Formal Analysis and Verification of an OFDM Modem Design

Abu Nasser Mohammed Abdullah

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Masters of Applied Science at

Concordia University

Montréal, Québec, Canada

February 2006

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Abu Nasser Mohammed Abdullah**

Entitled: **Formal Analysis and Verification of an OFDM Modem Design**

and submitted in partial fulfilment of the requirements for the degree of

Masters of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____

____________________________________ Dr. Haarslev, Volker

____________________________________ Dr. Ghrayeb, Ali

____________________________________ Dr. Tahar, Sofiène

____________________________________ Dr. Raut, Robin

Approved by ____

Chair of the ECE Department

__________ 2006 ____

Dean of Engineering

ABSTRACT

Formal Analysis and Verification of an OFDM Modem Design

Abu Nasser Mohammed Abdullah, M.A.Sc
Concordia University, 2006

In this thesis we formally specify and verify an implementation of the Orthogonal Frequency Division Multiplexing (OFDM) Physical Layer using theorem proving techniques based on the HOL (Higher Order Logic) system. The thesis is meant to follow a framework, developed at Concordia University, incorporating formal methods in the design flow of digital signal processing systems in a rigorous way. The design under verification is a prototype of IEEE 802.11 Physical Layer implemented using standard Very Large Scale Integration (VLSI) design flow, starting from a floating-point model to the fixed-point and then synthesized and implemented in Field Programmable Gate Array (FPGA) technology. The models were verified in HOL against the IEEE 802.11 specification ratified by the IEEE standardization body and implemented by almost all major wireless industry in the world. The versatile expressive power of HOL helped model the original design at all abstraction levels without affecting its integrity. The thesis also investigates the rounding error accumulated during ideal real to floating-point and fixed-point transitions and also from floating-point to fixed-point numbers at the algorithmic level. On top of the existing theories of HOL, we have built some helping theories, not so trivial, to aid the modeling of the design. The thesis successfully demonstrates the application of formal methods in verification and error analysis of complex telecommunications hardware such as the OFDM modem.

ACKNOWLEDGEMENTS

I would like to thank Professor Sofiene Tahar for his supervision throughout my research. I owe a great deal to him for his immense help in formulating the direction of the work and guiding me towards the end. My fellow researchers in Hardware Verification Group (HVG) of Concordia University had always been helpful and my first stop for asking any question related to research. I want to make special mention of Dr. Akbarpour for his dedicated help to carry out this work; the road behind would have been more rough without his inspiration.

During my stay in Montreal, I came in touch with some people of both head and heart who shared my emotional burden as an international student. I would like to thank Mr. Akhlaque-e-Rasul and Mr. Asif Iqbal and of course their families for compensating the other side of my research life. My friend—Haja on whom I relied on many matters deserve a great thanks for our years of friendship. I also want to thank Raihan for lending me his computer when I needed one.

I know that it is because of the love and prayer of my parents I made it to this far, no expression will be enough to acknowledge them. My uncle Dr. Abdul Mabud has always inspired me throughout my life and he also shares a great deal to my upbringing. Lastly, I want to thank Almighty whom I believe and who has given me so much when I do not deserve an iota of those.

To

My Father,
YOUNUS HALDER

And

My Mother,
SABERA KHATUN

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ACL2	A Computational Logic Applicative Common Lisp
ARQ	Automatic Repeat Request
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
BER	Bit Error Rate
CAD	Computer Aided Design
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DCF	Distributed Coordination Function
DFT	Discrete Fourier Transform
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
DLL	Data Link Layer
DQAM	Demodulation of QAM
DSP	Digital Signal Processing
DSSS	Direct Sequence Spread Spectrum
FEC	Forward Error Correction
FFT	Fast Fourier Transform
FRIDGE	Fixed-point pRogrammIng DesiGn Environment
HDL	Hardware Description Language
HDS	Hardware Design Systems
HOL	Higher Order Logic
IEEE	Inistitute of Electrical and Electronics Engineers
IFT	Inverse Fourier Transform
IFFT	Inverse FFT

IC	Integrated Circuit
IP	Intellectual Property
LAN	Local Area Network
LCF	Logic for Computable Functions
LLL	Logical Link Control
LSB	Least Significant Bit
MAC	Medium Access Control
MAN	Metropolitan Area Network
ML	Meta Language
MP	Modus Ponens
MSB	Most Significant Bit
OFDM	Orthogonal Frequency Division Multiplexing
OSI	Open System Interconnecti
PC	Point Coordinator
PCF	Point Coordination Function
PHY	Physical Layer
PLCP	Physical Layer Convergence Procedure
PMD	Physical Medium Dependent
PRISM	Probabilistic Symbolic Model Checker
PSDU	Physical Service Data Unit
PSK	Phase Shift Keying
PVS	Prototype Verification System
QAM	Quadrature Amplitude Modulation
RAM	Ransom Access Memory
RTL	Register Transfer Level
SMV	Symbolic Model Verifier
SPW	Signal Processing WorkSystem
TCP	Transmission Control Protocol

UDP	User Datagram Protocol
VGA	Video Graphics Adapter
VLSI	Very Large Scale Integration
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WLAN	Wireless LAN
WPAN	Wireless Personal Area Network

Chapter 1

Introduction

1.1 Motivation

Technology has always been the factor of human advancement. And, no other technology has ever created so much impact on every walk of life than the digital technology. From the current fad of *iPoD* video player for hours of entertainment, or gigahertz machine running on the desktop, to the mobile phone equipped with Video Graphics Adapter (VGA) quality video recorder, all are very sophisticated combination of hardware and software giving base to those digital systems. The compact implementation of such systems owe much to the advancement of very large scale integration technology of hardware. The industry is still holding on to the Moore's law and billion transistor processors are now a reality since the semiconductor fabrication moves from the current generation of 90 nanometer processes to the next 65nm and 45nm generations. But, this comes with a price, which is complexity of the system and thus very difficult verification and validation process to ensure bug-free product. Even hardware verified using state of the art simulation technique failed miserably and caused havoc in terms of economics and also in many cases valuable human lives. Examples are [48]: (1) The maiden flight of the Ariane 5 launcher (June 4 1996) ended in an explosion and later it was found that it was caused by an an exception occurred when a large 64-bit floating point number was converted to a 16 bit signed integer that eventually led the failure of the computer. The total loss was over 850 million. (2) Between June 1985 and January 1987, a computer-controlled radiation therapy machine, called the Therac-25, massively overdosed six people, killing two. Later it was found that the software was excluded from the safety analysis and an 1-bit error in the microswitch codes produced an ambiguous position message for the computer thus overdosing the patients. (3) The replacement of defective Pentium processors costs Intel corporation several hundreds of millions of dollars in 1995. (4) The April 30, 1999 loss of a Titan I , which cost 1.23-billion, was due to incorrect software (incorrectly entered roll rate filter constant). (5) The December 1999 loss of the Mars Polar Lander was due to an incomplete software requirement. A landing leg jolt caused engine shutdown of the Lander. (6) The Denver Airports computerized baggage handling system delayed opening by 16 months. And, the list can go longer. These incidents coupled with our increasing reliance on technology at every step dictates a sound and flawless verification methodology in order to produce bug free software and hardware. To add more complexity to the state of the art, the embedded systems and the rapid interfacing between wired and wireless world changing the dimension of the verification domain and pushing it to encompass a very rigorous approach to handle all the systems.

Usually, design verification is done using simulation by generating test cases and then the results are checked to see if they have complied with the desired behavior. But, simulation is inadequate to check all possible inputs of a design even of moderate size and thus leaves the design partially verified. Several approaches have been developed over the last decade or two to accelerate simulation, maximize the test case coverage or investigating alternative or complementary techniques. Formal verification is one such technique which has proved itself as a complement to simulation to achieve a rigorous verification. Although yet to be practiced widely in the industry, it is already applied in many large scale verification projects. Among the three main formal verification techniques theorem proving is particularly powerful for verifying complex systems at higher levels of abstraction.

In this thesis, we use theorem proving techniques to verify an implementation of the IEEE 802.11a [35] physical layer, an OFDM (Orthogonal Frequency Division Multiplexing) modem, implemented by Manavi [42]. In order to verify the implementation, both the design specification and the implementation are modeled in formal logic and then mathematical theorems are proved for correctness. Besides, we carry out a formal error analysis of the OFDM modem in order to analyze the round-off error accumulation while converting from one number domain to the other. Both the formal verification and error analysis used a very well established theorem proving tool called HOL[1] (Higher Order Logic) [23]. They are a direct application of a general methodology proposed by Akbarpour [1] for the formal modeling and verification of DSP (Digital Signal Processing) designs. The results of this thesis demonstrate the functional correctness of the OFDM system and proves the feasibility of applying formal methods for similar systems.

In next sections, we introduce the flow used in industry for DSP design and VLSI (Very Large Scale Integration) implementation. We then focus on the verification issue including a discussion on the difference between simulation and formal verification and an overview of some formal techniques used in hardware verification. Finally, we give an overview of the OFDM concept and modem design to be verified.

[1] Although the word HOL is the abbreviated version of *Higher Order Logic*, we mean the tool itself when we write HOL in this thesis.

1.2 Digital Signal Processing Design Flow

Digital signal processing (DSP) is the study of signals in a digital representation and the processing methods of these signals. The algorithms required for DSP are sometimes performed using specialized microprocessors, which are generally Application Specific Integrated Circuits (ASIC). Any DSP hardware is not bound to a unique hardware configuration therefore its capabilities are extended. The rapid miniaturization of transistor technology has given rise to DSPs that can process increasingly complex tasks. The design flow for DSP is the same as the one shown for generic digital design with a little addition to take care of the limited resources that a DSP can offer. For most DSP systems, the design has to result in a fixed-point implementation. Because, these systems are sensitive to power consumption, chip size and price per device. A typical DSP design flow is depicted in Figure 1.1 [37]. It starts from

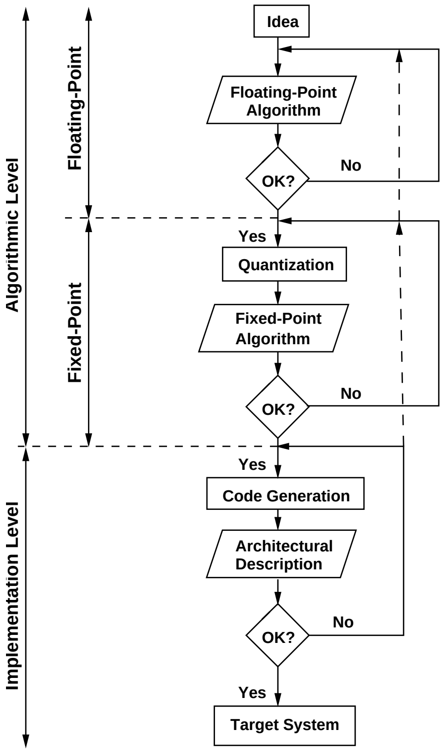

Figure 1.1: DSP Design Flow [37]

an algorithm in ideal real domain translated to a floating-point description and then to fixed-point realization. At every abstraction level, the design is checked whether it complies with the specification. As Figure 1.1 portrays, the conversion between the two domains is not perfect and it is depicted as *quantization* error which needs to be bounded according to the system requirement. The errors occurring in such transition need to be analyzed to demonstrate the robustness of a design and its implementation to ensure its fault free functionality. At the implementation level, the final fixed-point design is realized following standard steps of VLSI. Usually, the VLSI design process follows a very articulated flow (Figure 1.2). It starts from comprehensive system specification, where the system to be realized is explained abstractly with the functionality, interface, and overall architecture. Then a behavioral design is represented using any hardware description language (HDL) like VHDL or Verilog to analyze the functionality, performance, compliance to standard and other high-level issues. This behavioral design is manually converted to describe the data flow that will implement the circuit, which is called Register Transfer Level or RTL. In the next level, the RTL design is converted to gate level netlist, which is

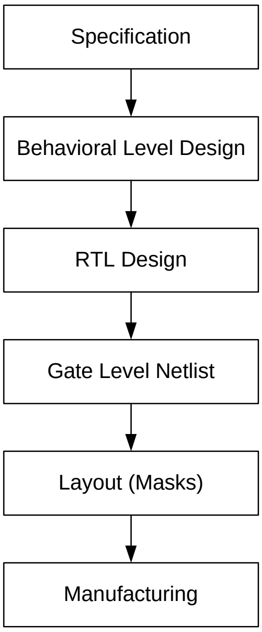

Figure 1.2: Digital Design Flow

a description of the circuit in terms of gates and connections between them. Finally, the netlist is provided as input to a “place and route” tool to generate the layout before manufacturing. Sometimes, the design is implemented on a programmable logic device called FPGA (Field Programmable Gate Array) which is a very efficient prototype platform and can be an alternative to costly production in an integrated circuit (IC) foundry for limited quantities.

1.3 Simulation and Formal Verification

Simulation is the standard verification technique used in industry. Hardware designers are at ease with simulation and it is a practical technique used for this purpose. Simulation is carried out by a design team by generating relatively few test cases, one at a time, and checking whether the results are correct. Towards the end of the design period the circuit is often simulated for an extended period of time. For example, if the design is a microprocessor, a design team runs some reasonably large programs on the simulated design. It is not uncommon to spend months of CPU (Central Processing Unit) time on mainframe computers simulating a final design. Considerable effort has been made to simply increase, in a brute-force manner, the simulation coverage and to cut down the time it takes to achieve this coverage. One approach is to run a number of independent simulations distributing test cases over a set of machines. Another brute-force approach is to design special purpose simulation hardware to increase the speed of a simulation by several orders of magnitude. But, such approach shows a tendency to force and verify a system which is inadequately covered by the simulation data. Theoretically, for example, it will take an impractical amount of time to fully verify even a trivial piece of 256 bit RAM (Random Access Memory) hardware [10], even though the clock speed of the fastest processor now reached 4 GHz and the kind of computation power a huge cluster of those machines can provide. Inherently and continually, at least as of this writing and the trend that is prevailing, simulation will have the following shortcomings:

- Simulation cannot generate perfect input sequences because, it exercises a small fraction of system operations and mostly the patterns are developed manually.

- Long simulation runs are required since effective input sequences are hard to generate

- Input patterns are generally biased towards anticipated sources of errors. Often, the errors occur where not anticipated

- It is sometimes difficult to compare results from different models and simulators

- The number of possible states grows exponentially with increased number of possible event combinations

- As the design grows larger the design team also grows larger and this often gives rise to sources of misunderstandings and inconsistencies

In contrast to simulation, formal verification [38] tries to answer some of the non-exhaustion problems of simulation by proving the correspondence between some abstract specification and the design at hand. But, this statement is no assertion that it can be a complete alternative to simulation. Formal techniques use mathematical reasoning to prove that an implementation satisfies a specification and like a mathematical proof the correctness of a formally verified hardware design holds regardless of input values. All possible cases that can arise in the system are taken care of in formal verification. Moreover, the formal solutions are scalable unlike simulation. There are three main techniques for formal verification: (i) Equivalence Checking, (ii) Model Checking and (iii) Theorem Proving.

Equivalence Checking

Equivalence Checking is used to prove functional equivalence of two design representations modeled at the same or different levels of abstraction [55]. It is divided into two categories: *Combinational Equivalence Checking* and *Sequential Equivalence Checking.* In the Combinational, the functions of the two circuits to be compared are converted into canonical representations of Boolean functions, typically Binary Decision Diagrams (BDDs) or their derivatives, which are then structurally compared. The drawback of this type of verification is that it cannot handle the *Equivalence Checking* between RTL and behavioral models. In *Sequential Equivalence Checking*, given two sequential circuits using the same state encoding, their equivalence can be established by building the product finite state machine and checking whether the values of two corresponding outputs are the same for any initial states of the product machine. It only considers the behavior of the two designs while ignoring their implementation details such as latch mapping and thus verifies the equivalence between RTL and behavioral models. The drawback of this technique is that it cannot handle large designs due to state space explosion.

Model Checking

Model Checking is an algorithm that can be used to determine the validity of formulas written in some temporal logic with respect to a behavioral model of a system. Two general approaches to model checking are used in practice today. The first, *temporal model checking* is a technique developed independently in the 1980s by Clarke and Emerson [5] and by Queille and Sifakis [51]. In this approach specifications are expressed in a temporal logic and systems are modeled as finite state transition systems. An efficient search procedure is used to check if a given finite state transition system is a model for the specification. In the second approach the specification is given as an automaton, then the system also modeled as an automaton is compared to the specification to determine whether or not its behavior conforms to that of the specification. Model Checking tools are effective debugging aids for industrial designs, and since they are fully automated, minimal user effort and knowledge about the underlying technology is required to be able to use them. However, one of the major drawbacks with this approach is the state space explosion as the number of state variables of a system increases. Model checking has been used to verify many IEEE protocols [5].

Theorem Proving

Theorem proving is an interactive technique where both the specification and the implementation are modeled using formal logic. Then a relationship is established between the two as a theorem in mathematics and logical techniques are used to prove that the implementation is equivalent or implying the specification. This mathematical approach answers the limitations of the other two formal verification techniques in terms of state explosion problem by handling designs of any complexity. Theorem provers are highly expressive in nature and are employed for solving problems of various domains. Both first-order and higher-order logic are used to develop theorem provers. The drawback with this approach is that, except some first-order logic provers, it needs human guidance to carry out the proof, and expertise in such act comes through experience. The theorem proving technology is explained in more details in Chapter 3.

1.4 Orthogonal Frequency Division Multiplexing

Orthogonal Frequency Division Multiplexing or OFDM is a modulation technique where data is spread over many channels and transmitted in parallel. Such parallel data transmission method is analyzed for the first time in a paper published in

1967 [54]. In this method, an available bandwidth is divided into several subchannels. These subchannels are independently modulated with different carrier frequencies. It was proved that the use of a large number of narrow channels combats delay and related amplitude distortion in a transmission medium effectively. Based on this concept, OFDM was introduced through a US patent issued in 1970 [17]. The name orthogonal comes from the fact that the subcarriers are orthogonal to each other. Such orthogonality eliminates the need of guard band and the carriers can be placed very close to each other without causing interference and thus conserving bandwidth. The key advantages of this technique are [49]:

- OFDM is an efficient way to deal with multipath; for a given delay spread, the implementation complexity is significantly lower than that of a single carrier system with an equalizer.

- In relatively slow time-varying channels, it is possible to significantly enhance the capacity by adapting the data rate per subcarrier according to the signal-to-noise-ratio of that particular subcarrier.

- OFDM is robust against narrowband interference, because such interference affects only a small percentage of the subcarriers

For such characteristics of OFDM, it is used in many applications—(i) Digital Audio Broadcasting standard in the European market. (ii) ADSL (Asymmetric Digital Subscriber Lline) standard. (iii) In IEEE 802.11a/g standard (iv) Latest WiMAX (Worldwide Interoperability for Microwave Access) technology, etc.

Next, we introduce an implementation of the OFDM technique described above, an OFDM modem [42], which will be the core of all our verification and error analysis work. A more detailed description is given in Chapter 2. The design is implemented in Xilinx Virtex II [61] FPGA. The modem is first modeled in Signal Processing Worksystems (SPW) [15]—a prototype builder, in floating-point and fixed-point format to analyze the performance of different bit sizes and to achieve optimum bit error rate. Then, a library native to the SPW is used to generate VHDL code automatically. The main RTL blocks identified for verification are the quadrature amplitude modulation (QAM) block used for the modulation of the data input; the demapper block to demodulate the received data in the receiver; the serial to parallel and parallel to serial blocks for manipulating data before and after the inputs from other blocks. The core computational blocks—Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) are chosen for error analysis since they are the only two computational blocks in the system. The FFT, IFFT and some of the memory modules of the OFDM implementation at hand are designed using Intellectual Property (IP) blocks, which are ready made parameterized blocks usually optimized for performance; in this case the Xilinx Coregen Library [61] has been used. For this reason, they are not considered for verification in this thesis as no access to the implementation details is provided. The design also implements a synchronizer using SPW, necessary for timing and synchronization of the OFDM system. For the rest of the design the implementation codes are generated manually.

1.5 Related Work

1.5.1 IEEE 802.11 Physical Layer (OFDM) Implementation

There are numerous research work done on the design and implementation of the IEEE 802.11a physical layer. Although no significant work is done about using theorem proving for the verification of the OFDM or part of the system, we still mention some important implementations of OFDM systems. In [19], a coded OFDM system was developed using the TMS320C6201 processor for telemetry applications in the racing and automotive environment. In [56] the authors developed a wireless LAN (Local Area Network) system using the TI C6x platform. A real time software implementation of OFDM modem optimized for software defined radio is implemented in [13]. Software modules representing discrete system blocks are created and sequentially called upon as needed in this implementation. This software reconfigurable system is developed on a TMS320C6201 evaluation module, which is based on a fixed-point processor. The work also explored different combinations of arithmetic precision and speed for the fixed-point operations. In this thesis, we consider the design of [42] described in the section above. Unlike [19], the design under verification is not optimized for telemetry applications and it does not use the coded OFDM technology. The OFDM design in [56] is targeted for a specific platform and used the high level procedural language subroutine provided by the platform extensively; whereas [42] used Xilinx library to implement some high performance computational blocks. The work described in [13] also designed OFDM system but it is optimized specially for software defined radio. Both [19] and [13] used the same processor platform, but [42] has a more generic design that can be accommodated in various applications.

1.5.2 IEEE 802.11 and Formal Methods

There exists a couple of work related to the application of formal methods for the IEEE 802.11. Both use probabilistic model checking but none of them analyzes the design or implementation of the system from the hardware viewpoint. The first one [40] models the two-way handshake mechanism of the IEEE 802.11 standard with a fixed network topology using probabilistic timed automata, a formal description mechanism, in which both nondeterministic and probabilistic choices can be represented. Then from the probabilistic timed automaton model a finite-state Markov decision process is obtained which in turn is verified using PRISM [39], a probabilistic model checking tool. In the second work [53], which identifies ways to increase the scope of application of probabilistic model checking to the 802.11 MAC (Media Access Control). It presents a generalized probabilistic timed automata model optimized through an abstraction technique. Here also the results were verified using

PRISM. In contrast to these related work, we focus completely in different direction. While the first work performs model checking on a IEEE 802.11 network setting and concentrates on the protocol issues, it is concerned more about the upper layers of the OSI (Open System Interconnect) model than the physical layer. The second work also uses model checking to verify the MAC protocol which resides just above the physical layer. In this thesis, we concentrate only on the physical layer and its hardware implementation. Moreover, instead of model checking we use theorem proving techniques based on HOL. The above two work are totally related with the protocol verification and address the verification issues related with the upper layers of OSI model and hence more related with software verification.

1.5.3 Error Analysis and Formal Methods

Previous work on the error analysis in formal verification was done by Harrison [25] who verified floating-point algorithms such as the exponential function against their abstract mathematical counterparts using the HOL Light theorem prover. As the main theorem, he proved that the floating-point exponential function has a correct overflow behavior, and in the absence of overflow the error in the result is bounded to a certain amount. He also reported on an error in the hand proof mostly related to forgetting some special cases in the analysis. This error analysis is very similar to the type of analysis performed for DSP algorithms. The major difference, however, is the use of statistical methods and mean square error analysis for DSP algorithms which is not covered in the error analysis of the mathematical functions used by Harrison. In this method, the error quantities are treated as independent random variables uniformly distributed over a specific interval depending on the type of arithmetic and the rounding mode. Then the error analysis is performed to derive expressions for the variance and mean square error. In another work, Huhn *et al.* [34] proposed a hybrid formal verification method combining different state-of-the-art techniques to guide the complete design flow of imprecisely working arithmetic circuits starting at the algorithmic down to the register transfer level. The usefulness of the method is illustrated with the example of the discrete cosine transform algorithms. In particular, the authors in [34] have shown the use of computer algebra systems like Mathematica or Maple at the algorithmic level to reason about real numbers and to determine certain error bounds for the results of numerical operations. In contrast to [34] and based on the findings from [25], Akbarpour [1] proposed an error analysis technique by realizing the system at different number domains and then subtracting the real number valuations of one domain from the other to get the error in transition from ideal real to fixed-point and floating-point, and then from floating-point to fixed-point. The feasibility of such analysis is also demonstrated by applying the technique for the error analysis of digital filters [3] and 16 point radix 2 FFT [4]. In this thesis, we intend to investigate error analysis in the same way as proposed by [1] but on a larger case study by choosing a combination of FFT-IFFT which is radix-4 and 64 point in computation. Our work proves that the approach in [1] is scalable.

1.6 Thesis Contributions and Organization

This thesis has two main contributions. Although none of the contributions adds to the elementary branch of knowledge but both are significant applications of formal verification techniques in digital design. The first contribution is the successful formal verification of RTL blocks of an OFDM modem implementation using the HOL theorem prover. This work can be seen as an example on how to apply formal methods in the verification of digital communication systems to check its compliance with standard specifications. The second contribution is the formalization of the error analysis of the OFDM modem by analyzing its two computational blocks—FFT and IFFT. A mathematical model of a radix-4 64 point FFT-IFFT combination is developed and extended with floating-point and fixed-point error parameters due to arithmetic operations. Then errors occurring in the transformation from different abstraction levels are also derived mathematically. At the end, the whole analysis is formalized in HOL. These three sub-steps are unique in themselves though. All the works done can be reused as off the shelf verification blocks or theorems for performing similar work.

The rest of the thesis is organized as follows. Chapter 2 provides an introduction to the IEEE 802.11 standards and describes details of the OFDM technique and modem implementation to be verified. In Chapter 3, the HOL theorem prover, its underlying logic, and usage for hardware verification are described. A sample is also explained. Chapter 4 presents the verification of RTL blocks of the OFDM system, one of the cores of the thesis. In Chapter 5, the error analysis of the OFDM modem is mathematically analyzed and then formalized using HOL. The last chapter concludes the thesis and provides hints for future work directions.

Chapter 2

IEEE 802.11 OFDM Modem and Verification Methodology

This chapter describes the issues which will help to understand the rest of the thesis. The focus is mainly on the IEEE 802.11a standard and one of its implementation which we formally verify in this thesis. To illustrate on the topics, a brief description of the IEEE standards is provided followed by the wireless networking architecture supported by it. Since the design at hand is the implementation of the physical layer of IEEE 802.11a, a section is dedicated to this matter also. In the last part of the chapter, we explain the methodology used to model and verify the design.

2.1 IEEE 802.11a Standard

2.1.1 IEEE 802.xx Standard

IEEE 802 refers to a family of IEEE standards about local area networks and metropolitan area networks. More specifically, the IEEE 802 standards are restricted to networks carrying variable-size packets. It follows the OSI or “The Open Systems Interconnection Reference Model” closely although not exactly, which is a layered abstract description for communications and computer network protocol design, developed as part of the Open Systems Interconnect initiative [16]. The services and protocols specified in IEEE 802 map to the lower two layers (Data Link and Physical) of the seven-layer OSI networking reference model. In fact, IEEE 802 splits the OSI Data Link Layer (DLL) into two sub-layers named Logical Link Control (LLC) and Media Access Control (MAC). Figure 2.1 shows the relationship of the layers. The IEEE 802 family of standards is maintained by the IEEE 802 LAN[1]/MAN[2] Standards Committee (LMSC). The most widely used standards are for the Ethernet family, Token Ring, Wireless LAN, Bridging and Virtual Bridged LANs. An individual Working Group provides the focus for each area. Some of the widely known standards are:

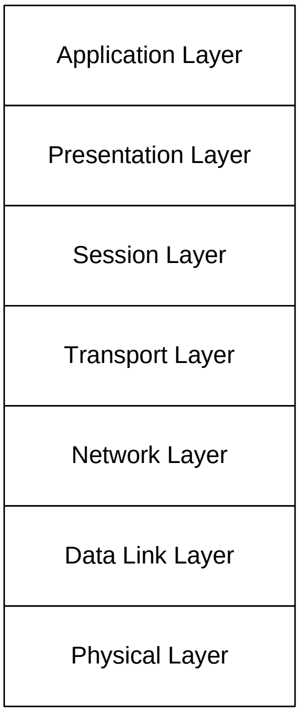

Figure 2.1: OSI Layer Reference Model

- IEEE 802.2, Logical link control

[1] LAN= Local Area Network

[2] MAN= Metropolitan Area Network

- IEEE 802.3, Ethernet
- IEEE 802.5, Token Ring
- IEEE 802.11, Wireless LAN (WLAN)
- IEEE 802.15, Wireless Personal Area Network (WPAN)
- IEEE 802.22, Wireless Regional Area Network

We now describe only IEEE 802.11 standard.

