − 2),64,31,1) ∧
   Finite Float (IN′′ (n)) ∧
   Finite Float (IN′′ (n − 1)) ∧
   Finite Float (IN′′ (n − 2)) ∧
   Finite Float (OUT′′ (n − 1)) ∧
   Finite Float (OUT′′ (n − 2))
   ⟹
   Notch_60_Float_Spec Float (a1′′) Float (a2′′)
   Float (a3′′) Float (a4′′) Float (a5′′) Float (IN′′)
   Float (OUT′′) ∧ Finite Float (OUT′′ (n)) ∧
   Isvalid Fxp (OUT′′ (n),64,31,1) ∧
   Val (Float (OUT′′ (n))) −
   value (Fxp (OUT′′ (n),64,31,1)) =
   (Val (Float (IN′′ (n))) * Val (Float (a1′′)) −
   value (Fxp (IN′′ (n),64,31,1)) *
   value (Fxp (a1′′,64,31,1))) +
   (Val (Float (IN′′ (n − 1))) * Val (Float (a2′′)) −
   value (Fxp (IN′′ (n − 1),64,31,1)) *
   value (Fxp (a2′′,64,31,1))) +
   (Val (Float (IN′′ (n − 2))) * Val (Float (a3′′)) −
   value (Fxp (IN′′ (n − 2),64,31,1)) *
   value (Fxp (a3′′,64,31,1))) +
   (Val (Float (OUT′′ (n − 1))) * Val (Float (a4′′)) −
   value (Fxp (OUT′′ (n − 1),64,31,1)) *
   value (Fxp (a4′′,64,31,1))) +
   (Val (Float (OUT′′ (n − 2))) * Val (Float (a5′′)) −
   value (Fxp (OUT′′ (n − 2),64,31,1)) *
   value (Fxp (a5′′,64,31,1))) +
   Floaterror (n, Float (a1′′), Float (a2′′),
   Float (a3′′), Float (a4′′), Float (a5′′),
   Float (IN′′), Float (OUT′′)) +
   Fxperror (n, Fxp (a1′′,64,31,1),
   Fxp (a2′′,64,31,1), Fxp (a3′′,64,31,1),
   Fxp (a4′′,64,31,1), Fxp (a5′′,64,31,1),
   Fxp (IN′′,64,31,1), Fxp (OUT′′,64,31,1)) +
```

5 Conclusion

In this paper we describe a novel approach for the application of formal specification and verification to SPW designs at different abstraction levels. We provided a shallow embedding of SPW descriptions at different levels in the HOL theorem proving environment. For the verification of the transition from floating-point to fixed-point levels, we propose an error analysis approach in which we consider the effects of finite precision in the implementation of DSP systems. The verification from fixed-point to RTL and netlist levels is performed using hierarchical hardware verification in HOL. We believe this is the first time a complete formal verification framework from floating-point algorithm to netlist implementation is considered, hence opening new avenues in using formal methods for the specification and verification of DSP systems as complement to traditional simulation. We demonstrated our methodology using the example of a second order Notch filter in SPW. In future work, we will intend to extend our case studies to larger industrial designs.

References

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

[2] 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, pages 129-156. North-Holland, 1992.

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

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

[5] J. R. Harrison, "Formal Verification of Floating Point Trigonometric Functions," In Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, Vol. 2152, Springer-Verlag, pp. 217-233, 2000.

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

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

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

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

[10] J. H. Wilkinson, Rounding Errors in Algebraic Processes, Englewood Cliffs, N. J., Prentice-Hall, 1963.