İSTANBUL TECHNICAL UNIVERSITY

ELECTRICAL - ELECTRONICS ENGINEERING FACULTY

Design And Verification Of An 16-Bit Floating Point Alu, Using Universal Verification Method

BSc Thesis by

Yasin Fırat Kula

040110020

Department: Electronics and Communication Engineering

Programme: Electronics and Communication Engineering

Supervisor: Assoc. Prof. Dr. Sıddıka Berna ÖRS YALÇIN

MAY 2016

PREFACE

First of all, thanks to all of my respectable college teachers who pass on their great knowledge throughout BSc program ,with a special thanks to my teacher and project advisor Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın for her invaluable information and support in the course of the project.

Also, I would like to thank Güler ÇOKTAŞ for her help and guidance during the process of the thesis.

Lastly, I present my eternal gratitude to my family, who've been always been with me from the beginning to the present, and bringing me where I am now.

MAY 2016 Yasin Fırat KULA

TABLE OF CONTENTS

LIST OF ABBREVIATONS

UVM : Universal Verification Methodology
ALU : Arithmetic Logic Unit
IVB : Incisive Verification Builder
IEEE : Institute of Electrical and Electronics Engineers
OVM : Open Verification Methodology
AVM : Advanced Verification Methodology
VMM : Verification Methodology Manual
HDL : Hardware Description Language
FPGA : Field Programmable Gate Array
IDE : Integrate Development Environment
SoC : System on Chip
ASIC : Application Specific Integrated Circuit
RTL : Register Transfer Level
DUT : Design Under Test
TLM : Transaction Level Modeling
DPI : Direct Programming Interface

LIST OF TABLES

LIST OF FIGURES

ON ALTI BİTLİK KAYAN NOKTA SİSTEMLİ ALU TASARIMI VE DOĞRULANMASI

ÖZET

Sayısal sistem tasarımında devrenin boyutu ve karmaşıklığı arttıkça, orantılı olarak sistemin doğru çalıştığının test edilmesi de daha zorlayıcı hale gelmektedir. Piyasada bulunan çeşitli basit benzetim araçlarının olmasına rağmen; yüksek miktarda özel hal, giriş kombinasyonları ve durumlar bu araçlar ile yapılan benzetimlerin izlenmesini ve bulunan tasarım hataların giderilmesini oldukça zor kılmaktadır ve bu da gözden kaçan hatalı durumlar veya istenmeyen davranışlar oluşması riskine sebep vermektedir. Sistemdeki hatanın, ürün piyasaya sürüldükten sonra ortaya çıkması durumu da sistemin hatalarını düzelterek yeniden tasarlamak oldukça masraflı olacağından üretim açısından büyük sorun teşkil etmektedir.

Bu sorunlar nedeniyle sistemlerin doğrulanması için daha gelişmiş yöntemler geliştirilmesi yoluna gidilmiştir; bu yöntemleri kullanabilecek yeterli bilgi düzeyine sahip kişilere duyulan ihtiyaç da doğrulama mühendisliğini doğurmuştur. Evrensel Doğrulama Metodu (UVM), bu geliştirilen doğrulama yöntemlerinden biridir ve projede bu metot kullanılmıştır.

Bitirme projesi iki aşamadan oluşmaktadır: İlk olarak , bilgisayar ve mikroişlemci sistemlerindeki yaygınlığı ve önemi nedeniyle, ardışık düzen mimarisine sahip ve kayan nokta aritmetiği yapabilen bir aritmetik lojik birim (ALU) ortalama karmaşıklık sunacak bir devre olarak tasarlanmıştır. İkinci aşamada, gerekli UVM kodları yazılarak ALU tasarımının doğrulanması amaçlanmıştır. Projenin üç bitirme öğrencisinin ortak çalışması olarak götürülmesi nedeniyle, bu aşamalar ALU'nun toplama-çıkarma birimi, çarpma birimi ve bölme birimi olarak üçe bölünmüş; her bölümün tasarım ve doğrulaması ayrı bir öğrenci tarafından yapılmıştır. Bu çalışmada bölme modülü üzerinde çalışılmıştır.

Bu amaçlar doğrultusunda; tasarımının adımları detaylı bir şekilde anlatılacak ve yapılan basit benzetiminin sonuçları gösterilecek; ardından kullanılan önemli UVM birimleri hakkında bilgiler verilecek ve doğrulama işleminin aşamaları gösterilecektir.

DESIGN AND VERIFICATION OF AN 16-BIT FLOATING POINT ALU, USING UNIVERSAL VERIFICATION METHOD

SUMMARY

In digital system design; as the size and complexity of the system increases, verifying its validity becomes proportionally challenging. Even though many basic simulation tools are produced by various corporations and individuals, the vast array of special cases, behaviours and input combinations makes the simulation process quite hard to track and restore design flaws, resulting in overlooked faulty cases in design that leads to errors and unintended behaviour. When a fault detected on a product system after its release, it becomes very costly for producers to re-release a fixed version of the product.

The problems described paved a way for development of more sophisticated system verification methods; and with the requirement of individuals with ample knowledge of those verification systems, verification engineering is born. Universal Verification Methodology (UVM), one of those verification methods that are developed, will be used as the verification standard during this project.

The graduation project contains two phases: Firstly, a pipelined floating point arithmetic logic unit (ALU) is designed as a system with moderate complexity, and for its commonness and importance in many computer systems and microprocessors. Secondly, required UVM codes are written and the ALU design is aimed to be verified by using these codes and special design tools. Since the project is a collaborated work, this phases are divided into three sections as addition-subtraction module, multiplication module and division module; where each module's design and verification have done by one of the collaborated graduate students. In this work, division module's design and verification process was expressed.

For this aspect; steps of the ALU design will be given in detail, a basic simulation will be applied to test its functionality; then the verification process will be shown step by step, after the introduction of the important UVM components that is used.

1. INTRODUCTION

In electronics engineering, digital system designs steps forward with various aspects like easier programmability and design process, high speed and lower cost; also, digital systems offers more reliability and precision compared to analog systems in many applications, even though analog systems are better and more suitable for various practices, digital systems are replacing analog systems whenever possible [1]. As the commonness of digital systems increases, their size and complexity increases; and this brings system validation problem with it. To cope with this problem, various verification methods are developed. The Universal Verification Methodology (UVM) is one of these methodologies and it is used in verifying the created system in this project[2]. UVM was chosen because of its high reliability, user-friendly construct and its current position as a highly accepted standard in verification process[3].

Generally, the process of creating a digital system consists of two steps; which are designing and verifying. The goal of the project is to see each step of this process in detail, by creating a digital system with moderate complexity and validating it afterwards. First, an arithmetic logic unit (ALU) design was created according to its specified requirements, then it was verified with the selected methodology.

The ALU is designed in Verilog HDL[4], using Xilinx ISE Design Suite[5] and later simulated in various aspects, at the same designing tool, in order to basically check its validity. The process of system design is expressed in detail at the fourth chapter of thesis.

In the following phase, information on UVM was gathered. It is followed by creating the necessary UVM codes. The code creation part was done by taking the UVM codes of the previously completed graduation project on verification with UVM as basis[2]. These codes were originally created by Incisive Verification Builder (IVB) tool[6], and modified as per the requirements. Similarly in this project, these codes created in the aforementioned graduation project are modified and appended, with the goal of further contributing the verification study performed on our university. Detailed information on verification is given in thesis Chapter 5.

Created UVM testbench had been ran on Mentor Graphics QuestaSim[7] tool. Steps of running the UVM testbench is expressed at Chapter 6.

2. BACKGROUND

2.1. Floating Point Numbers

2.1.1. General Information

Floating point is a number representation system that approximates to a real number, escalated by a balance between range and precision. Numbers are represented approximate to a fixed-point significand number, then scaled using an exponent with a fixed base number [8]. By these definitions, a floating point number can be exactly represented as follows:

$$Significand \ \ x \ Base^{Exponent} = Number \quad (2.1)$$

In digital world, different techniques for representing floating point numbers were defined. Currently, the most widely used floating point representation is the Institute of Electrical and Electronics Engineers' (IEEE) IEEE-754 floating point format [9].

IEEE-754 format offers binary and decimal based floating point formats in different precisions, which could be seen from Table 2.1. These representation named after their number base and their corresponding bit lenght that used to represent the number. More bit number means larger approximation to the actual real number. In the ALU design, "binary16" representation was chosen for smaller circuit size.

Table 2.1: Available formats of IEEE-754 number system [10]

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	−262142	+262143	not basic
decimal32		10	7	7	7.58	96	101	−95	+96	not basic
decimal64		10	16	16	9.58	384	398	−383	+384	
decimal128		10	34	34	13.58	6144	6176	−6143	+6144	

2.1.2. IEEE-754 Half Precision Floating Point System

An IEEE 754 format floating point number consists of three parts.

- Sign Bit: This is the most significant bit for all representation types in IEEE 754. High value of a sign bit represents negative sign, whereas low value of the sign bit represents positive sign.
- Exponent Bits: Exponent bits represents the power of the base number. During the calculation, a bias with the amount of ;

  $Base^{Number\ of\ Exponent\ bits-1} - 1$ (2.2)

  $Base^{\text{Number of exponent bits- 1}} - 1$

  added to the exponent value.
- Mantissa (Fraction) Bits: Mantissa bits represents the significand, in other words represents the precision of the number. It is the remaining bits from the sign and exponent, in all expressions. Mantissa bits assumed to have an extra leading bit as "1" for calculation purposes.

Below is the bit arrangement of "binary16" half precision system.

Figure 2.1 : Bit arrangement of "binary16" system[11]

By those definitions, decimal expression of a half precision floating number could be given as follows:

$-1^{\text{Sign}} x\ 2^{Exponent-15} x\ 1.Mantissa = Number$ (2.3)

Using this formula; it is found that the largest absolute value representable is 65504 = $(-1)^{sign} * 2^{(30-15)} * 1.1111111111$, and the smallest absolute value representable is 6,103515625e-5 = $(-1)^{sign} * 2^{(0-15)} * 1.0000000001$. Exponent value of 31 represents infinity cases.

2.2. Floating Point ALU

The arithmetic logic unit is a digital circuit that can perform arithmetic and logic operations for a processor system. Floating point ALU, also known as floating-point

unit, does these mathematical operations for floating point numbers, which its system is chosen depending on application. It can either be in a system as an integrated processor, or as an independent co-processor. Since applications like scientific calculation and signal processing requires operation capability on floating point numbers with good precision, a floating-point unit has quickly become a corner-stone component in current computer systems and digital signal processors.

During the project, a 16-bit floating point ALU using the IEEE 754 half precision system is designed. The unit kept simple with only containing the four basic mathematical operations.

Some processors contains an implementation named pipeline, in order to reduce the time consumed on tasks requiring high computing time (clock cycles)[12]. In pipeline, instructions are overlapped on the processors, before waiting for an instruction to be completed. The structure usually contains a handful of stages, each performing a specific task and are separated from each other with memory units. For example, an instruction that requires three clock cycles to complete could be overlapped consequently in a pipelined implementation; after four clock cycles, the second instruction (overlapped instruction) will produce its result, due to being computed in parallel lower stages of the design. In this manner, the instruction clock cycle is not reduced, rather its throughput time of consequent instructions is decreased.

During the implementation of ALU in this project, it is aimed to implement a pipelined design.

2.3. Verification

Verification is the process of validating a system's operation, by checking if it is working as planned. In digital designs, when the first time the verification process is introduced, designers tend to perform the verification step by checking the corresponding waveforms of a simulation or checking the outputs manually. However; as the size and complexity of the systems are increased, new methodologies are needed, since for the large systems even simply observing the waveform becomes inextricable [13].

In modern verification; certain languages and methodologies are used to validate the system, like SystemVerilog [14] and UVM, with support of object oriented

programming. The verification steps consist of creating suitable test codes and running the tests. Usually verification step is performed in parallel with design process. A verification engineer needs to have a good background knowledge on design to be tested, he/she needs to create suitable test that will cover specific and boundary values of the design; while also aiming to cover entire code to minimize any unexpected behaviour, and allow to designers fix them if any of these behaviour occurs.

2.4. Universal Verification Methodology (UVM)

The Universal Verification Methodology is a standard founded by Accellera [15]. It is firmly established on the existing OVM (Open Verification Methodology) [16] base code In this aspect the methodology is also a hybrid of various known verification methodologies such as Mentor Graphics' AVM (Advanced Verification Methodology) [17] and Synopsys' VMM (Verification Methodology Manual)[18] along with several new technologies[19]. UVM is aimed to be the standard in verification methodologies.

As stated in Section 2.3., first times of verification process was exercised with waveform simulations; and later evaluated to languages like Vera [20] and SystemVerilog, then followed by OVM based methodologies. Following the tradition, the UVM also offers the SystemVerilog based Base Class Libraries in order to make the process more systematic for developers. These libraries offer standard construct for many components that are used in UVM testbenches. For the desire of automating and reusing the process, these template libraries become quite handy for users as well. This way, a full testbench created could be used again for a different application with just a few modifications. The reusable and standardized construct hence saves a verification engineer from being constrained by the design tool chosen for the verification process; which was mostly the case before UVM.

3. DESIGN AND VERIFICATION TOOLS

3.1. Xilinx ISE Design Suite

ISE (Integrated Synthesis Environment) Design Suite [5] is a tool produced by Xilinx, for developing, compiling, analysing and synthesising HDL (Hardware Description Language) designs (supporting Verilog language). It features a wide scale of options for design implementation like timing analysis and power analysis; is able to show RTL and technology diagrams of created designs, also letting the developer to load and configure the HDL design into supported device. It should be noted that the software behind ISE is strictly coupled to the company's own produts' architectures and cannot be used with products from different producers.

ISE Design Suite also contains integrated sub-tools; which are used for tasks like logic simulation and device configuring.

This tool was used as a design environment and simulation tool during the timespan of the project. Verilog codes for floating point ALU and basic testbenches for its simulation were created using this tool.