IEEE 802.11, the Wi-Fi standard, denotes a set of WLAN standards developed by working group 11 of the IEEE LAN/MAN Standards Committee (IEEE 802). The term is also used to refer to the original 802.11, which is now sometimes called the the *802.11 legacy*. The original version of the standard IEEE 802.11 released in 1997 specifies two raw data rates of 1 and 2 Mbps to be transmitted via infrared (IR) signals or in the Industrial Scientific Medical frequency band at 2.4 GHz. The original standard also defines Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) as the media access method. A significant percentage of the available raw channel capacity is sacrificed (via the CSMA/CA mechanisms) in order to improve the reliability of data transmissions under diverse and adverse environmental conditions. Then, in 1999 an amendment to the original standard was ratified as 802.11b that has a maximum raw data rate of 11 Mbps and uses the same CSMA/CA media access method. Due to the CSMA/CA protocol overhead, in practice the maximum 802.11b throughput that an application can achieve is about 5.9 Mbps over TCP (Transmission Control Protocol) and 7.1 Mbps over UDP (User Datagram Protocol). A variation of DSSS (Direct-sequence spread spectrum) modulation technique is used in this amendment. The indoor range of 802.11b is 30 m at 11 Mbps and 90 m at 1 Mbps. Another amendment to the original standard was also passed in 1999 which is 802.11a. The 802.11a standard uses the same core protocol as the original standard, operates in 5 GHz band, and uses a 52-subcarrier orthogonal frequency-division multiplexing (OFDM) with a maximum raw data rate of 54 Mbps. The data rate is reduced to 48, 36, 24, 18, 12, 9 then 6 Mbps if required. 802.11a has 12 non-overlapping channels, 8 dedicated to indoor and 4 to point to point. It is not interoperable with 802.11b, except if using equipment that implements both standards. For brevity, we do not discuss more about IEEE 802.11 `x` standards and `x` ranges from `a` upto `w`. Work is under way to reach the data rate of 540 Mbps and two competing proposals from Intel and Philips are currently under consideration by IEEE.

2.1.2 IEEE 802.11a Networking Architecture

As described above, in the 802.11 standard, DLL consists of Logical Link Control (LLC) and Medium Access Control (MAC) sublayers. LLC hides the differences among 802 family members and Ethernet. It makes them indistinguishable as far as the network layer is concerned. MAC determines how to access the medium and send data by doing the required setup for the physical layer (PHY). PHY is dedicated to handle the details of data transmission and reception between two or more stations.

The MAC sublayer for all 802.11 families is common and the differences start to be evident only in PHY. In 802.11, MAC has two modes of operation: Distributed Coordination Function (DCF) and Point Coordination Function (PCF). The DCF is the basic access method of IEEE 802.11 standard. The DCF makes use of a simple CSMA algorithm. The DCF does not include a collision detection function (CSMA/CD) because collision detection is not practical on a wireless network. On the other hand in PCF, Point Coordinator (PC) gives right to stations to send their frames by asking them if they have any frame to send. Since the order of transmission data is completely controlled by a base station in PCF mode, no collision ever occurs. Such nature of access control requires WLAN standard to split the PHY into two generic components: the Physical Layer Convergence Procedure (PLCP) and Physical Medium Dependent (PMD). PLCP maps MAC frames by adding a number of fields to MAC frames, shown in Figure 2.2. It defines a method to make PHY Service Data Unit (PSDU) into a framing format. The format is suitable for transmitting data and management information between two or more stations using the associated PMD system. On the other hand, this is the PMD responsibility to transmit PLCP frames with radio waves through the air. In the other word, PLCP sublayer makes it possible that 802.11 MAC operates with minimum dependency on the PMD sublayer. By dividing the main layers into sublayers the standard makes the architecture of 802.11 MAC independent of PHY. One of the advantages of 802.11 can be highlighted as flexibility and adaptability of this standard while all of its complexity is hidden in its implementation.

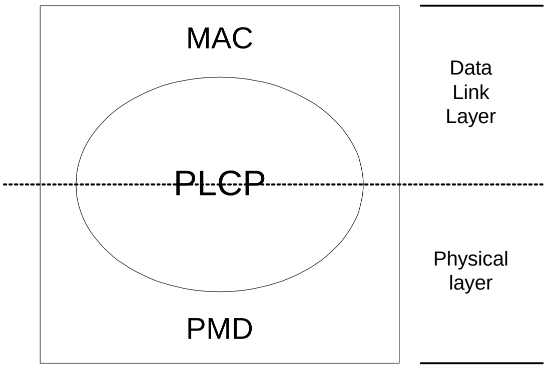

Figure 2.2: Relationship between MAC and PHY Layer

2.1.3 IEEE 802.11a Physical Layer

The PHY of IEEE 802.11 is based on orthogonal frequency division multiplexing (OFDM), a modulation technique that uses multiple carriers to mitigate the effects of multipath. Orthogonality means that the peak of each subcarrier is exactly happening when the other signals have zero amplitude. The advantage of such concept is that if data is multiplexed over a set of orthogonal subcarriers, more subcarriers will be transmitted through the bandwidth. This property increases the bandwidth efficiency of OFDM technique. OFDM distributes the data over a large number of carriers that are spaced apart at precise frequencies. The 802.11a standard supports multiple 20 MHz channels, with each channel being an OFDM modulated signal consisting of 52 carriers. Among the 52 carriers, 48 carry data and 4 carry pilot signals. Each carrier is 312.5 kHz wide and modulated using binary phase shift keying (BPSK) or quaternary phase shift keying (QPSK) or quadrature amplitude modulation (QAM). Instead of separating each of the 52 carriers with a guard band, OFDM overlaps them. But, this could lead to an effect known as intercarrier interference where the data from one carrier cannot be distinguished unambiguously from its adjacent carriers. OFDM avoids this problem because of its orthogonality property and by precisely controlling the relative frequencies and timing of the carriers.

Now, we describe the mathematical model of an OFDM symbol. An OFDM signal consists of a sum of digitally modulated subcarriers transmitted in parallel. In general form we have:

$$s(t) = \sum_{-\infty}^{\infty} s_n(t) \tag{2.1}$$

where $s_n(t)$ is the transmitted signal for the OFDM symbol number n. If this symbol starts at $t = t_s$, then one OFDM symbol is [49]:

$$\begin{aligned} s_n(t) &= Re\left\{ \sum_{i=-\frac{N_s}{2}}^{\frac{N_s}{2}-1} d_{i+\frac{N_s}{2}}\, exp(j2\pi(f_c - \frac{i+0.5}{T})(t-t_s)) \right\}, \quad t_s \le t \le t_s + T \\ s_n(t) &= 0, \quad t < t_s \quad \text{and} \quad t > t_s + T \end{aligned} \tag{2.2}$$

where, d_i is the complex QAM symbol; N_s is the number of subcarriers; f_c is the carrier frequency; T is the OFDM symbol duration; and $Re\{.\}$ denotes the real part of a complex variable. Often the equivalent complex baseband notation is used, which is written as [49],

$$\begin{aligned} s_n(t) \ &= \ Re\left\{\sum_{i=-\frac{N_s}{2}}^{\frac{N_s}{2}-1} d_{i+\frac{N_s}{2}}\, exp(j2\pi\frac{i}{T}(t-t_s))\right\}, \quad t_s \le t \le t_s+T \\ s_n(t) \ &= \ 0, \quad t < t_s \quad \text{and} \quad t > t_s+T \end{aligned} \tag{2.3}$$

The real and imaginary parts of Eq. (2.3) have to be multiplied by a cosine and sine of the desired carrier frequency to produce the final OFDM signal. Because of the orthogonality property, each subcarrier has an integer number of cycles in one OFDM symbol with period T. On the other hand, Eq. (2.3) is the mathematical definitions of inverse discrete Fourier transform for a QAM or BPSK input symbol d_i . At the receiver side, the transmitted jth subcarrier can be extracted by down converting it with a frequency of j/T and then integrating the signal over T seconds. So the QAM value for a particular subcarrier comes from [49]:

$$\begin{aligned} &\int_{t_s}^{t_s+T} exp(-j2\pi\frac{j}{T}(t-t_s)) \sum_{i=-\frac{N_s}{2}}^{\frac{N_s}{2}-1} d_{i+\frac{N_s}{2}}\, exp(j2\pi\frac{i}{T}(t-t_s))dt \\ &= \sum_{i=-\frac{N_s}{2}}^{\frac{N_s}{2}-1} d_{i+\frac{N_s}{2}} \int_{t_s}^{t_s+T} exp(-j2\pi\frac{i-j}{T}(t-t_s))dt \\ &= d_{j+\frac{N_s}{2}T} \end{aligned} \tag{2.4}$$

that gives the desired output $d_{j+\frac{N_s}{2}T}$ multiplied by a constant factor T. Since the frequency difference $\frac{i-j}{T}$ is an integer number of cycles within the integration interval T, the integration result is always zero except for $i = k$. Eq. (2.4) is the mathematical definition of the Fourier transform of $s_n(t)$. We do not show the equations related with windowing, guard insertion, synchronization.

2.2 An Implementation of OFDM

A standard block diagram implementation of OFDM is shown in Figure 2.3. For the verification purpose of the thesis we follow the implementation done by [42].

The block diagram shows the approach taken to implement the concept described above. The discussion below focuses only on the specific implementation at hand since it is more relevant to the verification to follow in the next chapter. The VHDL implementation details are postponed till the next chapter to help readers of this thesis understand the step by step approach taken to verify the blocks shown in the above design.

The first block is the random data generator, which is shown here merely for completion purpose. For the simualtion, a test bench is used in its place. The next block is quadrature amplitude modulation block (QAM). In fact, it can be any digital modulation block as stated in the 802.11a standard. For the specific implementation, 64-QAM is used. The block gives two outputs in real and imaginary format in 16 bit 2's complement, which are stored in a Dual Port RAM to use as input in the *inverse fast fourier transform* (IFFT) block. The real and imaginary components of mapped symbols are grouped in a vector of 48 words. The next block is serial to parallel block and it can also be found in the receiver side of the block diagram. All the serialized data from the QAM block is converted into parallel stream in this

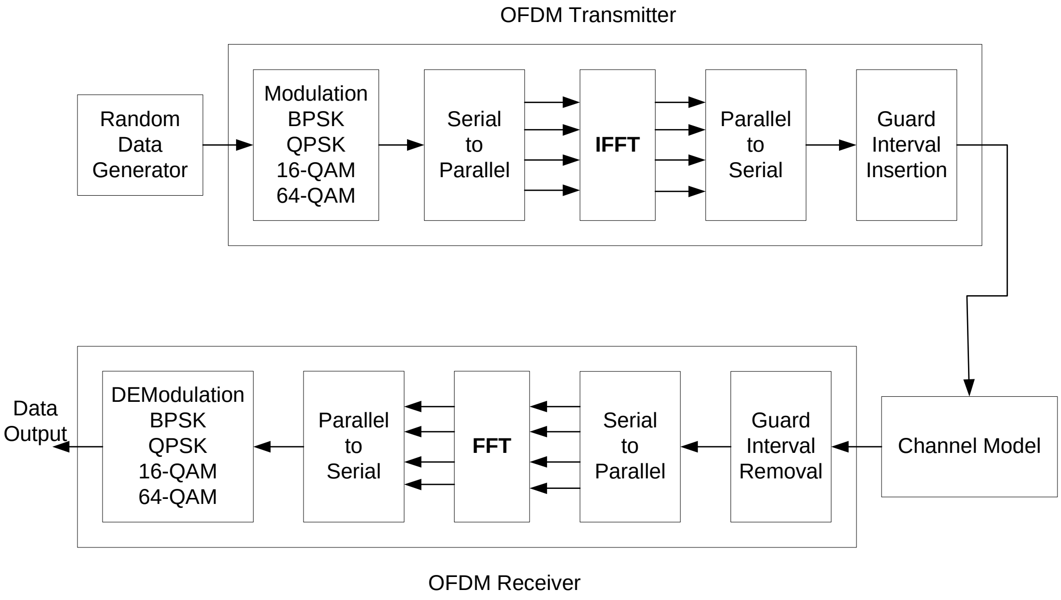

Figure 2.3: OFDM Block Diagram [42]

block to act as an input to the IFFT block and in the receiver side this also does the same after the received signal is stripped from the guards inserted in the transmitter side. This block is designed as a shift register in the implementation. But, it appears from the implementation that this block is stitched with the main design right before the QAM mapper, and this does not affect the overall design at all since the main idea is to give parallel input to FFT and IFFT and this is done systematically since RAMs are used to store the data. The next block is the IFFT block, one of the most important blocks of OFDM. The design uses a 64-point complex IFFT core from Xilinx Coregen Library. The input and output samples are vectors of 64 complex values represented as 16-bit 2's complement numbers for the chosen core. The fourier transform engine employs Cooley-Tukey radix-4 decimation in frequency IFFT. The core does not come with memory space but can be configured for *Single Memory Space(SMS)*, *Dual Memory Space(DMS)* and *Triple Memory Space(TMS)* for different performance requirement. For this implementation, *DMS* configuration is used which allows input, computation and output operations to be overlapped.

Although the FFT block comes later in the diagram but it can be explained with IFFT since it is the same IP core that is used for the implementation by adjusting a signal named *FWD_INV* that controls what kind of computation will take place– IFFT or FFT. The core has a latency of 17 clock cycles for the first 64 data, but later the result vectors appear at every 192 clock cycles. The clock speed for the core is 72 MHz and thus can compute the IFFT result within 3.2μs. The parallel to serial circuitry makes the next block. It does exactly the opposite of serial to parallel. But, this is also implemented in the design right after the serial to parallel block. The reason is, when the data is coming as input in random fashion in the system, this is latched in order to have a sizeable data to send to the system, so it is parallelized using the first block, but then the same data is needed by QAM mapper serially, so the parallel to serial block is used for this purpose. It can be said that the implementation in hand does not follow the block diagram for these two blocks since it relies on the RAMs for the parallel output of data when it is required, and it is no violation of standard but a designer's choice. The next block in the line is the cyclic extension or guard interval insertion circuitry. It is sued to eliminate the intercarrier interference. The OFDM symbol is cyclically extended in the guard interval. This ensures that delayed replicas of the OFDM symbol always have an integer number of cycles within the FFT interval, as long as the delay is smaller than the guard interval. This is implemented using three counters and two Dual Port RAM. The output of IFFT is fed into one of the RAM blocks and the counters controlled where the data to be stored. According to the implementation, 16 symbols are inserted into one OFDM symbol that creates a guard time equal to 800 ns. In the receiver side, the first block is guard interval removal block. It is implemented in the same way as the preceding block except that the role is now reversed and the functionality of the counter is changed. We move now to the QAM demapper (DQAM) block since we discussed the other blocks before. In the DQAM section, the intelligence of the signal is mapped back to its original form according to the 64-QAM demapping table. Although it is not as straightforward as QAM, but it is implemented exactly as the QAM using combinational logic except that the mapping focuses more towards range rather than exact bit to bit mapping as done before. This parallel to serial converter comes right after the mapping block and then the data is serialized again and the output is received sequentially.

The design flow choosen for the OFDM modem implementation under study started from the floating-point modeling. The FP model is prepared based on the mathematical model. This helped to explore and compare the performance of different algorithm and schemes. System modification and optimization is also done at this step of design. For this OFDM modem design, the environment used for floating-point modeling is the Signal Processing Worksystem (SPW) [15] from Cadence [11].

All libraries necessary for modeling and simulating an OFDM system exist in SPW. The second step in the design flow was fixed-point modeling and simulation. The environment used for this purpose is the Hardware Design System (HDS), which is a set of libraries from SPW. The blocks inside the FP model are replaced with the libraries from HDS with the execution of a command. A number of bits are assigned to each block in order to minimize calculation errors. The errors happened due to floating-point to fixed-point conversion and this is formally analyzed in a later chapter and forms one of the contributions of the thesis. The bit error rate (BER) curve of the two models are compared to find the optimum number of bits for the system. Then the design blocks of the floating-point model are replaced with HDS libraies and VHDL codes are generated automatically for the whole system using HDS also. But, for some blocks like FFT/IFFT there was no HDS counterpart and those were imported from Xilinx Coregen Library [61]. Some of the VHDL codes were prepared manually. After VHDL code generation, these are synthesized in Synopsys Design Compiler targeting FPGA as the hardware for implementation. Finally, the synthesized circuitry is mapped into FPGA using “Place and Route” technique and a bit file is generated.

The VHDL code which were manually prepared for the design and thus available for formal verification are for these blocks– QAM, DQAM, parallel to serial, serial to parallel, guard insertion and removal block. But, for FFT/IFFT, RAMs and multiplexers, the blocks are generated using Coregen Library and does not have RTL codes available due to proprietary issues imposed by Xilinx. The latter part inhibited the scope of verification to a certain degree.

2.3 Verification Methodology

The formal specification, verification and error analysis used in this thesis is adopted from DSP verification framework proposed by Akbarpour [1]. The proposal advocated for rigorous application of formal methods to the design flow of the DSP systems. The commutating diagram shown in Figure 2.4 demonstrates the basic idea of the framework. The methodology proposes that the ideal real specification of the DSP algorithms and the corresponding floating-point (FP) and fixed-point (FXP) representations be modeled as the RTL and gate level implementations in higher order logic. The overall methodology for the formal specification and verification of the DSP algorithms will be based on the idea of shallow embedding [6] of languages using the HOL theorem proving environment. In shallow embedding only the logical formulas are embedded directly in the language of the tool in contrast to deep embedding where the logical formulas are embedded as datatype. The latter

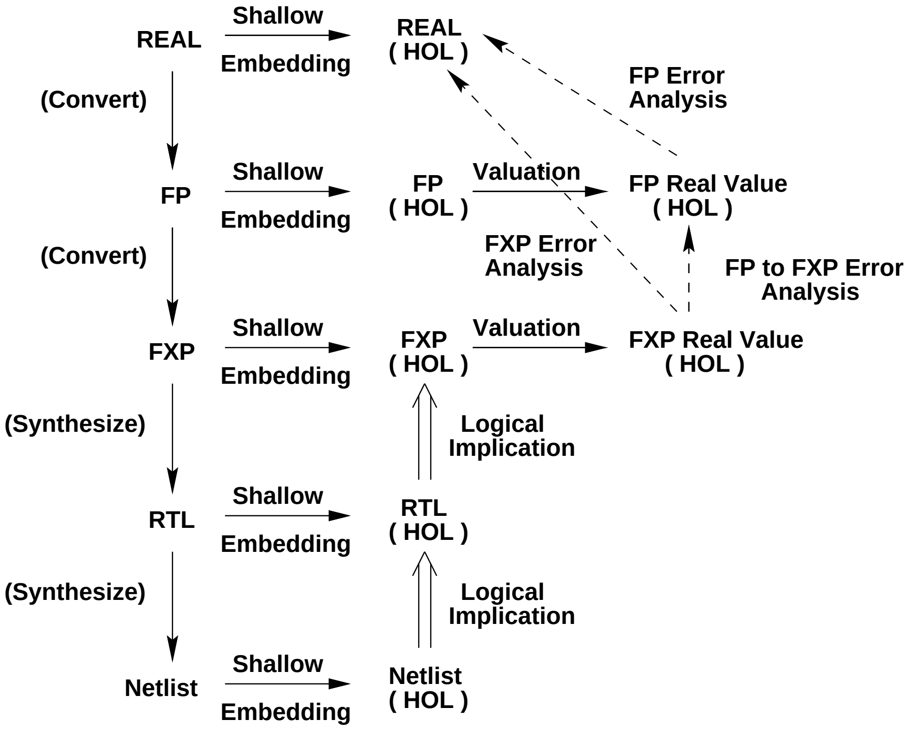

Figure 2.4: A DSP Specification and Verification Approach [1]

is more powerful but time consuming and theorems about the embedded design can be proved. While in shallow embedding, only theorems in the embedding is provable. But, the former suffices the purpose of the verification work we purport to do because, we make use of the existing mathematical theorems of the tool and all the necessary reasoning about them are already built-in in the system. For the thesis, we follow the methodology partially by making use of the first three levels of the diagram which are related with the error analysis.

For formal verification of RTL blocks, we model the QAM, QAM demapper, serial to parallel and parallel to serial blocks using HOL logic. There are ample theories to model standard VHDL design in formal logic [23]. In all blocks, the functionality of the designs are preserved while embedding formally. Then a specification of the design is selected either from the IEEE 802.11a standard or from any existing model for the generic design like serial to parallel. The specifications for all the blocks under verification are also embedded in HOL. Having both the specification and implementation embedded in the tool, we set a relationship between the specification and corresponding implementation as a mathematical theorem. This theorem is then proved by the logical techniques, which are functions written in higher order logic, in the tool. After proving such theorems for rest of the RTL blocks in the design, the whole formalization is saved as a theory which can be reused for any other verification that involves this system.

For the error analysis, we use existing theories in HOL pertaining to the construction of real and complex numbers [28, 24] and model the design in ideal natural number and real number, respectively. For the floating-point implementation of the same design, formalization of IEEE 754 standard based floating-point arithmetic [26] is used. For the fixed-point design, a fixed-point arithmetic formalization developed by Akbarpour et. al. [2] is used, which we extended to model some other functions required for the formalization of the design. Next, the valuation functions are used to find the real values of the floating-point and fixed-point outputs. At this point the error is defined as the difference between the values obtained using valuation function and the ideal real specification. We establish initial fundamental theorems, often called lemmas, on the error analysis of floating-point and fixed-point roundings and arithmetic operations against their abstract mathematical counterparts—the ideal domain. Finally, based on these lemmas, expressions for the accumulation of round-off error is derived. We carry the error analysis at the algorithmic level by directly formalizing the mathematical model. It would have been a better choice to apply the whole framework for the design at hand. But, part of the OFDM design which has used IP block from Xilinx does not have any RTL code provided by the vendor. The behavioral code of the IP block is too deep and thus too impractical to be embedded in HOL for analysis. More on these are described in the related chapters.

It is important to point out that there are errors in digital communication systems in transmission and receiving signals, which are quantified using various parameters like bit error rate (BER), signal to noise ratio and other parameters. There are techniques like forward error correction (FEC) [30] or automatic repeat request (ARQ) [30] to tackle such problem. But, none of these issues are related to the kind of error analysis we present here. Our focus is solely related to hardware and we only concentrate on the accumulation of round-off error that does not have any relationship with the communication error that occurs when the device is implemented and operated thereafter.

Chapter 3

HOL Theorem Proving

Probably it is the extreme philosophical reductionism that says anything in the world can be reduced to physics and mathematical modeling, which in itself can be reduced to a small number of axioms and which can be finally reduced to one formula [8]. Such assertion can be a very difficult theorem to be proved but there are many real life problems which can be stated in terms of mathematical formulas and the branch of knowledge that deals with this more solvable natured problems, unlike the one stated in the maiden sentence, is called formal methods. Theorem proving is one such technique used for formally specifying and verifying many systems. It is in fact a man-machine collaboration for proving mathematical theorems by computer program. Both hardware and software systems have seen large amount of theorem proving use besides other popular formal method techniques.

In hardware verification theorem proving is the only technique where any system that can be modeled using it can be of any size and still the logic can lead to a proof unlike other formal techniques as model checking that always have the problem of state space explosion if the problem space is too large. Theorem provers are highly expressive and work best for verifying hierarchical systems. The basic idea is to model an implementation of a system using formal logic, be it propositional, first order or higher order whichever is applicable, then the desired specification of the system is also formalized in the same logic. The relationship between specification and implementation is then stated as a mathematical theorem to be proven interactively using the tool. Regarding proof interactivity, it is to be noted that propositional logic is fully automated; first order logic is automated but not necessarily terminating; but higher order logic is mainly interactive. That is why there is no single theorem proving system to do all possible kind of verification without human intervention. There are hybrid theorem proving systems [58] which use model checking as an inference rule.The proof system generally consists of a set of axioms and inference rules and new theorems are built on top of these to ensure the soundness of the new theorems. Sometimes provers were written to prove a particular theorem, with a proof that if the program finishes with a certain result, then the theorem is true; e.g. *four color theorem* [21], a controversial solution though since the validity of the proof cannot be verified by hand due to the sheer size of the calculation. There are many theorem provers available but only few of them are in constant development and have large user base. Automated provers for first-order logic include 3TAP, ft, Gandalf, LeanTAP, METEOR, Otter, SATURATE and SETHEO [27]. Among the interactive higher-order logic based ones, we cite Coq, HOL, HOL Light, Isabelle, LEGO, Nuprl and ProofPower are known [27]. Some other known first-order and higher order provers are—ACL2, ALF, EVES, FOL/GETFOL, IMPS, KIV, Lambda Prolog, LARCH, Metamath, MIZAR, MuRal, NQTHM, OBJ3, OSHL, PVS and TPS [27]. Among all these provers HOL, HOL Light, Isabelle, PVS, ACL2 and MIZAR are most widely used. A comprehensive list of theorem provers besides other computer math systems is provided in [57]. In this thesis, we use HOL for all our verification work of OFDM. The reason for choosing HOL is due to the existence of a large amount of theorems about real number theory, floating-point, fixed-point and a comprehensive choice of logical reasoning to carry out the proof procedures. Moreover, some earlier error analysis works, as stated in related work section, used HOL and proved its effectiveness to carry out such analysis with the tool. In the next sections, we describe HOL, the logic on which it is based and its usage as a formal hardware verification tool.

3.1 Higher-Order Logic and HOL

The typed λ calculus [31] provides the theoretical foundation of higher order logic. The λ calculus is a logic that has propositions, models and a way of assigning truth values to each proposition to each model. It also has type expressions and terms and ways of assigning a meaning to each type expression and each term in each model. Higher order logic is derived from the typed λ calculus by selecting distinguished symbols and restricting the set of symbols so that each distinguished symbol is guaranteed to have a certain standard meaning. This logic is more powerful than first or second order because first order logic can only quantify over individuals, e.g., $\forall x, y.\, R(x, y) \rightarrow R(y, x)$; and second order can quantify over predicates and functions, e.g., $P \wedge Q \equiv \forall R.\, (P \rightarrow Q \rightarrow R) \rightarrow R$; whereas higher order logic can quantify over arbitrary functions and predicates. Since arguments and results of predicates and functions in higher order logic themselves be predicates or functions, this imparts a first-class status to functions, and allows them to be manipulated just like ordinary values. For example, a mathematical induction like this – $\forall P.\, [P(0) \wedge (\forall n.\, P(n) \rightarrow P(n+1))] \rightarrow \forall n.\, P(n)$, is impossible to express in first order logic. Any proposition of first order logic can be translated into a proposition of higher order logic, but the reverse does not hold. Higher order logic has, however, some disadvantages: (1) incompleteness of a sound proof system for most higher-order logics; (2) there is no complete deduction system for the second-order logic; (3) reasoning is more difficult in higher order than in first order logic; (4) need ingenious inference rules and heuristics; (5) inconsistencies can arise in higher-order systems if semantics not carefully defined, e.g. *Russell Paradox* [47].

The primary interface to HOL is the functional programming language ML—Meta Language [50]. HOL system can be used for directly proving theorems and also as embedded theorem proving support for application specific verification systems. The tool follows the LCF (logic for computable functions) approach of mechanizing formal proof that is due to Robin Milner [22]. LCF was intended for interactive automated reasoning about higher order recursively defined functions. The interface of the logic to the meta-language was made explicit, using the type structure of ML, with the intention that other logics eventually be tried in place of the original logic one. The HOL system is a direct descendant of LCF and this is reflected in everything from its structure and outlook to its incorporation of ML, and even parts of its implementation. Thus HOL satisfies the early plan to apply the LCF methodology to other logics [23]. The original version of HOL is called HOL88 and it has evolved to its current version HOL4 through HOL90 and HOL98. HOL88 used its own implementation of ML on top of Common Lisp while HOL4 used Moscow ML– an implementation of Standard ML (SML) [50]. HOL's logic, like λ calculus, has only four kinds of terms:

- **Variables.** These are sequences of letters or digits beginning with a letter, e.g., x, a, *hol_is_good*.
- **Constants.** These have the same syntax as variables, but stand for fixed values. Whether an identifier is a variable or a constant is determined by a theory; e.g., **T**, **F**.
- **Applications.** This represents the evaluation of a function f at an argument x; any term may be used in place of x and f.

