Tampere University

Debajit Saha

EVALUATION OF OPEN-SOURCE EDA TOOL “EDA PLAYGROUND”

Faculty of Information Technology
and Communication Sciences
Master of Science Thesis
October 2023

ABSTRACT

Debajit Saha: Evaluation of Open-Source EDA Tool “EDA Playground”.
Master of Science Thesis
Tampere University
Master’s Degree Programme in Electrical Engineering
Major: Electronics
October 2023
Examiners: Prof. Dr. Jari Nurmi, Dr. Hesam Zolfaghari

With the advancement of Information Technology, the design, verification, and manufacturing of Integrated circuits have been challenging and time consuming. Unlike the software domain, Electronic Design Automation (EDA) tools are mostly commercially available, and access is limited to the students. An open-source EDA tool might help the students to initialize the learning process. This thesis showcases an open-source EDA platform, EDA Playground, where users can practice their hardware description language (HDL) codes, create a testbench to simulate their designs and synthesize their code.

The thesis shows how EDA Playground provides its users with the ability to write code in various HDLs, enabling them to evaluate their designs using a range of both commercial and freely available simulators. Additionally, it also shows how the platform helps in identifying and resolving design failures through the utilization of waveform viewing tool, EPwave, developed my EDA Playground and logs. It is also highlighted how users have the ability to employ commercial synthesizers in order to combine their codes, thereby facilitating the assessment of device utilization and circuit diagram.

Another notable objective of the thesis is to highlight the application of EDA Playground to the incorporate of UVM 1.2. A step-by-step UVM testbench of a simple SystemVerilog adder was developed and simulated as a part of the thesis. Prospective users have the opportunity to gain knowledge about this methodology by accessing educational resources, which encompass various tools and examples provided for their advantage.

The thesis provides an extensive array of use cases that showcase the varied functionalities provided by EDA Playground. This thesis extensively employs and evaluates the diverse resources offered on EDA Playground to determine their usefulness.

Keywords: Hardware Description Language (HDL), Electronic Design Automation (EDA), Universal Verification Methodology (UVM), SystemVerilog, VHDL, EDA Playground.

PREFACE

This thesis was done as a part of the master's degree in electrical engineering for Tampere University.

First, I would like to thank my supervisor Prof. Dr. Jari Nurmi for constant suggestions and guidance during the thesis. I would also like to express my gratitude to Dr. Hesam Zolfaghari for his important feedbacks. I extend special thanks to Arto Oinonen of Tampere University whose master's thesis was of great help towards planning the use cases using SystemVerilog and UVM.

Finally, I would like to express my deepest gratitude to my parents and my wife for constantly pushing and motivating me to achieve my goal.

Tampere, 2nd October 2023

Debajit Saha

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CLB	Combinational Logic Blocks
CPU	Central Processing Unit
DRC	Design Rule Check
DSP	Digital Signal Processor/Processing
DUT	Device Under Test
EDA	Electronic Design Automation
ESL	Electronic System Level
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property
IOB	Input/Output Block
IT	Information Technology
LUT	Look-Up Table
LVS	Layout Vs. Schematic
ML	Machine Learning
PCB	Printed Circuit Board
PLD	Programmable Logic Device
RAM	Random Access Memory
RTL	Register Transfer Level
SoC	System on Chip
TLM	Transaction Level Modeling
UDP	User Defined Primitive
UVM	Universal Verification Methodology
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VIP	Verification IP
VLSI	Very Large-Scale Integration

1. INTRODUCTION

The field of information technology (IT) is centered around advanced technologies related to digital information and communication. The information technology sector has experienced significant growth in recent years, making it the industry with the highest rate of expansion. As globalization continues to shrink the world, the demand for enhanced communication devices and methods becomes increasingly crucial in the business realm. One of the crucial attributes of IT is its escalating requirement for exceedingly robust processing capabilities and extensive bandwidth to effectively manage time-sensitive applications, such as video streaming. These circumstances forced the development of accelerated and progressively more effective products to facilitate enhanced telecommunications capabilities. The increasing demand in the contemporary world is driving numerous countries to make substantial investments in the design of Very Large Scale Integration (VLSI) systems[1].