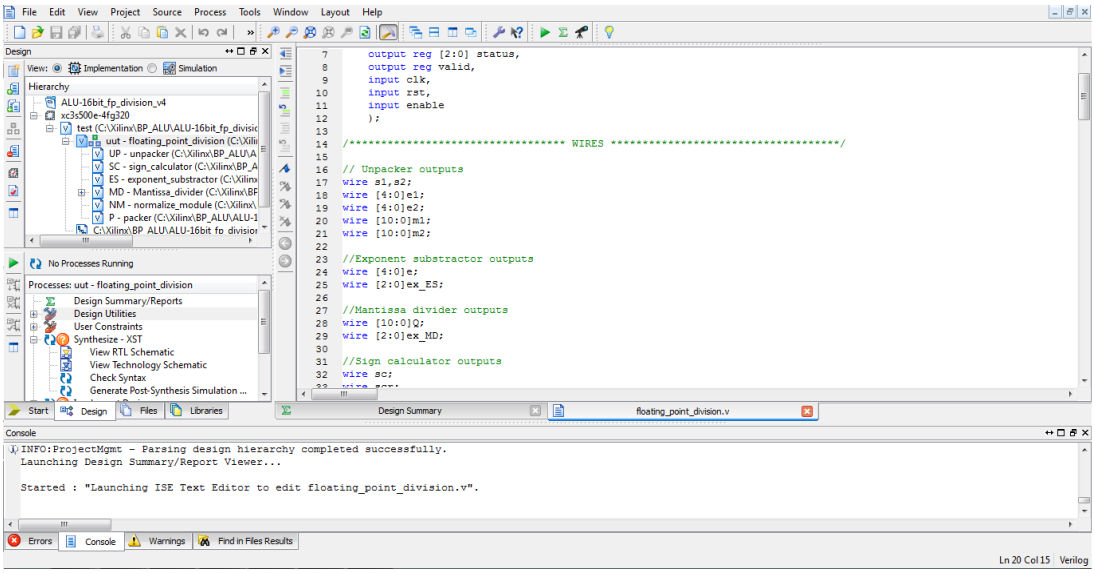

Figure 3.1: A screenshot of HDL design environment of ISE

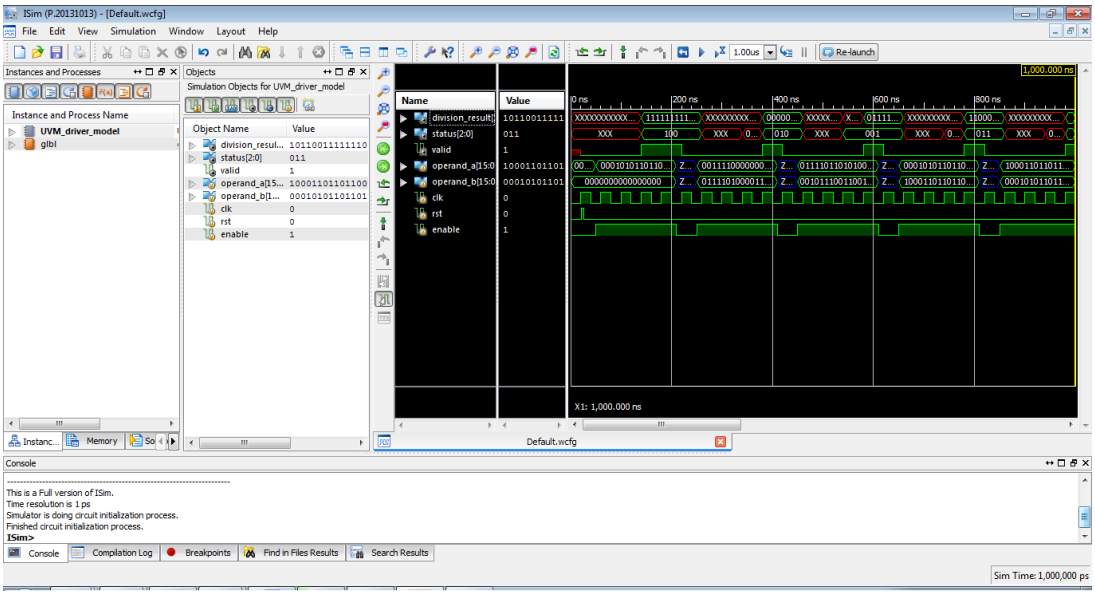

Figure 3.2: A simulation screen from ISE integrated ISIM simulator

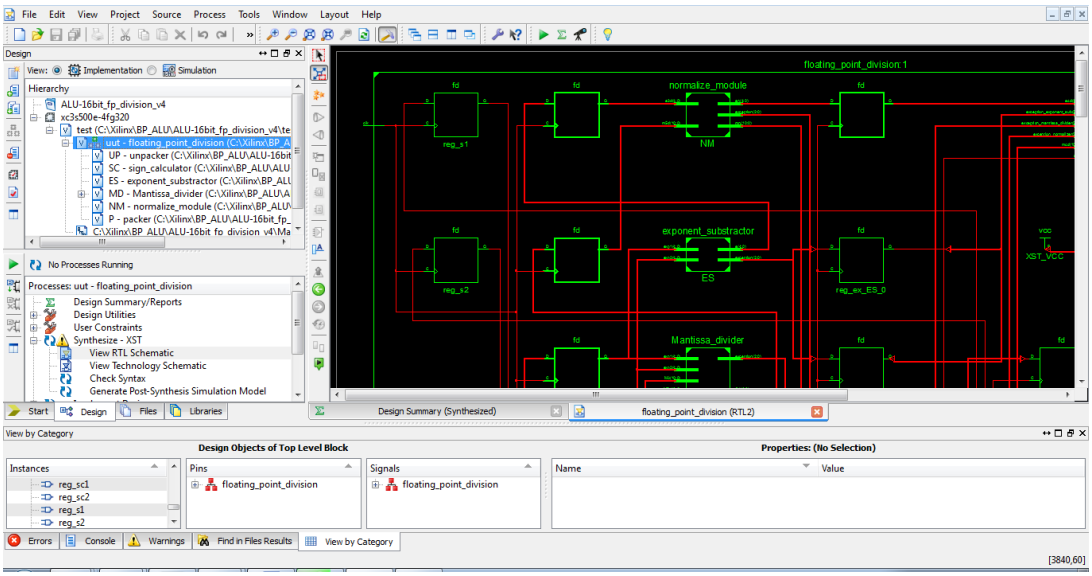

Figure 3.3: An RTL schematic in display at ISE

3.2. SVEditor

SVEditor is an Eclipse-based IDE (Integrated Development Environment) [21] supporting Verilog and SystemVerilog languages. It offers a colorized editor with syntax-checking, also supporting with source navigator, content and documentation assistance and SystemVerilog templates. It is also possible to import UVM libraries as source templates and properly check the code lines containing UVM based class extensions and methods.

SVEditor was used to develop and check UVM codes written in SystemVerilog.

.

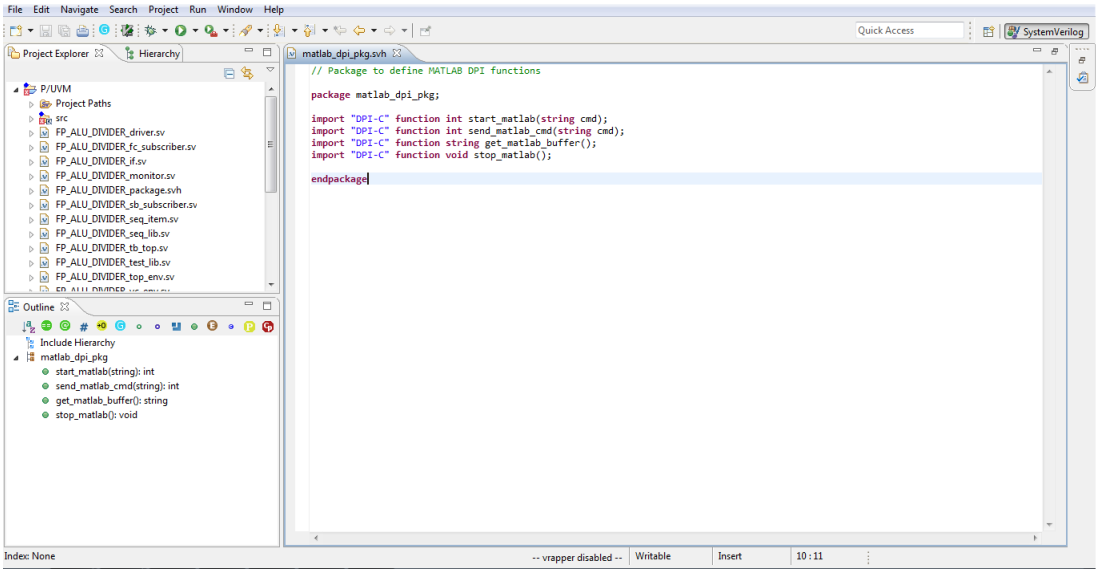

Figure 3.4: SVEditor design environment

3.3. QuestaSim

QuestaSim is a functional verification simulator of Mentor Graphics', developed as an integrated platform under Questa Advanced Simulator software, which is the core simulation and debugging engine of the Questa Verification Platform [7], and powered by ModelSim tool, which both are the software products belonging to same company.

QuestaSim offers higher capacity and supports larger FPGA and SoC (System on Chip), where ModelSim supports smaller designs. Moreover, an important advantage of QuestaSim is its compatibility with verification since it has a basis consisting of Questa Verification IP and Accelerated Coverage Closure technologies [22].

Because of its verification supportive nature, QuestaSim is compatible with a large array of languages containing Verilog, SystemVerilog and VHDL; and offers support for SystemVerilog based UVM libraries in hardware description and verification.

QuestaSim was used for running the complete UVM testbench in order to observe the verification results.

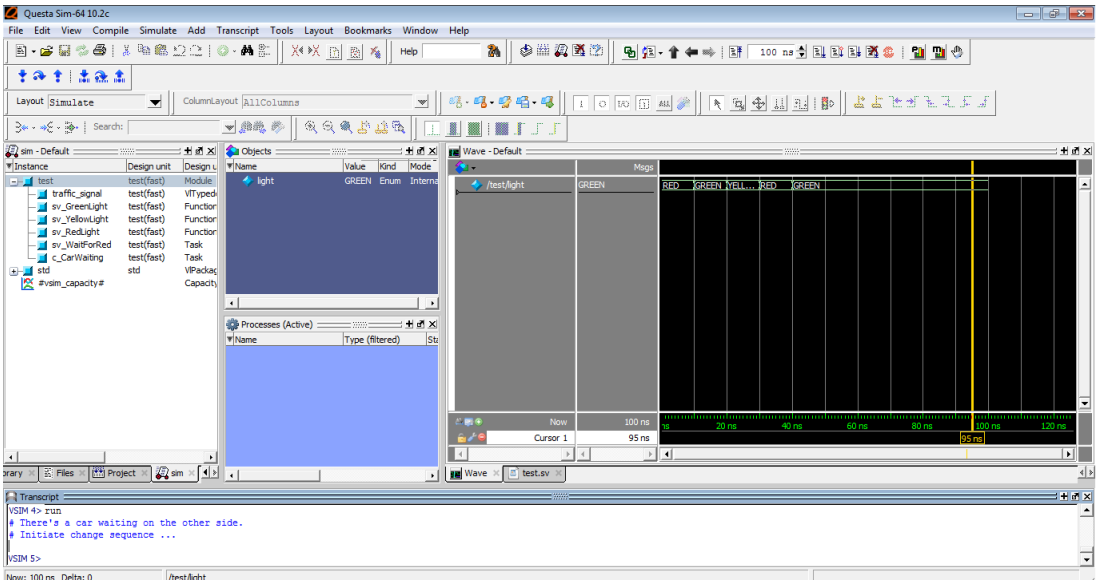

Figure 3.5: A screenshot from QuestaSim simulation environment

4. HARDWARE DESIGN

4.1. Overview

As the first step of the project, a design to be verified was required. Design to be tested needed to have enough complexity to be add new aspects to the previous work done in the same subject by one of the currently graduated students from ITU Department of Electronics and Communications Engineering, which was mainly aimed to introduce the UVM and therefore used a simple adder circuit as the design to be tested [3]. In order to add up to this work, the 16-bit floating point ALU design has been chosen. This ALU design was based of Aahrty M. and Dave Omkar R.'s paper titled "ASIC Implementation of 32 and 64 bit Floating Point ALU Using Pipeline" (Application Specific Integrated Circuit - ASIC) [23].

The floating point unit specifications are as listed as below:

- Based on IEEE-754 half precision number format
- Contains four operations: Addition, subtraction, multiplication and division.
- Contains a pipeline implementation

The hardware design process is separated into three as addition-subtraction module, multiplication module and division module; where each module have been designed by one of the contributing graduate students. The division module was designed in this work, hence the process of the division module design will be denoted in detail.

4.2. Floating Point Division Module

4.2.1. Top Module

The top architecture of the division module contains two 16-bit inputs to represent operands, named "operand_a" as the dividend and "operand_b" as the divisor; with three outputs called "division_result", which is an 16-bit output to show computation result, a one bit output named "valid", which sets itself when the division result has calculated, and a 3-bit long "status" output to show whether there is an exception occurred during computation or not. There are also three one-bit control inputs; one for the clock (positive edge triggering), one for the reset (active high reset) and one

as an enable signal. When the enable signal is low, the circuit stays idle during the rising clock cycles.

In the case of exceptions, which is represented by "status" output; there are five possible states that can occur after a division:

- Overflow: Occurs when the division result is greater than the highest absolute value that is representable. In this case, the result is set to the maximum absolute value representable with its corresponding sign bit.
- Underflow: Contrary of the overflow, it occurs whenever the computed result is smaller than the minimum absolute value that is representable. In this case, the result is set to zero.
- Division by Zero: Occurs when the divisor input acquired as zero. This situation sets all sixteen bits of the result output to high, representing the infinity, or unavailability.
- Result Zero: This status value is seen when the division result is zero, but there is no underflow occurred. In other words, it will be seen when the dividend input is zero.
- Normal Operation: Normal operation flag will be seen at status bits when none of the special conditions described above are present.

The bit codes for each of the cases described above are as follows:

Table 4.1: List of bit codes for each exception in division module

Status Case	**Bit Code**
Result Zero	000
Overflow	001
Underflow	010
Normal Operation	011
Divided by Zero	100

The top module consists of six submodules; unpacker, sign calculator, exponent subtractor, mantissa divider, normalizer and packer. Design of each module will be told in detail in their respective sections. All submodules are combinational circuits.

In terms of pipelining, four different layers are presented, each seperated with a set of register blocks. Therefore, the computation result is given in fourth rising edge of the clock after the computation starts.

The input and output schematic from RTL top layer is given below.

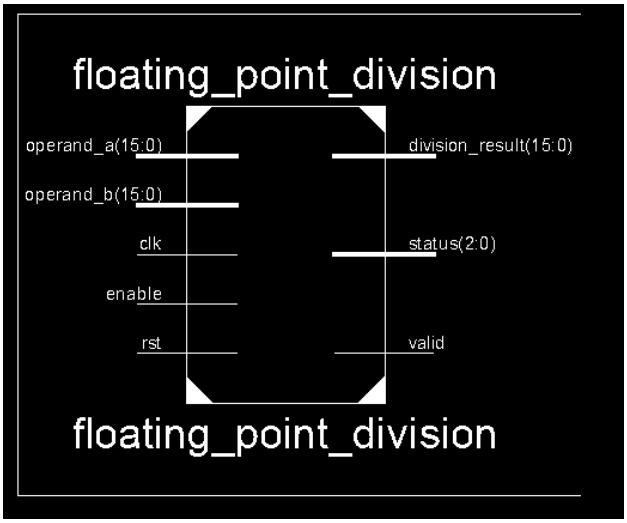

Figure 4.1: Top layer RTL schematic of top module

A schematic of the top layer circuit is presented below.

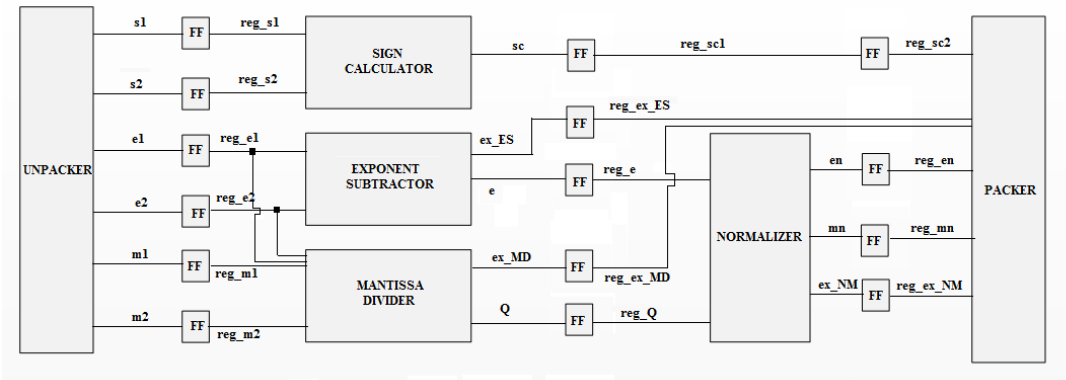

Figure 4.2: The circuit schematic of division module

The Verilog code for the top module can be examined from "FP_ALU_DIVIDER.v" file in the disk.