The following concepts are fundamental to the construction of HOL theorem prover using higher-order logic. Although there are many other complex theoretical basis on which the the tool is built, we only mention the ones which will frequently be encountered in the next chapters:

- **Abstractions**

  HOL provides λ terms, also called λ abstractions for denoting functions. Such a term has the form $\lambda x.\, y$ and denotes the function f defined by: $f(x) = y$. The variable x and term t are called respectively the bound variable and body of the λ expression $\lambda x.\, y$. An occurrence of the bound variable in the body is called a bound occurrence. If an occurrence is not bound it is called free.

- **Types**

  According to the augmentation of λ-calculus by Church [12] with a theory of types, every HOL term has a unique *type* which is either one of the basic types or the result of applying a type constructor to other types. Types are expressions that denote the sets of values, they are either atomic or compound. Examples of atomic types are: *bool, ind, num, real*; where these denote the sets of booleans, individuals, natural numbers and real numbers respectively. Compound types are built from atomic types using *type operators.* For example, if σ, σ_1 and σ_2 are types then so are: σ list and $\sigma_1 \rightarrow \sigma_2$, where *list* is an unary operator and → is an infixed binary type operator. Each variable and constant in a HOL term must be assigned a type. Variables with the same name but different types are regarded as different. If x has a type σ then it is written as $x : \sigma$. Explicit expression of types of variables can be omitted if they can be inferred from the context using type inference algorithm.

- **Inference Rules**

  These are procedures for deriving new theorems and represented as functions in ML. There are eight primitive inference rules, all other rules are derived from these and the axioms. The rules are—(1) Assumption introduction; (2)

Reflexivity; (3) Beta-conversion; (4) Substitution; (5) Abstraction; (6) Type instantiation; (7) Dischanrging an assumption (8) Modus Ponens

- **Theorems**

  A theorem is a sequent that is either an axiom or follows from theorems by a rule of inference. A sequent (Γ, t) consists of a set finite of boolean terms Γ called the assumptions together with a boolean term t called the conclusion. If (Γ, t) is a theorem then it is written as $\Gamma \vdash t$.

- **Theories**

  A theory consists of a set of types, type operators, constants, definitions, axioms and theorems. It contains an explicit list of theorems that have already been proved from the axioms and definitions. Theories can have other theories as parents; if `th1` is a parent of `th2` then all of the types, constants, definitions, axioms and theorems of `th1` are available for use in `th2`. Any theory can be extended by means of constant specification and type specification. For example, the most basic theory is `bool` and this has a descendant theory `ind` that introduces the type *ind*. There are many other theories established in HOL to reason about numbers, pairs, lists, words, bits, probabilities etc.

- **Proofs**

  A theorem is the last element of a *proof*. The only way to create theorems is by generating a proof. In HOL, this consists of applying ML functions representing rules of inference to axioms or previously generated theorems. A proof of a sequent (Γ, t) from a set of sequents Δ is defined to be a chain of sequents culminating in Δ such that every element of the chain either belongs to Δ or else follows from Δ and earlier elements of the chain by deduction. There are two types of proof approach, goal directed and forward proof. We use goal directed proof for all the theorems proved in this thesis work. Forward proof starts from the primitive inference rules and tries to prove the goal building theorems on top of these rules. But, forward proof is too low level and the notion of *tactic* is used. A tactic is a function that splits a 'goal' into 'subgoals' and keeps track of the reason why solving the subgoals will solve the goal.

- **HOL Notations**

  The HOL notations as used in the thesis are summarized in Table 3.1. The first column shows the terminologies used in standard logic paradigm; the second and third column gives the corresponding notations in HOL and standard format followed in literature, respectively. The last column describes the notations.

Kind of Term	HOL Notation	Standard Notation	Description
Truth	`T`	$\top$	*true*
False	`F`	$\perp$	*false*
Negation	`~t`	$\neg t$	*not t*
Disjunction	`t1 \/ t2`	$t1 \vee t2t$	*t1 or t2*
Conjunction	`t1 /\ t2`	$t1 \wedge t2t$	*t1 and t2*
Implication	`t1==>t2`	$t1 \Rightarrow t2$	*t1 implies t2*
Equality	`t1=t2`	$t1 = t2$	*t1 equals t2*
$\forall$-quantification	`!x.t`	$\forall x.\,t$	*for all x : t*
$\exists$-quantification	`?x.t`	$\exists x.\,t$	*for some x : t*
ϵ-term	`@x.t`	$\epsilon x.\,t$	*an x such that : t*

Table 3.1: Terms of the HOL Logic

In the rest of the thesis we have used some notations to pretty print the definitions and theorems. Any definition written in HOL is showed with a *vertical dash* and the word *def* written below it. For example, a simple definition to define the associativity that $a + b$ is equal to $b + a$ with a name MY_DEF is written as

MY_DEF $= \vdash_{def}$ $\forall$a b. a + b $=$ b + a

As we proceed with building the model of any system, eventually we prove theorems on them to establish the relationship between specification and implementation. We write all these theorems using only *vertical dash*. If we prove that, for all a, b and c, $a * b * c$ is equal to $b * c * a$ and save it as MY_THM, then this is shown as

MY_THM $= \vdash$ $\forall$a b c. a * b * c $=$ b * c * a

3.2 Hardware Verification in HOL

To verify a hardware design in HOL, at first the specification and implementation of the hardware is modeled then a goal is set to prove that the implementation meets the specification. An example can be an aid to understand how HOL is used to do such verification. For example, we take a simple circuit in Figure 3.1 and specify its intended behavior and structure as a predicate that relates the inputs to the outputs. The basic idea is that the predicate forms a constraint on the inputs and outputs. It is only true when desired relationship between the inputs and outputs is met. The following HOL term can be used to specify such relationship for the above circuit:

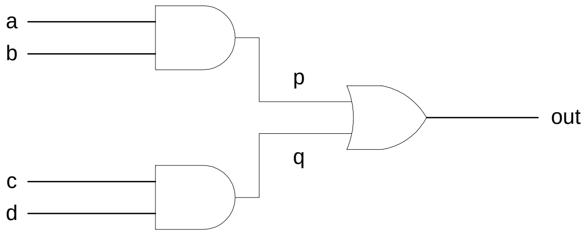

Figure 3.1: A Simple Boolean Circuit

```
⊢def  ∀ a b c d out.
               BEH(a,b,c,d,out) =
                    out = (a ∧ b) ∨ (c ∧ d)
```

The above code shows that specification BEH takes four arguments. In HOL the nesting can be avoided According to the diagram, this circuit has four inputs, one output, and two internal signals *p* and *q*. The implementation uses two AND gates and one OR gate. Presuming that predicates describing the behavior of a two input AND gate and two input OR gate exist, it can be used to create a description of the implementation, IMP. The following are the specifications for AND and OR gate.

```
|- ∀ a b c.
            AND(a,b,c) =
                 c = (a ∧ b)

|- ∀ a b c.
            OR(a,b,c) =
                 c = (a ∨ b)
```

Now, we need a predicate that constraints the inputs and outputs according to how the circuit is structured. It can be confirmed that *AND(a,b,p)* represents the constraints that we would like the AND gate to place on those lines. Extending that concept, we can have a predicate like this: *AND(a,b,p)* ∧ *AND(c,d,q)* ∧ *OR(p,q,out)*. A definition then can be written in HOL as,

```
|- ∀ a b c d out.
            IMP(a,b,c,d,out) =
                 AND(a,b,p) ∧
                 AND(c,d,q) ∧
                  OR(p,q,out)
```

The above definition looks perfect but there is a flaw. The signals *p* and *q* are free in the right hand side. They can be made as parameters of `IMP` but this would mean that the internal signals will be exposed and every circuit that will use it has to be concerned about these as well. That will defy the very hierarchical nature of constructing a hardware. The problem is solved using the existential quantification,

```
|- ∀ a b c d out.
           ∃ p q.
            IMP(a,b,c,d,out) =
                 AND(a,b,p) ∧
                 AND(c,d,q) ∧
                  OR(p,q,out)
```

For the task of verification, a goal is set to prove using HOL tactics that the *IMP* is actually what is defined as implemented at the very beginning. The constraints represented by the behavioral description *BEH* and the constraints represented by the structural description *IMP* are equivalent. That is, the same constraint is applied on the inputs and outputs whether it is specified directly or derived from the circuit design. The goal is written in HOL like this,

```
∀ a b c d out.
          IMP(a,b,c,d,out) = BEH(a,b,c,d,out)
```

For many circuits, a proof of equivalence is not possible due to other abstractions that is used in the circuit. In those cases it can be shown that the constraints represented by the implementation implies the constraints represented by the specification.

```
∀ a b c d out.
             IMP(a,b,c,d,out) ⟹ BEH(a,b,c,d,out)
```

The above is a weaker requirement but in many cases such implication proof does not affect the integrity of the verification framework.

For proving the above theorem in HOL, some special kind of "theorem proving functions" called tactic(s) can be used. A tactic reduces a goal to a list of subgoals, along with a function mapping a list of theorems to a theorem. The relation of theorems to goals is achievement and a theorem achieves a goal if the conclusion of the theorem is equal to the term part of the goal.

Chapter 4

Verification of RTL Blocks

In this chapter we describe the verification of the RTL blocks of OFDM using HOL according to the methodology described in Chapter 2.3 The whole design is segmented into different blocks and then modeled using HOL. The resulting model is in turn set against an ideal specification and the HOL tool is used interactively to prove its correctness. For all the blocks described below, the corresponding abstract models, parts of VHDL code, HOL models and parts of the proof strategy are provided to explain the verification in its entirety.

4.1 Verification of Quadrature Amplitude Modulation (QAM) Block

4.1.1 QAM Basics

QAM is a modulation scheme which conveys data by changing the amplitude of two carrier waves. These two waves, usually sinusoids, are out of phase with each other by 90° and are thus called quadrature carriers—hence the name of the scheme. It is a kind of *M-ary* signaling technique where one of M possible signals, $s_1(t)$, $s_2(t)$, $\ldots$, $s_M(t)$ may be sent during each signaling interval of duration T. Unlike *M-ary PSK*

(Phase Shift Keying), where in-phase and quadrature components of the modulated signals are interrelated in such a way that the envelope is constrained to remain constant, QAM has this constraint removed. The general form of *M-ary* QAM is defined by the transmitted signal in Equation (4.1)

$$s_i(t) = \sqrt{\frac{2E_0}{T}}\ a_i\ cos\ (2\pi f_c t)\ +\ \sqrt{\frac{2E_0}{T}}\ b_i\ sin\ (2\pi f_c t) \quad 0 \leq t \leq T \tag{4.1}$$

where E_0 is the energy of the signal with the lowest amplitude, and a_i and b_i are a pair of independent integers chosen in accordance with the location of the pertinent message point [30].

According to the IEEE 802.11a standard, the OFDM subcarriers shall be modulated by using *BPSK* (Binary Phase Shift Keying), *QPSK* (Quadrature Phase Shift Keying), *16-QAM*, or *64-QAM* modulation depending on the rate requested. The encoded and interleaved binary serial input data shall be divided into bit groups and converted into complex numbers representing BPSK, QPSK, 16-QAM or 64-QAM constellation points. The conversion shall be performed according to Gray-coded constellation mappings, illustrated in Figure 4.1, with the input bit, b0, being the earliest in the stream. The output values, d, are formed by multiplying the resulting $I + jQ$, where I and Q are the x-axis and y-axis of the constellation respectively, value by a normalization factor K_{MOD}

$$d\ =\ (I + jQ)\ K_{MOD} \tag{4.2}$$

The normalization factor, K_{MOD}, depends on the base modulation mode, as prescribed in Table 4.1. The purpose of the normalization factor is to achieve the same average power for all mappings. In practical implementations, an approximate value of the normalization factor can be used, as long as the device conforms with the modulation accuracy as specified in the draft standard of IEEE 802.11a in [35]. A question might arise in terms of what QAM constellation should be used for OFDM ? The answer lies in the fact that, although higher constellation gives more bits per symbol, if the mean energy is to remain the same, the points must be closer together and are thus more susceptible to noise and other corruption; this results in a higher bit error rate and so higher-order QAM can deliver more data less reliably than lower-order QAM.

Modulation	K_{MOD}
BPSK	1
QPSK	$\frac{1}{\sqrt{2}}$
16-QAM	$\frac{1}{\sqrt{10}}$
16-QAM	$\frac{1}{\sqrt{42}}$

Table 4.1: K_{MOD} Normalization

4.1.2 QAM Mapping Circuitry

For the OFDM design verified, 64-QAM constellation was chosen after simulating the floating-Point and fixed-point point model in Cadence SPW [15]. The circuitry used for QAM mapping is implemented using combinational logic. It maps the input integer data into a constellation point as shown in Figure 4.1. The VHDL modeling is done using a look-up table approach [42], as given in the VHDL code Listing 4.1. The *QAM* block takes only 3 bits as inputs and maps to an output of 16 bits as shown in Figure 4.2. It is evident that the mapping scale is different in the real implementation and it differs fundamentally if compared with the constellation diagram of 64-QAM, since this modulation scheme requires -7 to 7 to map the input data—I and Q. But, in Listing 4.1 it is mapped from -28672 to 28672 (4096×7). This is done in order to make the output numbers large enough to provide with more precise result of the computations done in the following IFFT block [42]. The QAM block is instantiated two times and designed to give the real and imaginary components

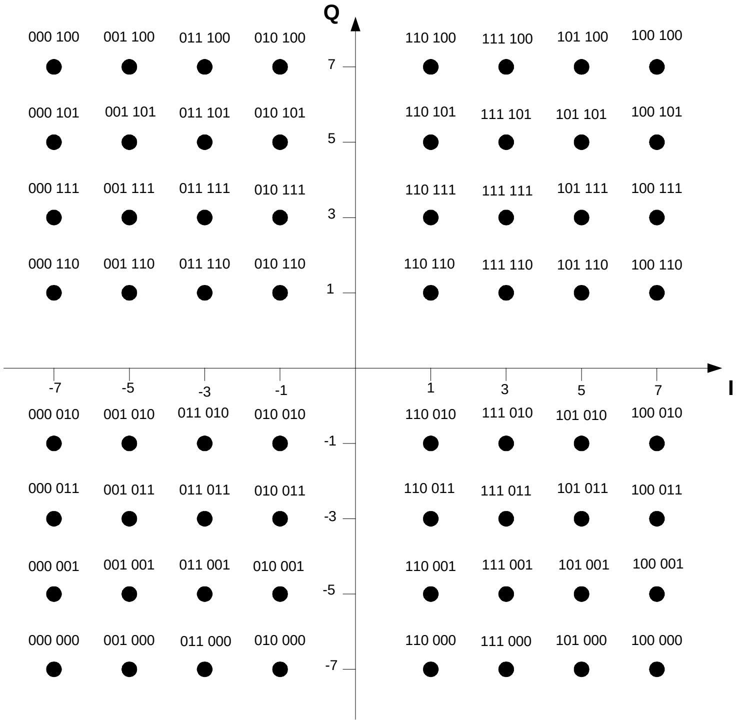

Figure 4.1: 64-QAM Constellation Bit Encoding

as two separated outputs. Each of them are formatted in 16-bit 2's complement against a 3-bit input chosen from an input of six for each block. These outputs

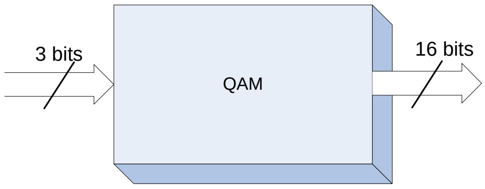

Figure 4.2: QAM Block

are shown by *out_qam_r* and *out_qam_i* in Figure 4.3. The circuitry is fed by the input continuously, therefore *out_qam_r* and *out_qam_i* are generated as continuous streams. The outputs are processed in groups of 48 symbols which are stored in two separated dual port RAMs called *"Dual Port RAM imag"* and *"Dual Port RAM real"* respectively. Since, this type of RAM is generated automatically using the Xilinx Coregen Library [61] it is not discussed further. The mapping process should

Listing 4.1: QAM Implementation in VHDL

```
......

......

 WITH input SELECT

  qam_out     <=   ”1001000000000000” WHEN ”000”  ,
                   ”1011000000000000” WHEN ”001”  ,
                   ”1111000000000000” WHEN ”010”  ,
                   ”1101000000000000” WHEN ”011”  ,
                   ”0111000000000000” WHEN ”100”  ,
                   ”0101000000000000” WHEN ”101”  ,
                   ”0001000000000000” WHEN ”110”  ,
                   ”0011000000000000” WHEN others;
.....

.....
```

be done within a 4μs time interval. The real and imaginary components of mapped symbols are grouped in a vector of 48 words, where each word is formatted in 16-bit 2's complement.

4.1.3 QAM Modeling in HOL

The modeling of QAM is done in HOL using different existing theories. IF-THEN-ELSE construct is used to embed the VHDL code as below:

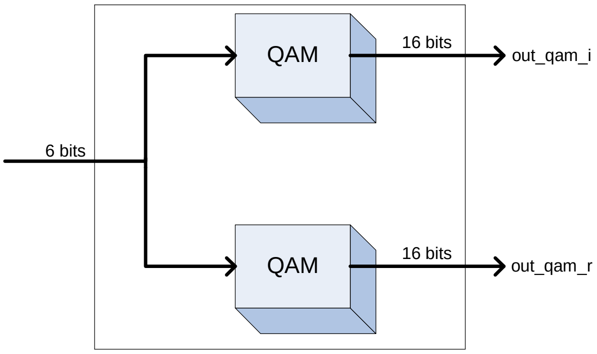

Figure 4.3: Instantiation of QAM Blocks

```
⊢def  ∀input qam_out.
        qam_imp (input qam_out) =
        (WORDLEN input = 3) ∧
        (if input = WORD [ F; F; F ] then
           qam_out = WORD [ T; F; F; T; F; F; F; F; F; F; F; F; F; F; F; F ]
         else
           (if input = WORD [ F; F; T ] then
              qam_out =
              WORD [ T; F; T; T; F; F; F; F; F; F; F; F; F; F; F; F ]
            else
              (if input = WORD [ F; T; F ] then
                 qam_out =
                 WORD [ T; T; T; T; F; F; F; F; F; F; F; F; F; F; F; F ]
               else
                 (if input = WORD [ F; T; T ] then
                    qam_out =
                    WORD [ T; T; F; T; F; F; F; F; F; F; F; F; F; F; F;
                    F ]
                   else
                     (if input = WORD [ T; F; F ] then
                        qam_out =
                        WORD
                          [ F; T; T; T; F; F; F; F; F; F; F; F; F; F; F; F]
                     else
                      (if input = WORD [ T; F; T ] then
                        qam_out =
                        WORD
                          [ F; T; F; T; F; F; F; F; F; F; F; F; F; F; F; F ]

                       else
                        (if input = WORD [ T; T; F ] then
                           qam_out =
                           WORD
                             [ F; F; F; T; F; F; F; F; F; F; F; F; F; F; F; F ]
```

```
          else
            qam_out =
            WORD
              [ F; F; T; T; F; F; F; F; F; F; F; F; F; F;
               F; F ])))))))
```

The above model is based on the *wordTheory* [60, 33]. The data types of VHDL can be modeled using this theory. The VHDL type *BIT* can be modeled using **T** and **F** where these represent **1** and **0** respectively. *BIT VECTOR* can be modeled using `WORD[...]` where the dots can be replaced with any sequence of **T** or **F** separated by *";"* as above. As an example, bit vector "110" can be modeled as `WORD[T;T;F]`. The above model is constrained using the condition `WORDLEN input = 3` since the input is always 3 bits and thus the model does not need to be generalized for n bits. Here, `WORDLEN` is a function that takes any `WORD` as input and returns the length of it. The model above now can be used (or in HDL terminology can be instantiated) as many times as required to model any complex design. For our case, it is used two times to embed the port-mapped component in HOL, and named as *qam_mod2*. We stick to the same nomenclature used by the designer. The VHDL code of the component is given in Listing 4.2. Below is the corresponding HOL modeling.

```
⊢def  ∀ input out_qam_r out_qam_i.
      qam_mod2_imp (input out_qam_r out_qam_i) =
      (WORDLEN input = 6) ∧ (WORDLEN out_qam_r = 16) ∧
      (WORDLEN out_qam_i = 16) ∧ qam_imp (WSEG 3 0 input) out_qam_i ∧
      qam_imp (WSEG 3 3 input) out_qam_r
```

This model has the same characteristics as the one before except the *input* is now constrained to six bits since the input of *qam_mod2* will always be six. Now that the modeling of the RTL block is completed it is time to model the specification of QAM in HOL. After that we will use the logical techniques of the tool to prove that the implementation is conformed to the specification.

Listing 4.2: Instantiation of QAM in VHDL

```
...... PORT(
      input     : in  std_logic_vector(5  downto 0);
      out_qam_r: out  std_logic_vector(15 downto 0);
      out_qam_i: out  std_logic_vector(15 downto 0)
     );

......  ......

BEGIN qam_i  : qam   port map
                  ( input   =>  input(2 downto 0),
                    qam_out=>  out_qam_i
                  );
qam_r  : qam   port map
                 ( input   =>  input(5 downto 3),
                   qam_out=>  out_qam_r
                 );
 END ;
.....

.....
```

Since the design is based on IEEE 802.11a we have used the standard [35] itself as a specification in order to verify the implementation. Accordingly, for every six bits entering the *qam_mod2* block, the bits are divided into three bits each, which acts as an input to the *qam* block. Then the *qam_mod2* block outputs, as describe in Section 4.1.2, two vectors containing real and imaginary parts of the modulated input. Table 4.2 shows the encoding of bits for I and Q. One point can be noticed from the two tables is the similarity of bit encoding both for I and Q and this helps us to model only one specification for both, while it is trivial to model them separately. Modeling a table in HOL can be done by using predicates as follows:

Input bits (b_0,b_1,b_2)	$I - out$
000	-7
001	-5
011	-3
010	-1
110	1
111	3
101	5
100	7

Input bits (b_3,b_4,b_5)	$Q - out$
000	-7
001	-5
011	-3
010	-1
110	1
111	3
101	5
100	7

Table 4.2: $64 - QAM$ Encoding Table [35]

```
val TABLES_QAM =
  ⊢def  ∀ I_OUT.
        TABLES_QAM (I_OUT) =
        (I_OUT (F,F,F) = ¬7) ∧ (I_OUT (T,F,F) = ¬5) ∧
        (I_OUT (T,T,F) = ¬3) ∧ (I_OUT (F,T,F) = ¬1) ∧
        (I_OUT (F,T,T) = 1) ∧ (I_OUT (T,T,T) = 3) ∧
        (I_OUT (T,F,T) = 5) ∧ (I_OUT (F,F,T) = 7)
```

In the above model *I_OUT* is a triplet which will accept three bits similar to the left columns of Table 4.2. For each and every argument of *I_OUT*, a unique number will be mapped as given in the tables and '∧' is used as a composition operator to construct all rows.

Having covered all the pertinent details about the implementation and a very reliable means to extract the specification, *qam_spec* can be written in terms of TABLES_QAM–

```
 ⊢def  ∀ b0 b1 b2 I_OUT.
       qam_spec (b0 b1 b2 I_OUT) =
       ∃OUT. TABLES_QAM OUT ∧ (I_OUT b0 b1 b2 = OUT (b0,b1,b2))
```

The specification *qam_spec* is mirrored, in the same way its implementation *qam_imp* is instantiated in *qam_mod2_imp*,

```
⊢def  ∀ input I_OUT_R I_OUT_I.
        qam_mod2_spec (input I_OUT_R I_OUT_I) =
        qam_spec (BIT 0 input) (BIT 1 input) (BIT 2 input) I_OUT_I ∧
        qam_spec (BIT 3 input) (BIT 4 input) (BIT 5 input) I_OUT_R
```

With the specification above we have finished all the groundwork to set the goal for verification of the QAM RTL block. The next subsection will discuss the verification in details and the proof strategies adopted to bolster the correctness of RTL implementation.

4.1.4 Verification of QAM

The general goal is to prove that for all inputs and outputs the correctness theorem holds, under certain constraints, which can be stated as

```
∀  n inputs outputs.
     constraints  ⊃   (implementation ≡  specification) [9]
```

The equivalence can be replaced by implication which will set space for some allowance in the correctness theorem by proving only specific behaviors of the system, which will certainly weaken the sole purpose of verification [46]. But, there are cases where the engineer (or anybody who is carrying out the proof work) can categorically exclude some cases given the certainty that those will never occur. For our case, it is an implication due to the constraints we have imposed in the definitions since we are certain that there can be no other combination occurring other than those. This justification leaves us only to state our goal, except we need one more definition to do so, which is as follows:

```
⊢def  ∀ x.
        TCOMP_VAL x =
        ¬& (BV ()) * 2 pow 3 + & (BV (BIT 2 x)) * 2 pow 2 +
        & (BV (BIT 1 x)) * 2 pow 1 + & (BV (BIT 0 x))
```

It is a simple definition based on *boolLibrary* of HOL to convert a `bool word` into its real number equivalent. The function `TCOMP_VAL` accepts a `bool word` and returns a real number. The "&" symbol is an overloaded HOL operator that converts any natural number to real number. And, `BV` is also a function, defined in theory *numTheory*, that uses another function to convert the boolean value into a natural number.

$\vdash_{def}$ ∀ `b. BV b = (if b then SUC 0 else 0)`

The function `SUC` takes a natural number and returns the consecutive natural number. So, `SUC 0` will return `1`. And, `BIT x input` chooses a particular bit positions from *input* defined in *x*. Now, we can state that our goal as - *for all input and output and constraints, the QAM implementation implies the QAM specification*

```
∀ n inputs outputs.
   constraints  ⊃   (QAM implementation ⟹  QAM specification)
```

formalized in HOL as

```
∀ input qam_out.
  qam_imp (input qam_out) ⟹
  qam_spec (BIT 0 input) (BIT 1 input) (BIT 2 input)
    (λ b0 b1 b2. TCOMP_VAL (WSEG 4 12 qam_out))
```

The definition `WSEG m k WORD` selects a portion of `WORD` from `k` to `k+m-1`. The function *qam_spec* takes three arguments and gives a corresponding output. One λ abstraction is used to convert the selected *qam_out* word into real number. Now, the stage is set to apply the tactics of HOL to prove the goal. We have used the existing theories of *wordTheory* and *realTheory* to build many helpful definitions and lemmas to prove the above goal and thus established the correctness of the RTL block formally. We prove the theorem and name it as *qam_imp_spec_correct.* Due to textual brevity, we do not include the whole proof procedure here line by line. But, proving this theorem just ensures us about the QAM block and we are yet to prove the implementation of *qam_mod2_imp*. In order to do so we set a goal as -

```
∀ input out_qam_r out_qam_i.
   qam_mod2_imp (input out_qam_r out_qam_i) ⟹
   qam_mod2_spec input (0 b1 b2. TCOMP_VAL (WSEG 4 12 out_qam_r))
     (λ b0 b1 b2. TCOMP_VAL (WSEG 4 12 out_qam_i))
```

We use the same libraries as before to prove this goal too. Below is the HOL proof steps.

```
REPEAT GEN_TAC THEN
ARW_TAC [qam_mod2_spec,qam_mod2_imp]THEN
ARW_TAC[BIT_WSEG_input]THEN
ARW_TAC [qam_imp_spec_correct]THEN
ARW_TAC[BIT_WSEG_input]THEN
ARW_TAC [qam_imp_spec_correct]
```

We use only built in tactics. The `REPEAT GEN_TAC` tactic removes all the universal and existential quantifications. Next, `ARW_TAC` is a tactic defined using a rewriting tactic `RW_TAC` using simpset [33] `arith_ss`. This defined tactic is used to rewrite the the goal with the specifications and proved theorems as shown in the code segments above. We name this last proved theorem as *qam_imp_spec_correct*.