The process of manufacturing VLSI systems on chips is a complex undertaking that involves several key activities. These activities include the *design* of VLSI systems using electronic design automation (EDA) tools, functional and formal *verification*, the utilization of computer aided design (CAD) in the *development* and *manufacturing* of VLSI chips, the involvement of foundries in the entire process from the silicon wafer to the final *packaging* and *testing* of integrated circuits (ICs). Along with utilizing EDA tools, the digital system design steps require significant capital investment. On the other hand, the design steps require a significant amount of knowledge and experience.

Due to the proliferation and rapid expansion of applications, the demand for VLSI system application designers has surpassed that of professionals specializing in chip technology, making it a challenging and intriguing field[2].

The escalation in design complexity imposes a significant load on digital system engineers, resulting in a design activity that surpasses the threshold of optimal productivity. *Figure 1* illustrates a depiction of design and verification gaps commonly referred to as the disparities between effort needed and productivity of the system.

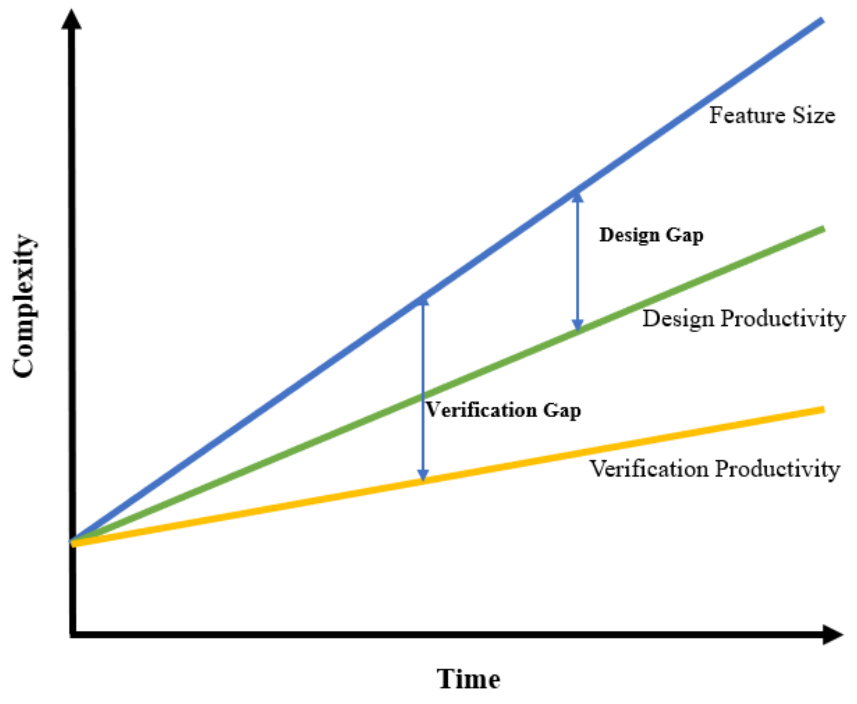


Figure 1. *Design Gap Vs Verification Gap[3]*

The study conducted by Wilson Research Group in 2014 revealed that while the design gap has been reduced due to an increase in design reuse, the same level of maturity has not been achieved in verification reuse. Consequently, it is anticipated that the growth of the design gap will decelerate, whereas the verification gap will persist in its expansion. It has been observed that the proportion of time dedicated to verification in application specific integrated circuit (ASIC) projects has been increasing significantly [3].

Significantly, there has been a consistent increase in the number of projects in which the duration of verification activities accounts for more than 70% of the total project time, as observed in each iteration of the study [1]. The difficulties encountered during verification are a result of the complicated design features, verification techniques, and methodology employed. Common complex design characteristics include embedded memories and multiple clocks.