4.2.2. Unpacker

Unpacker module is where the operands are separated into their sign, exponent and mantissa bits. Moreover, the hidden high bit in front of each mantissa is added here. Separated bits later sent to their appertaining modules to be processed. The module's inputs are the two operands, and outputs are their separated sign, exponent and mantissa bits.

The unpacker alone also forms the first layer of the pipeline.

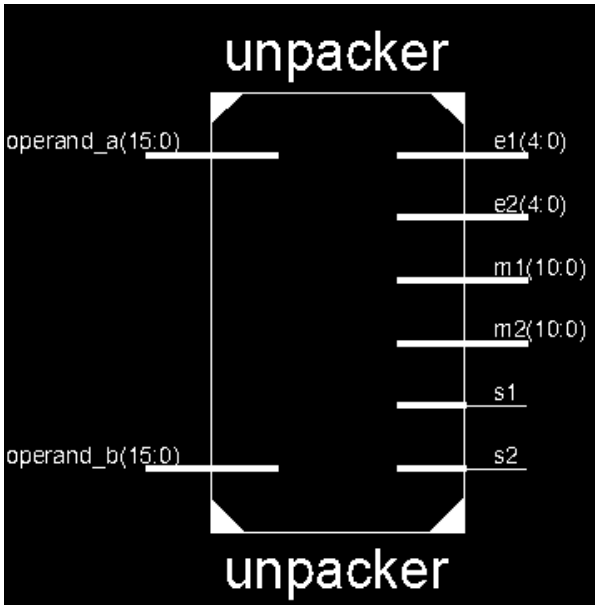

Figure 4.3 : Top layer RTL schematic of unpacker

The Verilog code for the unpacker is presented at Appendix A.1.

4.2.3. Sign Calculator

The sign calculator determines the output sign by evaluating operand sign bits. Since the division of two positive operands (sign bit zero) or two negative operands (sign bit one) are positive and division of operands with different signs results in negative sign, a simple XOR operation is used to determine the result sign.

This module is one of the three elements in second pipelining layer.

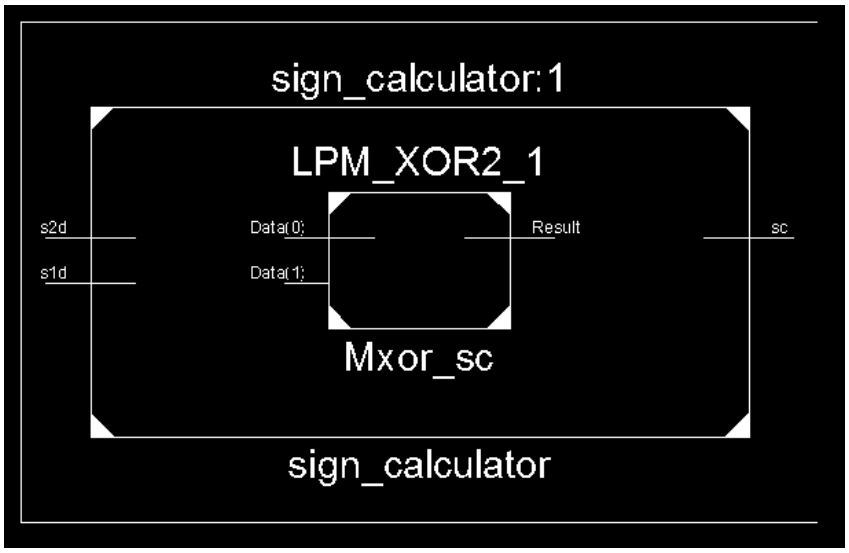

Figure 4.4: RTL schematic of sign calculator

The Verilog code for this module can be seen from Appendix A.2.

4.2.4. Exponent Subtractor

At this module, exponents of the two operands are subtracted, the divisor's exponent is subtracted from dividend's exponent, and then the bias (found as 15 from Equation 2.2) is added. Afterwards, the calculation result is driven to the module output.

After the computation, the resulted exponent must be checked if it is greater than 30 or smaller than 0, which in this cases they won't be represented as they should and an overflow or underflow will happen. Result of the check is stored in the exception output, to be later passed into the packer module in order to be evaluated in status decision.

This module is one of the three modules in that form the second pipelining layer.

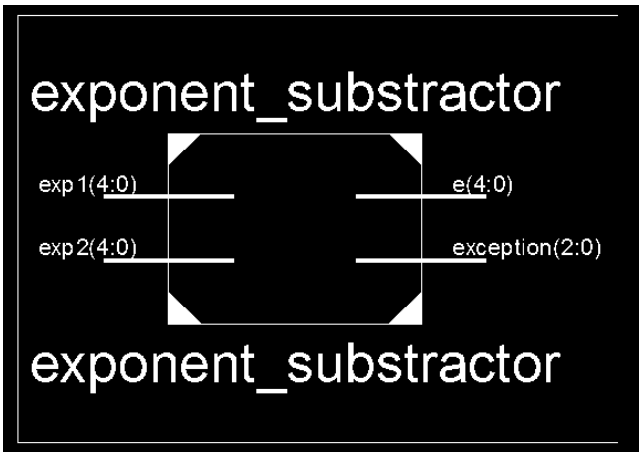

Figure 4.5: Top layer RTL schematic of exponent subtractor

The Verilog code for this module is given at Appendix A.3.

4.2.5. Mantissa Divider

Mantissa divider is the module where the significand's division is performed. There are more than one algorithm present for performing binary division in this module. The restoring division algorithm is used.

Restoring division algorithm performs one division for each bit in mantissa (including the hidden bit). The process starts from the initial mantissa inputs delivered from unpacker module. It is checked if the dividend is greater or lesser than the divisor mantissa (checking if dividend mantissa is completely dividable with divisor mantissa) and it is processed according to this.

To express better, the algorithm will be expressed in coding jargon;

Let the dividend mantissa called A and the divisor mantissa called B, initially;

```
for (i=10, i<=0 , i--)
{
    if(A >= B)
    {
         Q[i] = '1';
         R = A - B;
    }
    else
    {
        Q[i] = '0';
        R = A;
    }

A = 2*R;
}

D = Q;
```

In this expression, Q is the quotient bit; when it is '1' it means there is one times B in A, and when it is '0' means there is no B in A. Q is eleven bit long output where each bit represents the division result for its corresponding cycle, forming a eleven bit long mantissa division result when there are 11 cycles completed. The Q bits are filled starting from the most significant bit to the least significant bit. The R represents the remainder; if there is no B in A, the remainder is set as equal to A, and if there is one B in A, remainder is set to their subtraction. At the end of the cycle, the remainder is shifted right one time, then it becomes the new dividend and a new cycle starts. This operation is performed until all quotient bits are filled. Finally, the quotient bits form the mantissa division result, which is called D.

This loop operation was performed using three different cell blocks in Verilog; one for the initial iteration and one for the last iteration and one for second to tenth iterations. It is done so since the first and last iterations have differently connected input and output schemes than the second to tenth iterations,

It should also be noted that, although there is no need to store each block's dividend, remainder and quotient; it is still done so in order to be able to easily check them during simulation process.

There is also an exception output present in the module, to check if dividend or the divisor is zero, setting the exception bits accordingly.

This module is the last element of the second pipelining layer.

The Verilog code for this module can be examined from Appendix A.4.

4.2.6. Normalizer

Normalizing module is where the division alignment is performed. It takes the mantissa divider result and exponent subtractor result as inputs. In this module, the computed mantissa is restored to the form with the hidden bit '1' in the most significant bit ($1.mantissa_2$). To achieve this, the mantissa is shifted right until there is a '1' comes to the most significant bit. No shift is required if there is already a '1' in the most significant bit. If there will be no one's in the mantissa stream, a zero will be put into the most significant mantissa bit instead. After the shift operations are performed, the number of shifts performed must be deducted from the exponent.

After the subtraction from the exponent, it is required to check if the exponent became less than zero. In the case of exponent being less than zero, the exception bits will be set to underflow values. Exceptions will be set to normal operation values if exponent is equal or greater than zero.

In the Verilog code, it is preferred to write the exception checking part in the form of logical expressions.

This module solely forms the third layer of the pipelining implementation.

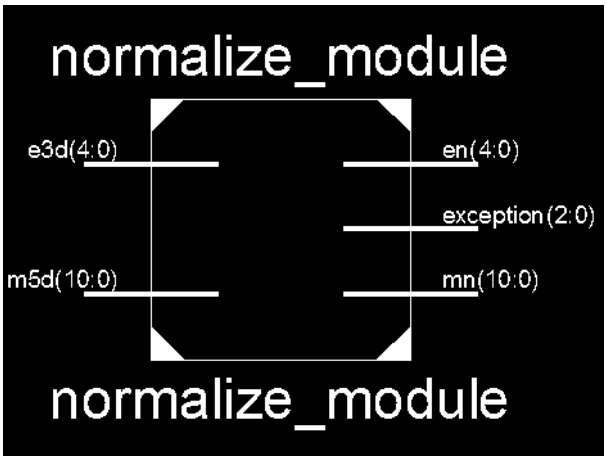

Figure 4.6: Top layer RTL schematic of normalizer

The Verilog code for the normalizer is presented at Appendix A.5.

4.2.7. Packer

Packer module is the location where the computed sign, exponent and mantissa are combined together to form the actual division result. The hidden bit of the mantissa is also removed here before repacking.

The module acts as an exception processer as well; all the exception outputs from previous modules are collected here and the status output of the top module is determined according to assigned priorities of the collected exceptions. Priority order is constructed as division by zero, result zero, overflow, underflow and normal operation; from highest to lowest.

Packer module also sets the special values for the output, like the infinity value in the case of division by zero and rounding down to the highest absolute value in case of an overflow.

Packer module is the fourth and the last layer of the pipelining implementation.

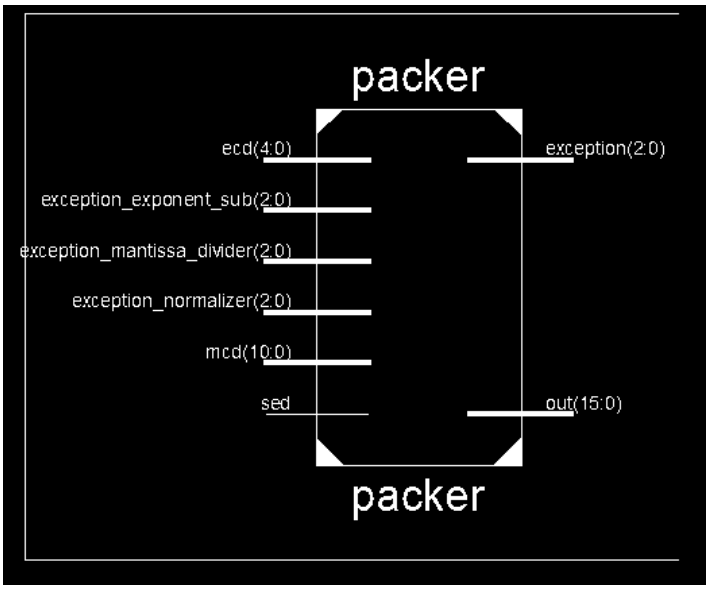

Figure 4.7: Top layer RTL schematic of packer

The Verilog code for the packer module can be examined from Appendix A.6.

4.3. Behavioural Simulation

Right after the design, before going into the verification step; the created circuit is put upon two different behavioural simulations; using samples for all special cases and for divisor and dividend pairs with different degrees, along with several random stimuli. The values that calculated are checked using a half precision calculator.

It should be noted that there may be small differences between the results like a difference in the least significant bit. This is a result of the usage of different structures and algorithms between unit under test and the reference calculator, Since the differences occur in the bits with less significance, the error should be safely assumed absolutely minimal.

The simulations are performed with ISE Design Suite's ISim tool.

4.3.1. Basic Simulation

For the first simulation, user defined the stimuli has given to the circuit inputs, each specially determined to cover boundary values and all five status states that could occur.

This first simulation does not benefit from the pipeline structure. Rather, the inputs are given in a controlled manner; five clock cycles at a time.

Below is the simulation result of the division module. Since it is not possible to show the half precision floating point equivalents of the circuit on the simulation environment, they are denoted in the test code as comments. The test code can be seen at "UVM_driver_model.v" file in the disk.

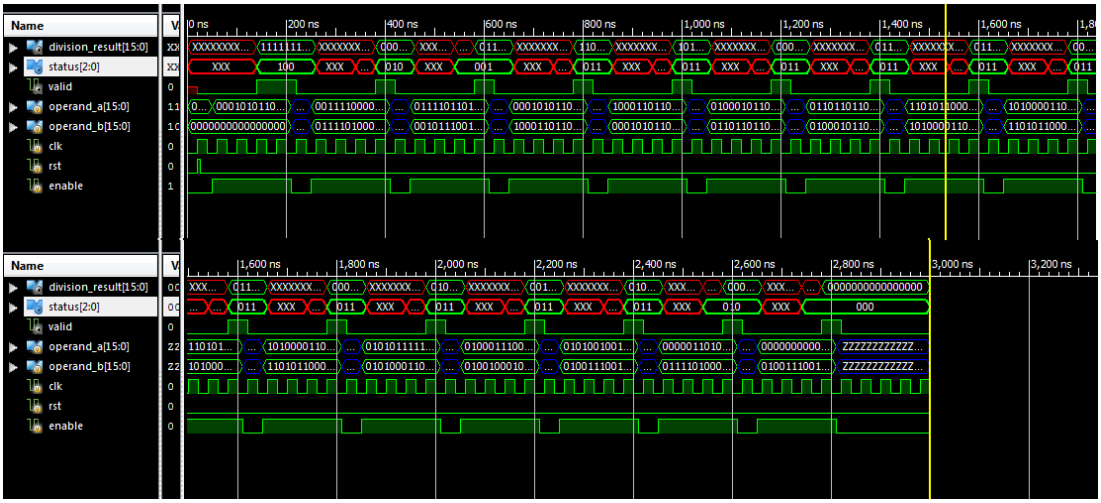

Figure 4.8: Waveforms of division module's basic simulation

Each of the results are checked using a half precision calculator.

The control mechanism used in this test was also used later in the verification part as one of the UVM components (UVM_driver), information on this subject can be seen at Section 5.4.5.

4.3.2. Pipeline Testing

Second test uses pipelining implementation and random inputs. With each rising clock edges, a pair of random stimuli applied to the circuit and after four clock cycles the particular inputs are given, the result is expected to be seen in the outputs corresponding to that input pair.

The test code was written so as the user can define the number of input pairs to be driven to the division module in one line.

A simulation waveform is given below, it used ten test input pairs for easier showcase.

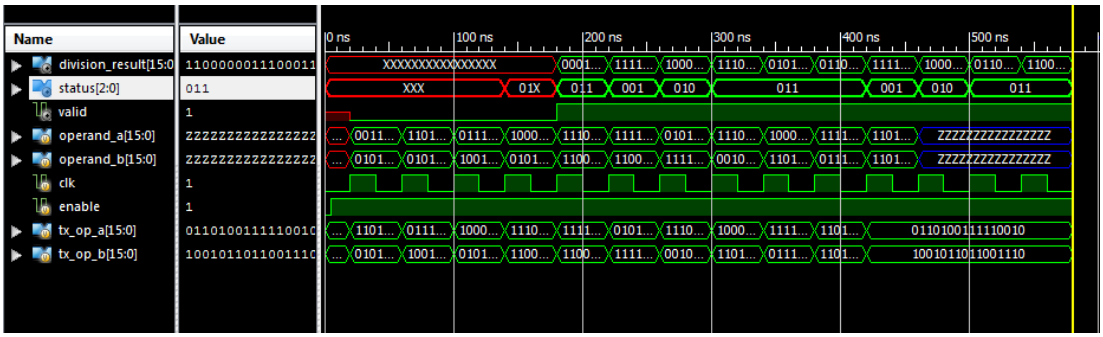

Figure 4.9: Waveforms of division module's pipelining simulation

Seeing from the simulation result, it could be observed that the circuit can give the result for the consecutive inputs with each rising clock edge. This validates that the pipeline implementation is successful.