Having proved the correctness of *qam_mod2_imp* and *qam_imp* using the theorems *qam_imp_spec_correct* and *qam_mod2_imp_spec_correct* it can be concluded that the *QAM* is formally verified. The implementation conforms the specification given in the standard.

4.2 Verification of the Serial to Parallel Block

4.2.1 Serial to Parallel Basics

In this section we will verify the serial to parallel block, later written as *S/P* , which is an indispensable part of the whole OFDM system. Most of the basics related to *S/P* are similar to those of the Parallel to Serial block, to be discussed later, and thus will cover almost all the important aspects of both blocks in this section. The concept of serial to parallel conversion is trivial. A long stream of data is divided into several equal or approximately equal length of chunks which can all be operated upon at the same time. From the mathematical point of view, it is the manipulation of a vector into several columns of a matrix. However, *S/P* conversion is very important in OFDM. The length of the blocks produced in *S/P* determine the number of spectral coefficients to be used by the IFFT, which is essential in choosing how many frequencies are to be used. Usually, the block length is a power of 2, which makes the IFFT and FFT algorithms most computationally efficient. Moreover, in OFDM, the data is divided among a large number of closely spaced carriers. Since, the entire bandwidth is filled from a single source of data, it is necessary to transmit in parallel way so that only a small amount of the data is carried on each carrier, and by this lowering of the bitrate per carrier, the influence of intersymbol interference is significantly reduced.

4.2.2 S/P Circuitry

The *S/P* circuitry is very simple to implement. It has its presence both in the transmitter and receiver of the system. In transmitter side, it is placed between *QAM* and *IFFT* block, and in the receiver side between *Guard Removal* and *FFT* block. The design at hand has the same functionality of of *"Bits to fixp"* block of SPW [15] in fixed-point model. It consists of a shift register and a latch, which are both clocked with the same rate as the input data. Six bits from input stream are serially shifted into a register. Then they are latched for six clock cycles. There are two control signals *enable* and *clear* to synchronize the whole process. Figure 4.4 shows a simple block diagram of a serial to parallel circuit functionality with a clock signal only. The VHDL code for this block is in Listing 4.3. Now, if we analyze the VHDL code we will find three signals *shift_reg*, *count* and *hold* are defined to use in the *PROCESS*, which is controlled by four inputs of the block as a sensitivity list. Signal *shift_reg* acts as a buffer for the manipulation of the data; *count* keeps track of the number of iterations and *hold* latches all the data to be assigned to the output port *out_parallel* after certain clock cycles.

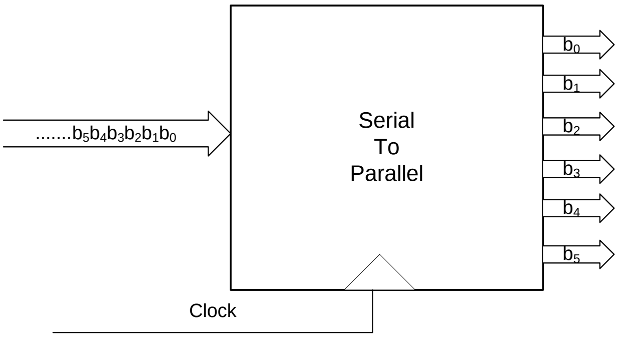

Figure 4.4: Block Diagram of a Typical Serial-Parallel Design

4.2.3 S/P Modeling in HOL

Modeling of the *S/P* block in HOL is done in a different way than what we have seen in Section 4.1.3. The modeling is not exactly one to one mapping because a VHDL *PROCESS* is involved here. In fact, a *PROCESS* never terminates itself, and it can only be controlled using *WAIT* statements and sensitivity lists. After executing the last statement, a *PROCESS* will be suspended only to be resumed later on an event in the sensitivity list. This last behavior poses a difficulty in modeling it in HOL due to non-termination problem. Higher order logic is a logic of total function and it does not allow the definition of any partial function. But, there are exceptions which motivates us to define our specification for *S/P* in a simpler way without resorting to

Listing 4.3: VHDL code for Serial to Parallel Block

```
.........
ARCHITECTURE rtl OF  serial_to_parallel IS
SIGNAL shift_reg   : std_logic_vector(5 downto 0):=”000000”;
SIGNAL count       : std_logic_vector(2 downto 0):=”000”;
SIGNAL hold        : std_logic_vector(5 downto 0):=”000000”;
..........
  PROCESS( clk  , clear  ,enable,input )
  BEGIN
    IF clear = ’1’ THEN hold <= ”000000”;
    ELSIF clk’event AND clk=’1’ THEN
    IF enable = ’1’ THEN
     IF  count /= 5 THEN
         count <= count + 1;
         shift_reg(4 downto 0) <= shift_reg (5 downto 1) ;
         shift_reg(5)<= input ;
     ELSE
     shift_reg(4 downto 0) <= shift_reg (5 downto 1) ;
     shift_reg(5)<= input ;
     hold (5) <= shift_reg (0);
     hold (4) <= shift_reg (1);
     hold (3) <= shift_reg (2);
     hold (2) <= shift_reg (3);
     hold (1) <= shift_reg (4);
     hold (0) <= shift_reg (5);
     count <= ”000”;
END IF;
END IF;
END IF;
END PROCESS;
out_parallel  <=  hold ;
END ;
```

complex definition. For example, the following is a total and non-recursive function that uses the expressive power of HOL [33]:

```
λ x. if (? n. P (FUNPOW g n x)) then
                  FUNPOW g (@n. P (FUNPOW g n x) ∧
                      !m. m < n ==>  P (FUNPOW g m x)) x
     else ARB
```

The function FUNPOW is a tail recursive function defined in the theory *arithmetic-Theory* to define function iteration. The above function does a case analysis on the iterations of function g. The finite ones return the first value at which P holds and the infinite ones are mapped to a constant named ARB that holds all the arbitrary values. ARB is a way to convert partial-functions into total functions in HOL. But, using ARB will only complicate our model without any added benefit. A VHDL *PROCESS* is more than a simple loop and we have no cases to deal with infinity rather we only have finite sets of statements to be dealt infinitely. This discussion is to justify why we did not use certain features of HOL to model our system which seems apparently helpful in doing so. The other aspect of the model is that three signals *clk, enable,* and *clear* are not used since we are verifying this module independently of other blocks, and there are no pipelining issues involved here. Having said that we introduce the implementation of *S/P* in HOL -

```
⊢def  ∀ cnt out_parallel input.
         Serial_Parallel_IMP (cnt out_parallel input) =
         ∃ shift_reg.
           (WORDLEN out_parallel = 6) ∧
           (shift_reg input = SHRN_bit cnt input out_parallel)
```

Apparently a simplification of the corresponding VHDL code but a little analysis will support its correct functionality. From the code, the variable cnt is a natural number whose type is defined as num; out_parallel is a bool word and input is of bool type. The implementation takes three arguments where cnt is defined to keep track of the time or bit index which is a model of the `signal count`. The second variable has the same name of its VHDL counterpart and so is the last one - `input`. A function `shift_reg` is defined as `shift_reg:bool→bool word` to mimic the VHDL signal of the same name. Variable `out_parallel` is constrained to six using WORDLEN function as before because the design specifies so. Since the system will receive only one input at a time and then latches all till it fills the whole shift register, so we write another definition in HOL to manipulate every new bit entering the system and filling the empty places with zeros

```
⊢def  ∀ N M w.
          SHRN_bit (N M w) =
          WCAT (WORD (REPLICATE (WORDLEN w − (N + 1)) F),WORD [ M ])
```

This definition uses `WCAT` which concatenates two lists is defined in *word_baseTheory* [33] as

```
⊢def  ∀ l1 l2. WCAT (WORD l1,WORD l2) = WORD (l1 ++ l2)
```

The symbol '++' is an infix operator that appends two lists in the above definition. The recursive definition of `REPLICATE` is in the theory *rich_listTheory* which replicates any variable repeatedly as specified. It is defined as

```
⊢def  (∀ x. REPLICATE 0 x = []) ∧
        ∀ n x . REPLICATE (SUC n) x = x::REPLICATE n x
```

Here the `REPLICATE` function fills the rest of the places of the shift register with 'F' depending on the current value passed to it by the function and then adds the `input` to it. In this way at the end of the iteration the whole register will be populated with serial data and will be ready to be latched out.

Having completed the modeling of implementation we describe the specification of the block so that we can explain the verification in the next section. We state the specification of the block as

```
⊢def  ∀ t out input.
         Serial_Parallel_SPEC (t out input) = (BIT t out = input)
```

It simply puts the relation between the input and output of the block it terms of bit position. At every time `t`, we have one input entering the block which goes in the bit position related to the current index of `t` of the output. A more general approach would be to use the modulo arithmetic to model the specification, but it is not required here due to the proof strategy we followed in Section 4.2.4.

4.2.4 S/P Verification

Unlike the verification strategy of QAM explained in Section 4.1.4 we adopt case analysis approach to prove the goal. We can define the goal as following:

```
∀ out input t.
           (0  ≤ t ∧ t ≤ 5 ) ⟹
                Serial_Parallel_IMP (t out input) ⟹
                     Serial_Parallel_SPEC (t out input)
```

It has a very generic pattern like any other goal except the constraint which bounds t as, $0 \leq t \leq 5$. Bounding t helps to get over with the problem of looping which we stated earlier in subsection 4.2.3. We flatten one whole iteration which is enough to demonstrate the functional correctness of the given block. That is why we bound the variable only to check the cases starting from $t = 0$ to $t = 5$. Once we finish with case analysis we prove following trivial lemma

```
∀ t.
           (0 ≤ t ∧ t ≤ 5) ⟹
           (t = 0) ∨ (t = 1) ∨ (t = 2) ∨
           (t = 3) ∨ (t = 4) ∨ (t = 5)
```

which simply states that when t is bound between 0 and 5, then the only values for which the correctness theorem needs to hold are $t = 0$, 1, 2, 3, 4 5. We proved the goal and thus verified the functionality of the said RTL block. The part of the proof procedures will be explained in the Section 4.3.4 due to its similarity in the way of doing the proof.

4.3 Verification of Parallel to Serial Block

4.3.1 P/S Basics

This section presents the verification of Parallel to Serial RTL block, later referred as *P/S* . As mentioned before, this block is similar to *S/P* and thus the functionality and design bears resemblance with that. For the OFDM system *P/S* is flanked by *IFFT* and *Guard Interval Insertion* at the transmitter side; and by *FFT* and *64-QAM Demodulator* at the receiver side. Principally, it takes parallel data as input and outputs serial data stream. In OFDM, this block is used to to convert data from frequency domain to time domain and let to insert guard intervals between IFFT frames as required. Figure 4.5 shows a block diagram of a typical *P/S* block.

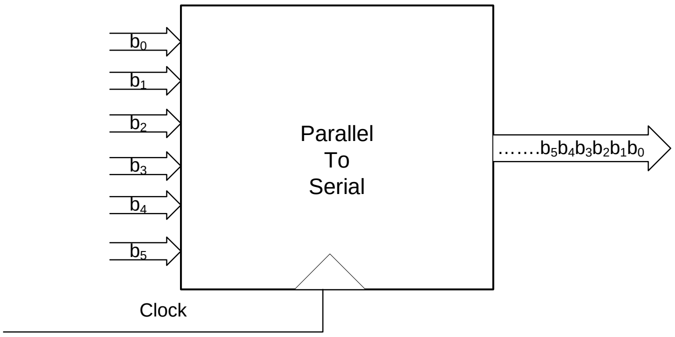

Figure 4.5: Block Diagram of a Typical Parallel-Serial Design

4.3.2 P/S Circuitry

Generally, the construction of a *P/S* circuitry is simple. Most of the generic circuits load the data in different registers clocked using a global timer at the same time but outputs the data from a register containing LSB or MSB bit while shifting the bits simultaneously from the other direction. Figure 4.5 shows the concept graphically. We verify a similar design as a part of our OFDM system verification. The input and output variables are similar to *S/P* except that the size of the vectors have swapped roles. The `VHDL` code for the design is given in Listing 4.4. One signal namely `out_at_each_clock` is defined to latch data to the output port at regular intervals. Unlike *S/P* , the `PROCESS` in this design contains two variables: `shift_reg` and `count`. As before, `shift_reg` is a *shift register* and *count* keeps track of the iterations. Any active event on `clk`, `clear`, `enable` or `input` will trigger the process. While count is at ‘000’, the input is assigned to shift_reg. Every fifth bit of the `shift_reg` is then shifted to `out_at_each_clk` and the counter is increased. When the count reaches ‘5’, it is initialized to ‘000’ again. After every iteration of a `PROCESS` the circuit will give one bit as output.

4.3.3 P/S Modeling in HOL

In this subsection we discuss the modeling of the circuitry in HOL. The discussion on looping in Section 4.2.3 equally applies here. Rather we concentrate on the issues of modeling signals and variables in HOL. A signal is a VHDL model of wire. Anything defined as signal is assigned the value after a delta (δ) delay. Whereas a variable can only be defined inside a `PROCESS`, and value can be assigned instantaneously unlike signal. From the higher order logical point of view, signal and variable can be treated as same. The notion of time delay is useful in simulation to understand the timing parameters of the system. But, it plays very little role in verifying the functional correctness of the design we are verifying. Thus we verify a delay-free model. The same can be said above all the verifications done above. We define the implementation of *P/S* circuit in HOL as following:

Listing 4.4: VHDL Code for *P/S* Circuit

```
ARCHITECTURE rtl OF parallel_to_serial  IS

SIGNAL out_at_each_clk        : std_logic;

BEGIN
  PROCESS(clk,clear,enable,input)
  VARIABLE shift_reg   : std_logic_vector(5 downto 0):=”000000”;
  VARIABLE count       : std_logic_vector(2 downto 0):=”000”;
  BEGIN
    IF clear = ’1’ THEN
      out_at_each_clk <= ’0’;
    ELSIF clk’event AND clk=’1’ THEN
      IF enable = ’1’ THEN
                  IF  count = 0 THEN

                   shift_reg(5 downto 0):= input ;
                       out_at_each_clk      <= input(5);
                   shift_reg(5 downto 1)     := shift_reg(4 downto 0);
                  ELSE

                   out_at_each_clk           <= shift_reg(5);
                   shift_reg(5 downto 1)     := shift_reg(4 downto 0);
                  END IF;

                  IF count = 5 THEN
                                          count := ”000”;
                      ELSE
                             count := count + 1;
                      END IF;
      END IF;
    END IF;
```

```
⊢def  ∀ cnt out input.
       Parallel_Serial_IMP (cnt out input) =
       ∃ shift_reg.
         (WORDLEN input = 6) ∧ (shift_reg cnt = SHLN cnt input) ∧
         (out cnt = BIT 5 (shift_reg cnt))
```

This time, we constrain the input of the circuit to six bits and we use another definition, SHLN, in order to accomplish this:

```
⊢def  ∀ N w. SHLN (N w) = WCAT (WSEG (WORDLEN w − N) 0 w,WORD (REPLICATE N F))
```

This definition also uses WCAT, WSEG and REPLICATE. The arguments cnt and input are passed to the function in order to get the number of bits to be extracted from the input in order to copy it back to the register. The replication of zero(s)in the vacant place is determined by the second list.

To define the specification for verification we use modulo arithmetic. The use of t MOD 6 helps to index the correct output in the specification below:

```
⊢def  ∀ t out input.
       Parallel_Serial_SPEC (t out input) =
       (out t = BIT (5 − t MOD 6) input)
```

MOD is built into HOL and some basic theorems are defined on it:

```
MOD_EQ_0 = ⊢  ∀ n. 0 < n ⟹  ∀k. (k * n) MOD n = 0
ZERO_MOD = ⊢  ∀ n. 0 < n ⟹  (0 MOD n = 0)
MOD_PLUS = ⊢  ∀ n. 0 < n ⟹  ∀j k. (j MOD n + k MOD n) MOD n
         = (j + k) MOD n
MOD_MOD  = ⊢  ∀ n. 0 < n ⟹  ∀k. k MOD n MOD n = k MOD n
```

The specification takes counter, input and output as argument and makes sure that the bit assigned to the output is the correct one by decreasing the index of the bit selection number. This specification can be used to verify any such circuit.

4.3.4 P/S Verification

In this section we will verify the circuit and as usual we will start with defining the goal to be proved:

```
∀ out input t.
0 ≤ t ∧ t ≤ 5 ⟹
Parallel_Serial_IMP t out input ⟹
Parallel_Serial_SPEC t out input
```

The value of t is constrained since we are only interested to verify it for that number of iterations only, which means one whole functional cycle is enough to prove its correctness. Below we will show the main steps of the proof done and the associated lemmas used. We start with removing all the quantifications and then rewrite using the definitions of `Parallel_Serial_IMP`, `Parallel_Serial_SPEC`, `SHLN` and `t_0_5`, where `t_0_5` is the same lemma we have stated in Section 4.2.4. Then we start case analysis on `t`. The code below shows the way to input these into the HOL tool:

```
REPEAT GEN_TAC
RW_TAC list_ss [Parallel_Serial_IMP,Parallel_Serial_SPEC, t_0_5]
ASM_REWRITE_TAC [SHLN]
Cases_on ‘t=0‘
```

This tactic generates two subgoals:

```
    Parallel_Serial_SPEC t out input
    ------------------------------------
      0.  t ≤ 5
      1.  Parallel_Serial_IMP t out input
      2.  ¬(t = 0)

    Parallel_Serial_SPEC t out input
    ------------------------------------
      0.  t ≤ 5
      1.  Parallel_Serial_IMP t out input
      2.  t = 0
```

We now use the theorem WCAT0 and a trivial lemma wordlen_6_inp to help prove the first subgoal:

```
RW_TAC arith_ss [wordlen_6_inp]
RW_TAC arith_ss [REPLICATE]
RW_TAC arith_ss [WCAT0]
```

Where the two are defined as:

```
wordlen_6_inp =
               ⊢_def  ∀ input. (WORDLEN input = 6)⟹
                          (WSEG 6 0 input = input)
WCAT0 = ⊢_def  ∀ w. (WCAT (WORD ],w) = w) ∧
                           (WCAT (w,WORD ]) = w)
```

In the next step, we do case analysis on t=1. It returns another two subgoals. We use a couple of lemmas to prove them. All of theses lemmas are proved using theorems and functions which are already explained in the previous sections: REPLICATE_1 is to prove that replicating 'F' one time is equal to a WORD with only one bit.

```
REPLICATE_1 = ⊢  WORD (REPLICATE 1 F) = WORD [ F ]
```

BIT_LEMMA_01 proves that for WORD w1 with length n1 and w2 with length n2; and any number k greater or equal to n2 and less than the sum of n1 and n2 implies that the kth bit of the concatenated WORD from w1 and w2 is equal to the (k-n2)th bit of w1.

```
BIT_LEMMA_01 =   ⊢  ∀n1 n2 w1 w2 k.
        (WORDLEN w1 = n1) ∧ (WORDLEN w2 = n2) ∧ n2 ≤ k ∧
        k < n1 + n2 ⟹  (BIT k (WCAT (w1,w2)) = BIT (k − n2) w1)
```

BIT_LEMMA_02 states that if wordlength of w is n and any number m and k is less than or equal to n, then it implies that the wordlength of the word segment of w taken from k to k+m-1 is equal to m.

```
BIT_LEMMA_02 =
  ⊢  ∀ n w m k.
       (WORDLEN w = n) ∧ m + k ≤ n ⟹  (WORDLEN (WSEG m k w) = m)
```

BIT_LEMMA_03 is a trivial lemma proved to state that any WORD with length 6 implies that if a segment taken from the word from 0 to 4, then its length is 5.

```
BIT_LEMMA_03 = ⊢  ∀ w. (WORDLEN w = 6) ⟹
                                  (WORDLEN (WSEG 5 0 w )= 5)
```

BIT_LEMMA_04 states that the wordlength of a single boolean element is 1.

```
BIT_LEMMA_04 = ⊢  WORDLEN (WORD F]) = 1
```

BIT_LEMMA_05 is an important lemma which states that given a WORD w with length n and the addition of two numbers m and k that is less than or equal to n implies that when another number j is less than m, then jth bit of the word segment of w from k to k+m-1 is equal to the (j+k)th bit of w.

```
BIT_LEMMA_05 =
  ⊢  ∀ n w m k j.
        (WORDLEN w = n) ∧ m + k ≤ n ⟹
        j < m ⟹
        (BIT j (WSEG m k w) = BIT (j + k) w)
```

Initially, we do modus ponens (MP) and specialize the lemmas along with some rewriting steps. MP helps us to introduce new lemmas in the proof procedure and then we specialize them according to our need. The rest of the proof steps involves repeated use of case analysis until t=5. For this, we prove another 14 lemmas and use some conversion tactic. The proof steps shown in this section delineates a typical approach which is specific to this proof. As mentioned in Chapter 3, there are many ways to do a proof and it mainly depends on the nature of the goal, and also the skill and experience of the user.

4.4 Verification of QAM Demodulator

4.4.1 Demodulator Basics

The demodulator is the last block of the OFDM system. It recreates the original message from the degraded version of the transmitted signal. The operating principles of the demodulator block is not straightforward as the modulator. The incoming signal is first downconverted and then demodulated. In between transmitter and receiver the medium hampers the quality of the signal by factors including atmospheric noise, competing signals, and multipath or fading. A modulator deals with data which has no distortion and thus engage no time to sort the intelligent signals from its input. But the demodulator has to handle the stochastic nature of data which cannot be mapped as easily as the constellation diagram shown in Section 4.1.1. Because the existence of noise causes the constellation points to spread, the demodulator has to establish decision regions in order to map the received signals to the correct QAM value. Generally, the demodulation involves a number of steps: (1) carrier frequency recovery, (2) symbol clock recovery, (3) signal decomposition to I and Q components, (4) determining I and Q values for each symbol, (5) decoding and de-interleaving, (6) expansion to original bit stream, (7) digital-to-analog conversion, if required. Both the symbol-clock frequency and phase must be correct in the receiver in order to demodulate the bits successfully and recover the transmitted information. A symbol clock could be at the right frequency but at the wrong phase. If the symbol clock was aligned with the transitions between symbols rather than the symbols themselves, the demodulation would be unsuccessful.

4.4.2 Demodulator Circuitry

The demodulator circuitry, as designed in [42] consists of combinational logic. This block is modeled in SPW as `General Slicer` to map QAM symbols from I and Q format to integer numbers. The functionality of OFDM demodulator is intertwined with the P/S block. Figure 4.6 shows the input and output of the demodulator to be

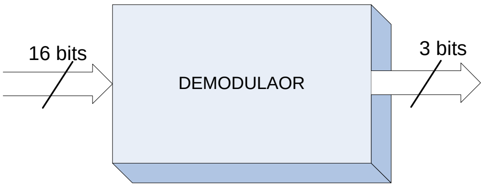

Figure 4.6: Demodulator Block

verified. It is implemented using if-then-else construct in VHDL. Listing 4.5 shows part of the code. It takes one input of 16 bits and outputs 3 bits. The MSB of the input is used as a sensitivity list to activate the process. When the sensitivity input is bit '0' then it maps the QAM symbols to a certain conditional block. For example, when the input (in hexadecimal format) is "0180" then it is mapped to "100", in binary, which is the top-right corner constellation point as given in Figure 4.1. On the other hand, if the MSB of input is '0', then it is mapped to another conditional clause. The values associated to a certain QAM symbol are mapped according to the decision regions given in the code. We describe it more in the later subsections. This demodulator block is then instantiated two times in order to map both real and imaginary symbols coming from the FFT block. Figure 4.7 shows the instantiated block diagram. Listing 4.6 shows the VHDL code for this.

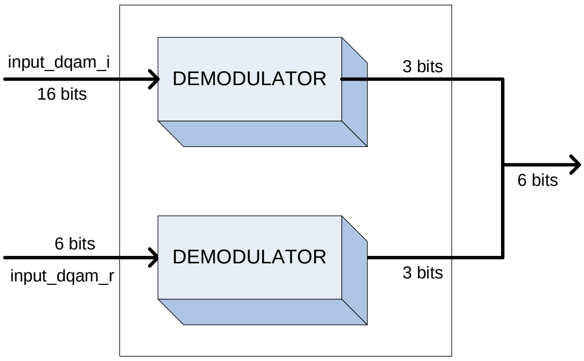

Figure 4.7: Instantiation of the Demodulator Block

Listing 4.5: VHDL code of OFDM Demodulator

```
BEGIN

 process(input)

 begin

      IF input(15) = '0' THEN

         IF      input >= x"0180" THEN
            dqam_out <= "100" ;
     ELSIF input >= x"0100" AND  input < x"0180"  THEN
                dqam_out <= "101" ;
     ELSIF input >= x"0080" AND  input < x"0100"  THEN
                dqam_out <= "111" ;
     ELSIF input < x"0080"  THEN
                dqam_out <= "110" ;
     END IF;
      ELSE
     IF      input >= x"FF80"  THEN
                dqam_out <= "010" ;
     ELSIF input >= x"FF00"  AND  input < x"FF80"  THEN
                dqam_out <= "011" ;
     ELSIF input >= x"FE80"  AND  input < x"FF00"  THEN
                dqam_out <= "001" ;
     ELSIF input < x"FE80"  THEN
                dqam_out <= "000" ;

         END IF;
      END IF;
 end process;

  END ;
```

Listing 4.6: VHDL Code for Demodulator Instantiation

```
.....
.....

dqam_r  :  dqam   port map (
                  input      =>   input_dqam_r(15 downto 0),
                  dqam_out   =>   output( 5 downto 3 )
                  );
dqam_i  :  dqam   port map (
                  input      =>   input_dqam_i(15 downto 0),
                  dqam_out   =>   output( 2 downto 0)
                  );
END ;
```

4.4.3 Demodulator Modeling in HOL

Modeling the implementation in HOL follows the same precept like any other block. The noticeable difference between demodulator and modulator is that the latter is designed with a process. But, unlike the design of *P/S* and *S/P* it does not have any looping in terms of the sequential statements. We model the demodulator using IF-THEN-ELSE constructs.

```
⊢def ∀input dqam_out.
       dqam_imp (input dqam_out) =
       (if BIT 15 input = F then
          (if BNVAL input ≥ 384 then
             dqam_out = WORD [ T; F; F ]
           else
             (if BNVAL input ≥ 256 ∧ BNVAL input < 384 then
                dqam_out = WORD [ T; F; T ]
              else
                (if BNVAL input ≥ 128 ∧ BNVAL input < 256 then
                   dqam_out = WORD [ T; T; T ]
                 else
                   dqam_out = WORD [ T; T; F ]))) 
```

```
    else
      (if BNVAL input >= 65408 then
         dqam_out = WORD [ F; T; F ]
       else
         (if BNVAL input >= 65280 / BNVAL input < 65408 then
            dqam_out = WORD [ F; T; T ]
          else
            (if BNVAL input >= 65152 / BNVAL input < 65280 then
               dqam_out = WORD [ F; F; T ]
             else
               dqam_out = WORD [ F; F; F ]))))
```

The HOL model of demodulator takes two `bool word` *input* and *dqam_out* as input and output, respectively. We manually convert the hexadecimal numbers defined in the VHDL code into natural numbers. The model is similar with the QAM except the usage of one new definition. The function `BNVAL`, defined in the theory *bword_numTheory*, takes one `bool word` and converts it into natural number.`BNVAL` helps to compare the QAM symbols easily:

$\vdash_{def}$ ∀ l. BNVAL (WORD l) = LVAL BV 2 l

Next, we define the demodulator instantiation using the demodulator model:

```
⊢def ∀ input_dqam_r input_dqam_i output.
       dqam_mod (input_dqam_r input_dqam_i output) =
       (WORDLEN input_dqam_r = 16) ∧
       (WORDLEN input_dqam_i = 16) ∧
       (WORDLEN output = 3) ∧
       dqam_imp input_dqam_r (WSEG 3 3 output) ∧
       dqam_imp input_dqam_i (WSEG 3 0 output)
```

where `input_dqam_r`, `input_dqam_i` and `output` are arguments to `dqam_mod_imp`. `WSEG` is used to map the desired portion of the word. We constrain both the inputs to a size of 16 bits and both the outputs to a size of 3 bits based on the design at hand. Such constraints make the proof relatively less complicated without modifying any parameter of the design.