Another type of verification challenge occurs while implementing Nanometer ICs. The implementation of ICs at the nanometer scale presents notable concerns related to signal integrity (SI), which must be thoroughly examined to prevent potential malfunctions in chip functionality. The aforementioned factors include reduced feature size, minimized in wire spacing, decreased power supply voltages, and a shrinkage of threshold voltages. As each successive process technology is developed, there is a progressive increase in the number of wire levels that are densely packed in close proximity. Consequently, there is a significant increase in the proportion of total wire capacitance that is attributed to lateral coupling. Therefore, this phenomenon is linked to a substantial rise in on-chip crosstalk noise. Another issue encountered in nanometer designs is an increase of clock frequencies accompanied by accelerated on-chip slew rates. Higher slew rates result in an amplified generation of switching noise and a corresponding elevation in instantaneous power consumption. In nanometer process technology, designs that satisfy physical verification criteria, such as design rule checking (DRC) and layout vs. schematic (LVS), may exhibit unexpected functional behavior due to the presence of above mentioned additional electrical side effects. This phenomenon becomes particularly pronounced at 90 nanometers and below [4]. Prominent EDA vendors are offering an extensive array of verification tools to tackle these issues. Cadence's "Physical Verification System", Siemens's "Calibre nmDRC", and Synopsys' "IC Validator" are examples of such tools [5].

1.1 Motivation and scope behind the thesis

As the design gap and verification gap increases with time, experts in the field of manufacturing VLSI chips are leading to saturation. This might impact in productivity and prolonged time to market.

This thesis provides an overview of an EDA platform, "EDA Playground" which will allow students without access to EDA tools through universities to make themselves acquainted and competent through practicing hobby projects. It was originally developed by Victor Lyuboslavsky as a part of Victor EDA, which was later acquired by Doulos in 2019. Users can edit, simulate, observe waveforms, synthesize, and share their Hardware Description Language (HDL) code using the free web tool EDA Playground. Its objective is to speed up design and testbench development learning through more convenient code exchange and more straightforward access to simulators and libraries. EDA Playground was created especially for quick prototypes and examples.

1.2 Thesis Outline

The thesis is divided into 6 parts. **Chapter 2** explains different types of Digital systems, the basic System on Chip (SoC) design flow and implementation styles of digital systems with both ASICs or Field Programmable Gate Arrays (FPGA)s. Then the Hardware Description Languages, which are used for design and verification are described in brief. Lastly, Electronic Design Automation (EDA) tools and flow are highlighted. **Chapter 3** introduces the SystemVerilog language and its constructs for both the design and verification. The purpose of this chapter is to explain the basic concepts which can be tried out in EDA playground. **Chapter 4** provides an approach for understanding Universal Verification Methodology (UVM). UVM is necessary for implementing the verification tasks in EDA Playground. To broaden the perspective and highlight the opportunities offered by integrating UVM in the verification process, a few examples of advanced concepts are also provided.

The EDA playground platform and its various applications are demonstrated in **Chapter 5**. There are two different kinds of design and simulation shown in this chapter. The first example is a design and simulation in VHDL. The second one is an advanced verification method known as UVM simulation and verification of a SystemVerilog design.

Lastly, **chapter 6** concludes the thesis with discussion and suggesting future work.

2. LITERATURE REVIEW

This section presents the literature review from the related studies which helped to achieve the objective of this thesis. Firstly, different types of Digital systems are discussed where design and verification are applied using the EDA tools in modern digital world. The basic SoC design flow is explained in details and implementation styles pertaining to both ASICs or Field Programmable Gate Arrays (FPGA)s are discussed. Then the Hardware Description Languages, which are used for design and verification are described in brief. Lastly, EDA tools and flow are highlighted.

2.1 Digital Systems

A Digital System refers to a collection of devices that have been specifically designed to process, store and transfer digitally represented data or physical quantities that are represented in a digital format. In this context, digital implies that the quantities have only discrete values, as opposed to continuous ones. In the realm of digital circuitry, signals are commonly expressed through discrete states or logic levels. Digital signals are characterized by their non-continuous nature, as they undergo discrete changes in individual steps. Digital systems play a significant role in various domains such as computation and data processing, control systems, communications, and measurement. There is a wide range of digital systems which are frequently used in our day-to-day life[6].