The test code for this testbench is present at Appendix A.7.

5. FLOATING POINT ALU DIVISION MODULE VERIFICATION

5.1. General Structure

In digital system verification, the main construction is a top module that contains a design to be tested, with various component to create a systematic stimulus feeding and result checking module, called tests. Depending on the application and preference, there might be more than one test modules.

On the main layer of the verification environment, there is an interface component used to perform communication between the design under test (DUT) and the test module or modules.

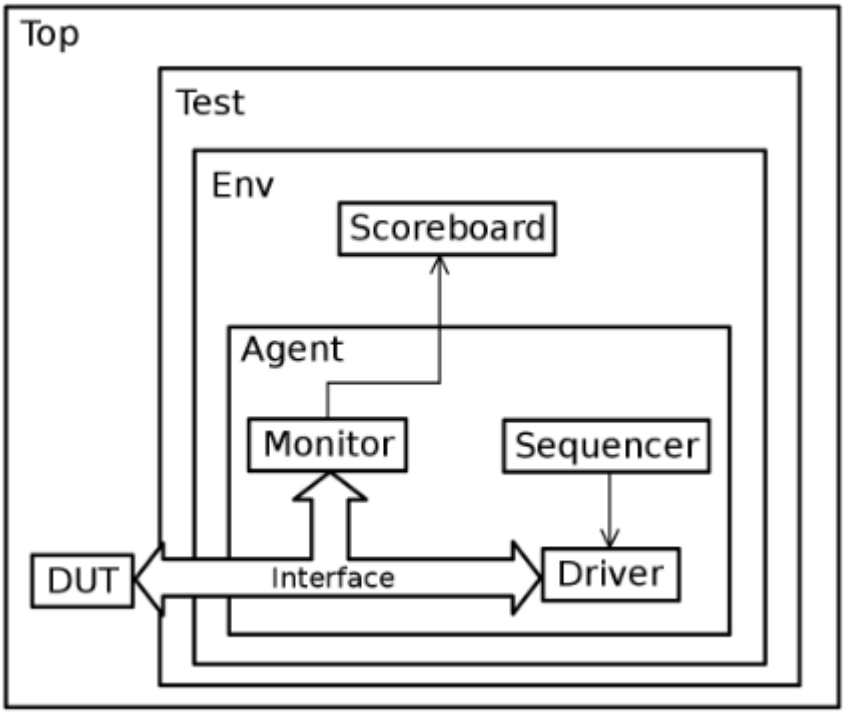

Figure 5.1: A typical UVM top module [24]

Summarily, in a typical UVM testbench; the stimuli are generated in the sequencer and then fed into the driver. Driver is responsible to give the stimuli into the communication bus in a controlled manner, the data then gets evaluated in the DUT and then gets sampled at the monitor, where the input of the DUT is sampled as well.

The sampled inputs and outputs later driven into the scoreboard to be evaluated and compared. Scoreboard generally contains a predictor section to compare DUT outputs with predicted results.

5.2. UVM Top Module

A top module in UVM is where the main test objects and DUT is mounted. It contains created samples of these both entities. A top module can contain more than one testbeches to be used according to application needs.

The DUT and the test top modules are connected with units called interfaces. Interface furnishes the input/output traffic between these components. Interfaces also have an internal branch between monitor and driver components of the testbench, more information will be given on the specified sections of this components. In this project, the clock and reset signal used to control the system is also generated here.

In the coding of the top module; instances of DUT, testbench and intefrace should be created; along with a clock and reset to control the environment. Therefore, all the required declarations such as sub-module files to be included, UVM package importations and DUT source codes. Moreover, the connections to attach interface and the other components are needed to be done here. The files to be imported are referenced from a header file called package. Codes for top module and package are given in Appendix B.1. and Appendix B.2.respectively.

Below is a basic schematic of a UVM top module.

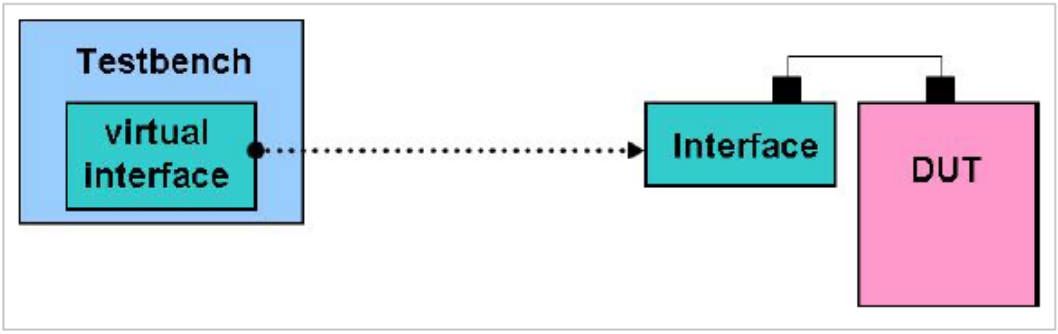

Figure 5.2: Interaction of top module elements [25]

5.3. Interface

It was previously stated that the interface is the communication component between the DUT and the test top module. It delivers the test stimulus created to the DUT from the outside of the test module; and delivers it to the monitor and also returns the DUT outputs to the same or a different monitor component inside the agent module (see Sections 5.4.2. and 5.4.3.), using virtual interfaces as braches. All these communication must be done in a controlled manner, so there should be several controlling protocols, which is generally run by driver module (see Section 5.4.5.). In this aspect, the interface can be thought as a bundle of intelligent wires that provides synchronization and connectivity [26].

In the coding of the interface; the connectors declared at the top module must be appropriately attached. An error handling part also needed because during the data delivering phase, there should be no X or Z (X is the unknown logic value and Z is the high impedance symbol in Verilog and SystemVerilog) fetched.

Code for interface is given at Appendix C.

Below is a general view of a testbench-DUT connection via interface.

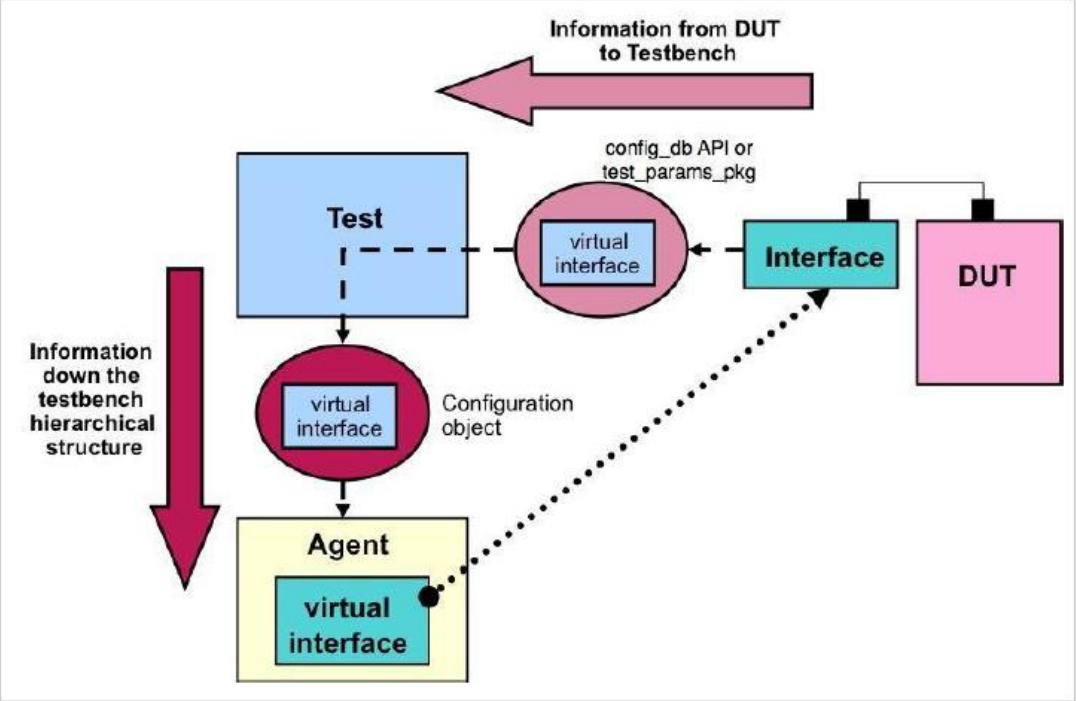

Figure 5.3: Interface's connection with other components [25]

5.4. UVM Components

5.4.1. Test Module and Environment

In UVM test module and environment components are used to implement verification environments. Environment is a part extended from uvm_env_class. A test object contains virtual systematic steps to be constructed; including building the components, connecting the components, running the test and reporting. An environment elaborates these steps using virtual methods inherited from uvm_env_classs, and they are specially called UVM phases.

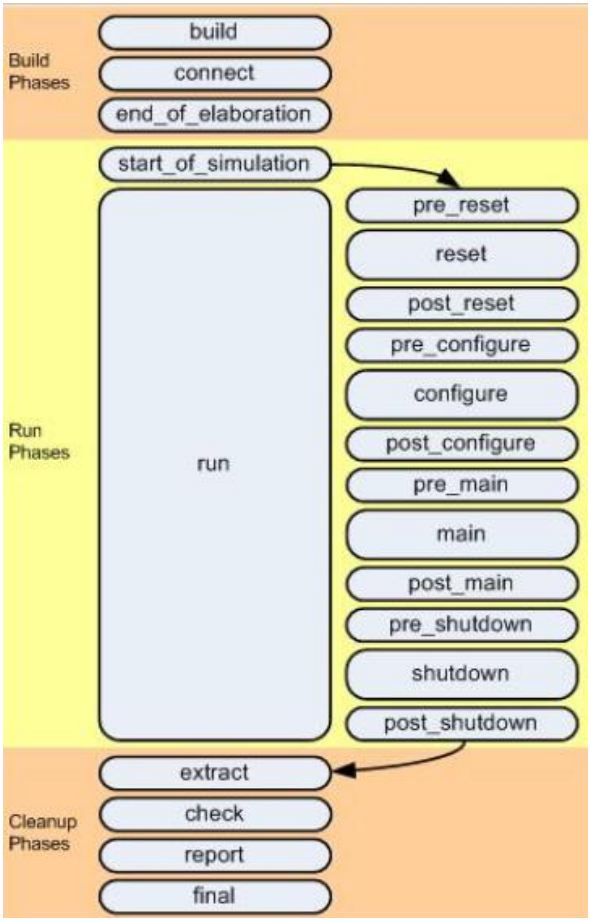

Figure 5.4: List of UVM phases [27]

Build phases are used in occasions where test components are created and configured. Running phases are used for management during simulation runtime of the testbench. Cleanup phases are used at the end, tasked to collect test results and reporting [3].

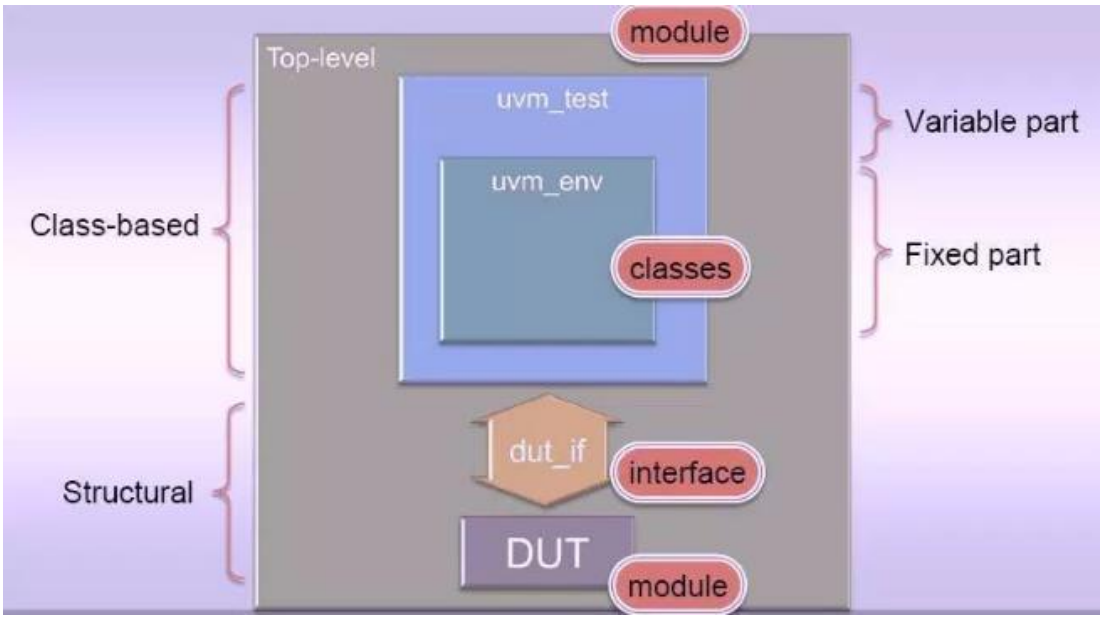

Figure 5.5: A showcase of top layer elements [28]

Test module contains an instance of the environment and specifies the application-specific test functionality [29]. Environment component makes declarations of virtual interfaces. These virtual interfaces are branched from the physical interface located in the top module, and they are pointed to this main interface in code. Henceforth, test module's section that is not in junction with the environment is called as variable part, meaning it is a dynamic section that is differing form application to application; whereas environment is a static structure and called as the fixed part. Since both are created from main UVM class extensions, this two components are also called class-based.

The environment used in the project has two partitions. One is the top environment and the other contains the environment components.

Codes for test and environment components are given at Appendices D.1, D.2 and D.3 respectively.

5.4.2. Agent

An UVM agent is derived from the uvm_component class. Agent contains the monitor, sequencer and driver components and connects them together. Since there is no simulation process present in the agent, it will only have build and connect phases. When active, agent is connected to all those three components mentioned above, where just being connected with scoreboard during passive phase. These connections are made with analysis ports.

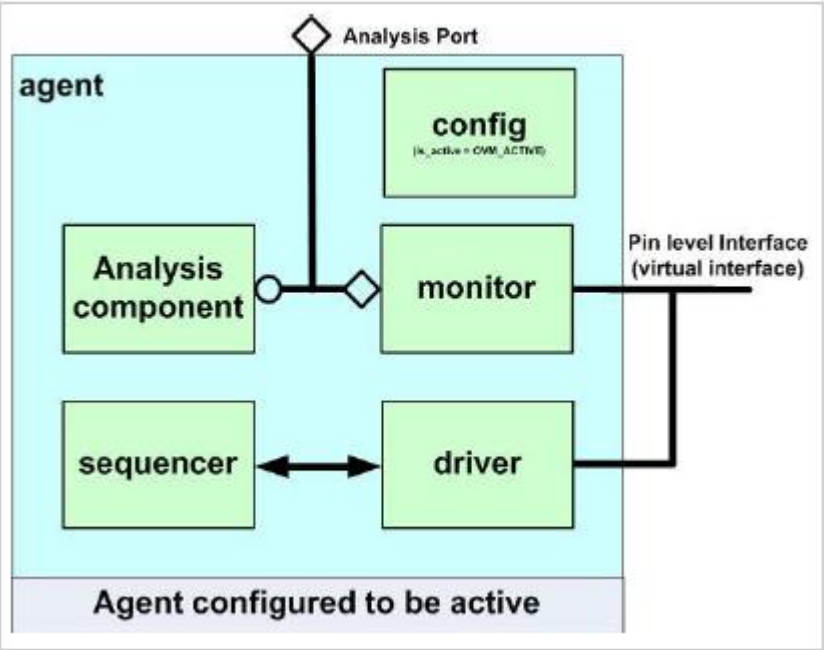

Figure 5.6: An active agent [30]

Analysis ports are based from transaction level modelling (TLM). TLM ports defines various functions and methods to allows communication of transaction objects.