We now concentrate on the model of the specification of the system before resorting to verification. But, finding a specification for a demapper is not an easy task. Unlike its predecessor blocks, where the specification can be derived from the standards or from existing theory, the demodulator is dependent on the design decision of the hardware designer. The demapping takes place according to the partition of the constellation plane into decision regions. Figure 4.8 shows the mapping of the regions used to model the specification. We rely on the design itself to extract

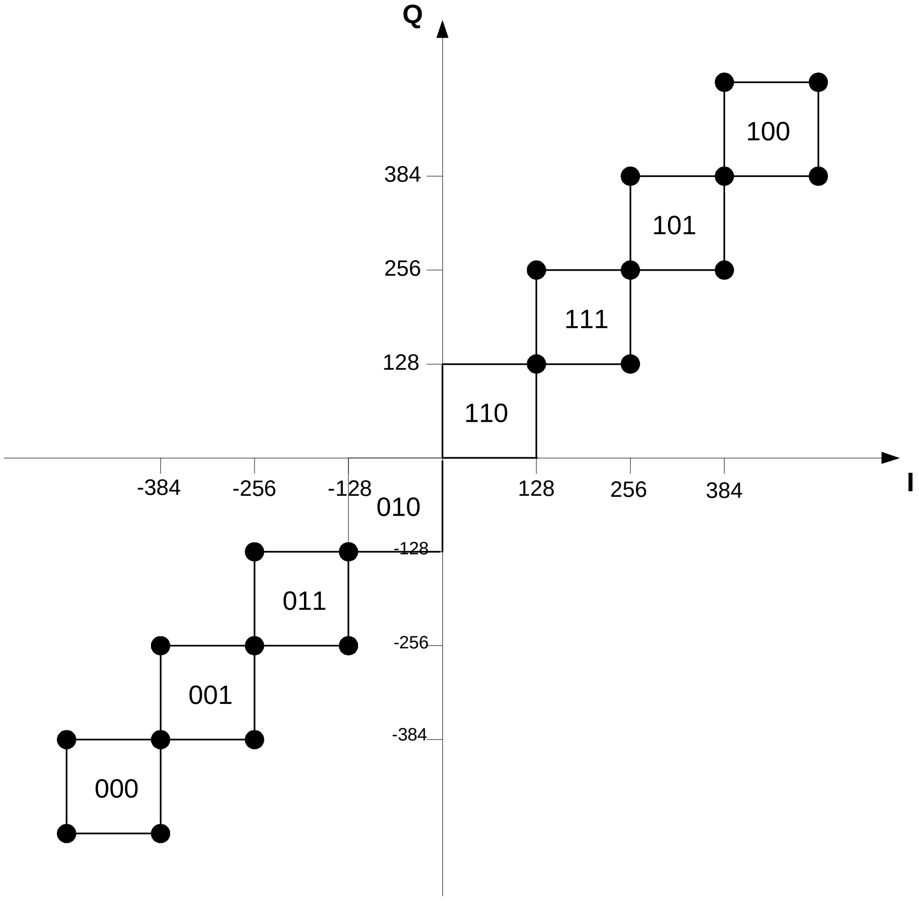

Figure 4.8: Decision Region for Demapping

the specification since there is no way to know how the FFT block is implemented as the designer has port-mapped the design to a Xilinx FFT IP, which contains proprietary RTL code that cannot be accessed. We model the whole decision region in a table format. It can be assumed that the specification is residing in a ROM

Input (I & Q in Real Number)	*Output*
-385	000
-384	001
....	”
....	”
-257	001
-256	011
....	”
....	”
-129	011
-128	010
....	”
....	”
0	010
1	110
....	”
....	”
127	110
128	111
....	”
....	”
255	111
256	101
....	”
....	”
383	101
384	100

Table 4.3: Demapping Table for OFDM Demodulator

where the whole table is saved. We construct only one table, Table 4.3, for both I and Q because there is no difference between them. The QAM symbol -385 or below maps to 000, where the minimum value depends on the size of the ROM. Any symbol between −384 and −257 maps to 001 and the rest follows suit. We show in Section 4.4.4 how we handle values smaller than −385 and greater than 384. Next, we formalize the table in HOL as follows:

```
⊢def  ∀ REV_I_OUT.
       TABLES_DQAM REV_I_OUT =
       (REV_I_OUT ¬385 = WORD F; F; F]) ∧
       (REV_I_OUT ¬384 = WORD F; F; T]) ∧
         .
         .
       (REV_I_OUT ¬257 = WORD F; F; T]) ∧
       (REV_I_OUT ¬256 = WORD F; T; T]) ∧
         .
         .
       (REV_I_OUT ¬129 = WORD F; T; T]) ∧
       (REV_I_OUT ¬128 = WORD F; T; F]) ∧
         .
         .
       (REV_I_OUT 0 = WORD F; T; F]) ∧
       (REV_I_OUT 1 = WORD T; T; F]) ∧
         .
         .
       (REV_I_OUT 127 = WORD T; T; F]) ∧
       (REV_I_OUT 128 = WORD T; T; T]) ∧
         .
         .
       (REV_I_OUT 255 = WORD T; T; T]) ∧
       (REV_I_OUT 256 = WORD T; F; T]) ∧
         .
         .
       (REV_I_OUT 383 = WORD T; F; T]) ∧
       (REV_I_OUT 384 = WORD T; F; F])
```

We define a function REV_I_OUT which takes real value as input that acts as a key to choose the corresponding bool word value in the table. In order to keep the code short, we show only representative value of the table. Next, we define the model of the demodulator based on this table.

```
⊢def  ∀ input demapper.
       dqam_spec input demapper =
       ∃ mapVal.
         TABLES_DQAM mapVal ∧
         (demapper (& (BNVAL input)) = mapVal (& (BNVAL input)))
```

The definition of *dqam_spec* describes the mapping using a bool word variable and two functions where the latter can be existentially quantified. For every input through function mapVal one unique value will be mapped using the defined table. For the modeling of the instantiated demodulator block dqam_mod2_imp we use the model dqam_spec and use it as an argument:

```
⊢def  ∀ input_I input_R demapper_mod2.
      dqam_mod2_spec input_I input_R demapper_mod2 =
      dqam_spec input_I demapper_mod2 ∧ dqam_spec input_R demapper_mod2
```

Here, input_I and input_R are defined as bool word and demapper_mod2 is a function defined as $real \rightarrow boolword$. The model uses conjunction of two demodulator specification to pass the input arguments to be checked against the table implemented. The hierarchical nature of hardware design benefits the verification in the same way it does the former.

4.4.4 Demodulator Verification

In this section we verify the demodulator block by proving that the specification implies the implementation. Our reference is Table 4.3 and we check the implementation against it. Like before, we constrain input to 16 bits because it will always be of constant length; whereas we constrain `out` to 3 bits since each demodulator instantiation contributes exactly half of the bits of the whole RTL block. We use the theories—*realTheory* and *wordTheory*, to formulate our goal. Initially we set a goal to verify the demodulator:

```
∀ input out.
      ((WORDLEN input = 16) ∧
       (WORDLEN out = 3))    ⟹
                (dqam_imp input out ⟹ dqam_spec input (λ mapreal. out));
```

The goal states that for all input and output, the demodulator implementation implies that it will return the same output as stated in the VHDL code, and the same output should be received by the demodulator specification from the demapping table for the same input. We prove this goal using the existing tactics of HOL and some lemmas built on top of existing theories. To complete the verification we state our final goal to verify the instantiations of the demodulator blocks:

```
∀ input_i input_r out.
      ((WORDLEN input_i = 16) ∧
       (WORDLEN input_r = 16) ∧
       (WORDLEN out = 6))  ⟹
            (dqam_mod2_imp input_i input _r out ⟹
                           dqam_mod2_spec input_i input_r  (λ mapreal.out))‘;
```

The goal above constrains both `input_i` and `input_r` to 16 bits and `out` to 6 bits. Both of the constraints are logically sound as the design does not have those parameters with any other word size. The function `dqam_mod2_imp` takes `input_i` and `input_r` as arguments and returns the corresponding mapping to boolean words. This boolean word must match the one returned by `dqam_mod2_spec` and thus establishes the implication. We prove this goal by rewriting using `dqam_spec` and other existing tacticals and theorems. By proving this goal we formally establish the functional correctness of demodualtor block.

4.5 Discussion

The modeling, specification and verification done above for the OFDM RTL blocks demonstrate a way to incorporate formal methods in the verification of digital systems. We have described the implementation of the RTL blocks in HOL using formal logic. For the QAM block, it was straightforward to embed the if-then-else HDL code in HOL and the specification is obtained from IEEE 802.11 specification. Although the demodulator block has a similar implementation and its formal description was similar to the QAM block, but finding a specification to check the design could not be done using IEEE standard since this block resides in the receiver side and the designer has the freedom to choose any way to implement it. Both the specifications for QAM and demodulator are based on look-up tables and the implementations were proved against those. For the *S/P* and *P/S* blocks, the specifications and implementations were also formalized after much consideration about the VHDL *PROCESS*. The verification of all blocks were done using existing theories in HOL on real numbers, natural numbers, boolean logic, lists, words and others. Many lemmas were proved in order to aid the proof steps. Some lemmas were very trivial but HOL requires each and every proof step to be sound and complete and that is why there is no ambiguity in the HOL proof. The built-in rewriting tactics `RW_TAC` was heavily used with the powerful simplification sets augmented with the required lemmas and theorems. In most cases, the proof strategy starts with a rough proof sketch by hand and then formalized in HOL. But, some lemmas and intermediate theorems were simple enough to not resort to this approach.

The main purpose for using formal verification was to find bugs in the design. We did not find any bug in the blocks. But, some comments are in order. Namely, for the QAM block, it is given in the standard that the input for a 64-QAM modulation must follow the constellation diagram shown in Figure 4.1. The constellation gives output between -7 to 7 but the implementation used 16 bit 2's complement number to represent these numbers while 3 bits would have done the same job. If the standard is followed exactly, then this issue might have resulted in a bug in the design. But, the standard gives some flexibility to the designers in order to have more precise results from the IFFT block, as explained before in Section 4.1.2. As, we were aware about it at the time of verification, we constrained the implementation using the proper number of bits. The same comments are applied to the DQAM block. For the rest of the blocks we did not find any issue like this.

A pertinent question can be raised about the higher-order logic used for the modeling and verification of OFDM that - whether first-order logic can also be used for this purpose. The reason is of course automation of proofs and completeness in some cases. It is mentioned in Chapter 3 that higher-order logic is expressive and the variables in this logic can be functions and predicates those in turn can take functions and predicates as arguments and return them too. Whereas first-order logic can only quantify over objects and variables. For the design verified none of the RTL blocks can be specified or verified using first-order logic fully. For instance, the QAM block cannot be modeled completely using first-order logic, although the implementation of the block—a pure combinational logic circuit—can be modeled in first-order since simple predicate logic is used. But, the instantiated specification of QAM block needs universal quantification on functions which were used to access the tables of the I and Q values for modulation. For, the S/P block, first-order logic cannot be used due to the use of existential quantification on the shift register in the formal modeling of the implementation. The same can be told for the P/S block. The implementation of the DQAM block can be modeled using first-order logic but the instantiation of the demapping function for modeling the decision region for specification needs to be universally quantified.

There are other blocks in the OFDM that we did not verify; namely, guard interval insertion and guard interval removal. The reason is that the RTL codes for those blocks were not available for the design at hand. The guard insertion block in the transmitter side has a portion of its behavioral code but the whole code mostly contains port-mapping [7] to the IP blocks. In general, the whole design contains many IP blocks and thus the verification of the design in its entirety is not practical using any theorem-proving tool like HOL. Still, this chapter demonstrates the scope and feasibility of formal methods in a comprehensive way in parts of the OFDM RTL blocks.

Chapter 5

Error Analysis of OFDM Modem

Digital systems are approximate models of its analog counterpart. They give more control over the output, programmable and have short product cycle. The implementation of any system in digital domain is the closest possible imitation using the resources and expertise available. The design process starts with simulation to analyze the effect of different parameters without considering finite precision arithmetic—the ultimate form of realization. A designer is very careful in choosing number of bits to represent the hardware he/she is going to design, but still various errors do show up due to finite word-length. This contributes to errors while converting designs from real to floating-point and lastly to fixed-point domain. This section describes such error analysis of OFDM modem in a formal way. Mainly we focus on the two computational blocks of the design—FFT and IFFT. Both blocks are probably the most widely used DSP cores in the digital world and considered as *raison d'être* of OFDM system. We use HOL to model the computational blocks and the accumulated errors due to the conversion from one domain to the next using different established theories and lemmas built on top of it. Such formalization of error analysis demonstrates the feasibility of the DSP framework developed by [1] in larger application domain.

5.1 Preliminaries

5.1.1 Finite Word-Length Effect and Error Analysis

Any implemented DSP system suffers from the finite word-length constraint which manifests in many ways. When the signal is converted to digital form, the restriction posed by the precision bits degrade the original signal and add errors. Even when the signal is being processed, the arithmetic errors due to the precision of processor add to error. Lastly, when the signal is converted back from digital sequence to analog form, yet another round of error is added. These errors can be categorized as quantization error. For example, given two m bit numbers—b_0 and b_1, multiplying them results in a number $2m$ bit long. Since most of the DSP operations are iterative, say for filters, if the resulting $2m$ bit number is multiplied with another m bit number then will generate some number of $3m$ bits and so on. In this case, truncation and rounding is done in order to use the limited resource of registers to store the increasing length of bits. Both fixed-point and floating-point arithmetic have different ways to handle overflow and round-off error. Designers put certain restrictions in terms of the input that can be used in order to get optimum output and thus categorizes a single product in multiple versions for different user need based on precision and performance. Finite word-length effect is a phenomena which can only be handled, but cannot be eliminated, by increasing the register length. But, such measure cannot be of infinite length since we consider here realizable system not the hypothetical ones. Error analysis shows how a system behaves in certain domain because errors due to limited precision are non-linear and nonlinearity can lead to instability, which is beyond the scope this thesis.

5.1.2 Fast Fourier Transform (FFT)

A prelude to discrete Fourier transform (DFT) is needed before we introduce fast Fourier transform or commonly known with its acronym FFT. The DFT is a mathematical procedure used to determine the harmonic content of a discrete signal sequence. Its origin is continuous Fourier transform but digital computer has helped to define the notation in discrete frequency domain sequence as

$$A(p) = \sum_{n=0}^{N-1} x(n)\ e^{-j2\pi np/N} \tag{5.1}$$

where $x(n)$ is a discrete sequence of time domain sampled values of the continuous variable $x(t)$ and $j = \sqrt{-1}$. Normally, $e^{-j2\pi np/N}$ is written as $(W_N)^{np}$ for brevity, where $(W_N) = e^{-j2\pi/N}$, which is complex roots on unity and called as twiddle factors. For N input time domain sample values, the DFT determines the spectral content of the input at N equally spaced frequency points. This complicated, although straightforward, looking DFT equation is equally inefficient for larger DFT points. Evaluation of those sums would take $O(n^2)$ arithmetic operations which means it has quadratic complexity. In 1965, Cooley and Tuckey [14] described an efficient algorithm to implement DFT, now known as FFT. FFT can compute the same result in $O(n\ log\ n)$ operations and thus its computation complexity is linearithmic. We take radix-2 FFT algorithm as an example to illustrate the concept. In radix-2 the DFT size has to be an integral power of two, say, $N = 2^m$ and m is a positive integer. If we want to calculate an 8-point DFT, then according to Equation (5.1) we have to perform N^2 or 64 complex multiplications. For the same result using FFT the number of complex multiplications to be performed can be approximated as:

$$\frac{N}{2}\ .\ \log_2 N$$

This is a significant reduction comparing with the DFT computation for large number of N. When N=8192, the DFT must calculate 1000 complex multiplications for each complex multiplication in the FFT [41]. FFT can be represented best using a signal flow graph [44] that follow four rules—(1) signals travel along branches only in the direction of the arrows; (2) a signal travelling along any branch is multiplied by the transmission of that branch; (3) the value of any node variable is the sum of all signals entering the node; (4) the value of any node variable is transmitted on all branches leaving that node. Such a graph for an 8-point DFT is shown in Figure 5.1. Every node in the figure corresponds to a 2-point DFT. FFT structures can be decimation-in-time (DIT) and decimaion-in-frequency (DIF). DIT coresponds to having the twiddles before the 2-point DFT node points, while DIF corresponds to having the twiddles after the two-point DFT. We explain only the DIF structure, which can be easily extended to develop the DIT structure [52].

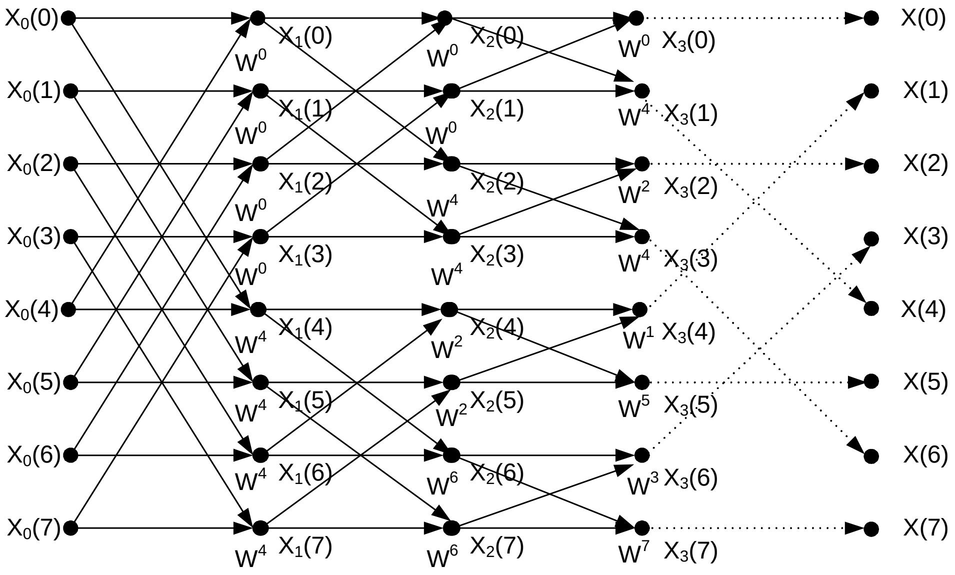

Figure 5.1: An 8-point Radix-2 FFT Signal Flow Graph

To illustrate DIF, we define Equation (5.1) using the twiddle factors in following manner

$$A(p) = \sum_{n=0}^{N-1} x(n)\ (W_N)^{np} \tag{5.2}$$

Let each integer $p,\ p = 0, 1, 2, \dots, N-1$, be expanded into a binary form as

$$p \;=\; 2^{m-1}p_0 + 2^{m-2}p_1 + \cdots + 2p_{m-2} + p_{m-1}, \qquad p_k = 0 \text{ or } 1 \tag{5.3}$$

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

$$p^* \;=\; 2^{m-1}p_{m-1} + 2^{m-2}p_{m-2} + \cdots + 2p_1 + p_0 \tag{5.4}$$

and $\{A_k(p)\}_{p=0}^{N-1}$ denote the N complex numbers calculated at the kth step. Then the DIF FFT algorithm can be expressed as [36]

$$A_{k+1}(p) = \begin{cases} A_k(p) + A_k(p + 2^{m-1-k}) & \text{if } p_k = 0 \\ [A_k(p - 2^{m-1-k}) - A_k(p)]\ w_k(p) & \text{if } p_k = 1 \end{cases} \tag{5.5}$$

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

$$z_k(p) = 2^k\ (2^{m-1-k}p_k + 2^{m-2-k}p_{k+1} + \cdots + 2p_{m-2} + p_{m-1}) - 2^{m-1}p_k \tag{5.6}$$

Equation (5.5) is carried out for $k = 0, 1, 2, \dots, m-1$, with $A_0(p) = x(p)$. It can be shown that at the last step $\{A_m(p)\}_{p=0}^{N-1}$ is the discrete Fourier coefficients in rearranged order [20]. Specifically, $A_m(p) = A(p^*)$ with p and p^* expanded and defined as in Equations 5.3 and 5.4, respectively. We will see how we extend this DIF structure of radix-2 to radix-4 64-point FFT in a later section in order to model the FFT used in the OFDM design under verification.

5.1.3 Inverse FFT

Inverse fast Fourier transform, later written as IFFT, is used to convert signals from frequency domain to time domain. For this, the existing FFT algorithm can be used with little modification. We explain two methods to carry out such computation [41].

Inverse FFT Method I

Equation (5.1) can be rewritten for IFFT as

$$x(n) = \frac{1}{N} \sum_{p=0}^{N-1} A(n)(W_N)^{-np} \tag{5.7}$$

The only changes made are, multiplying the FFT equation with $\frac{1}{N}$ and changing the sign of the twiddle factor. In the first approach, we take complex conjugate of both sides of Equation (5.7) to give us

$$x^*(n) = \frac{1}{N} \left[\sum_{p=0}^{N-1} A(n)(W_N)^{-np} \right]^* \tag{5.8}$$

Now, we apply the properties of complex numbers— *"the conjugate of a product is equal to the product of the conjugates"*. That is, if $c = ab$, then $c^* = (ab)^* = a^*b^*$. Applying this property to the right hand side of Equation (5.8) we can show that

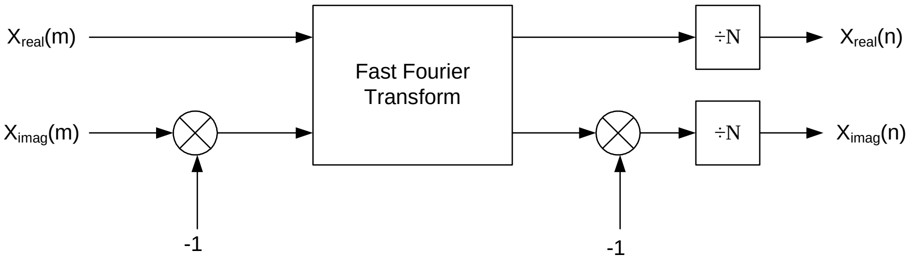

Figure 5.2: Method I for IFFT Calculation

$$\begin{aligned} x^*(n) &= \frac{1}{N} \sum_{p=0}^{N-1} A(n)^*((W_N)^{-np})^* \\ &= \frac{1}{N} \sum_{p=0}^{N-1} A(n)^*(W_N)^{np} \end{aligned} \tag{5.9}$$

There is a similarity between the original forward DFT expression in Equations (5.1) and 5.9. By performing a forward DFT on the conjugate of the $A(p)$ in Equation (5.9), and divide the result by N, we get the conjugate of desired time samples $x(n)$. Taking the conjugate of both sides of last equation gives a more straightforward expression for $x(n)$

$$x(n) = \frac{1}{N} \left[\sum_{p=0}^{N-1} A(n)^*(W_N)^{np} \right]^* \tag{5.10}$$

Figure 5.2 illustrates this method.

Inverse FFT Method II

The second method does not apply the conjugation rather it uses very clever FFT scheme. The real and imaginary parts of the complex data sequence are swapped to accomplish the act. Figure 5.3 shows the idea. To see why this process works, we look at the DFT Equation (5.1) again while separating the input $A(p)$ term into real and imaginary parts and remembering the Euler's formula from complex analysis that, $e^{j\theta} = \cos\theta + j\sin\theta$. This time we dont use the twiddle factor verson of inverse DFT. We expand Equation (5.1) into real and imaginary parts

$$\begin{aligned} x(n) &= \frac{1}{N}\sum_{p=0}^{N-1} A(p)e^{j2\pi np/N} \\ &= \frac{1}{N}\sum_{p=0}^{N-1}[A_{real}(p) + jA_{imag}(p)][\cos(2\pi np/N) + j\sin(2\pi np/N)] \end{aligned} \tag{5.11}$$

Multiplying the complex terms in Equation (5.11) gives us

$$\begin{aligned} x(n) = \frac{1}{N}\sum_{p=0}^{N-1}&[A_{real}(p)\cos(2\pi np/N) - A_{imag}(p)\sin(2\pi np/N)] \\ &+ j[A_{real}(p)\sin(2\pi np/N) + A_{imag}(p)\cos(2\pi np/N)] \end{aligned} \tag{5.12}$$

With $A(p) = A_{real}(p) + jA_{imag}(p)$, then swapping these terms gives us

$$A_{swap}(p) = A_{imag}(p) + jA_{real}(p) \tag{5.13}$$

The forward DFT of $A_{swap}(p)$ is

$$\frac{1}{N}\sum_{n=0}^{N-1}[A_{imag}(p) + jA_{real}(p)][\cos(2\pi np/N) - j\sin(2\pi np/N) \tag{5.14}$$

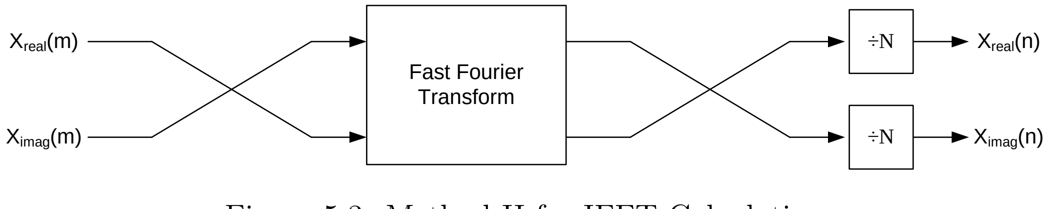

Figure 5.3: Method II for IFFT Calculation

Multiplying the complex terms in Equation (5.14) gives us

$$\begin{aligned} x(n) = \frac{1}{N} \sum_{n=0}^{N-1} &[A_{imag}(p)\cos(2\pi np/N) + A_{real}(p)\sin(2\pi np/N)] \\ &+ j[A_{real}(p)\cos(2\pi np/N) - A_{imag}(p)\sin(2\pi np/N)] \end{aligned} \tag{5.15}$$

Swapping the real and imaginary parts of the result and dividing the output by N gives us the desired IFFT result, which is exactly equal to Equation (5.12).

$$\begin{aligned} x(n) = \frac{1}{N} \sum_{n=0}^{N-1} &[A_{real}(p)\cos(2\pi np/N) - A_{imag}(p)\sin(2\pi np/N)] \\ &+ j[A_{imag}(p)\cos(2\pi np/N) + A_{real}(p)\sin(2\pi np/N)] \end{aligned} \tag{5.16}$$

Above computation method for IFFT saves both space and resources in any hardware implementation by reusing the FFT block itself. Figure 5.3 illustrates the method graphically.

5.2 FFT-IFFT Combination as a Model for Error Analysis

In this section we describe the combination of FFT and IFFT which forms the basis for error analysis of the OFDM modem. Before that, we explain on what basis we take FFT-IFFT combination as the model for the error analysis of OFDM modem. The design at hand has eight main blocks—quadrature amplitude modulator, demodulator, parallel to serial converter, serial to parallel converter, FFT and IFFT. Among all the blocks only FFT and IFFT are computational blocks which does arithmetic operation. Other blocks carry out merely mapping operations of bits from one domain to another. For example, the quadrature amplitude modulator performs arithmetic operation in theory, but the digital implementation of this modulator is just a look-up table by mapping incoming data to some other. No arithmetic operation is involved during the mapping process and so no error occurs in this block. For the demodulator, there are comparison operations preceding a look-up table and this also does not generate any error. The parallel to serial and serial to parallel block latches data for certain clock cycles and outputs data either bit by bit or in parallel fashion without altering the data by means of any computation. But, this does not imply that there is no error involved in the communication system. Of course, parameters like bit error ratio (BER) in QAM or error in transmission and receiving all are still involved but has no effect in the hardware design and not a point of interest in the current discussuion. We only concentrate here in the error issues related with the computation inside any block pertaining to finite word-length, as described in Section 5.1.1. Every block has floating-point or fixed-point data as input and output, but the resolution of the number used is sufficient to handle all possible input as well as output, so there cannot be any underflow or overflow. Moreover, the operation inside the blocks does not alter the data except mapping it to different domain. But, the same cannot be told about the FFT and IFFT blocks. Both have same types of data as input or output, but the multiplication and addition operation causes overflow or underflow. Converting from floating-point to fixed-point gives rise to such phenomena and the lost precision contributes to the error. Bolstered by above justification, we continue the error analysis with FFT-IFFT combination. The combination is very simple to form and we use Equation (5.1) and 5.7 in a nested format to do that. We apply IFFT on $x(n)$ since inverse transform precedes FFT in the OFDM modem, and then we compute FFT on the result,

$$\begin{aligned} x(n) &= \sum_{n=0}^{N-1}\left[\frac{1}{N}\sum_{p=0}^{N-1} x(n)(W_N)^{-np}\right](W_N)^{-np} \\ &= \frac{1}{N}\sum_{n=0}^{N-1}\left[\sum_{p=0}^{N-1}\left[x(n)(W_N)^{-np}\right]\right](W_N)^{-np} \end{aligned} \tag{5.17}$$

Above, we apply only the basic equation to substitute corresponding values. Later, we use Equation (5.17) to derive more complicated nesting for error analysis.