2.1.1 Embedded Systems

An embedded system refers to a specially designed computer system that is typically integrated within a larger system. A computational engine is formed by integrating software as well as hardware elements, which collectively enable the execution of a specific task. Embedded systems can only be used for certain tasks, while regular computers such as personal computers (PC) can be used for many different purposes. Embedded systems often have to work in reactive and time-constrained situations, examples can be airbags or Anti-lock Braking System (ABS) of a car. An embedded system can be roughly divided into two parts: the hardware and the software. Hardware provides the speed and reliability needed for the functionality (and other system properties, like security) and ensures lower power consumption. The software provides features and flexibility.

A conventional embedded system receives input from sensors to perceive the surrounding environment, and subsequently uses actuators to manipulate and control the said environment. Embedded systems are required to meet performance standards that align with the surrounding environment, the reason why Embedded systems are commonly called as reactive systems. A reactive system needs the utilization of both hardware and software elements in tandem. The issue is further complicated because these external events can exhibit either periodicity and predictability, or aperiodicity and unpredictability. In the context of scheduling events in an embedded system, it is imperative to consider both periodic and aperiodic events, while ensuring that the system's performance is guaranteed even under the worst-case execution rates [7].

The important characteristics of embedded system are:

- Embedded systems receive a set of analog signals through sensors from the environment such as pressure, temperature, vibration, etc. and then using the specific algorithms, to control the actuators or display in some format.
- Embedded systems continuously process information which involves the meaningful manipulation of data obtained from sensors, which may include tasks such as data compression/decompression and side impact detection.
- Embedded systems are developed for specific tasks, for example, airbag deployment, digital still cameras, and cell phones. Embedded systems can also be specifically engineered to handle the execution of control rules, Finite State Machines (FSMs), and signal processing algorithms. In addition to their primary functions, embedded systems are required to possess the capability to identify and respond suitably to malfunctions occurring within their internal computational infrastructure, as well as in the external systems with which they interact.

2.1.2 Digital Signal Processor

Digital Signal Processors (DSPs) are specialized electronic devices that perform mathematical operations on digitized real-world signals, such as voice, audio, video, temperature, pressure, or position. Analog devices in practical applications can detect various types of signals, like sound, light, temperature, and pressure. These signals carry analog information that needs analysis and conversion. Analog-to-Digital converters are utilized to transform real-world analog signals into a digital representation consisting of binary digits, namely 1's and 0's. At this point, the DSP assumes control by acquiring the digitized data and conducting subsequent processing. Subsequently, the digitized information is utilized in practical applications. In case an analog signal is necessary, a digital to analog signal is used to convert the digitized data to an analog signal. All these phenomena transpire at exceedingly rapid rate.

Due to its programmability, a DSP possesses the capability to be used across a diverse range of applications. Individuals have the option to develop their own software or utilize software available in the market to design a DSP solution for a specific application [8].

2.1.3 System on Chip (SoC)

A system on a chip (SoC) refers to an integrated circuit that integrates multiple components of a computer system onto a single chip. A SoC invariably comprises a central processing unit (CPU), although it may additionally incorporate components such as system memory, peripheral controllers (e.g., for Universal Serial Bus (USB) and storage), and more sophisticated peripherals like graphical processing units (GPUs), specialized neural network circuitry, radio modems (for Bluetooth or Wi-Fi), among others [9].

The SoC architecture differs from the conventional PC architecture, which consists of a central processing unit (CPU) chip along with distinct controller chips, a graphical processing unit (GPU), and random-access memory (RAM) which can be substituted, enhanced, or interchanged as required. The utilization of SoCs results in the reduction of computer size, enhancement of processing speed, cost reduction, and decreased power consumption [9].