In TLM, there are two aspects called consumer and producer which are connected together. For the verification testbench case, the consumer uses a function that takes the transactions as argument, where the producer uses that same function in passing the expected transaction to the consumer [31]. The two type of elements present in TLM communication is ports and exports. A port can be associated to just one export, but like in UVM verification; there may be cases when a port that can be plugged into more than one exports are required. From this need, TLM involves one more connection type called as analysis ports.

An analysis ports task is same as a normal port, the difference is in the connectivity. It can be connected to several exports and can be triggered by any of the connected exports on the same line whenever a function request arrives from them.

Table 5.1: Table of port types in TLM [31]

Symbol	Type	Port declaration
□	Port	uvm_blocking_put_port #(transaction) port_name
○	Export	uvm_blocking_put_imp #(transaction, classname) export_name
◇	Analysis Port	uvm_analysis_port #(transaction) analysis_port_name

This kind of communication is present in UVM, with agent through monitors among coverage, scoreboard and metric analyzers if any (metric analyzers not used in this project).

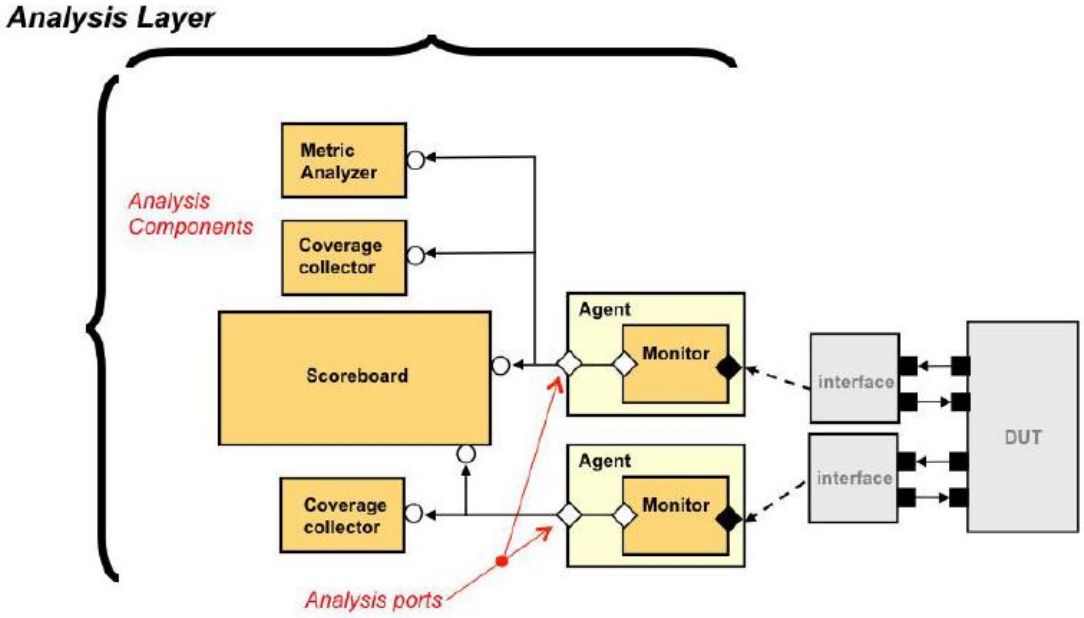

Figure 5.7: A picture of TLM analysis ports and exports in analysis layer [32]

The code for agent is presented at Appendix E.

5.4.3. Monitor

Monitor is a component within the agent, like the agent it is extended from uvm_component class. A monitor's task is to sample DUT outputs and driver inputs, then moving them to the analysis ports within the agent. These samples will be used at coverage and result checking steps later. Since it doesn't drive any signals into the DUT it is also a passive component. To avoid faulty execution, a control signal is required to inform the monitor when to take a DUT output sample. The valid signal of the division module is used to perform this control (Valid signal gets set high when the result is calculated). The monitor also needs to return error when unexpected behaviour occurs in the protocol.

Depending on the design, an environment can have more than one monitor.

Monitor code can be examined at Appendix F.

5.4.4. Sequencer and Sequences

The verification design needs to send testing inputs with specific data type to the DUT in order to perform any kinds of validation testing. The creation and first hand control of this test data is covered by transaction, sequence and sequencer component types in the UVM applications.

A transaction is an object which is generally extended from uvm_transaction or uvm_sequencer_item classes of uvm_components. The transaction is the smallest core part of what type of stimulus will be used for DUT inputs. It is mainly consisted of random variables of some data types, optionally specialized by various constraints or methods to increase accuracy of the tests to be performed.

Sequences are the elements that creates a series of transactions. A sequence is extended from uvm_sequence of uvm_component class. A sequence collects a specified amount of transactions and packs them to be sent to the driver when called. The sequence can further customize the transaction sets to be suitable to whatever specific aspects of the design should be tested.

There is one more class that's used to take the created sequence to the driver. It is named as sequencer and is generally used with the default sequencer class of UVM instead of extending it, since the default class usually be sufficient for applications.

It should be noted that the sequences and sequencer are not aware of the communication protocol in the verification environment, meaning they have a re-usability property in verification works when they are correctly programmed. Interacting these components with the communication protocol is the task of the driver component, that is described at Section 5.4.5.

Figure 5.8 shows the operation among sequence items, sequence and sequencer.

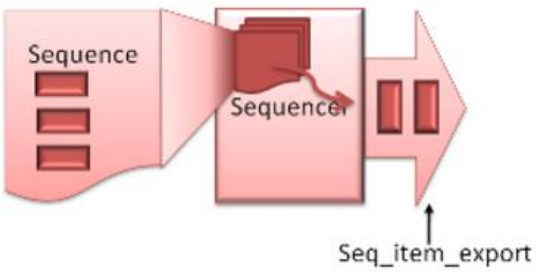

Figure 5.8: Image of the links between sequencer, sequence and sequence item [33]

Corresponding codes for sequence and sequencer are given at Appendices G.1 and G.2.

5.4.5. Driver

Driver is a component contained in the agent, and is extended from uvm_driver of uvm_component class. A driver interacts with the DUT by pulling sequence items from the sequencer and directing them to the DUT inputs with respect to a specified protocol, so the driver operates with pin level connection. A driver automatically terminates it's operation when the specified number of sequence items are driven. The driver's interaction with the sequence family can be seen from Figure 5.9.

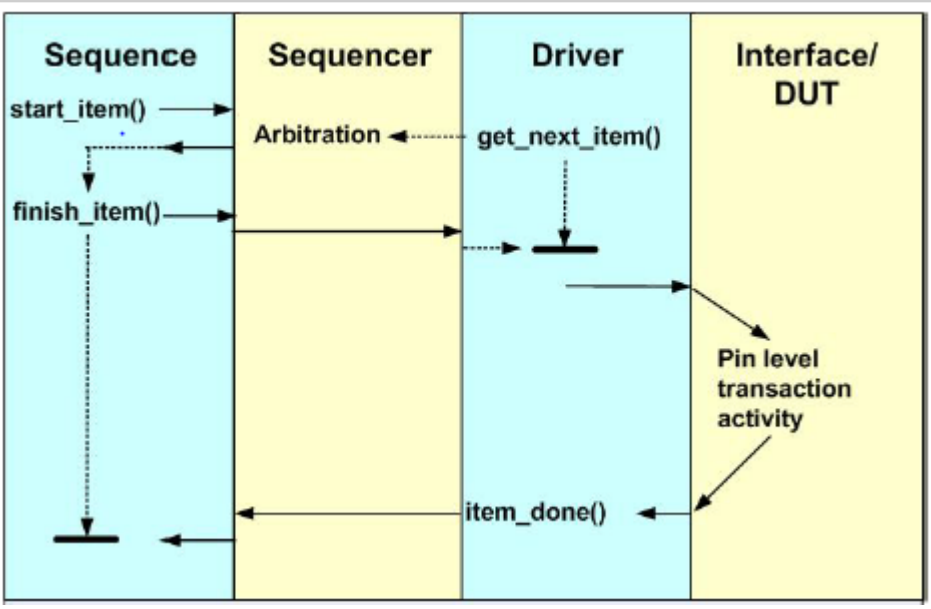

Figure 5.9: Sequencer-Driver-DUT transaction [34]

Before coding the driver protocol, it is recommended to implement it as a normal testbench first, using preferred design tool; for being sure it's operation is suitable to the needs. On this account; in the coding of the driver, the same protocol that's seen in the behavioural simulation in Section 4.3.1 and Figure 4.8 is modelled. The driver drives the input and sets the enable to start DUT's execution, then waits until the required time to pass for required clock cycles, then clears and the input and drives a new one. This protocol is essential for being sure an input will not be driven before the DUT finishes its calculation. Driver code is available at Appendix G.3.

5.4.6. Subscriber

Subscriber is a class extended from uvm_component class. It provides export to the analysis ports that receive transactions. Subscribers mostly used for helping coverage analysis on analyze ports. The subscriber class contains a "write" method for this exporting purpose [35]. When the write function gets a call with transactions as arguments, each subscriber points to the corresponding sequence item. That way it is possible for sequence items to be processed without mixing into each other.

There are two subscriber types present in the project. One used for code coverage and the other used for scoreboard ports.

Code coverage determines the percentage of how much of the code lines are covered by checking if every possible partition of the code have been moved upon. Code coverage percentage determines the quality of the tests performed.

Subscriber code used for analysis ports of this project can be seen at Appendix H.1.

5.4.7. Scoreboard

Scoreboard components are extended from uvm_scoreboard. Task of a scoreboard is to create predictions based on input sequences sent into DUT, then comparing these prediction results with actual DUT results. An error will be asserted when comparing result is unfavorable. These input and output streams are driven to the scoreboard ports from the monitor.

Scoreboard of the division module is designed together with its scoreboard subscriber element, that is it's analysis port subscribers.

Scoreboards contains a predictor section to create values for comparing with DUT outputs. Predictors does not have to be written in the SystemVerilog language. For complex designs, support from foreign languages required. It is possible to communicate with other programming languages. The way to performs this discussed in the next section.

Code for scoreboard is presented at Appendix H.2.

5.4.7.1. SystemVerilog Direct Programming Interface (DPI)

In some verification applications, core SystemVerilog language's offerings may not be sufficient to perform certain tasks. To cope with it, SystemVerilog offers a solution named as direct programming interface (DPI). DPI allows SystemVerilog to interact with foreign programming languages, like C or MATLAB. The user can use existing foreign language code after importing it with a special syntax, the import "DPI" declarations, or export the SystemVerilog code to another supported language [36].

Because it was not possible to perform half precision division in a small code piece with the current SystemVerilog data types, DPI feature was used to call a created MATLAB function.

5.4.7.2. MATLAB Predictor Module

UVM testbenches are able to interact with MATLAB [37]; mostly in the forms of using MATLAB functions as DUT's (because MATLAB functions' ability to be transformed into RTL (register-transfer level) library blocks.), and scoreboard checker. MATLAB provides some mechanisms for interacting with UVM. These mechanisms could be presented as following:

- Running in parallel with HDL simulators for computation.
- Supporting Verilog modules with MATLAB functions
- By using SystemVerilog DPI, MATLAB functions can be compiled into a shared library to be used in SystemVerilog environment [38].

The predictor function is a quick recreation of the half precision division algorithm used in the ALU divider. The inputs are sent from the SystemVerilog DPI as signed integers, so the string based binary conversion functions presented in MATLAB are used to convert incoming integers to half precision floating point system (MATLAB does not include a built-in data type for half precision floating point). Evaluated results then also will be sent to verification environment on DPI as signed integers as well, hence the processed data needed to be re-converted into signed integers at the end.

It should be noted that the MATLAB file to be used as DPI function must be written in function declaration format. Files written in script format are not supported in DPI.

Required explanations about methods used in development of the predictor function are present as comment lines on the code.

MATLAB code for the predictor can be seen at "float16div.m" file presented on the disk.

The C file to be used as a connector between DPI and MATLAB is at Appendix I.1.

SystemVerilog definitions of the MATLAB functions are included in a.svh file, which is given in Appendix I.2.

6. RUNNING THE UVM TESTBENCH WITH QUESTASIM

The last step of the project was to run the program using the QuestaSim tool and observe the verification results. However, due to various complications during the usage of the program, the simulation couldn't be performed and observed. It is aimed to fix the errors occurred and perform the simulation until the presentation of the project.

7. CONCLUSION

In this graduation project, the main goal was to create a 16-bit floating point ALU design from scratch and perform a verification process, chosen as UVM, on the created design; with the collaboration of three different BSc students.

The ALU design has been divided into three parts, and the division part is designed in this project. The design has been performed in Verilog language with Xilinx ISE tool. Afterwards, some basic simulations are generated in order to make a general control on the circuit to see if it seemingly performs the operations as desired. Then, each three parts are planned to be verified separately. In order to perform this, a verification environment was built using UVM libraries with SystemVerilog language. Each component for this verification environment has been created individually and then all of them have been put on a top module accordingly, with the design under test and test module on a top module; connected with an interface component. The test module is set so as it will generate a series of random input transactions to test the design under test. The results then tracked with their corresponding input values and compared with the expected values, predicted using an external programming language code, using the SystemVerilog's DPI feature. A MATLAB code has been designed as a predictor in this case.

At the end of the project, the created design was moved to the QuestaSim environment in order to perform the verification. However, the verification has not been performed due to various errors and lack of configurations.

In conclusion, it seems that the UVM verification is an hard and long process to learn, but it is though that once it could learnt, it would be easier to design different verification environments for different digital circuits, due to the reusability and convertibility of the UVM testbenches. It is also predicted that the importance of the UVM will keep increasing greatly as the digital design evolves.

REFERENCES

[1] **Shiva, S.G**., 1998. Introduction to Logic Design (2nd Ed.), University of Alabama in Huntsville Huntsville, Alabama.

[2] **Accellera,** 2011. Universal Verification Methodology (UVM) 1.1 User's Guide

[3] **Çoktaş G.,** 2014. Bir Sayısal Sistem Tasarımının Evrensel Doğrulama Metodu ile Doğrulanması, *BSC Thesis*, I.T.U. Faculty of Electrical and Electronics Engineering, İstanbul.

[4] **IEEE,** 2006, IEEE Standart for Verilog Hardware Description Language, 10016-5997, New York.

[5] **Xilinx,** ISE Design Suite, http://www.xilinx.com/products/design-tools/ise-design-suite.html

[6] **Cadence Website**, Demonstration on Advanced UVM Using Incisive Platform - Part 2, [Reference Date: 10 March 2016], http://www.cadence.com/alliances/languages/pages/uvm.aspx

[7] **Mentor Graphics**, Questa Verification Environment, [Reference Date: 13 March 2016], https://www.mentor.com/products/fv/questa/

[8] **Wikipedia,** Floating Point, [Reference Date: 10 March 2016], https://en.wikipedia.org/wiki/Floating_point

[9] **Muller et al.**, Handbook of Floating Point Arithmetic, Birkhauser Boston, Basel, Berlin.