5.3 Abstract Modeling of FFT-IFFT

In this section we develop the equations behind FFT-IFFT combination in order to model them in HOL. We make extensive use of the equations developed above. In OFDM, 64-point radix-4 FFT/IFFT is used. In the later text, FFT and IFFT can be understood interchangeably. The reason why we did not formally verify the RTL design of FFT is due to impracticability of its verification using HOL. The design uses a Xilinx high performance 64-point complex FFT/IFFT IP core which has codes comprised of more than 300 pages spanning multiple files. In comparison with previous IP cores of FFT this version is more optimized but distributed in nature since it uses many base components shared by many other IP blocks of Xilinx. This adds to the complexity of embedding such huge amount of design in HOL with networks of entities, architectures, port-mapped components, procedures and so on. So, we stick to the theoretical derivation of FFT and IFFT. Figure 5.4 shows the FFT-IFFT combination based on the discussion in former sections. We are to derive the equations for this system. We rewrite Equation (5.3) in terms of radix-4,

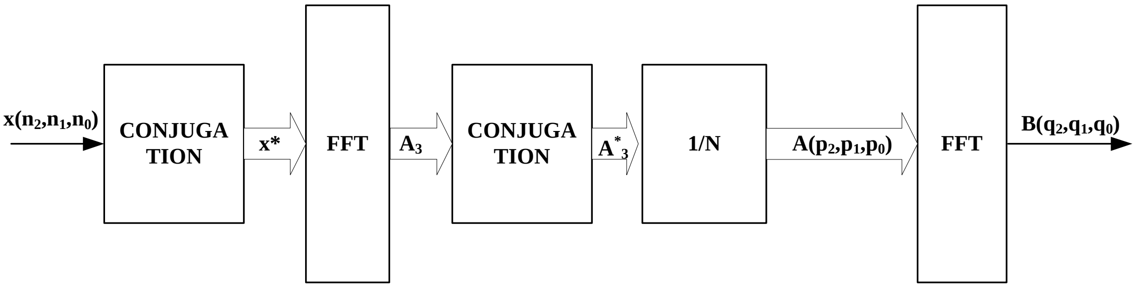

Figure 5.4: Construction of FFT-IFFT

$$p = 4^{m-1}p_0 + 4^{m-2}p_1 + \cdots + 4p_{m-2} + p_{m-1}, \quad p_k = 0,\ 1,\ 2,\ 3 \tag{5.18}$$

Since 64-point FFT is used, writing Equation (??) in terms of $N = 64 = 4^3$ gives us

$$p = p_0 + 4p_1 + 16p_2 \quad \text{where } p_2, p_1, p_0 = 0,\ 1,\ 2,\ 3 \tag{5.19}$$

The n can also be written in exactly same way

$$n \;=\; n_0 + 4n_1 + 16n_2 \quad \text{where } n_2, n_1, n_0 = 0,\ 1,\ 2,\ 3 \tag{5.20}$$

The derivation is composed of three nested summation. When input signal is $x(n_2, n_1, n_0)$, then the conjugate of it is written as $x^*(n_2, n_1, n_0)$. This conjugated signal is the innermost loop in the derivation and we apply FFT based on Equation (5.1) which results in $A_1(p_0, n_1, n_0)$. A_1 acts as input of the middle summation and the result, $A_2(p_0, p_1, n_0)$, in turn is the input of exterior summation. W_{64} is the twiddle factor due to $N = 64$. Now, we take a look at the formulation

$$\begin{aligned}
A_1(p_0, n_1, n_0) &= \sum_{n_2=0}^{3} x^*(n_2, n_1, n_0)(W_{64})^{16p_0n_2} \\
A_2(p_0, p_1, n_0) &= \sum_{n_1=0}^{3} A_1(p_0, n_1, n_0)(W_{64})^{(4p_1+p_0)4n_1} \\
A_3(p_0, p_1, p_2) &= \sum_{n_0=0}^{3} A_2(p_0, p_1, n_0)(W_{64})^{(16p_2+4p_1+p_0)n_0} \\
A(p_2, p_1, p_0) &= \frac{1}{64} A_3^*(p_0, p_1, p_2)
\end{aligned} \tag{5.21}$$

Expanding the term “A” gives us the full expression of IFFT,

$$A(p_2, p_1, p_0) = \sum_{n_0=0}^{3} \sum_{n_1=0}^{3} \sum_{n_2=0}^{3} x(n_2, n_1, n_0)(W_{64})^{-[16p_0n_2+(4p_1+p_0)4n_1+(16p_2+4p_1+p_0)n_0]} \tag{5.22}$$

The same derivations are used in a different way to develop radix-4 64-point FFT equations. In lieu of n, q is used for this derivation-

$$q \;=\; q_0 + 4q_1 + 16q_2 \quad \text{where } q_2, q_1, q_0 = 0,\ 1,\ 2,\ 3 \tag{5.23}$$

We formulate FFT as Equation (5.24) and the input is $A(p_2, p_1, p_0)$ which according to Figure 5.4 accepts the output of IFFT as its input and apply the fast fourier transform. This justifies why we share the variable p in both derivation

$$\begin{aligned}
B_1(q_0, p_1, p_0) &= \sum_{p_2=0}^{3} A(p_2, p_1, p_0)(W_{64})^{16q_0p_2} \\
B_2(q_0, q_1, p_0) &= \sum_{p_1=0}^{3} B_1(q_0, p_1, p_0)(W_{64})^{(4q_1+q_0)4p_1} \\
B_3(q_0, q_1, q_2) &= \sum_{p_0=0}^{3} B_2(q_0, q_1, p_0)(W_{64})^{(16q_2+4q_1+q_0)p_0} \\
B(q_2, q_1, q_0) &= B_3(q_0, q_1, q_2)
\end{aligned} \tag{5.24}$$

Expanding the term "B" gives the expanded version of IFFT with all the summations

$$B(q_2, q_1, q_0) = \sum_{p_0=0}^{3}\sum_{p_1=0}^{3}\sum_{p_2=0}^{3} A(p_2, p_1, p_0)(W_{64})^{-[16q_0n_2+(4q_1+q_0)4p_1+(16q_2+4q_1+q_0)p_0]} \tag{5.25}$$

Rewriting Equation (5.25) with Equation (5.24) we get the full expression of FFT-IFFT combination

$$\begin{aligned}
B(q_2, q_1, q_0) = \frac{1}{64}&\sum_{p_0=0}^{3}\sum_{p_1=0}^{3}\sum_{p_2=0}^{3}\sum_{n_0=0}^{3}\sum_{n_1=0}^{3}\sum_{n_2=0}^{3} x(n_2, n_1, n_0) \\
&(W_{64})^{-[16p_0n_2+(4p_1+p_0)4n_1+(16p_2+4p_1+p_0)n_0]} \\
&(W_{64})^{[16q_0n_2+(4q_1+q_0)4p_1+(16q_2+4q_1+q_0)p_0]}
\end{aligned} \tag{5.26}$$

The above equation looks clumsy and verbose, we rewrite it as following

$$B(q_2, q_1, q_0) = \frac{1}{64}\sum_{\mathbf{p}}\sum_{\mathbf{n}} x(n_2, n_1, n_0)(W_{64})^{(L-M)}$$

where,

$$\begin{aligned}
\sum_{\mathbf{p}} &= \sum_{p_0=0}^{3}\sum_{p_1=0}^{3}\sum_{p_2=0}^{3} \\
\sum_{\mathbf{n}} &= \sum_{n_0=0}^{3}\sum_{n_1=0}^{3}\sum_{n_2=0}^{3} \\
L &= 16q_0n_2 + (4q_1 + q_0)4p_1 + (16q_2 + 4q_1 + q_0)p_0 \\
M &= 16p_0n_2 + (4p_1 + p_0)4n_1 + (16p_2 + 4p_1 + p_0)n_0
\end{aligned} \tag{5.27}$$

With this final equation we finish all the required mathematics needed to develop the formal modeling of the error analysis. The idea is to apply inverse fast fourier transform on one signal and then the output is applied with fast fourier transform. In the ideal case, the final output should be equal to input. But, in real implementation it is never the case and that is the topic of discussion in the text to follow.

5.4 Modeling of FFT-IFFT Combination in Different Number Domains

In this section we explain the model developed in former section in real, floating-point and fixed-point domain. Illustrating the model in one number system can easily be extended for others.

We start with the Real number domain. The signal $x(n)$ and twiddle factor W_{64} are complex numbers and can be written in terms of real and imaginary component of theirs. In Equation (5.27) these two functions are multiplied with each other. From the basic properties of complex numbers, we know that if two complex numbers $a + jb$ and $c + jd$ are multiplied with each other, then the resulting complex number can be written as $m + jn$ where, $m = ac - bd$ and $n = ad + bc$ due to the properties of $j = \sqrt{-1}$. Based on this discussion, we denote the real and imaginary part of $x(n)$ and W_{64} like this

$$C_0 = Re[x] \tag{5.28}$$

$$D_0 = Im[x] \tag{5.29}$$

$$U_{64} = Re[W_{64}] \tag{5.30}$$

$$V_{64} = Im[W_{64}] \tag{5.31}$$

The notations C_0 and U_{64} denote the real parts; D_0 and V_{64} denote the imaginary parts of $x(n)$ and W_{64}, respectively. Next, we denote $C(q_2, q_1, q_0)$ as the real part and $D(q_2, q_1, q_0)$ as the imaginary part of $B(q_2, q_1, q_0)$ and rewrite the equation as following,

$$C(q_2, q_1, q_0) = \frac{1}{64} \sum_{\mathbf{p}} \sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(U_{64})^{(L-M)} - D_0(n_2, n_1, n_0)(V_{64})^{(L-M)} \tag{5.32}$$

$$D(q_2, q_1, q_0) = \frac{1}{64} \sum_{\mathbf{p}} \sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(V_{64})^{(L-M)} + D_0(n_2, n_1, n_0)(U_{64})^{(L-M)} \tag{5.33}$$

Mimicking the analysis of real numbers we ought to define the equations for floating-point and fixed-point number and state $fl(.)$ and $fxp(.)$ as floating-point and fixed-point respectively. The characters prime and double primes are used to point to floating-point and fixed-point numbers and we will stick to this convention in any analysis set forth. Using these notations we state the following short-hands,

$$C_0^{'} = fl(Re[x]) \tag{5.34}$$

$$C_0^{''} = fxp(Re[x]) \tag{5.35}$$

$$D_0^{'} = fl(Im[x]) \tag{5.36}$$

$$D_0^{''} = fxp(Im[x]) \tag{5.37}$$

Unlike $Re[.]$ and $Im[.]$, the notations $fl(.)$ and $fxp(.)$ do not extract any real and imaginary parts, rather they convert the number to the nearest floating and fixed point, respectively. A question of precision might be arisen and that is the core of this chapter which we purport to explain. We can now write Equation (5.27) in terms of floating-point and fixed-point number using the equations (5.34) to (5.37)

$$C^{'}(q_2, q_1, q_0) = fl\left(\frac{1}{64} \sum_{\mathbf{p}} \sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(U_{64})^{(L-M)} - D_0(n_2, n_1, n_0)(V_{64})^{(L-M)}\right) \tag{5.38}$$

$$D^{'}(q_2, q_1, q_0) = fl\left(\frac{1}{64} \sum_{\mathbf{p}} \sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(V_{64})^{(L-M)} + D_0(n_2, n_1, n_0)(U_{64})^{(L-M)}\right) \tag{5.39}$$

$$C^{''}(q_2, q_1, q_0) = fxp\left(\frac{1}{64}\sum_{\mathbf{p}}\sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(U_{64})^{(L-M)} - D_0(n_2, n_1, n_0)(V_{64})^{(L-M)}\right) \tag{5.40}$$

$$D^{''}(q_2, q_1, q_0) = fxp\left(\frac{1}{64}\sum_{\mathbf{p}}\sum_{\mathbf{n}} C_0(n_2, n_1, n_0)(V_{64})^{(L-M)} + D_0(n_2, n_1, n_0)(U_{64})^{(L-M)}\right) \tag{5.41}$$

where, $C^{'}$ and $D^{'}$ are the floating-point equivalents of C and D; and $C^{''}$ and $D^{''}$ are the fixed-point equivalents of C and D.

5.5 Error Analysis of FFT-IFFT Combination

In this section we discuss the error analysis of FFT-IFFT combination. We start with an introduction of floating-point and fixed-point error model and then introduce error in the models developed in previous section. In order to derive the error while converting from real number to floating-point, the two corresponding equations are subtracted and for the real to fixed-point conversion the error analysis is the same. It is also required to analyze the error incurred in conversion between floating-point number to fixed-point and that can be calculated using the direct equations as done above or subtracting the result of the real to floating-point and real to fixed-point.

5.5.1 Error Analysis Models

Floating-Point Error Model

In analyzing the effect of floating-point roundoff, the effect of rounding is presented multiplicatively. The following theorem is the most fundamental in floating-point rounding theory which gives a convenient expression for the relative error committed if a given real number x is rounded to the closest floating-point number x_R [59, 18, 1].

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

Here, p is the precision of the floating-point format. Now, we apply this theorem to the arithmetic operations. Let $*$ denote any of the operations $+$, $-$, $\times$, $\div$ and we use $fl(.)$ to state that,

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

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. A point to note is that the error accumulates for any floating-point operation multiplicatively and this is what makes the final accumulated error a large magnitude.

Fixed-Point Error Model

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 can be stated as [59, 18, 1],

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

and $fracbits$ is the number of bits that are to the right of the binary point in the given fixed-point format. The fractional X-bit two's complement number representation evenly distributes 2^X quantization levels between -1 and $1 - 2^{-(X-1)}$. The spacing between quantization levels is then,

$$\frac{2}{2^X} = 2^{-(X-1)} = \Delta_X$$

Any signal value falling between two levels is assigned to one of the two levels. Now, as before we state $*$ to denote any of the fixed-point arithmetic operations, $+$, $-$, $\times$

or ÷ with a given format X. Then

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

5.5.2 Introducing Error in Design

In this section we analyze the error introduced in the floating-point and fixed-point design only. The real part of floating-point, C' , can be written with all the errors due to floating-point round-off as follows,

$$\begin{aligned} C^{'}(q_2,q_1,q_0) \ = \ & \frac{1}{64}\left[\sum_{\mathbf{p}}\sum_{\mathbf{n}}\Big(\Big(C_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)}\right. \\ & (1+\delta_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0})\Big)- \\ & \Big(D_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)} \\ & (1+\epsilon_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0})\Big)\Big) \\ & (1+\xi_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0}) \\ & \left.\prod_{\substack{i=1024p_2+256p_1\\+64p_0+16n_2\\+4n_1+n_0}}^{4095}(1+\lambda_i)\right](1+\tau)(1+\rho) \end{aligned} \tag{5.42}$$

where $\boldsymbol{\delta}$ it accounts for the round-off error due to multiplication of $C_0^{'}$ and $(U_{64})^{(L-M)}$ according to Theorem 1. The function $\boldsymbol{\epsilon}$ represents the error due to the round-off error after the multiplication of $D_0^{'}$ and $(V_{64})^{(L-M)}$. The error due to the subtraction of $[C_0^{'}(U_{64})^{(L-M)} - D_0^{'}(V_{64})^{(L-M)}]$ is represented using $\boldsymbol{\xi}$. Based on the errors due to one single iteration, error due to the two summations $\sum_{\mathbf{p}}\sum_{\mathbf{n}}$ (which is an abbreviation for six summations— $\sum_{p_0=0}^{3}\sum_{p_1=0}^{3}\sum_{p_2=0}^{3}\sum_{n_0=0}^{3}\sum_{n_1=0}^{3}\sum_{n_2=0}^{3}$) can be stated as products of $\boldsymbol{\lambda}$ where the upper index is set as 4095 due to six iterations each ranging from 0 to 3 giving $4\times4\times4\times4\times4\times4-1\ = 4095$. It should have eclipsed all the rounding errors in the whole system of equation, but still the fraction $\frac{1}{64}$ incurs two round-off errors. One of them due to the division of 1 by 64, denoted as $\boldsymbol{\tau}$ and the other is for the multiplication thereafter with the rest of the equations, denoted as $\boldsymbol{\rho}$. These errors can be generalized on the same line of reasoning for the other equations. The error related with the imaginary part $D^{'}$ of the floating-point can be written as

$$\begin{aligned} D^{'}(q_2,q_1,q_0) \;=\; & \frac{1}{64}\Bigg[\sum_{\mathbf{p}}\sum_{\mathbf{n}}\Big(\Big(C_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)} \\ & (1+\delta^{''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0})\Big)- \\ & \Big(D_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)} \\ & (1+\epsilon^{''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0})\Big)\Big) \\ & (1+\xi^{''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0}) \\ & \prod_{\substack{i=1024p_2+256p_1\\+64p_0+16n_2\\+4n_1+n_0}}^{4095}(1+\lambda_i^{''})\Bigg](1+\tau^{'})(1+\rho^{'}) \end{aligned} \qquad (5.43)$$

where, the previous function symbols used in Equation (5.42) are modified with double/single prime, namely $\boldsymbol{\delta}^{''}$, $\boldsymbol{\epsilon}^{''}$, $\boldsymbol{\xi}^{''}$, $\boldsymbol{\lambda}^{''}$, $\boldsymbol{\tau}^{'}$, $\boldsymbol{\rho}^{'}$; but the meaning remains the same. A point to emphasize is that all the error functions are in multiplication relation with the variable and this is what makes the floating-point round-off error much complicated. For the fixed-point domain, the error functions are additive as explained in Theorem 2. So, the functions are not multiplied but rather added at the end of the equation. The error functions for the real part of fixed-point number $C^{''}$ are denoted as $\boldsymbol{\delta}^{'}$, $\boldsymbol{\epsilon}^{'}$, $\boldsymbol{\xi}^{'}$, $\boldsymbol{\lambda}^{'}$,$\boldsymbol{\tau}^{''}$, and $\boldsymbol{\rho}^{''}$; while referring to the same type of error functions as discussed before.

$$
\begin{aligned}
C^{''}(q_2,q_1,q_0) \;=\; \frac{1}{64}\Bigg[\sum_{\mathbf{p}}\sum_{\mathbf{n}} & C_0^{''}(n_2,n_1,n_0)(U_{64})^{(L-M)} - D_0^{''}(n_2,n_1,n_0)(V_{64})^{(L-M)} + \\
& \delta^{'}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \epsilon^{'}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \xi^{'}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \sum_{\substack{i=1024p_2+256p_1\\+64p_0+16n_2\\+4n_1+n_0}}^{4095} \lambda_i^{'}\Bigg] + \tau^{''} + \rho^{''}
\end{aligned}
\tag{5.44}
$$

Equation (5.45), which is the imaginary part of fixed-point number is rewritten with the error functions denoted as $\boldsymbol{\delta}^{'''}$, $\boldsymbol{\epsilon}^{'''}$, $\boldsymbol{\xi}^{'''}$, $\boldsymbol{\lambda}^{'''}$,$\boldsymbol{\tau}^{'''}$, and $\boldsymbol{\rho}^{'''}$.

$$
\begin{aligned}
D^{''}(q_2,q_1,q_0) \;=\; \frac{1}{64}\Bigg[\sum_{\mathbf{p}}\sum_{\mathbf{n}} & C_0^{''}(n_2,n_1,n_0)(V_{64})^{(L-M)} - D_0^{''}(n_2,n_1,n_0)(U_{64})^{(L-M)} + \\
& \delta^{'''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \epsilon^{'''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \xi^{'''}_{1024p_2+256p_1+64p_0+16n_2+4n_1+n_0} + \\
& \sum_{\substack{i=1024p_2+256p_1\\+64p_0+16n_2\\+4n_1+n_0}}^{4095} \lambda_i^{'''}\Bigg] + \tau^{'''} + \rho^{'''}
\end{aligned}
\tag{5.45}
$$

Adding the error parameters leaves us just one step away before we start to formalize the analysis after deriving the error that occurred in the conversion from one domain to another. We start with the real to floating-point conversion and the round-off error difference between the complex floating-point implementation and complex real implementation of FFT-IFFT denoted as $e(q_2,q_1,q_0)$

$$
\begin{aligned}
e(q_2,q_1,q_0) &= [C^{'}(q_2,q_1,q_0) + jD^{'}(q_2,q_1,q_0)] - [C(q_2,q_1,q_0) + jD(q_2,q_1,q_0)] \\
&= C^{'}(q_2,q_1,q_0) - C(q_2,q_1,q_0) + j[D^{'}(q_2,q_1,q_0) - D(q_2,q_1,q_0)]
\end{aligned}
\tag{5.46}
$$

Rewriting the above equation using Equations (5.38) and (5.39) gives the expanded version Equation (5.47)

$$\begin{aligned} e(q_2,q_1,q_0) = \frac{1}{64}\Bigg[\sum_{\mathbf{p}}\sum_{\mathbf{n}}\Big(& C_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)} - D_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)} \\ & - C_0(n_2,n_1,n_0)(U_{64})^{(L-M)} + D_0(n_2,n_1,n_0)(V_{64})^{(L-M)}\Big) \\ & + j\Big(C_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)} + D_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)} \\ & - C_0(n_2,n_1,n_0)(V_{64})^{(L-M)} - D_0(n_2,n_1,n_0)(U_{64})^{(L-M)}\Big)\Bigg] \end{aligned} \tag{5.47}$$

Further rearranging in terms of $(U_{64})^{(L-M)}$ and $(V_{64})^{(L-M)}$ gives

$$\begin{aligned} e(q_2,q_1,q_0) = \frac{1}{64}\Bigg[\sum_{\mathbf{p}}\sum_{\mathbf{n}}\Bigg(& C_0^{'}(n_2,n_1,n_0) - C_0(n_2,n_1,n_0) \\ & + j\Big(-D_0^{'}(n_2,n_1,n_0) + D_0(n_2,n_1,n_0)\Big)\Bigg)(U_{64})^{(L-M)} \\ & + \Bigg(D_0^{'}(n_2,n_1,n_0) - D_0(n_2,n_1,n_0) \\ & + j\Big(C_0^{'}(n_2,n_1,n_0) - C_0(n_2,n_1,n_0)\Big)\Bigg)(V_{64})^{(L-M)}\Bigg] \end{aligned} \tag{5.48}$$

In order to have a terse format for the final error analysis, we assume

$$e_0(q_2,q_1,q_0) = C_0^{'}(n_2,n_1,n_0) - C_0(n_2,n_1,n_0) + j\Big(D_0^{'}(n_2,n_1,n_0) - D_0(n_2,n_1,n_0)\Big) \tag{5.49}$$

Using the property of complex numbers and Equation (5.49) and taking the common terms, Equation (5.48) is written in terms of e_0,

$$e(q_2,q_1,q_0) = e_0(n_2,n_1,n_0)\left[(U_{64})^{(L-M)} + j(V_{64})^{(L-M)}\right] \tag{5.50}$$

Noting that both $(U_{64})^{(L-M)}$ and $(V_{64})^{(L-M)}$ are just real and imaginary parts of $(W_{64})^{(L-M)}$ as stated in Equations (5.30) and (5.31), we can write

$$e(q_2,q_1,q_0) = e_0(n_2,n_1,n_0)(W_{64})^{(L-M)} \tag{5.51}$$

Finally, we write the error functions separately as $\mathbf{f}(\mathbf{n},\mathbf{p})$ in order to distinguish the element from the rest. Following equation expresses the round-off error accumulated due to real to floating-point conversion,

$$e(q_2,q_1,q_0)=\frac{1}{64}\left[\sum_{\mathbf{p}}\sum_{\mathbf{n}}e_0(n_2,n_1,n_0)(W_{64})^{(L-M)}+\mathbf{f}(\mathbf{n},\mathbf{p})\right] \tag{5.52}$$

where, $\mathbf{f}(\mathbf{n},\mathbf{p})$ is written according to Equations (5.42),(5.43) and (5.48)

$$\begin{aligned}\mathbf{f}(\mathbf{n},\mathbf{p}) &= C_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)}\big[(1+\delta_{(p,n)})(1+\xi_{(p,n)})\prod_{i=(p,n)}^{4095}(1+\lambda_i)(1+\tau)-1\big]\\ &- D_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)}\big[(1+\epsilon_{(p,n)})(1+\xi_{(p,n)})\prod_{i=(p,n)}^{4095}(1+\lambda_i)(1+\tau)-1\big]\\ &+ j\Bigg[C_0^{'}(n_2,n_1,n_0)(V_{64})^{(L-M)}\big[(1+\delta^{''}_{(p,n)})(1+\xi^{''}_{(p,n)})\prod_{i=(p,n)}^{4095}(1+\lambda^{''}_i)(1+\tau^{'})-1\big]\\ &- D_0^{'}(n_2,n_1,n_0)(U_{64})^{(L-M)}\big[(1+\epsilon^{''}_{(p,n)})(1+\xi^{''}_{(p,n)})\prod_{i=(p,n)}^{4095}(1+\lambda^{''}_i)(1+\tau^{'})-1\big]\Bigg]\end{aligned} \tag{5.53}$$

The two variables n and p are used for the function as a short-hand for $n=n_2,n_1,n_0$ and $p=p_2,p_1,p_0$.