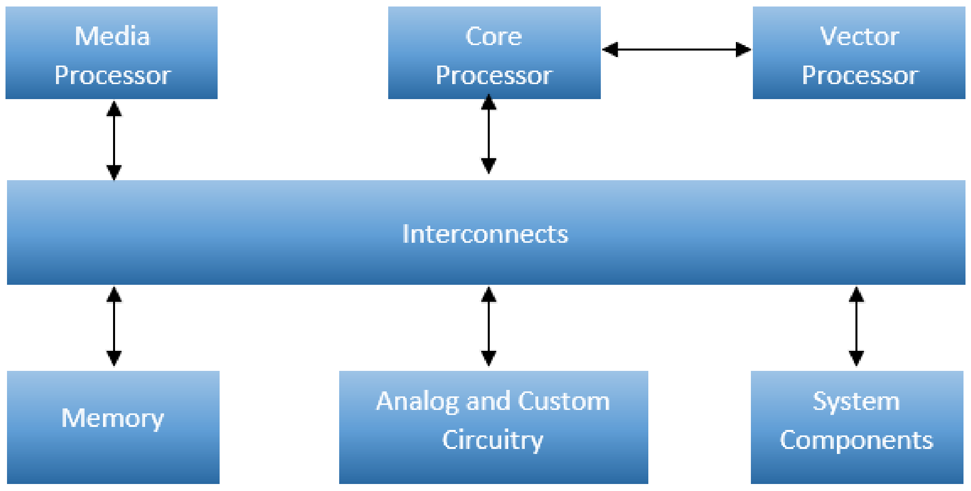


Figure 2. *A Basic Model of SoC [10].*

Figure 2 depicts some of the fundamental components of a SoC system. These consist of a few heterogeneous processors linked to one or more memory components with interconnects, maybe with an array of programmable logic. The SoC frequently includes analog circuitry for handling sensor data and analog-to-digital conversion, as well as to facilitate wireless data transfer [10].

The popular SoC designs are implemented either in ASICs or in FPGAs depending on the requirements, volume, cost, and time to market.

2.1.3.1 Field Programmable Gate Array (FPGA)

FPGAs are programmable logic devices where the programming can be done by the end user. A configuration file, also known as bit file, is uploaded to an FPGA to program it [11] . This configuration file contains data to execute a specific function FPGAs mostly depends on Static Random-Access Memory (SRAM) and integrate memory and Look-Up Tables (LUTs) in order to implement the logic blocks.

FPGAs exhibit distinct characteristics that set them apart from other Programmable Logic Devices (PLDs), typically offering the most complex logic capacity among their counterparts. An FPGA is composed of a collection of Complex Logic Blocks (CLBs), which are encapsulated by programmable Input/Output Blocks (IOBs) and interconnected through a programmable interconnection network. The IOBs play a crucial role in managing the connection between the pins of the input-output package and the internal signal lines. On the other hand, the programmable interconnect resources serve the purpose of establishing the necessary pathways to link the inputs and outputs of CLBs and IOBs to their respective networks. The implementation of combinational logic in logic cells can be implemented through physical means such as a small memory LUT or a combination of multiplexers and gates. An LUT is a memory array where the memory address lines serve as inputs to the logic block, while the output of the memory functions as the output of the lookup table.

An FPGA may have tens of thousands of logic blocks that can be set up in different ways, and even more flip-flops. The activation of the user's logic function is achieved by toggling the switches in the grid that correspond to the logic function of each logic cell. Then, to make the desired circuit, these simple blocks are put together to make more complicated functionalities.

The benefits of FPGAs are that they are flexible, reprogrammable, and are cost-efficient. For example, the reprogrammable nature of the product allows designers and manufacturers to modify its design or distribute updates after distribution. Many designers often use this feature to make prototypes based on FPGAs so that the design can be fully debugged, tested, and updated before they are sent to mass production. Even though the one-time costs are low and they have reduced manufacturing time, some of an FPGA's resources, for example speed or power consumption are compromised [12].

Also, comparing the cost of production to the number of units made shows that using FPGAs is more expensive than using ASICs as the number of units made goes up [2]. Also, almost all FPGAs lack analog blocks. So, in case there is a need of using mixed signals in FPGAs, unique analog blocks must be added to FPGA systems. Most of the time, these functions need to be added by external integrated circuits (IC)s, which makes the product even larger and more expensive [11].