[10] **Wikipedia**, IEEE Floating Point, [Reference Date: 10 March 2016], https://en.wikipedia.org/wiki/IEEE_floating_point

[11] **Wikipedia**, Half-Precision Floating-Point Format, [Reference Date: 10 March 2016], http://en.wikipedia.org/wiki/Half_precision_floating-point_format

[12] **Ball, S.R.,** 2002. Embedded Microprocessor Systems: Real World Design (3rd Ed), Elsevier Sceince, USA

[13] **Araujo, P**., UVM Guide for Beginners, [Reference Date: 13 March 2016], https://colorlesscube.com/uvm-guide-for-beginners/

[14] **IEEE,** 2012, SystemVerilog 1800-2012 IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language

[15] **Accellera,** Accellera Solutions Website, http://www.accelerasolutions.com/

[16] **Wikipedia**, Open Verification Methodology, [Reference Date: 23 May 2016], https://en.wikipedia.org/wiki/Open_Verification_Methodology

[17] **Mentor Graphics,** Verification Academy AVM Cookbook, https://verificationacademy.com/cookbook/avm

[18] **Synopsys,** VMM Verification Methodology, [Reference Date: 23 May 2016], https://www.synopsys.com/community/interoperability/pages/vmm.aspx

[19] **Mentor Graphics,** Verification Academy UVM Cookbook, Introduction, [Reference Date: 29 March 2014], https://verificationacademy.com/cookbook/uvm

[20] **Synopsys,** OpenVera Website, http://www.open-vera.com/

[21] **Eclipse,** Eclipse Website, https://eclipse.org/

[22] **Wikipedia**, Mentor Graphics, (2013). [Referene Date: 13 March 2016], http://en.wikipedia.org/wiki/Mentor_Graphics[23] **Aarthy M. and Omkar D.R**., 2014, ASIC Implementation of 32 and 64 bit Floating Point ALU using Pipelining, *International Journal of Computer Applications,* **94**,0975 – 8887.

[24] **Araujo, P**., UVM Guide for Beginners - Defining the Verification Environment, [Reference Date: 5 April 2016] https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/

[25] **Mentor Graphics,** Verification Academy UVM Cookbook, Connections to DUT Interfaces, [Reference Date: 2 April 2016], https://verificationacademy.com/cookbook/uvm

[26] **Spear, C**., 2008, SystemVerilog for Verification (2nd Ed.), Springer, Marlboro MA, USA.

[27] **Mentor Graphics,** Verification Academy UVM Cookbook, Phasing, [Reference Date: 3 April 2016], https://verificationacademy.com/cookbook/uvm

[28] **Singhal, M.,** Application of Virtual Interface and uvm_config_db**,** [Reference Date: 5 April 2016], https://verificationacademy.com/sessions/uvm-sequences-and-tests

[29] **Krishna, G. and Maddipati, N.,** Easy Labs: UVM - Phase 3: Environment and Testcase, http://www.testbench.in/UL_06_PHASE_3_ENVIRONMENT_N _TESTCASE.html

[30] **Mentor Graphics,** Verification Academy UVM Cookbook, Agent, [Reference Date: 8 April 2016], https://verificationacademy.com/cookbook/uvm

[31] **Araujo, P**., UVM Guide for Beginners - Monitor, [Reference Date: 5 April 2016] https://colorlesscube.com/uvm-guide-for-beginners/chapter-6-monitor/

[32] **Mentor Graphics,** Verification Academy UVM Cookbook, Analysis Components and Techniques, [Reference Date: 11 April 2016] https://verificationacademy.com/cookbook/uvm

[33] **Krishna, G. and Maddipati, N.,** Easy Labs: UVM - Phase 5: Sequencer and Sequence, https://www.testbench.in/UL_08_PHASE_5_SEQUENCER_N_SEQUE NCE.html

[34] **Mentor Graphics,** Verification Academy UVM Cookbook, Driver/Sequence API, [Reference Date: 11 April 2016] https://verificationacademy.com/cookbook/uvm

[35] **Mentor Graphics**, UVM Class Reference 1.1c, [Reference Date: 14 April 2016] https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1c/html/files/comps/uvm_subscriber-svh.html#uvm_subscriber.write

[36] **Aynsley, J**., Doulos, SystemVerilog DPI Tutorial, https://www.doulos.com/knowhow/sysverilog/tutorial/dpi/

[37] **MathWorks,** MATLAB, http://www.mathworks.com/

[38] **Mentor Graphics,** Verification Academy UVM Cookbook, MATLAB/Integration, [Reference Date: 4 May 2016] https://verificationacademy.com/cookbook/uvm

APPENDICES

APPENDIX A.1

unpacker.v

APPENDIX A.2

sign_canculator.v

APPENDIX A.3

exponent_subtractor.v

APPENDIX A.4

```
/*-------------------------------------------------------------------------------------
Name : divider_module.v
Description: Performs mantissa division
-------------------------------------------------------------------------------------*/

`timescale 1ns / 1ps

module Mantissa_divider(
    input [10:0] MA,
    input [10:0] MB,
     input [4:0]exp1,
     input [4:0]exp2,
    output [10:0]Q,
     output [2:0]exception
    );

   wire [10:0]R[10:1];
    wire [10:0]A[10:1];
    wire [10:0]temp_Q;

   // Division
    Mantissa_divider_cell10 MDC10(MA,MB,temp_Q[10],R[10],A[10]);

   genvar i;
   generate
     for(i = 9; i > 0 ; i = i-1)
     begin: division
        Mantissa_divider_cell MDCi(A[i+1],MB,R[i+1][10:0],temp_Q[i],R[i],A[i]);
     end
   endgenerate

 Mantissa_divider_cell0 MDC0(A[1][10],MB,R[1][10:0],temp_Q[0]);

// (exp1==0 and MA==0) represents operand_a == 0, thus the division result will be set to 0
assign Q = (exp1==0 && MA[9:0]==0) ? 0 : temp_Q;

// Determining status
 assign exception = (exp2==0 && MB[9:0]==0) ? 4 :
                    (exp1==0 && MA[9:0]==0) ? 0 :  3 ;

endmodule
```

APPENDIX A.5

```
/*----------------------------------------------------------------------
Name : normalize_module.v
Description: Normalizes the mantissa back to the form of 1.[mantissa]
----------------------------------------------------------------------*/

`timescale 1ns / 1ps

module normalize_module(
    input [4:0] e3d,
    input [10:0] m5d,
    output [4:0] en,
    output [10:0] mn,
     output [2:0]exception
    );

  /* if the MSB of the result mantissa is 1, no normalization required */

 assign mn =       (m5d[10]==1)? m5d   :
                   (m5d[9]==1) ? 2*m5d :
                        (m5d[8]==1) ? 4*m5d :
                        (m5d[7]==1) ? 8*m5d :
                        (m5d[6]==1) ? 16*m5d :
                        (m5d[5]==1) ? 32*m5d :
                        (m5d[4]==1) ? 64*m5d :
                        (m5d[3]==1) ? 128*m5d :
                        (m5d[2]==1) ? 256*m5d :
                        (m5d[1]==1) ? 512*m5d :
                        (m5d[0]==1) ? 1024*m5d : 11'b00000000000;

assign en =       (m5d==0)    ? 0     :
                      (m5d[10]==1)? e3d   :
                  (m5d[9]==1) ? e3d-1 :
                       (m5d[8]==1) ? e3d-2 :
                       (m5d[7]==1) ? e3d-3 :
                       (m5d[6]==1) ? e3d-4 :
                       (m5d[5]==1) ? e3d-5 :
                       (m5d[4]==1) ? e3d-6 :
                       (m5d[3]==1) ? e3d-7 :
                       (m5d[2]==1) ? e3d-8 :
                       (m5d[1]==1) ? e3d-9 :
                       (m5d[0]==1) ? e3d-10 : e3d-11;

assign exception = (m5d[10]==1) ? 3 :
                   (m5d[9]==1)  ? ( ((~e3d[4] & ~e3d[3] & ~e3d[2] & ~e3d[1] & ~e3d[0])==1) ? 2 : 3) :
                        (m5d[8]==1)  ? ( ((~e3d[4] & ~e3d[3] & ~e3d[2] & ~e3d[1])==1) ? 2 : 3) :
                        (m5d[7]==1)  ? ( ((~e3d[4] & ~e3d[3] & ~e3d[2] & (~e3d[1] | ~e3d[0]))==1) ? 2 : 3) :
                        (m5d[6]==1)  ? ( ((~e3d[4] & ~e3d[3] & ~e3d[2])==1) ? 2 : 3) :
                        (m5d[5]==1)  ? ( ((~e3d[4] & ~e3d[3] & ( ~e3d[2] | (~e3d[1] & ~e3d[0]) ))==1) ? 2 : 3) :
                        (m5d[4]==1)  ? ( ((~e3d[4] & ~e3d[3] & (~e3d[2] | ~e3d[1]))==1) ? 2 : 3) :
                        (m5d[3]==1)  ? ( ((~e3d[4] & ~e3d[3] & (~e3d[2] | ~e3d[1] | ~e3d[0]))==1) ? 2 : 3) :
                        (m5d[2]==1)  ? ( ((~e3d[4] & ~e3d[3])==1) ? 2 : 3) :
                        (m5d[1]==1)  ? ( ((~e3d[4] & ( ~e3d[3] | ( ~e3d[2] & ~e3d[1] & ~e3d[0]) ))==1) ? 2 : 3) :
                        (m5d[0]==1)  ? ( ((~e3d[4] & ( ~e3d[3] | ( ~e3d[2] & ~e3d[1]) )   )==1) ? 2 : 3) : 2;

endmodule
```

APPENDIX A.6

```
/*----------------------------------------------------------------------
Name : packer.v
Description: Combines the calculated sign, exponent and mantissa
----------------------------------------------------------------------*/

`timescale 1ns / 1ps

module packer(
  input sed,
  input [4:0]ecd,
  input [10:0]mcd,
  input [2:0]exception_exponent_sub,
  input [2:0]exception_mantissa_divider,
  input [2:0]exception_normalizer,
  output [15:0]out,
  output [2:0]exception
    );

  /* Combine the calculated sign,mantissa and exponent to form the result. Drop the implied bit from the
     mantissa. */
  assign out = (exception_mantissa_divider==4) ? {1'b1,5'b11111,10'b1111111111} : // Division by zero will result in NaN
                  (exception_exponent_sub==1)  ? {sed,5'b11110,10'b1111111111}: // Overflow will bound the result at 65504
                   (exception_exponent_sub==2)  ? {sed,5'b00000,10'b0000000000}: // Underflow will round down the result to zero
                   (ecd==5'b11111)              ? {sed,5'b00000,10'b0000000000}: {sed,ecd,mcd[9:0]}; // Underflow will
                                                                                 // round down the result to zero
                                                                  // Assign the computed result if none of the exceptions ocurr.
  assign exception = (exception_mantissa_divider==4) ? 4 :
                     (exception_mantissa_divider==0) ? 0 :
                      (exception_exponent_sub==1)     ? 1 :
                            (ecd==5'b11111)                 ? 2 :
                            (exception_exponent_sub==2)     ? 2 :
                      (exception_normalizer==2)       ? 2 : 3;

endmodule
```

APPENDIX A.7

```
/*-----------------------------------------------------------------
Name : pipeline_test.v
Description: A testbench with a series of random stimulus, designed
    to test pipeline structure.
-----------------------------------------------------------------*/


`timescale 1ns / 1ps

module pipeline_test;

    // Inputs
    reg [15:0] operand_a;
    reg [15:0] operand_b;
    reg clk;
    reg rst;
    reg enable;

    // Outputs
    wire [15:0] division_result;
    wire [2:0] status;
    wire valid;

   // Integers
    integer counter_finish;

    reg [15:0]tx_op_a;
    reg [15:0]tx_op_b;

    // Instantiate the Unit Under Test (UUT)
    floating_point_division uut (
       .operand_a(operand_a),
       .operand_b(operand_b),
       .division_result(division_result),
       .status(status),
       .valid(valid),
       .clk(clk),
       .rst(rst),
       .enable(enable)
   );

parameter no_of_stim = 10;  // Number of test stimulus

// Initial values
initial begin
       clk = 1'b0;
       rst = 1'b0;
       enable = 1'b0;

       counter_finish = 0;

       tx_op_a <= $random;
      tx_op_b <= $random;
       #5;
       enable = 1'b1;
end

//Generates clock
initial begin
   forever #20 clk = ~clk;
end
```

```
// Specified amount of test results will be generated in this time period, by means of the pipeline structure.

always@(posedge clk)
begin
   counter_finish = counter_finish + 1;

    if(counter_finish < no_of_stim + 2 )
    begin
    operand_a <= tx_op_a;
    operand_b <= tx_op_b;
    tx_op_a <= $random;
    tx_op_b <= $random;
    end

    else
    begin
    operand_a <= 16'hz;
    operand_b <= 16'hz;
    end

    // Wait four more positive clock edges to allow DUT perform it's last operation
    if(counter_finish == no_of_stim + 5) $finish;
end

endmodule
```

APPENDIX B.1

```
/*-----------------------------------------------------------------------------
 Name : FP_ALU_DIVIDER_tb_top.sv
 Description   : Top testbench module - Contains DUT,interface and test components
-----------------------------------------------------------------------------*/

`include "FP_ALU_DIVIDER.sv"     // Import DUT
`include "FP_ALU_DIVIDER_if.sv"  // Import Interface
`include "FP_ALU_DIVIDER_package.svh"  // Import all other components which are
                                       // required as submodules

module FP_ALU_DIVIDER_tb_top;

  // Import UVM Package
  import uvm_pkg::*;

  // Import the FP_ALU_DIVIDER UVC Package
  import FP_ALU_DIVIDER_package::*;

  // Include the test library
  `include "FP_ALU_DIVIDER_test_lib.sv"

  // DUT I/O regs
  reg [15:0] operand_a;
  reg [15:0] operand_b;
  reg [15:0] division_result;
  reg [2:0] status;
  reg valid;
  reg clk;
  reg rst;
  reg enable;

// Interface instance to be connected with DUT
FP_ALU_DIVIDER_if dut_if();

// Create the DUT, connect it with the interface
FP_ALU_DIVIDER dut(
  .operand_a(dut_if.operand_a),
  .operand_b(dut_if.operand_b),
  .division_result(dut_if.division_result),
  .status(dut_if.status),
  .valid(dut_if.valid),
  .clk(dut_if.sig_clock),
  .rst(dut_if.sig_reset),
  .enable(dut_if.enable)
);

initial begin
  uvm_config_db #(virtual FP_ALU_DIVIDER_if)::set(null, "uvm_test_top", "vif", dut_if);
  run_test();
end

initial begin
  dut_if.sig_reset <= 1'b1;
  dut_if.sig_clock <= 1'b1;
  #21 dut_if.sig_reset <= 1'b0;
end

// Clock Generator
always
  #20 dut_if.sig_clock = ~dut_if.sig_clock;

initial
begin
   $recordfile ("./test.trn");
   $recordvars ("depth=0", FP_ALU_DIVIDER_tb_top);
end

endmodule
```

APPENDIX B.2

```
/*-----------------------------------------------------------------------------------------------
 Name : FP_ALU_DIVIDER_package.svh
 Description: Package is where all the files that used in UVC (Universal Verificatin Component)
 are imported
-----------------------------------------------------------------------------------------------*/

// Import UVM macros
`include "uvm_macros.svh"
//Import Matlab DPI
`include "matlab_dpi_pkg.svh"

// Package Definition
package FP_ALU_DIVIDER_package;

    // UVM class library compiled in a package
    import uvm_pkg::*;

    `include "FP_ALU_DIVIDER_defines.sv"
    `include "FP_ALU_DIVIDER_seq_item.sv"
    `include "FP_ALU_DIVIDER_monitor.sv"
    `include "FP_ALU_DIVIDER_driver.sv"
    `include "FP_ALU_DIVIDER_agent.sv"
    `include "FP_ALU_DIVIDER_fc_subscriber.sv"
    `include "FP_ALU_DIVIDER_sb_subscriber.sv"

    `include "FP_ALU_DIVIDER_seq_lib.sv"
    `include "FP_ALU_DIVIDER_vc_env.sv"

endpackage : FP_ALU_DIVIDER_package
```

```
// Description   :  This part declares the UVC seq_item.

typedef enum bit {FP_ALU_DIVIDER_EVEN, FP_ALU_DIVIDER_ODD } FP_ALU_DIVIDER_data_type_e;
```