The above analysis can be adopted similarly to come at the following error function, $e^{'}(q_2,q_1,q_0)$, for the round-off error due to conversion from real to fixed-point domain

$$e^{'}(q_2,q_1,q_0) \;=\; C^{''}(q_2,q_1,q_0)-C(q_2,q_1,q_0)+j\big[D^{''}(q_2,q_1,q_0)-D(q_2,q_1,q_0)\big] \tag{5.54}$$

Denoting the error as $\mathbf{f}'(\mathbf{n},\mathbf{p})$ the final error can be written as

$$e^{'}(q_2,q_1,q_0)=\frac{1}{64}\left[\sum_{\mathbf{p}}\sum_{\mathbf{n}}e_0(n_2,n_1,n_0)(W_{64})^{(L-M)}+\mathbf{f}'(\mathbf{n},\mathbf{p})\right] \tag{5.55}$$

where, $\mathbf{f}'(\mathbf{n},\mathbf{p})$ is constructed according to Equations (5.38), (5.38), (5.44), and

(5.45).

$$\begin{aligned}\mathbf{f}'(\mathbf{n},\mathbf{p}) =&\delta^{'}_{(p,n)} + \epsilon^{'}_{(p,n)} + \xi^{'}_{(p,n)} + \sum_{i=(p,n)}^{4095} \lambda^{'}_{i} + \tau^{'} \\ &+ j\left[\delta^{'''}_{(p,n)} + \epsilon^{'''}_{(p,n)} + \xi^{'''}_{(p,n)} + \sum_{i=(p,n)}^{4095} \lambda^{'''}_{i} + \tau^{'''}\right]\end{aligned} \tag{5.56}$$

Equation 5.56 is much more simplified than its real to floating-point counterpart since this error is additive but not multiplicative To derive the errors due to floating-point to fixed-point conversion we do not resort to derive those mammoth equations as above, rather we use the previous derivations. If the two error results derived previously are subtracted then the result gives the error we are looking for. Denoting this error as $e^{''}(q_2, q_1, q_0)$, it can be written as

$$e^{''}(q_2, q_1, q_0) = e^{'}(q_2, q_1, q_0) - e(q_2, q_1, q_0) \tag{5.57}$$

To summarize all the derivations above, we denote $\aleph$ as any of the three errors - $\mathbf{f}(\mathbf{n},\mathbf{p})$, $\mathbf{f}'(\mathbf{n},\mathbf{p})$ or $\mathbf{f}''(\mathbf{n},\mathbf{p})$ for $e^{''}$; and η to denote the degree of the primes of e, then it can be written

$$e^{\eta}(q_2, q_1, q_0) = \frac{1}{64}\left[\sum_{\mathbf{p}}\sum_{\mathbf{n}} e_0(n_2, n_1, n_0)(W_{64})^{(L-M)} + \aleph\right]$$

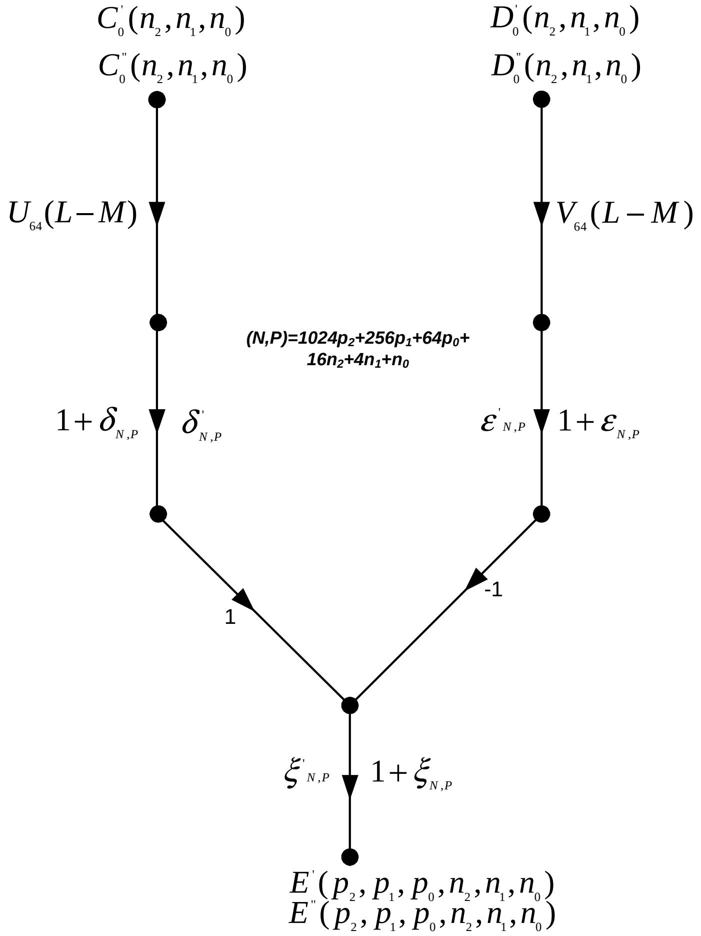

Figure 5.5: Error Flow Graph for C' and C''

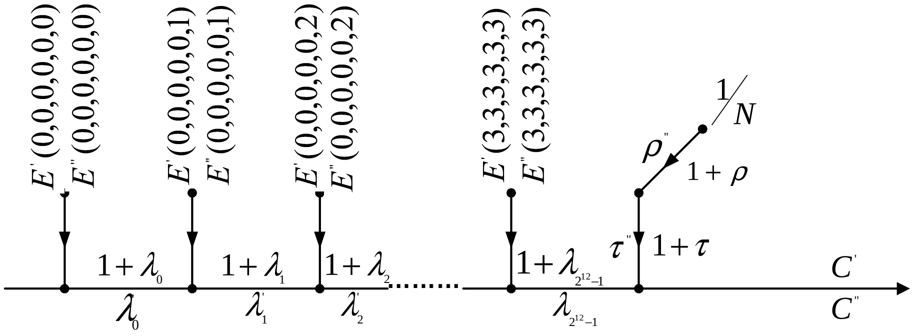

Figure 5.6: Error Flow Graph for C' and C'' contd.

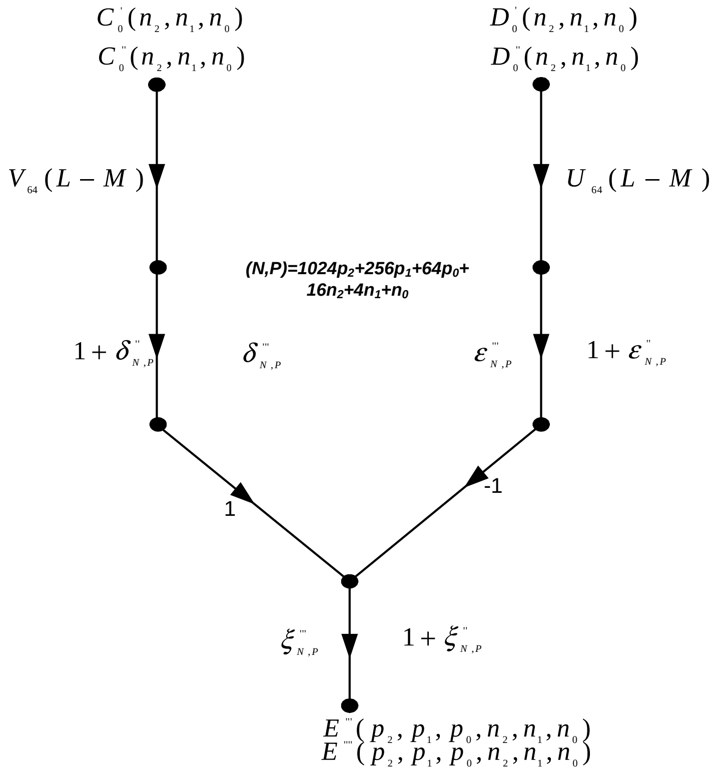

Figure 5.7: Error Flow Graph for D' and D''

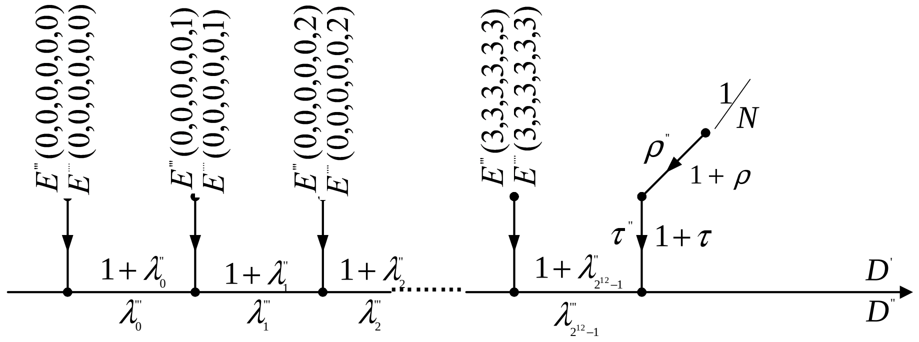

Figure 5.8: Error Flow Graph for D' and D'' contd.

Figures 5.5 to 5.7 summarize all the error analysis in a flow-graph format discussed so far. Figure 5.5 and 5.6 refer to the errors incur in the real parts of the floating-point and fixed-point model. Starting with $C_0^{'}(n_2, n_1, n_0)$ from the left branch of Figure 5.5 it is multiplied with $U_{64}(L - M)$ as shown between the edge of the first two nodes. Next, the error occurred in the previous operation is multiplied in the edge between second and third node. In the same way we can reach in the similar calculation for $D_0^{'}(n_2, n_1, n_0)$. When the two branches meet in the bottom node, they are subtracted from each other due to multiplication of the $D_0^{'}$ with -1 and this operation adds to the next error which is expressed as $1 + \xi_{N,P}$. Here, (N, P) refers to $1024p_2 + 256p_1 + 64p_0 + 16n_2 + 4n_1 + n_0$. Now, this error is labeled as $E^{'}(p_2, p_1, p_0, n_2, n_1, n_0)$.

If the same calculation is repeated for $C_0^{''}(n_2, n_1, n_0)$ and $D_0^{''}(n_2, n_1, n_0)$, the error at the end is labeled as $E^{''}(p_2, p_1, p_0, n_2, n_1, n_0)$. But, this time the error functions are all additive not multiplicative. We now look into figure 5.6 which is the continuation of figure 5.5 to define the errors related with the six summations each having four iterations. As stated before, the error here is denoted as $1 + \lambda_i$ for floating-point and λ_i for fixed-point. The error calculation starts from $E^{'}(0, 0, 0, 0, 0, 0)$ and $E^{''}(0, 0, 0, 0, 0, 0)$ and the corresponding errors are multiplied or added till $E^{'}(3, 3, 3, 3, 3, 3)$ and $E^{''}(3, 3, 3, 3, 3, 3)$. At this point, we are left with two more errors. The branch starting with node label $\frac{1}{N}$ adds errors due to division operation, which are denoted as $\rho^{'}$ and $1 + \rho$ for fixed-point and floating-point respectively. And then the same constant is multiplied with rest of the what is calculated so far and adds another error denoted as $\tau^{'}$ and $1 + \tau$ for fixed-point and floating-point respectively. In the end $C^{'}$ and $C^{''}$ are found as we calculated earlier.

The above discussion can be applied as it is for the calculation of the imaginary part of floating-point and fixed-point model as shown in Figure 5.7 and 5.8.

5.6 Formal Error Analysis in HOL

Any complex function can be modeled in HOL in a hierarchical way. It is evident from preceding sections that a hierarchical approach is followed all the way to derive the final equations keeping in mind that hardware designs are hierarchical and this analysis then can be used readily to formally analyze the accumulation of round-off error. We use different theories established in HOL to model the final error analysis. It requires to start from the construction of complex number and then complex sum in all three number domains. And, then these theories will be used to model FFT and IFFT formally at the algorithmic level, which in turn will be used to model the FFT-IFFT combination. Finally, the error analysis lemmas and theories are developed to end with the formalization.

5.6.1 Ideal Complex Number Modeling

In order to define complex numbers in HOL a mechanism exists using type bijection that defines new subtypes based on the existing ones. Such types are defined by introducing a new type constant and asserting an axiom that characterizes it as denoting a set in bijection with a non-empty subset of an existing type [32]. The complex numbers are isomorphic to $\mathbb{R} \times \mathbb{R}$, where $\mathbb{R}$ denotes real numbers. The seminal work of John Harrison [24] established the theory of real numbers in HOL. A mutually inverse type bijection of complex number is given as

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

where *complex* : $\mathbb{R}^2 \rightarrow \mathbb{C}$ and *coords* : $\mathbb{C} \rightarrow \mathbb{R}^2$. Although, all the complex number theories are well developed, it is imperative that some important definitions are mentioned again for the sake of clarity. The real and imaginary part of a complex number is defined using the *pairTheory* [33] as follows:

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

where, FST and SND extracts the first and second element of a pair, respectively.

Since, conjugation is one of the major component in IFFT we define it as `CNJ` by simply inverting the sign of imaginary part,

```
CNJ = ⊢def  ∀ z. CNJ z = complex (Re z,¬Im z)
```

The principal n-roots of unity is defined using Euler's identity which makes it very easy to implement

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

A trivial definition of the complex constant multiplied with the rest of the summations is defined as,

```
complex_64 = ⊢def  complex_64 = complex (1 / 64,0)
```

We omit to define `compadd`, `compsub`, `compmul`, `complex_add`, `complex_sub`, `complex_mul` and some other definitions to retain brevity of the thesis. Here, `compadd`, `compsub`, and `compmul` take a pair of real number and then returns a pair of real number by adding, multiplying or subtracting the numbers. Whereas, `complex_add`, `complex_sub`, and `complex_mul` uses `complex_tybij` to convert the corresponding addition, subtraction and multiplication done using real numbers, but this time the input type is of complex number rather than real. Since, `complex_sum`—a recursive definition used heavily throughout the definitions to follow, we mention it here

```
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))
```

it models the summation shown in the equations described in the previous sections, which takes a pair (n,0) for the upper and lower index and a function f. Having described the most necessary definitions we formalize the IFFT model described in Equation (5.24). In order to have a one to one relation between the mathematical symbols of the equations and HOL code, we tried utmost to use the same symbols as in the equations, although, it is not possible in some cases due to some syntactical restrictions. For the definition below, REAL_IFFT_F , the variables can be traced with the mathematical model. The let....in construct helps in local binding of the summations to express the final output in terms of A_3

```
⊢def  ∀ x. REAL_IFFT_F x = (λp2 p1 p0.
(let A1 p0 n1 n0 = complex_sum (0,4) (λn2. CNJ (x n2 n1 n0) *
     principal_root_1 (16 * & p0 * & n2) 64)
in
let A2 p0 p1 n0 = complex_sum (0,4) (λn1. A1 p0 n1 n0 *
     principal_root_1 ((4 * & p1 + & p0) * (4 * & n1)) 64)
in
let A3 p0 p1 p2 = complex_sum (0,4) (λn0. A2 p0 p1 n0 *
     principal_root_1 ((16 * & p2 + 4 * & p1 + & p0) * & n0) 64)
in
complex_64 * CNJ (A3 p0 p1 p2)))
```

Now, the following theorem is proved in order to formalize the expansion derived in Equation (5.22). The three summations of the equation are written as complex_64 with their corresponding indexes using λ abstraction. The signal x is then multiplied with the twiddle factor modeled as principal_root_1 with all the exponents consisting of p_0, p_1, p_2, n_0, n_1, n_2. This theorem will be used for rewriting as further theorems are developed.

```
REAL_IFFT_EXPAND =
⊢  ∀ A x. (A = REAL_IFFT_F x) ⟹
         ∀ p0 p1 p2. A p2 p1 p0 =
           complex_64 *
           complex_sum (0,4) (λn0.
           complex_sum (0,4) (λn1.
           complex_sum (0,4) (λn2. x n2 n1 n0 *
                 principal_root_1
                  ¬(16 * & p0 * & n2 + (4 * & p1 + & p0) * (4 * & n1) +
                    (16 * & p2 + 4 * & p1 + & p0) * & n0)) 64)))
```

The FFT mathematical model in Equation (5.24) is also written in the same way as IFFT,

```
REAL_FFT_F =
⊢def  ∀ A. REAL_FFT_F A = (λq2 q1 q0.
(let B1 q0 p1 p0 = complex_sum (0,4)(λp2. A p2 p1 p0 *
     principal_root_1 (16 * & q0 * & p2) 64)
in
let B2 q0 q1 p0 = complex_sum (0,4)(λp1. B1 q0 p1 p0 *
    principal_root_1 ((4 * & q1 + & q0) * (4 * & p1)) 64)
in
let B3 q0 q1 q2 = complex_sum (0,4)(λp0. B2 q0 q1 p0 *
    principal_root_1 ((16 * & q2 + 4 * & q1 + & q0) * & p0) 64)
in
B3 q0 q1 q2))
```

Now, Equation (5.25), the expanded version of FFT is proved as a theorem in HOL for the same purpose described before. The output A from IFFT is fed as an input of FFT. The two main building blocks of the system are at hand and these can now be used to model FFT-IFFT in HOL. According to our description both the blocks can be modeled in way of function and we define it as such:

```
⊢def  ∀ x. REAL_IFFT_FFT_F x = REAL_FFT_F (REAL_IFFT_F x)
```

If two of the previous theorems are true, then the expansion of FFT-IFFT combination should result in Equation (5.26) and it is proved as a theorem in HOL as following,

```
REAL_IFFT_FFT_EXPAND =
⊢  ∀ B x. (B = REAL_IFFT_FFT_F x) ⟹
    ∀ q0 q1 q2. B q2 q1 q0 =
     complex_64 *
     complex_sum (0,4) (λp0. complex_sum (0,4) (λp1.
     complex_sum (0,4) (λp2. complex_sum (0,4) (λn0.
     complex_sum (0,4) (λn1. complex_sum (0,4) (λn2.
       x n2 n1 n0 * principal_root_1 ¬(16 * & p0 * & n2 +
                    (4 * & p1 + & p0) * (4 * & n1) +
                    (16 * & p2 + 4 * & p1 + & p0) * & n0)) 64 *
                    principal_root_1 (16 * & q0 * & p2 +
                    (4 * & q1 + & q0) * (4 * & p1) +
                    (16 * & q2 + 4 * & q1 + & q0) * & p0) 64))))))
```

5.6.2 Real Number Modeling

For modeling the design in real numbers, we define `Z` and `OMEGA` to have a compact representation of the theorems. Both definitions take nine arguments,

```
Z = ⊢def  ∀ n0 n1 n2 p0 p1 p2 q0 q1 q2.
Z n0 n1 n2 p0 p1 p2 q0 q1 q2 =
  16 * & q0 * & p2 + (4 * & q1 + & q0) * (4 * & p1) +
  (16 * & q2 + 4 * & q1 + & q0) * & p0 −
  (16 * & p0 * & n2 + (4 * & p1 + & p0)*
  (4 * & n1)+ (16 * & p2 + 4 * & p1 + & p0) * & n0)


OMEGA = ⊢def  ∀ n0 n1 n2 p0 p1 p2 q0 q1 q2.
OMEGA n0 n1 n2 p0 p1 p2 q0 q1 q2 =
  principal_root_1 (Z n0 n1 n2 p0 p1 p2 q0 q1 q2) 64
```

The `Z` is used in lieu of the very long exponent, abbreviated as L and M is Equation 5.27. The `OMEGA` is defined using `Z` as an argument to the previously defined principal_root_1. At this point in modeling, we define C and D as stated in Equation. (5.32) and (5.33). Initially, $Re[.]$ and $Im[.]$ is used on the ideal model to extract the real and imaginary components.

```
IFFT_FFT_RE =
⊢def  ∀ x n0 n1 n2. IFFT_FFT_RE x n0 n1 n2 =
                        Re (REAL_IFFT_FFT_F x n0 n1 n2)
IFFT_FFT_IM =
⊢def  ∀ x n0 n1 n2. IFFT_FFT_IM x n0 n1 n2 =
                        Im (REAL_IFFT_FFT_F x n0 n1 n2)
```

Then the expansion of both of them is proved as theorems in HOL. An important thing to be noted is the use of sum instead of complex_sum. It is done so due to the use of either real and imaginary expansion in the theorems, unlike in previous theorems where a complex number is used. For the same reason complex_64 is replaced with inv 64 to denote a simple inversion of a natural number. The definitions for OMEGA_RE and OMEGA_IM are not shown here. The theorems REAL_IFFT_FFT_RE_EXPAND and REAL_IFFT_FFT_IM_EXPAND are proved formally for the two equations of C and D, respectively.

```
REAL_IFFT_FFT_RE_EXPAND = ⊢  ∀ x q2 q1 q0.
      IFFT_FFT_RE x q2 q1 q0 =
         inv 64 *
         sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
         sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
             Re (x n2 n1 n0) * OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2 −
             Im (x n2 n1 n0) * OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2))))))

REAL_IFFT_FFT_IM_EXPAND = ⊢  ∀ x q2 q1 q0.
      IFFT_FFT_IM x q2 q1 q0 =
         inv 64 *
         sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
         sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
             Re (x n2 n1 n0) * OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2 +
             Im (x n2 n1 n0) * OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2))))))
```

5.6.3 Floating-Point Modeling

The formal definitions required for the floating-point modeling is similar to the ones described above. Here also a mutual inverse type bijection is required and the definitions for real and imaginary component can be written in straightforward way. We define `float_complex_tybij` in the same way we did for normal complex number, to set the stage to use complex numbers of float type. Then, `float_Re` and `float_Im` are defined to use the real and imaginary part of a floating point complex number in the modeling. The function `float_OMEGA` is defined in the same way we did for complex `OMEGA` to abbreviate the twiddle factor in floating point domain. We do not write the definitions of these here explicitly, rather we show the more complicated definition of `float_complex_sum` used to model the complex summation in floating-point number

```
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))
```

In [25], John Harrison explains the IEEE floating-point formalization and its rounding issues. The rounding maps the real number to the nearest floating-point number, towards zero or towards positive or negative infinity. Here, the first format *To_nearest* is used and also the floating-point single precision denoted as `float`. The definition `float_complex_round` is used to round a complex number to its nearest floating-point. The `round` function takes three arguments—a pair denoting the attribute for floating-point precision, `float_format`; the rounding format, `To_nearest`; and the number to be rounded in real format, `Re z` or `Im z`. The definition `float_complex_64` instantly demonstrates the use of this definition by applying it on `complex_64`.

```
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)))
float_complex_64 =
⊢def  float_complex_64 = float_complex_round complex_64
```

Next, we discuss the real number valuation of floating-point numbers. It is just the inverse of rounding. The definition `Val` is used in conjunction with type bijection for complex number to get the real number equivalent of any floating-point complex number. A detailed description with the consideration of all corner cases can be found in [25].

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

where, `Val` is defined as

```
Val = ⊢def  ∀ a. Val a = valof float_format (defloat a)
```

A detailed definition for the functions `valof` and `defloat` can be found in the HOL theory *floatTheory* [32]. The definitions for other related functions are as real number modeling, and an elaboration is not provided here on them. Rather, we prove two theorems for the mathematical models derived in Equations (5.38) and (5.39) to formalize the floating-point models of FFT-IFFT design denoted as, C' and D'. Following theorems, `FLOAT_IFFT_FFT_RE` and `FLOAT_IFFT_FFT_IM` prove the real and imaginary part of the floating-point design as stated earlier. Starting with the rounding of `inv 64`, the theorems show a nesting of six floating-point summations

```
FLOAT_IFFT_FFT_RE = ⊢  ∀ x q0 q1 q2. FLOAT_IFFT_FFT_RE x q0
q1 q2 = float (round float_format To_nearest (inv 64)) * float_sum
(0,4) (λp0. float_sum (0,4) (λp1. float_sum (0,4) (λp2. float_sum
(0,4) (λn0. float_sum (0,4) (λn1. float_sum (0,4) (λn2.
   float_Re (x n2 n1 n0) * FLOAT_OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2 −
   float_Im (x n2 n1 n0) * FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2))))))
```

```
FLOAT_IFFT_FFT_IM = ⊢  ∀ x q0 q1 q2. FLOAT_IFFT_FFT_IM x q0
q1 q2 = float (round float_format To_nearest (inv 64)) * float_sum
(0,4) (λp0. float_sum (0,4) (λp1. float_sum (0,4) (λp2. float_sum
(0,4) (λn0. float_sum (0,4) (λn1. float_sum (0,4) (λn2.
   float_Re (x n2 n1 n0) * FLOAT_OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2 −
   float_Im (x n2 n1 n0) * FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2))))))
```

5.6.4 Fixed-Point Modeling

The fixed-point modeling is different from what is done so far due to the primitive parameters for arbitrary attributes related with fixed-point numbers. In order to establish the required theorems for the design at hand, we define the mutual inverse type bijection for fixed-point and the related definitions for real and imaginary part extraction by fxp_Re and fxp_Im as done above. Definitions of arithmetic operations can be found in [2] and the recursive complex summation is defined in a fairly complex manner that deserves mentioning.

```
 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)))
```

The two fixed-point number arguments for `fxp_complex` are passed through functions *streamlength* : *num#num#num* → *num* and `REPLICATE` to set the number of digits on the right hand side of the binary point of a fixed-point number. The *fracbit* notation is now a part of any fixed-point operation as shown above. In the same manner fixed-point summation can also be defined.

The rounding of fixed-point numbers takes an infinitely precise real number and converts it into a fixed-point number. There are seven quantization modes formalized in HOL *fxpTheory* for this purpose. The definition below defines a function that takes a complex number and its attribute to convert it into a rounded fixed-point complex number.

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

For the real number valuation of fixed-point numbers both signed and unsigned numbers are considered in [2]. The function `value` [2] is defined that returns the corresponding real value of a fixed-point number. We use it here to valuate fixed-point complex number. The definition `fxp_complex_value` shows how it is done in HOL,

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

Now, the real and imaginary parts of FFT-IFFT can be modeled in fixed-point domain using all the definitions described above (some are omitted). Two theorems are proved following the mathematical models described in Equations (5.40) and (5.41). `FxpAdd`, `FxpSub`, and `FxpMul` are used in conjunction with attribute `X` to do addition, subtraction and multiplication on fixed-point numbers. Both the theorems have used two new variables in universal quatification, `M` and `V`. `M` is used to specify different rounding modes and, `V` to define one of the five overflow modes. To note further that we use `fxp_round` instead of `Fxp_round` because we are dealing with fixed-point number itself not fixed-point complex number. While, the `fxp_round` takes a triplet, a real number and an overflow mode to return the rounded value in fixed-point format; the `Fxp_round` uses `fxp_round` as one of its argument chooses the first element of the pair returned from the function. The two theorems are

```
FXP_IFFT_FFT_RE = ⊢  ∀ X M V x q0 q1 q2.
      FXP_IFFT_FFT_RE X M V x q0 q1 q2 =
      FxpMul X (FST (fxp_round X M (inv 64) V))
       (fxp_sum (0,4) X (λp0. fxp_sum (0,4) X (λp1.
        fxp_sum (0,4) X (λp2. fxp_sum (0,4) X (λn0.
        fxp_sum (0,4) X (λn1. fxp_sum (0,4) X (λn2.
           FxpSub X (FxpMul X (fxp_Re (x n2 n1 n0))
                      (FXP_OMEGA_RE X n0 n1 n2 p0 p1 p2 q0 q1 q2))
                    (FxpMul X (fxp_Im (x n2 n1 n0))
                      (FXP_OMEGA_IM X n0 n1 n2 p0 p1 p2 q0 q1 q2))))))))


FXP_IFFT_FFT_IM = ⊢  ∀ X M V x q0 q1 q2.
      FXP_IFFT_FFT_IM X M V x q0 q1 q2 =
        FxpMul X (FST (fxp_round X M (inv 64) V))
        (fxp_sum (0,4) X (λp0. fxp_sum (0,4) X (λp1.
         fxp_sum (0,4) X (λp2. fxp_sum (0,4) X (λn0.
         fxp_sum (0,4) X (λn1. fxp_sum (0,4) X (λn2.
             FxpAdd X (FxpMul X (fxp_Re (x n2 n1 n0))
                        (FXP_OMEGA_RE X n0 n1 n2 p0 p1 p2 q0 q1 q2))
                      (FxpMul X (fxp_Im (x n2 n1 n0))
                        (FXP_OMEGA_IM X n0 n1 n2 p0 p1 p2 q0 q1 q2))))))))
```

5.6.5 Error Analysis

Now that we have finished all the modeling of FFT-IFFT design in HOL formally in all three domains, we are left with the final part of the modeling towards error analysis of the design. We need some more definitions for that reason and starting with the recursive definition of a product of functions in `mul(n,m) f` we model $\prod_{i=m}^{n+m-1} f(i)$.