2.1.3.2 Application Specific Integrated Circuit (ASIC)

An application-specific integrated circuit (ASIC) refers to an IC that has been specifically designed and tailored to fulfill a particular task or application. ASIC designs are customized at the initial stages of the design process in order to cater to specific requirements. (ASICs) have the capability of utilizing pipelining and massively parallel processing, thereby yielding designs that are both faster and more cost-effective. ASIC designs are considered appropriate when there is an anticipation of high-volume production [1].

Due to their semi- or fully custom nature, ASICs face substantial development costs, often amounting to millions of dollars, throughout the design and implementation phases. Furthermore, it should be noted that once ASICs are manufactured, they lack the capability to be reprogrammed. This means that changes to the design cost additional money. Even though ASICs have relatively high one-time costs, these costs associated with ASICs are justified because,

- ASICs have higher density, which enables the integration of complex functionalities into a single chip, which in turn, allows the size, power, and cost to decrease.
- The customizability of ASICs enables accurate consideration of transistor count, minimizing resource wastage in the design process.
- When producing substantial quantities of designs tailored for a specific purpose, ASICs emerge as the optimal choice[13].

The difference between ASICs and FPGAs is summarized in the following table:

Table 1. *Difference Between ASIC and FPGAs while implementing SoCs.*

Differentiating Parameters	**ASIC**	**FPGA**
Time to Market	Slow	Fast
Design Flow	Complex	Simple
NRE	High	Low
Power Consumption	Low	High
Performance	High	Medium
Unit Size	Low	Medium
Unit Cost	Low	High

2.2 Digital System Design Flow

The primary stages of the System-on-Chip design flow, in accordance with the product's life cycle, include Exploration, Development, and Production [10]. The initial stage of decision-making involves establishing and examining the requirements in relation to the existing technologies, scheduling constraints, and financial considerations. The development process covers both the design of hardware and software components, as well as the verification and integration of these components. Once chip samples that meet the required standards have been obtained and the software has been stabilized, mass production can commence. The nature of the work is iterative; however, it requires the strict sequential execution of numerous tasks within each phase [10] [14] .

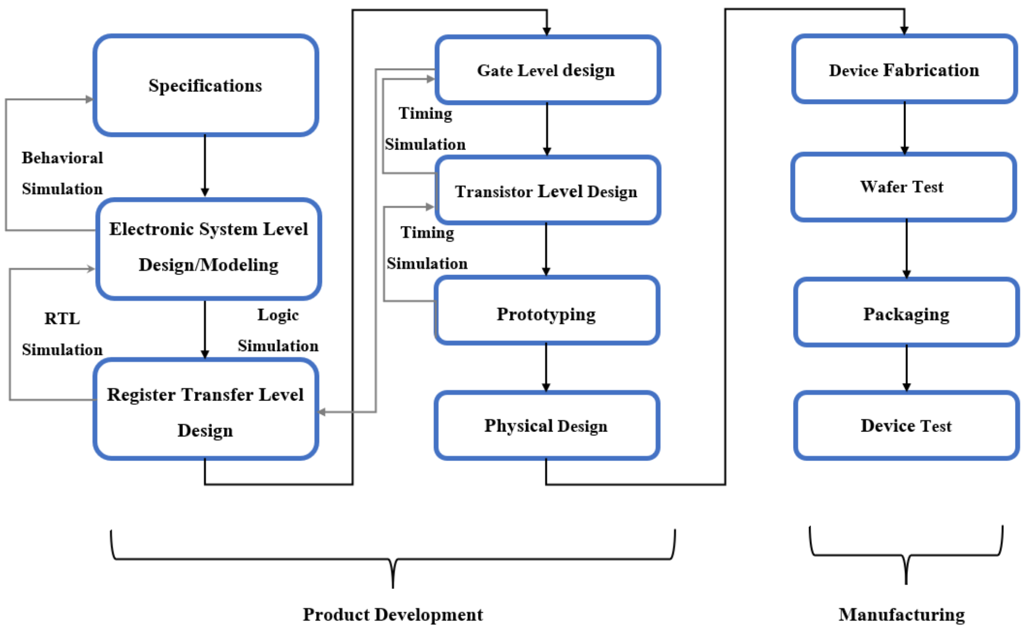


Figure 3. *A simple digital system design flow [15].*

2.2.1 Specification

The initial stage of digital design involves establishing a set of system specifications. Specifications can be determined either by the consumer of a particular product or by a governing entity that establishes functional criteria for a dependable and effective design, covering aspects such as power, area, and performance, among others.