APPENDIX C

```
/*------------------------------------------------------------------------
Name : FP_AU_DIVIDER_if.sv
Description  : Establishes connection between test environment and DUT
------------------------------------------------------------------------*/

// Define interface
interface FP_ALU_DIVIDER_if ();

  // Import UVM macros and package
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  // Interface I/O - DUT signals will be connected to these
    logic sig_clock;
    logic sig_reset;
    logic enable;
    logic valid;
    logic [15:0]operand_a;
    logic [15:0]operand_b;
    logic [15:0]division_result;
    logic [2:0]status;

  // Control flags
  bit has_checks = 1;
  bit has_coverage = 1;

  // SVA default clocking
  wire uvm_assert_clk = sig_clock && has_checks;
  default clocking master_clk @(negedge uvm_assert_clk);
  endclocking

 // SVA Default reset
 default disable iff (sig_reset);

 // Data must not be X or Z during Data Phase (when valid is raised)
 assertValidAndData: assert property (
   ($rose(valid) |=> !$isunknown(division_result)))
   else
  `uvm_error("ERR_DATA_XZ", "Data went to X or Z during Data Phase")

endinterface : FP_ALU_DIVIDER_if
```

APPENDIX D.1

```
/*-----------------------------------------------------------------------*
 Name : FP_ALU_DIVIDER_test_lib.sv
 Description   : This file implements two kinds of test in the testbench.
    A test file verifies one or more cases in the test plan.
-----------------------------------------------------------------------*/

`include "FP_ALU_DIVIDER_top_env.sv"

class FP_ALU_DIVIDER_base_test extends uvm_test;

  `uvm_component_utils(FP_ALU_DIVIDER_base_test)

  FP_ALU_DIVIDER_top_env top_env0;
  uvm_table_printer printer;

  function new(string name = "FP_ALU_DIVIDER_base_test", uvm_component parent);
    super.new(name,parent);
    printer = new();
  endfunction : new

  // UVM build() phase ------------------------------------------------------
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Enable transaction recording for everything
    set_config_int("*", "recording_detail", UVM_FULL);
    // Create the testbench
    top_env0 = FP_ALU_DIVIDER_top_env::type_id::create("top_env0", this);
  endfunction

  // UVM start_of_simulation() phase
  virtual function void start of simulation phase(uvm phase phase);
    super.start_of_simulation_phase(phase);
    printer.knobs.depth = 5;
    this.print(printer);
  endfunction

endclass : FP_ALU_DIVIDER_base_test

//------------------------------------------------------------------------------
//
// TEST: itugp_seq_adder_first_test - sets the first sequences
//
//------------------------------------------------------------------------------
class FP_ALU_DIVIDER_first_test extends FP_ALU_DIVIDER_base_test;
  `uvm_component_utils(FP_ALU_DIVIDER_first_test)

  function new(string name = "FP_ALU_DIVIDER_first_test", uvm_component parent);
    super.new(name,parent);
  endfunction

  // UVM build() phase
  virtual function void build_phase(uvm_phase phase);
    int unsigned itr;

    super.build_phase(phase);
  endfunction
  extern virtual task run_phase(uvm_phase phase);
endclass

// ------------------------------------------------------------------------------
```

```
task FP_ALU_DIVIDER_first_test::run_phase(uvm_phase phase);

    FP_ALU_DIVIDER_illegal_transmit_seq seq;

   `uvm_info(get_type_name(),"In build() function of FP_ALU_DIVIDER_first_test class", UVM_MEDIUM)
    seq = FP_ALU_DIVIDER_illegal_transmit_seq::type_id::create("seq");
    // -----------------
    assert(seq.randomize());
    // -----------------
    phase.raise_objection(this, "Starting the sequence.");
    // -----------------
    seq.start ( top_env0.FP_ALU_DIVIDER.agent_inst.sequencer );
    // -----------------
    phase.drop_objection(this, "Finishing the sequence.");

endtask : run_phase
```

APPENDIX D.2

```
/*-----------------------------------------------------------------
Name : FP_ALU_DIVIDER_top_env.sv
Description   : Models the top layer of the environment component
-----------------------------------------------------------------*/

// Derive top environment from umv_env class
class FP_ALU_DIVIDER_top_env extends uvm_env;

  // Provide implementations of virtual methods such as get_type_name and create
  `uvm_component_utils(FP_ALU_DIVIDER_top_env)

  // FP_ALU_DIVIDER environment
  FP_ALU_DIVIDER_vc_env FP_ALU_DIVIDER;

  // Constructor - required syntax for UVM automation and utilities
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

// Additional class methods
// UVM build() phase
function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  // set vif property for child elements
  uvm_config_db#(virtual FP_ALU_DIVIDER_if)::set(this,"*","vif",FP_ALU_DIVIDER_tb_top.dut_if);
  FP_ALU_DIVIDER = FP_ALU_DIVIDER_vc_env::type_id::create("FP_ALU_DIVIDER", this);
endfunction

// UVM start_of_simulation() phase
function void start_of_simulation_phase(uvm_phase phase);
  super.start of simulation phase(phase);

  uvm_test_done.set_drain_time(this, 1000);
endfunction
endclass
```

APPENDIX D.3

```
/*------------------------------------------------------------------------------
 Name : FP_ALU_DIVIDER_vc_env.sv
 Description   : Models the verification component layer of the environment
------------------------------------------------------------------------------*/

// Derive UVC environment from umv_env class
class FP_ALU_DIVIDER_vc_env extends uvm_env;

  // Virtual Interface variable
  protected virtual interface FP_ALU_DIVIDER_if vif;

  // The following two bits are used to control whether checks and coverage are
  // done both in the bus monitor class and the interface.
  bit checks_enable = 1;
  bit coverage_enable = 1;

  // Components of the environment
   FP_ALU_DIVIDER_agent          agent_inst;
   FP_ALU_DIVIDER_fc_subscriber  fc_sub;
   FP_ALU_DIVIDER_scoreboard     scoreboard;

  /* Call macros to provide attributes for environment. uvm_component_utils_begin
     macro should be updated to get required utilities. */
  `uvm_component_utils_begin(FP_ALU_DIVIDER_vc_env)
    `uvm_field_int(checks_enable, UVM_DEFAULT)
    `uvm_field_int(coverage_enable, UVM_DEFAULT)
  `uvm_component_utils_end

  // Constructor - required syntax for UVM automation and utilities
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  // Additional class methods
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);
  extern protected task update_vif_enables();
  extern virtual task run_phase(uvm_phase phase);

endclass : FP_ALU_DIVIDER_vc_env

  // UVM build() phase
  function void FP_ALU_DIVIDER_vc_env::build_phase(uvm_phase phase);
    super.build();
    agent_inst = FP_ALU_DIVIDER_agent::type_id::create("agent_inst", this);
    fc_sub = FP_ALU_DIVIDER_fc_subscriber::type_id::create(.name("fc_sub"), .parent(this));
    scoreboard = FP_ALU_DIVIDER_scoreboard::type_id::create(.name("scoreboard"), .parent(this));

    if(!uvm_config_db#(virtual FP_ALU_DIVIDER_if)::get(this,"","vif",vif))
    `uvm_error("NOVIF","virtual if not configured");
  endfunction

 // UVM connect() phase
   function void FP_ALU_DIVIDER_vc_env::connect_phase(uvm_phase phase);
      super.connect();
      agent_inst.FP_ALU_DIVIDER_ap.connect(fc_sub.analysis_export);
      agent_inst.FP_ALU_DIVIDER_ap.connect(scoreboard.FP_ALU_DIVIDER_analysis_export);
   endfunction

  // ---------------------------------------------------------------------------------------
  // TASK -> update_vif_enables() -> Function to assign the checks and coverage bits
  // ---------------------------------------------------------------------------------------
  task FP_ALU_DIVIDER_vc_env::update_vif_enables();
```

```
  // Make assignments at time zero based upon config
  vif.has_checks <= checks_enable;
  vif.has_coverage <= coverage_enable;
  forever begin
    // Make assignments whenever enables change after time zero
    @(checks_enable || coverage_enable);
    vif.has_checks <= checks_enable;
    vif.has_coverage <= coverage_enable;
  end
endtask : update_vif_enables

// UVM run() phase
task FP_ALU_DIVIDER_vc_env::run_phase(uvm_phase phase);
  super.run_phase(phase);
  fork
    update_vif_enables();
  join_none
endtask
```

APPENDIX E

```
/*-----------------------------------------------------------------------------
 Name : FP_ALU_DIVIDER_agent.sv
 Description: Agent contains monitor,sequencer and driver components and implements
 connections between aformentioned modules
-----------------------------------------------------------------------------*/

// Derive an agent from uvm_agent class
class FP_ALU_DIVIDER_agent extends uvm_agent;

  FP_ALU_DIVIDER_monitor  monitor;
  FP_ALU_DIVIDER_driver   driver;
  /* Derive and define a sequencer from uvm_sequencer class that uses seq_item,
     called FP_ALU_DIVIDER_sequencer, then create an instance */
  typedef uvm_sequencer #(FP_ALU_DIVIDER_seq_item) FP_ALU_DIVIDER_sequencer;
  FP_ALU_DIVIDER_sequencer sequencer;

  // Creating analysis port
  uvm_analysis_port #(FP_ALU_DIVIDER_seq_item) FP_ALU_DIVIDER_ap;

  /* Call macros to provide attributes for agent. uvm_component_utils_begin macro
     should be updated to get required utilities. */
  `uvm_component_utils_begin(FP_ALU_DIVIDER_agent)
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT)
  `uvm_component_utils_end

  // Constructor - required syntax for UVM automation and utilities
  function new (string name, uvm_component parent);
    super.new(name, parent);
    FP_ALU_DIVIDER_ap = new("FP_ALU_DIVIDER_ap",this);
  endfunction : new

// UVM build() phase
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  monitor = FP_ALU_DIVIDER_monitor::type_id::create("monitor", this);
  if(is_active == UVM_ACTIVE) begin
    sequencer = FP_ALU_DIVIDER_sequencer::type_id::create("sequencer", this);
    driver    = FP_ALU_DIVIDER_driver::type_id::create("driver", this);
  end
endfunction

// UVM connect() phase
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  monitor.item_collected_port.connect(FP_ALU_DIVIDER_ap); //connect monitor to agent via analysis port
  if(is_active == UVM_ACTIVE) begin
    // Binds the driver to the sequencer using consumer-producer interface
    driver.seq_item_port.connect(sequencer.seq_item_export);
  end
endfunction

endclass : FP_ALU_DIVIDER_agent
```

APPENDIX F

```
/*------------------------------------------------------------------------*
 Name : FP_ALU_DIVIDER_monitor.sv
 Description   : This file implements the monitor. Monitors
   the activity of its interface bus.
------------------------------------------------------------------------*/

// Derive monitor from uvm_monitor
class FP_ALU_DIVIDER_monitor extends uvm_monitor;

  // Virtual Interface for monitoring DUT signals
  protected virtual interface FP_ALU_DIVIDER_if vif;

  int num_col; // To hold collected seq_item count

  // The following two bits are used to control whether checks and coverage are
  // done in the monitor
  bit checks_enable = 1;
  bit coverage_enable = 1;

  // This TLM port is used to connect the monitor to the scoreboard
  uvm_analysis_port #(FP_ALU_DIVIDER_seq_item) item_collected_port;

  // Currently monitored seq_item
  protected FP_ALU_DIVIDER_seq_item seq_item;

  // Covergroup for seq_item
  covergroup seq_item_cg;
    option.per_instance = 1;
  endgroup : seq_item_cg

// Provide UVM automation and utility methods
`uvm_component_utils_begin(FP_ALU_DIVIDER_monitor)
  `uvm_field_int(checks_enable, UVM_DEFAULT)
  `uvm_field_int(coverage_enable, UVM_DEFAULT)
`uvm_component_utils_end

// Constructor - required syntax for UVM automation and utilities
function new (string name, uvm_component parent);
  super.new(name, parent);
  // Create the covergroup only if coverage is enabled
  void'(get_config_int("coverage_enable", coverage_enable));
  if (coverage_enable) begin
    seq_item_cg = new();
    seq_item_cg.set_inst_name("seq_item_cg");
  end
  // Create the TLM port
  item_collected_port = new("item_collected_port", this);
endfunction : new

// Additional class methods
extern virtual task run();
extern virtual protected task collect_seq_item();
extern virtual protected function void perform_checks();
extern virtual protected function void perform_coverage();
extern virtual function void report_phase(uvm_phase phase);

// UVM build() phase
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
```

```
    if(!uvm_config_db#(virtual FP_ALU_DIVIDER_if)::get(this,"","vif",vif))
    `uvm_error(get_type_name(),"virtual if not configured");

  endfunction
endclass : FP_ALU_DIVIDER_monitor

  // UVM run() phase
  task FP_ALU_DIVIDER_monitor::run();
    fork
      collect_seq_item();
    join_none
  endtask : run

  task FP_ALU_DIVIDER_monitor::collect_seq_item();
    seq_item = FP_ALU_DIVIDER_seq_item::type_id::create("seq_item", this);
    forever begin
    @(posedge vif.sig_clock iff vif.valid === 1);
      // Begin transaction recording
      void'(begin_tr(seq_item, "FP_ALU_DIVIDER Monitor"));

    seq_item.operand_a = vif.operand_a;
    seq_item.operand_b = vif.operand_b;
    seq_item.division_result = vif.division_result;
    seq_item.status = vif.status;

      @(posedge vif.sig_clock iff vif.valid === 0);
     // End transaction recording
     end_tr(seq_item);
     `uvm_info(get_type_name(),
       $sformatf("seq_item collected :\n%s",
       seq_item.sprint()), UVM_HIGH)
     if (checks_enable)
        perform_checks();
     if (coverage_enable)
        perform_coverage();
     // Send seq_item to scoreboard via TLM write()
     item_collected_port.write(seq_item);
     num_col++;
   end
 endtask : collect_seq_item

 // perform_seq_adder_seq_item_checks
 function void FP_ALU_DIVIDER_monitor::perform_checks();
   // Add checks here
 endfunction : perform_checks

 // Triggers coverage events
 function void FP_ALU_DIVIDER_monitor::perform_coverage();
   seq_item_cg.sample();
 endfunction : perform_coverage

 // UVM report() phase
 function void FP_ALU_DIVIDER_monitor::report_phase(uvm_phase phase);
   `uvm_info(get_type_name(),
     $sformatf("\nReport: FP_ALU_DIVIDER monitor collected %0d seq_items", num_col),
     UVM_LOW)
 endfunction
```

APPENDIX G.1

```
/*-------------------------------------------------------------
 Name: FP_ALU_DIVIDER_seq_item.sv
Description   :  Specifies the sequence item. Declare constraints
                 depending the verification strategy
-------------------------------------------------------------*/

class FP_ALU_DIVIDER_seq_item extends uvm_sequence_item;

rand bit [15:0] operand_a;
rand bit [15:0] operand_b;
bit      [15:0] division_result;
bit      [2:0] status;
bit      [1:0] output_timing; // 00 --> LEGAL_OUTPUT_TIMING, 01 --> MISSING_OUTPUT, 10 -->
// EARLY_OUTPUT_TIMING, 11 --> LATE_OUTPUT_TIMING

`uvm_object_utils_begin(FP_ALU_DIVIDER_seq_item)
   `uvm_field_int(operand_a, UVM_DEFAULT)
   `uvm_field_int(operand_b, UVM_DEFAULT)
   `uvm_field_int(division_result, UVM_DEFAULT)
   `uvm_field_int(status, UVM_DEFAULT)
`uvm_object_utils_end

  // Constraints
  /*
  constraint const_a {a >= 0; a < 10;}
  constraint const_b {b >= 3; b < 13;}
  */

  // Constructor - required syntax for UVM automation and utilities
  function new (string name = "FP_ALU_DIVIDER_seq_item");
    super.new(name);
  endfunction : new

endclass : FP_ALU_DIVIDER_seq_item
```