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

To formalize errors introduced to the real and imaginary components of both floating-point and fixed-point designs, the effect of `Val` and `value` on arithmetic operations is defined in HOL which is used as assumptions in all the theorems to be followed. We did so because, the definitions of both the functions in HOL covers all the cases including the crucial corner cases. Such a definition gives numerous subgoals as we proceed deep inside the goal using tactics and tacticals. For example, in order to prove a simple associativity theorem of three floating-point numbers, even without any valuation functions applied on it gives a huge theorem to solve. The theorems added in the assumption is later proved separately to prove that the all the proofs done based on it are sound. Following Theorem 1, the rounding error due to valuation of floating-point in real number are defined in [25], where `a` and `b` are fixed-point numbers and `e` is a real

```
∀ a b. ∃e. Val (a + b)  =  (Val a + Val b) * (1 + e)
∀ a b. ∃e. Val (a − b) =  (Val a − Val b) * (1 + e)
∀ a b. ∃e. Val (a * b)  =  (Val a * Val b) * (1 + e)
```

Following Theorem 2 the corresponding errors due to valuation of fixed-point numbers are defined below, where `a` and `b` are floating-point numbers and `e` is a real

```
∀ a b X. ∃e. value (FxpAdd X a b) = value a +  value b + e
∀ a b X. ∃e. value (FxpSub X a b) = value a − value b + e
∀ a b X. ∃e. value (FxpMul X a b) = value a *  fvalue b + e
```

A confusion might arise as to why the operators are not overloaded for fixed-point operations while it is done for floating-point? There is no syntactical or semantical reason for doing so, neither any limitation in terms of the tool, rather it was an arbitrary decision which has no bearing with HOL. Having the theories at hand we can formalize the error incurred in FFT-IFFT real number rounding operation as derived in Equations (5.42) and (5.43). Before that we define a function ER_K to abbreviate the formal description of errors

```
ER_K =
⊢def  ∀ n0 n1 n2 p0 p1 p2.
      ER_K n0 n1 n2 p0 p1 p2 =
         1024 * p2 + 256 * p1 + 64 * p0 + 16 * n2 + 4 * n1 + n0
```

The floating-point valuation function Val is used in the corresponding formal definition of Equations (5.42) and (5.43) and the errors are also modeled as defined in the mathematical derivation. For example the error due to the multiplication of float_Im (x n2 n1 n0) and FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2 is defined as 1 + e (ER_K n0 n1 n2 p0 p1 p2). The following two theorems are proved in order to formally establish the floating-point errors:

```
Real Part of Floating Point Round-Off Error:
--------------------------------------------
∀ x q0 q1 q2. ∃ t p l d e z. Val (FLOAT_IFFT_FFT_RE x q0 q1
q2) =
 inv 64 *
 sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
 sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
 Val (float_Re (x n2 n1 n0)) *
 Val (FLOAT_OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2) *
     (1 + d (ER_K n0 n1 n2 p0 p1 p2)) −
 Val (float_Im (x n2 n1 n0)) *
 Val (FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2) *
     (1 + e (ER_K n0 n1 n2 p0 p1 p2))) *
     (1 + z (ER_K n0 n1 n2 p0 p1 p2)) *
     mul (ER_K n0 n1 n2 p0 p1 p2, 4097 − ER_K n0 n1 n2 p0 p1 p2)
     (λi. 1 + l i))))))) * (1 + t) * (1 + p)
```

```
Imaginary Part of Floating Point Round-Off Error:
-------------------------------------------------
∀ x q0 q1 q2. ∃ t′ p′ l″ d″ e″ z″. Val (FLOAT_IFFT_FFT_IM
x q0 q1 q2) =
 inv 64 *
 sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
 sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
 Val (float_Re (x n2 n1 n0)) *
 Val (FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2) *
     (1 + d″ (ER_K n0 n1 n2 p0 p1 p2)) −
 Val (float_Im (x n2 n1 n0)) *
 Val (FLOAT_OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2) *
     (1 + e″ (ER_K n0 n1 n2 p0 p1 p2))) *
     (1 + z″ (ER_K n0 n1 n2 p0 p1 p2)) *
     mul (ER_K n0 n1 n2 p0 p1 p2, 4097 − ER_K n0 n1 n2 p0 p1 p2)
     (λi. 1 + l″ i))))))) * (1 + t′) * (1 + p′)
```

For the fixed-point error formalization, the function `value` is used on the previously proved theorems to get the real number valuation. Since the errors are additive in this case, we write all the errors at the end of the code. Other than the use of a different valuation function the other differences this code has is the use of the function `sum` instead of `mul` since it is going to be summation of functions. Below are the two theorems proved according to the equations developed in (5.44) and (5.45)

```
 Real Part of Fixed-Point Round-Off Error:
---------------------------------------------------------
∀ X x q0 q1 q2. ∃ t'' p'' l' d' e' z'.
   value (FXP_IFFT_FFT_RE X M V x q0 q1 q2) =
   value (FST (fxp_round X M (inv 64) V)) *
   sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
   sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
   value (fxp_Re (x n2 n1 n0)) *
   value (FXP_OMEGA_RE X n0 n1 n2 p0 p1 p2 q0 q1 q2) -
   value (fxp_Im (x n2 n1 n0)) *
   value (FXP_OMEGA_IM X n0 n1 n2 p0 p1 p2 q0 q1 q2) +
   d' n0 n1 n2 p0 p1 p2 q0 q1 q2 + e' n0 n1 n2 p0 p1 p2 q0 q1 q2 -
   z' n0 n1 n2 p0 p1 p2 q0 q1 q2 +
   sum (ER_K n0 n1 n2 p0 p1 p2, 4096 - ER_K n0 n1 n2 p0 p1 p2)
       (λi. l' i))))))) + p''+t''
```

```
Imaginary Part of Fixed Point Round-Off Error:
------------------------------------------------
∀ X x q0 q1 q2. ∃ t''' p''' l''' d''' e''' z'''.
   value (FXP_IFFT_FFT_RE X M V x q0 q1 q2) =
   value (FST (fxp_round X M (inv 64) V)) *
   sum (0,4) (λp0. sum (0,4) (λp1. sum (0,4) (λp2.
   sum (0,4) (λn0. sum (0,4) (λn1. sum (0,4) (λn2.
   value (fxp_Re (x n2 n1 n0)) *
   value (FXP_OMEGA_IM X n0 n1 n2 p0 p1 p2 q0 q1 q2) −
   value (fxp_Im (x n2 n1 n0)) *
   value (FXP_OMEGA_RE X n0 n1 n2 p0 p1 p2 q0 q1 q2) +
   d''' n0 n1 n2 p0 p1 p2 q0 q1 q2 + e''' n0 n1 n2 p0 p1 p2 q0 q1 q2 −
   z''' n0 n1 n2 p0 p1 p2 q0 q1 q2 +
   sum (ER_K n0 n1 n2 p0 p1 p2, 4096 − ER_K n0 n1 n2 p0 p1 p2)
       (λi. l''' i))))))) + p'''+ t'''
```

The error calculation in Equation (5.47) gives the mathematical model for error from ideal domain to floating point. We simply formalize it by subtracting the ideal FFT-IFFT representation from the error prone floating-point one as follows,

```
IFFT_FFT_REAL_TO_FP_ERROR_def = ⊢  ∀ x q0 q1 q2.
     IFFT_FFT_REAL_TO_FP_ERROR x q0 q1 q2 =
      complex (Val (FLOAT_IFFT_FFT_RE
         (λq0 q1 q2. float_complex_round (x q0 q1 q2)) q0 q1 q2) −
      IFFT_FFT_RE x q0 q1 q2,
               Val (FLOAT_IFFT_FFT_IM
         (λq0 q1 q2. float_complex_round (x q0 q1 q2)) q0 q1 q2) −
      IFFT_FFT_IM x q0 q1 q2)
```

As derived earlier in order to have a concise representation, we defined e_0 and define it in HOL as ERROR_0_def,

```
ERROR_0_def =
⊢def  ∀ x q0 q1 q2.
     ERROR_0 x q0 q1 q2 =
     complex (Val (float_Re
         ((λq0 q1 q2. float_complex_round (x q0 q1 q2)) q0 q1 q2)) −
         Re (x q0 q1 q2),
            Val   (float_Im
         (λq0 q1 q2. float_complex_round (x q0 q1 q2)) q0 q1 q2)) −
         Im (x q0 q1 q2))
```

We do not show the formal definitions of Equations (5.49), (5.50) and (5.51) since we have already seen something similar above. Now, to complete the error analysis of REAL to floating-point, we prove the following theorem to establish that the error analysis derived in Equation (5.54) is the valid analysis for such case. It can be seen from the code that all variables, functions and definitions are previously discussed and can be mapped to its mathematical counterpart easily.

```
∀ x q0 q1 q2. ∃ f. (IFFT_FFT_REAL_TO_FP_ERROR x q0 q1 q2 =
complex_64 * complex_sum (0,4) (λp0. complex_sum (0,4) (λp1.
complex_sum (0,4) (λp2. complex_sum (0,4) (λn0. complex_sum (0,4)
(λn1. complex_sum (0,4) (λn2.
   ERROR_0 x n2 n1 n0 * OMEGA n0 n1 n2 p0 p1 p2 q0 q1 q2 +
   f n0 n1 n2 p0 p1 p2 q0 q1 q2))))))) ∧
∃ t l d e z t′ l″ d″ e″ z″. f n0 n1 n2 p0 p1 p2 q0 q1 q2 =
complex ( Val (float_Re ((λn0 n1 n2. float_complex_round (x n0 n1
n2)) n0 n1 n2)) *
                  Val (FLOAT_OMEGA_RE n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      ((1 + d n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      (1 + z n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      mul (ER_K n0 n1 n2 p0 p1 p2,4097 − ER_K n0 n1 n2 p0 p1 p2)
                      (λi. 1 + l i) * (1 + t) − 1) −
Val (float_Im ((λn0 n1 n2. float_complex_round (x n0 n1 n2)) n0 n1 n2)) *
                  Val (FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      ((1 + e n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      (1 + z n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      mul (ER_K n0 n1 n2 p0 p1 p2,4097 − ER_K n0 n1 n2 p0 p1 p2)
                      (λi. 1 + l i) * (1 + t) − 1),
Val (float_Re ((λn0 n1 n2. float_complex_round (x n0 n1 n2)) n0 n1 n2)) *
                  Val (FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      ((1 + d″ n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      (1 + z″ n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      mul (ER_K n0 n1 n2 p0 p1 p2,4097 − ER_K n0 n1 n2 p0 p1 p2)
                      (λi. 1 + l″ i) * (1 + t′) − 1) −
Val (float_Im ((λq0 q1 q2. float_complex_round (x q0 q1 q2)) q0 q1 q2)) *
                  Val (FLOAT_OMEGA_IM n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      ((1 + e n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      (1 + z n0 n1 n2 p0 p1 p2 q0 q1 q2) *
                      mul (ER_K n0 n1 n2 p0 p1 p2,4097 − ER_K n0 n1 n2 p0 p1 p2)
                     (λi. 1 + l i) * (1 + t′) − 1))
```

The round-off error from real number to fixed-point domain can be written in the same way as real to floating-point error analysis. The only extra thing is the use of rounding and exception handling parameters for fixed-point operations.

```
IFFT_FFT_REAL_TO_FXP_ERROR_def = ⊢def  ∀ X M V x q0 q1 q2.
             IFFT_FFT_REAL_TO_FXP_ERROR X M V x q0 q1 q2 =
             complex
              (value
                (FXP_IFFT_FFT_RE X M V
                  (λq0 q1 q2. fxp_complex_round X (x q0 q1 q2)) q0 q1 q2) −
            IFFT_FFT_RE x q0 q1 q2,
            value
              (FXP_IFFT_FFT_IM X M V
                 (λq0 q1 q2. fxp_complex_round X (x q0 q1 q2)) q0 q1 q2) −
            IFFT_FFT_IM x q0 q1 q2)
```

We also define ERROR'_0 to have some functions under one definition for concise representation,

```
⊢def  ∀ X M V x q0 q1 q2.
ERROR'_0 X M V x q0 q1 q2 =
  complex (value (fxp_Re ((λq0 q1 q2.
                 fxp_complex_round X (x q0 q1 q2)) q0 q1 q2)) −
                 Re (x q0 q1 q2),
           value (fxp_Im ((λq0 q1 q2.
                 fxp_complex_round X (x q0 q1 q2)) q0 q1 q2)) −
                 Im (x q0 q1 q2))
```

Formalizing REAL to fixed-point error is less complicated since the additive property of fixed-point helps to isolate the error functions—$\boldsymbol{\delta}$, $\boldsymbol{\epsilon}$, $\boldsymbol{\xi}$, $\boldsymbol{\lambda}$,$\boldsymbol{\tau}$, $\boldsymbol{\rho}$. The following theorem is proved in HOL to establish the error analysis derived for REAL to fixed-point in Equations (5.55) and (5.56).

```
∀ X M V x q0 q1 q2. ∃ f′. (IFFT_FFT_REAL_TO_FXP_ERROR X M V x q0 q1 q2 =
           complex_64 *
            cmplex_sum  (0,4) (λp0. complex_sum (0,4) (λp1.
            complex_sum (0,4) (λp2. complex_sum (0,4) (λn0.
            complex_sum (0,4) (λn1. complex_sum (0,4) (λn2.
              ERROR′_0 X M V x n2 n1 n0 * OMEGA n0 n1 n2 p0 p1 p2 q0 q1 q2 +
              f′ n0 n1 n2 p0 p1 p2 q0 q1 q2))))))) ∧
               ∃t′ l′ d′ e′ z′ t′′′ l′ d′′′ e′′′ z′′′.
                f′ n0 n1 n2 p0 p1 p2 q0 q1 q2 =
                   complex (d′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           e′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           z′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           sum (ER_K n0 n1 n2 p0 p1 p2,4096 −
                           ER_K n0 n1 n2 p0 p1 p2)(λi. l′ i) + t′,
                           d′′′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           e′′′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           z′′′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           e′ n0 n1 n2 p0 p1 p2 q0 q1 q2 +
                           sum (ER_K n0 n1 n2 p0 p1 p2,4096 −
                           ER_K n0 n1 n2 p0 p1 p2) (λi. l′′′ i) + t′′′)
```

We are only left with the error that occurs between floating-point to fixed-point and we formalize it using Equation (5.57). The error occurred during REAL to floating-point—“e” and during REAL to fixed-point—“$e^{'}$” is used directly to formalize the error definition,

```
∀ X M V x q0 q1 q2. IFFT_FFT_FP_TO_FXP_ERROR X M V x q0 q1 q2
= IFFT_FFT_REAL_TO_FP_ERROR x q0 q1 q2 − IFFT_FFT_REAL_TO_FXP_ERROR
X M V x q0 q1 q2
```

Now, we prove the final theorem for floating-point to fixed-point error. We show a skeleton of the actual HOL code rather writing the complete one since it is just the right hand side of the previous two error analysis theorems

```
∀ X M V x q0 q1 q2. IFFT_FFT_FP_TO_FXP_ERROR X M V x q0 q1 q2
= right−hand side of [ REAL to FP  error theorem ] −
  right−hand side of [ REAL to FXP error theorem ]
```

5.7 Discussion

The error analysis done above covers the OFDM rounding error analysis thoroughly between different number domains. To establish the complete theory of error analysis we proved three main theorems with the help of formalized real and imaginary part of FFT-IFFT expansion and also the theorems related to the error for arithmetic operations. All definitions were derived heavily from existing theories, e.g., *realTheory*, *boolTheory*, *ieeeTheory*, *floatTheory*, *fxpTheory*, *wordTheory*, etc. There is a very strong relationship between mathematical models and their formal counterpart which might have been observed above. The definitions built on top of established theories in turn helped to build the FFT and IFFT components; which build the theory for the FFT-IFFT combinations. Then this theory is extended and the operators are overloaded for establishing the real, floating-point and fixed-point counterparts of the design using the *floatTheory* and *fxpTheory*.

For all the theorems and assumptions in the whole error analysis work it is imperative that higher-order logic be used. The error analysis is based on the floating-point and fixed-point theory of HOL, which are two of the most important additions in HOL's rich theory base. Besides quantification over variables and objects, there are many theorems in both theories that make use of quantification over functions. Moreover, almost all the definitions required to model the FFT-IFFT combination needed higher-order logic for the same reason. The error analysis functions of both floating-point and fixed-point—$\boldsymbol{\delta}$, $\boldsymbol{\epsilon}$, $\boldsymbol{\xi}$, $\boldsymbol{\lambda}$,$\boldsymbol{\tau}$ and $\boldsymbol{\rho}$—are all existentially quantified in the main theorems proved and these theorems construct the core of the final result.

Throughout the proof of the theories built-in tactics and tacticals were used. In many of these proofs case analysis and induction were used. Our main approach to prove the theorems was to have a rough paper and pencil sketch of the approach and then formalize it using the techniques available in the HOL tool. Many times it happened that it was hard to prove the theorem as a whole in one shot and then we break the goal in manageable size to prove the parts separately to combine later. To accomplish this in a different way sometimes theorems are assumed in the proof to concentrate in the core goal and later the assumed theorem is proved. Thus we prove the theorems till the final error analysis between floating-point to fixed-point. Through the course of the modeling and proof, many lemmas are developed, some are trivial but essential and some are crucial to move to the next step in establishing a theorem. But, it is important to mention that the current theorems can be proved in a better way which is realized gradually as we moved to much complicated proofs and so the latter proofs are better and concise than the previous ones.

Another important issue needs to be addressed and also equally applicable for all the theorems proved in Chapter 4 is that how it can be assured that the definitions created by the user themselves are sound and really characterize what the system user intends to formalize. In short, there is no way to verify that the modeling in HOL done by the user reflects the hardware exactly. The tool can check all the type requirements based on the initial information of the system provided by the user, and if these information are wrong then the final formalization will also be wrong. The HOL system is based on five axioms and eight primitive inference rules. All the HOL theories are built on top of them and this is another reason of the lengthy installation time required since all the built-in theories are to be proved before becoming part of the initial system. This is why there is no chance to have ambiguity in the proof system of HOL. Although highly improbable, but a wrong implementation can be verified against a wrong specification. Each and every possible scenario can happen. The tool itself might not be free from bugs. That is, however, why a tool like HOL needs expert users who have good knowledge of formal methods and also of the system under verification. The same can be told about the real RTL design in simulation where only the functionality of the system can be verified but it can never be assured completely that the final product will exactly behave as the specification due to manufacturing difficiencies or other factors.

Since HOL is an interactive tool where the user needs to guide every step of the proof, it is also possible that the theorem prover can be guided to falsely proof a system. But, HOL strongly checks the type of the terms and functions entered into the system. This particular constraint also makes it very difficult to make simple mistakes in defining wrong theorems thus also answers partially the concern mentioned in the previous paragraph. Still, if any user wants to trick the tool to generate proof arbitrarily, he/she has to use *oracle* [32] mechanism that enables arbitrary formulas to become elements of the `thm` type. By use of this mechanism, HOL can utilize the results of arbitrary proof procedures. To avoid unsoundness, a tag is attached to any theorem coming from an oracle. This tag is propagated through every inference that the theorem participates in and if falsity becomes derived, the offending oracle can be found by examining the tags component of the theorem. A theorem proved without use of any oracle will have an empty tag, and can thus be considered to have been proved solely by deductive steps in the HOL logic. Thus, the tool ensures its security against misuse.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

This thesis is mainly concerned to demonstrate the use of formal verification techniques, here, using theorem proving, to verify an implementation of an OFDM modem based on the IEEE 802.11a physical layer standard. The OFDM design is fairly complex and some important design blocks were chosen for verification purposes. We formally modeled the quadrature amplitude modulation and demodulation blocks, serial to parallel block and the parallel to serial block in HOL. The specifications of all the blocks are also modeled using the same formalism. Then, a functional correctness proof is done using implication with the help of many lemmas developed as the proof required for it and also using many built-in tactics. The end result showed the flawless functionality of the original implementation after abstracting the required functionality from the original design.

We also analyzed the errors in the OFDM system occurring at the time of converting from one number domain to the other. We used the IFFT-FFT combination as a model for the error analysis of the whole system. The mathematical equations for 64 point radix-4 FFT and IFFT are developed in graphic details. We derived mathematical expressions for all three domains—ideal real, floating-point, and fixed-point numbers. Then we derived fundamental theorems for the accumulation of round-off error in the OFDM system. This formalization can be considered as a large application of the formal error analysis framework described before and shows the viability of such analysis even for larger scale systems as the one analyzed.

Having a totally bug-free hardware is a daunting task and it still did not attain nowhere near where it can be said with confidence that a hardware will behave according to desired specification. Formal verification shows light in alleviating such problems but it is yet to develop its niche to be accepted as a novel technique to completely integrate with the whole VLSI design flow. It is digital designer's dream to have their designs formally verified from specification to final product. A beacon of hope is that state-of-the-art formal techniques have shown good trend to complement the simulation technique and successfully applied in industry in critical design verification. It can be concluded with certainty that, among all the formal verification techniques, theorem proving will always have a special place in the formal verification and error analysis of digital designs containing complex computational blocks.

6.2 Future Work

The future work that can be carried out pertaining to this thesis might elucidate new and interesting ideas and some suggestions are following:

1. Verifying the RTL implementation of OFDM block using the clocking constraints. This work can complement the current thesis by taking into account clock transitions and then it would be possible to verify the generation of control signals and latching of data at specific times. For this, a good model of a RAM is needed to mimic the one that is implemented in the current design. Moreover, standard RTL models of FFT and IFFT can also be attached to the system to stich all the signals together to have a comprehensive verification of the total system. If a design can be realized or obtained using the IEEE standard as it is, it would be a perfect combination to have it verified using HOL.

2. Development of a parameterized error analysis pattern for any FFT or IFFT design of arbitrary computing point and radix. Such parametric technique can follow the approach of Wong [60] where he developed the *wordtheory*—a HOL theory that can generate another theory of any bit as required by the user. Such work can eliminate the tedious process of specifying designs formally again and again for different computation points and radixes. And, in turn the error analysis can be parameterized so that a comprehensive error analysis can be obtained for any arbitrary FFT-IFFT combination or each component itself.

3. Verifying the OFDM system using a combination of HOL and another powerful computer algebra system such as Maple [43] or Mathematica [45]. For instance, a system like Maple incorporates a high-level programming language, which allows the user to define his/her own procedures; it also has packages of specialized functions, which may be loaded to do work in other fields. Similar to Maple, the Mathematica system is based on term-rewriting and supports both functional and procedural programming. The computer algebra systems are extremely powerful and flexible but often give results which require careful interpretation. In contrast, theorem provers are very reliable but lack the powerful specialized decision procedures and heuristics of the former. Both approaches can be complemented by combining them in a novel way (e.g. [29]) to verify the OFDM system and similar designs.

Bibliography

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

[2] 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.

[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] E. M. Clarke et. al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[6] C. M. Angelo, L. J. M. Claesen, and H. De Man. Degrees of Formality in Shallow Embedding Hardware Description Languages in HOL. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of Lecture Notes in Computer Science, pages 89–100. Springer-Verlag, 1994.

[7] P. J. Ashenden. *Designer's Guide to Vhdl*. Morgan Kaufmann, 2001.

[8] D. Basin, A. Brucker, J. G. Smaus, and B. Wolff. Computer Supported Modeling and Reasoning, Department of Computer Science Swiss Federal Institute of Technology Zurich. `http://www.infsec.ethz.ch/education/permanent/csmr`, 2005.

[9] G. Birtwistle, S. K. Chin, and B. Graham. new theory 'HOL';; An Introduction to Hardware Verification in Higher Order Logic. 1994.

[10] R. E. Bryant. Tutorial on Formal Verification of Hardware. In *Proceedings of the 28th Design Automation Conference*, pages 397–402, San Francisco, California, USA, June 1991.

[11] Cadence Design Systems Ltd. See. `http://www.cadence.com`, 2005.

[12] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[13] A. L. Cinquino. A Real-Time Software Implementation of an OFDM Modem Suitable for Software Defined Radios. Master's thesis, Department of ECE, Concordia University, Montreal, QC, Canada, 2004.

[14] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965.

[15] CoWare Inc. *Manual for the tool* Signal Processing Worksystems *(SPW)*, May 2002.

[16] J. D. Day and H. Zimmermann. The OSI Reference Model. *Conformance Testing Methodologies and Architectures for OSI Protocols*, pages 38–44, 1995.

[17] R. W. Chang, *Orthogonal Frequency Division Multiplexing.* US Patent No. 3,488,455. Filed on November 14, 1966, Issued on January 6, 1970.

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

[19] F. Frescura, S. Andreoli, S. Cacopardi, and E. Sereni. An OFDM Radio Transmitter based on TMS320C6000 DSP for Telemetry Applications. In *Proceeding of International Conference Signal Processing Applications and Technology*, Dallas, Texas, USA, October 2000.

[20] W. M. Gentleman and G. Sande. Fast Fourier Transforms - For Fun and Profit. In *AFIPS Fall Joint Computer Conference*, volume 29, pages 563–578, Washington, USA, 1966.

[21] G. Gonthier. *A Computer-Checked Proof of the Four Colour Theorem*. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2005.

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

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

[24] J. Harrison. Constructing The Real Numbers in HOL. *Formal Methods in System Design*, 5(1-2):35–59, 1994.

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

[26] J. Harrison. A Machine-checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics*, volume 1690 of Lecture Notes in Computer Science, pages 113–130. Springer-Verlag, 1999.

[27] J. Harrison. A list of theorem provers. `http://www.cl.cam.ac.uk/users/jrh/ar.html`, 2005.

[28] J. Harrison. Complex Quantifier Elimination in HOL. In *Supplemental Proceedings of Theorem Proving in Higher Order Logics*, pages 159–174. Edinburgh, UK, September 2001.

[29] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[30] S. Haykin. *Communication Systems.* Wiley, 1994.

[31] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

[32] HOL Sourceforge Project. *The HOL System Description.* http://hol.sourceforge.net, Sept 2005.

[33] HOL Sourceforge Project. *The HOL System Reference.* http://hol.sourceforge.net, September 2005.

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

[35] IEEE 802.11 Working Group. *IEEE Std 802.11a-1999.* The Institute of Electrical and Electronics Engineers,Inc, 1999.

[36] 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.

[37] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A Fixed-Point Design and Simulation Environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 429–435, Paris, France, 1998.

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

[39] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation, Modelling Techniques and Tools*, volume 2324 of Lecture Notes in Computer Science, pages 200–204. Springer-Verlag, 2002.

[40] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In *Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, volume 2399 of Lecture Notes in Computer Science, pages 169–187. Springer-Verlag, 2002.

[41] R. Lyons. *Understanding Digital Signal Processing.* Pearson Education, 2004.

[42] F. Manavi. Implementation of OFDM Modem for the Physical Layer of IEEE 802.11a Standard Based on XILINX VIRTEX-II FPGA. Master's thesis, Department of ECE, Concordia University, Montreal, QC, Canada, 2004.

[43] Maplesoft. Waterloo Maple Inc. `http://www.wolfram.com/products/mathematica/index.html`, 2006.

[44] S. J. Mason. Feedback Theory-Further Properties of Signal-Flow Graphs. In *Proceeding of IRE*, volume 44, pages 920–926, 1956.

[45] Mathematica. Wolfram Research Inc. `http://www.maplesoft.com/products/maple/index.aspx`, 2006.

[46] T. F. Melham. Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic. Technical report, University of Cambridge Computing Laboratory, Cambridge, UK, 1990.

[47] E. Mendelson. *Introduction to Mathematical Logic.* CRC Press, 1997.

[48] NASA Langley Formal Methods Team. *Why is Formal Methods Necessary?* `http://shemesh.larc.nasa.gov/fm/fm-why-new.html`, 2005.

[49] R. V. Nee and R. Prasad. *OFDM for Wireless Multimedia Communications.* Artech House Publishers, 2000.

[50] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[51] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CÆSAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 3576 of Lecture Notes in Computer Science, pages 337–351. Springer-Verlag, 1982.

[52] L. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing.* Prentice-Hall, 1975.

[53] A. Roy and K. Gopinath. Improved Probabilistic Models for 802.11 Protocol Verification. In *Computer Aided Verification*, volume 3576 of Lecture Notes in Computer Science, pages 239–252. Springer-Verlag, 2005.

[54] B. Saltzberg. Performance of an Efficient Parallel Data Transmission System. *IEEE Transaction on Communication*, 15(6):805–811, December 1967.

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

[56] M. Tariq, Y. Baltaci, T. Horseman, M. Butler, and A. Nix. Development of an OFDM based High Speed Wireless LAN Platform using the TI C6x DSP. In *Proccedings of IEEE International Conference on Communications*, pages 522–526, New York, NY, USA, May 2002.

[57] F. Wiedijk. An overview of systems implementing mathematics in the computer. `http://www.cs.ru.nl/~freek/digimath/index.html`, 2005.

[58] Wikipedia. Theorem provers. `http://en.wikipedia.org/wiki/Theorem_prover`, 2005.

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

[60] W. Wong. Modeling Bit Vectors in HOL: The Word Library. In *Higher Order Logic and Its Applications*, volume 780 of Lecture Notes in Computer Science, pages 371–384. Springer-Verlag, 1994.

[61] Xilinx Inc. Xilinx Coregen Library. `http://www.xilinx.com/ipcenter/coregen`, 2005.