2.2.2 Modeling

The development phase of SoC design begin with the process of modeling. The primary objective of modeling is to enable informed decision-making, as the overall flow is complex and extensive in nature. This step also allows to check the feasibility and practicality of the specification, given that significant alterations at a later stage can incur substantial costs. During this phase, the system requirements are established and examined using abstract system models in C, Matlab or SystemC, that can be executed, simulated, and analyzed. The system includes a substantial entity comprising the architecture, modules, and interfaces for both hardware and software components. The implementation of various components of the abstract system model can be generated, such as the automated generation of Verilog HDL code from the IP-XACT models [14].

2.2.3 Hardware Design

The next stage involves the extensive hardware design process. One of the most crucial aspects in the field of digital hardware design involves the utilization of HDL code at the register transfer level (RTL) abstraction, commonly referred to as RTL design. Moreover, scripts are utilized to create and manipulate the codebase, as well as to integrate intellectual property (IP) blocks sourced from third-party vendors. The RTL design methodology exhibits a greater resemblance to software programming rather than circuit design within the context of mixed signal design. When mixed signal designs are done, implementation of both analog and digital designs can be done parallelly[15].

2.2.4 Verification

Verification is a crucial procedure aimed at ensuring that the SoC design aligns with its specified requirements. Typically, this process is executed through simulations. SoC verification can take up to 70% of the time of the entire flow because the design needs to be bug free. It can be costly if bugs are reported in the manufacturing phase [3]. Both functional and formal verification techniques are employed during the flow. One of the popular verification methodologies used widely in the industry is the Universal Verification Methodology (UVM) standard [16], which acts as the foundational framework, alongside the System Verilog language.

2.2.5 Synthesis

Synthesis is the process that translates RTL designs into technology-dependent netlists that are tailored to a specific technology, while also optimizing them according to a predetermined set of constraints. Synthesis is mostly an automated process where the input to the EDA is behavioral RTL design and a set of constraints and the output is a gate level netlist which is optimized for either speed, area or power depending on the requirements [2] [12].

2.2.6 Prototyping

The term prototyping refers to the process of conducting physical tests on the chip. During this step, the RTL description of the chip is synthesized to an FPGA which subsequently serves as an emulator for the chip. The primary emphasis lies on functionality, as the timing and clock speeds vary significantly between the final SoC and FPGA [1].

2.2.7 Physical design

Physical design refers to the process of generating the transistor layout on the silicon die. The input of this process consists of a netlist obtained during the synthesis phase, and its purpose is to generate the placement and routing of the transistors. During the process of place and route, the EDA tool is responsible for determining the optimal layout and interconnection of logic elements within the targeted FPGA device. This is done while ensuring compliance with any specified user-defined settings or constraints. The EDA tool automatically chooses suitable resources, interconnection paths, and pin locations as its default behavior. When users allocate logic to particular device resources, the EDA tool attempts to fulfill those requirements by matching them accordingly. Subsequently, the tool proceeds to fit and optimize any remaining design logic that is not subject to constraints. In the event that the EDA tool is unable to fit the design within the target device, it will cease the compilation process and generate an error message. In case of an ASIC, the typical perspective of a designer is limited to the observation of standard cells. A standard cell consists of gates, registers, memories, interconnects, and other components. The individual transistors are exposed to the designer only in case of a full-custom design [14].