APPENDIX G.2

```
/*-------------------------------------------------------------
Name: FP_ALU_DIVIDER_seq_lib.sv
Description   : This file implements several sequence kinds
-------------------------------------------------------------*/

/********************* legal transmit sequence ****************************/

// Extend sequence from uvm_sequence class - using seq_item
class FP_ALU_DIVIDER_legal_transmit_seq extends uvm_sequence #(FP_ALU_DIVIDER_seq_item);

  `uvm_object_utils(FP_ALU_DIVIDER_legal_transmit_seq )

  // Constructor
  function new(string name="FP_ALU_DIVIDER_legal_transmit_seq ");
    super.new(name);
  endfunction

  // Sequence body definition
  virtual task body();
    begin
      `uvm_info(get_type_name(), "Executing...", UVM_MEDIUM)
       `uvm_do_with(req, { a <= 5; b >= 4;} )
    end
  endtask

endclass

/********************* illegal transmit sequence ****************************/

class FP_ALU_DIVIDER_illegal_transmit_seq  extends uvm_sequence #(FP_ALU_DIVIDER_seq_item);
  `uvm_object_utils(FP_ALU_DIVIDER_illegal_transmit_seq)

  // Sequence that will be called in this sequence
  FP_ALU_DIVIDER_legal_transmit_seq FP_ALU_DIVIDER_seq;

  // Parameter for this sequence

  // Constructor
  function new(string name="FP_ALU_DIVIDER_illegal_transmit_seq");
    super.new(name);
  endfunction

  // Sequence body definition
  virtual task body();
    uvm_component parent = get_sequencer();
    begin
      for(int i = 0; i < 50; i++) begin
        `uvm_do(FP_ALU_DIVIDER_seq)
      end
    end
  endtask

endclass
```

APPENDIX G.3

```
/*------------------------------------------------------------------
Name: FP_ALU_DIVIDER_driver.sv
Description   : This files implements the driver functionality.
------------------------------------------------------------------*/

class FP_ALU_DIVIDER_driver extends uvm_driver #(FP_ALU_DIVIDER_seq_item);

 /*****************************************************************************
  IVB-NOTE : REQUIRED : DRIVER functionality : DRIVER
  ---------------------------------------------------------------------------
  Modify the following methods to match your protocol:
    o drive_seq_item() - Handshake and seq_item driving process
    o reset_signals() - signal reset values
  Note that if you change/add signals to the physical interface, you must
  also change these methods.
  *****************************************************************************/

  // The virtual interface used to drive and view HDL signals.
  protected virtual interface FP_ALU_DIVIDER_if vif;

  FP_ALU_DIVIDER_seq_item seq_item;

  // Count seq_items sent
  int num_sent;

  // Provide implmentations of virtual methods such as get_type_name and create
  `uvm_component_utils(FP_ALU_DIVIDER_driver)

  // Constructor - required syntax for UVM automation and utilities
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  // Additional class methods
  extern virtual task run_phase(uvm_phase phase);
  extern virtual protected task get_and_drive();
  extern virtual protected task reset_signals();
  extern virtual protected task drive_seq_item();
  extern virtual function void report_phase(uvm_phase phase);

  // UVM build() phase
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    if(!uvm_config_db#(virtual FP_ALU_DIVIDER_if)::get(this,"","vif",vif))
    `uvm_error(get_type_name(),"virtual if not configured");

  endfunction
endclass : FP_ALU_DIVIDER_driver

  // UVM run() phase
  task FP_ALU_DIVIDER_driver::run_phase(uvm_phase phase);
        super.run_phase(phase);

    fork
      get_and_drive();
      reset_signals();
    join

  endtask
```

```
// ---------------------------------------------------------------------------------------
// TASK -> get_and_drive() ->Gets seq_items from the sequencer and passes them to the driver.
// ---------------------------------------------------------------------------------------
task FP_ALU_DIVIDER_driver::get_and_drive();


  `uvm_info(get_type_name(), "Reset not rised", UVM_MEDIUM)
  @(negedge vif.sig_reset);
  `uvm_info(get_type_name(), "Reset riseed", UVM_MEDIUM)
  forever begin
    @(posedge vif.sig_clock iff vif.enable === 0);
    `uvm_info(get_type_name(), "FP_ALU_DIVIDER_driver inside get_and_drive task 1", UVM_MEDIUM)
    // Get new item from the sequencer
    seq_item_port.get_next_item(seq_item);
    `uvm_info(get_type_name(), "FP_ALU_DIVIDER_driver inside get_and_drive task 2", UVM_MEDIUM)
    // Drive the item
    drive_seq_item();
      `uvm_info(get_type_name(), "FP_ALU_DIVIDER_driver inside get_and_drive task 3", UVM_MEDIUM)
    // Communicate item done to the sequencer
    seq_item_port.item_done();
    `uvm_info(get_type_name(), "FP_ALU_DIVIDER_driver inside get_and_drive task 4", UVM_MEDIUM)
  end
endtask : get_and_drive
```

```
// ---------------------------------------------------------------------------------------
// TASK -> reset_signals() ->Reset all signals.
// ---------------------------------------------------------------------------------------
task FP_ALU_DIVIDER_driver::reset_signals();
  forever begin
    @(posedge vif.sig_reset);
    `uvm_info(get_type_name(), "Reset observed", UVM_MEDIUM)
  vif.operand_a      <= 'hz;
  vif.operand_b      <= 'hz;
  vif.enable <= 'h0;
  end
endtask : reset_signals

// ---------------------------------------------------------------------------------------
// TASK -> drive_seq_item() ->Gets a seq_item and drive it into the DUT
// ---------------------------------------------------------------------------------------
task FP_ALU_DIVIDER_driver::drive_seq_item();
     `uvm_info(get_type_name(), "Inside drive_seq_item1", UVM_MEDIUM)
  vif.enable <= 1'b1;
  vif.operand_a <= seq_item.operand_a;
  vif.operand_b <= seq_item.operand_b;
  @(posedge vif.sig_clock iff vif.enable === 1);
      `uvm_info(get_type_name(), "Inside drive_seq_item2", UVM_MEDIUM)
   #160 vif.enable <= 1'b0;
      vif.operand_a<= 'hz;
      vif.operand_b<= 'hz;
      @(posedge vif.sig_clock);

   num_sent++;
  `uvm_info(get_type_name(), $sformatf("Item %0d Sent ...", num_sent),
    UVM_HIGH)
endtask : drive_seq_item

// UVM report() phase
function void FP_ALU_DIVIDER_driver::report_phase(uvm_phase phase);
  `uvm_info(get_type_name(),
    $sformatf("\nReport: FP_ALU_DIVIDER driver sent %0d seq_items",
    num_sent), UVM_LOW)
    stop_matlab();
endfunction
```

APPENDIX H.1

```
/*-----------------------------------------------------
Name : FP_ALU_DIVIDER_fc_subscriber.sv
Description   : functional coverage subscriber
-----------------------------------------------------*/

// Derive from uvm_subscriber class
class FP_ALU_DIVIDER_fc_subscriber extends uvm_subscriber #(FP_ALU_DIVIDER_seq_item);
// Include utility macros
`uvm_component_utils(FP_ALU_DIVIDER_fc_subscriber)

FP_ALU_DIVIDER_seq_item pkt;

int pkt_cnt;

covergroup cov1;
operand_a_cov: coverpoint pkt.operand_a {bins operand_a[8]  = {[0:15]};} //coverage register
operand_b_cov: coverpoint pkt.operand_b {bins operand_b[8] = {[0:15]};}  //coverage register
cross operand_a_cov, operand_b_cov;
endgroup : cov1

function new( string name , uvm_component parent);
super.new( name , parent );
cov1 = new();
endfunction

function void write(FP_ALU_DIVIDER_seq_item t);
real current_FP_ALU_DIVIDER_fc_subscriber;
pkt = t;
pkt_cnt++;
cov1.sample();
// cause sampling of covergroup
current_FP_ALU_DIVIDER_fc_subscriber = $get_coverage();

uvm_report_info("FP_ALU_DIVIDER_FC_SUBSCRIBER",$psprintf("%0d FP_ALU_DIVIDER_seq_items sampled,
FP_ALU_DIVIDER_fc_subscriber = %f%% ", pkt_cnt,current_FP_ALU_DIVIDER_fc_subscriber));

endfunction
endclass:FP_ALU_DIVIDER_fc_subscriber
```

APPENDIX H.2

```
/*-------------------------------------------------------------
Name : FP_ALU_DIVIDER_sb_subscriber.sv
Description   : Compares predicted results with actual DUT results
-------------------------------------------------------------*/

typedef class FP_ALU_DIVIDER_scoreboard;

class FP_ALU_DIVIDER_sb_subscriber extends uvm_subscriber #(FP_ALU_DIVIDER_seq_item);
`uvm_component_utils(FP_ALU_DIVIDER_sb_subscriber)

function new( string name , uvm_component parent);
super.new( name , parent );
endfunction:new

function void write(FP_ALU_DIVIDER_seq_item t);
    FP_ALU_DIVIDER_scoreboard FP_ALU_DIVIDER_sb;
    /*******************************/
    FP_ALU_DIVIDER_seq_item m_out_item;
    string msg, cmd, cmd_rsp1,cmd_rsp2;

    m_out_item = FP_ALU_DIVIDER_seq_item::type_id::create("m_out_item");

    //t is the input sequence item (transaction). Process t and then
    // write the new processed output to the m_output_ap.
    m_out_item.do_copy(t);

    $sformat(msg, "INPUT: A = %s, B = %s",t.operand_a.convert2string(), t.operand_b.convert2string());
    `uvm_info(get_name(),msg, UVM_HIGH);

    $sformat(cmd, "result = float16div_r(%0d,%0d);", t.operand_a, t.operand_b);
    `uvm_info(get_name(), cmd, UVM_HIGH);
```

```
// Call our MATLAB function with our transaction inputs
void'(send_matlab_cmd(cmd));

// Readback the MATLAB buffer with our output
cmd_rsp1 = get_matlab_buffer();

`uvm_info(get_name(), $sformatf("MATLAB Buffer is %s", cmd_rsp1), UVM_HIGH);

if (!$sscanf(cmd_rsp1, ">> %d", m_out_item.division_result)) begin
    `uvm_warning(get_name(), "Error parsing MATLAB response");
end

$sformat(cmd, "flg = float16div_s(%0d,%0d);", t.operand_a, t.operand_b);
`uvm_info(get_name(), cmd, UVM_HIGH);

// Call our MATLAB function with our transaction inputs
void'(send_matlab_cmd(cmd));

// Readback the MATLAB buffer with our output
cmd_rsp2 = get_matlab_buffer();

`uvm_info(get_name(), $sformatf("MATLAB Buffer is %s", cmd_rsp2), UVM_HIGH);

if (!$sscanf(cmd_rsp2, ">> %d", m_out_item.status)) begin
    `uvm_warning(get_name(), "Error parsing MATLAB response");
end

m_output_ap.write(m_out_item);
/*******************************/
```

```
      $cast( FP_ALU_DIVIDER_sb, m_parent );
      FP_ALU_DIVIDER_sb.check_FP_ALU_DIVIDER_checker(t);
   endfunction: write

endclass:FP_ALU_DIVIDER_sb_subscriber

//------------------------------------------------------------------------------
//
// CLASS: itugp_seq_adder_scoreboard
//
//------------------------------------------------------------------------------
class FP_ALU_DIVIDER_scoreboard extends uvm_scoreboard;
   `uvm_component_utils(FP_ALU_DIVIDER_scoreboard)

   uvm_analysis_export#(FP_ALU_DIVIDER_seq_item) FP_ALU_DIVIDER_analysis_export;
   local FP_ALU_DIVIDER_sb_subscriber FP_ALU_DIVIDER_sb_sub;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      if (!start_matlab("matlab -nosplash")) begin
        `uvm_fatal(get_name(), "Unable to start MATLAB");
      end

      void'(send_matlab_cmd("addpath ./MATLAB;"));
      m_output_ap = new("m_output_ap", this);
```

```
      FP_ALU_DIVIDER_analysis_export = new( .name("FP_ALU_DIVIDER_analysis_export"), .parent(this));
      FP_ALU_DIVIDER_sb_sub = FP_ALU_DIVIDER_sb_subscriber::type_id::create(.name("FP_ALU_DIVIDER_sb_sub"), .parent(this));
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      FP_ALU_DIVIDER_analysis_export.connect(FP_ALU_DIVIDER_sb_sub.analysis_export);
   endfunction: connect_phase
//------------------------------------------------------------------------------
//TASK => check_itugp_seq_adder_checker()
//------------------------------------------------------------------------------

   virtual function void check_FP_ALU_DIVIDER_checker(FP_ALU_DIVIDER_seq_item FP_ALU_DIVIDER_tx);
      uvm_table_printer p = new;

    shortint division_result_expected;    //expected value of output
    shortint division_result_actual;       //actual value of output
    int status_expected;    //expected value of output status
    int status_actual;       //actual value of output status

    division_result_expected = m_out_item.division_result;
    division_result_actual = FP_ALU_DIVIDER_tx.division_result;

    status_expected = m_out_item.status;
    status_actual = FP_ALU_DIVIDER_tx.status;


    //Comparator
    if ((division_result_expected == division_result_actual) && (status_expected == status_actual)) begin
        `uvm_info("division_result_scoreboard",
                        { "s=a+b.\n", FP_ALU_DIVIDER_tx.sprint(p) }, UVM_LOW);
        end
    else begin
            `uvm_error("division_result_scoreboard",
                        { "s/=a+b!\n", FP_ALU_DIVIDER_tx.sprint(p) });
        end

   endfunction: check_FP_ALU_DIVIDER_checker
endclass: FP_ALU_DIVIDER_scoreboard
```

APPENDIX I.1

```
/*
* Name : matlab_dpi.c
*
* Description : DPI Functions to enable SV communication to MATLAB
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "engine.h"

#define BUFSIZE 256

Engine *ep;
mxArray *T = NULL, *result = NULL;
char buffer[BUFSIZE+1];

int start_matlab(char *cmd)
{
    if (!ep && !(ep = engOpen(cmd))) {
       fprintf(stderr, "\nCan't start MATLAB engine\n");
       return 0;
       }

       engOutputBuffer(ep, buffer, BUFSIZE);
       return 1;
}

int send_matlab_cmd(char *cmd)
{
     return engEvalString(ep, cmd);
}

char *get_matlab_buffer()
{
     return buffer;
}

void stop_matlab()
{
     engClose(ep);
}
```

APPENDIX I.2

```
// Name : matlab_dpi_pkg.svh
// Package to define MATLAB DPI functions

package matlab_dpi_pkg;

import "DPI-C" function int start_matlab(string cmd);
import "DPI-C" function int send_matlab_cmd(string cmd);
import "DPI-C" function string get_matlab_buffer();
import "DPI-C" function void stop_matlab();

endpackage
```

RESUME

Name - Surname: Yasin Fırat Kula

Place and Date of Birth: Tekirdağ, 1993

High school: Tekirdağ Science High School 2007-2011

BSC: Istanbul Technical University, Electronics and Communications Engineering 2011-2016