2.2.8 Floorplanning

Floorplanning is a crucial step of the hierarchical design methodology. The circuit blocks are assembled into the chip optimizing it for the metrics such as area and wire length, which directly affects the cost. The circuit blocks may exhibit either flexible or rigid configurations in terms of their physical shapes. **Placement** refers to the procedure of allocating circuit components within a designated region on a chip. The problem at hand can be characterized as a constrained floorplanning problem involving rigid blocks that exhibit certain similarities in their dimensions. Following the placement stage, the **routing** process is responsible for determining the specific routes for conductors that facilitate the transmission of electrical signals on the chip layout. These routes are designed to interconnect all terminals that possess electrical equivalence. Following the routing stage, a series of physical verification procedures, including **design rule checking** (DRC), performance checking, and reliability checking, are conducted to ensure compliance with design rules and specifications in terms of geometric patterns, circuit timing, and electrical effects [15].

2.3 Hardware Description Language

A hardware description language (HDL) is a language used for the purpose of specifying the behavior or structure of ICs in the realm of digital circuits. HDLs are further used for the purpose of circuit stimulation and verification. There exists a wide range of HDLs, among which VHDL and Verilog have emerged as the most widely adopted and prevalent options. The majority of the CAD tools currently available in the market provide support for these HDLs. VHDL is an acronym that stands for "very high-speed integrated-circuit hardware description language." Both VHDL and Verilog are recognized as official IEEE (Institute of Electrical and Electronics Engineers) standards. Additional high-level description languages (HDLs) include Java HDL (JHDL) and proprietary HDLs, such as Active-HDL developed by Cypress Semiconductor Corporation [12].

HDLs are commonly used for the programming of systems based on PLDs and FPGAs. Intel and AMD(Xilinx) corporations offer free restricted editions (intended for educational purposes) of CAD software and tools, facilitating the programming of FPGA-based development boards. The CAD tools have a range of essential components, such as a schematic editor, a VHDL/Verilog editor, compilers, libraries, design simulators, and a variety of utility tools.

HDLs differ from conventional computer programming languages as it encompasses several distinctive language components. These include vector-shaped wire nets and registers, as well as non-blocking assignments within the process.

HDLs have been developed to address various requirements within the design process.

- It allows a system’s hierarchy, that is how a system is split into subsystems and how these subsystems are interconnected.
- It supports the ability to define the functionality of a system by employing commonly used programming language constructs.
- It allows the simulation of a system's design prior to its production, thereby enabling designers to efficiently evaluate different options and verify accuracy without the need for costly and time-consuming hardware prototyping.
- Thus, this approach enables the synthesis of a comprehensive design structure from a higher-level specification, thereby enabling designers to focus on strategic design choices and minimizing time-to-market.

2.3.1 Verilog HDL

Verilog HDL is a hardware description language used for the textual representation of the structure and behavior of digital system hardware. The representation of logical circuit diagrams, logical expressions, and the logical functions executed by digital logic systems can be achieved through this language. The development of the language took place during the mid-1980s by Gateway Design Automation, a company that was subsequently acquired by Cadence in 1989. It is a IEEE standard[17].

The primary objective of Verilog is to serve as a hardware description language that shares a fundamental syntax resemblance with the C programming language. The extensive use of the C language in various domains during the early stages of Verilog design leads to the inclusion of several language elements from C. A hardware description language that bears resemblance to the C language has the potential to facilitate the learning and acceptance process for circuit designers. Typically, designers proficient in the C programming language will likely acquire proficiency in the Verilog hardware description language at an accelerated pace [17].

2.3.1.1 Verilog Constructs

The fundamental structural component that Verilog uses to describe hardware is a module. The design and development of complex electronic circuits primarily involves the interconnection and integration of modules. The inclusion of modules is simplified by the utilization of the keywords "module" and "endmodule". The Verilog module has resemblance to the function found in the C programming language. Module is capable of defining both input and output ports. This module has the capability of calling other modules through instantiation, and it can also be called by instances of other modules. The module has the capability to incorporate both combinational logic and process timing[18].

```
//Verilog Code for design
module fulladder (input [3:0] a,
                  input [3:0] b,
                  input c_in,
                  output c_out,
                  output [3:0] sum);
  always @ (a or b or c_in) begin
    {c_out, sum} = a + b + c_in;
  end
endmodule
```

Program 1. *Example of simple adder design with Verilog*

Program 1 above illustrates the design of a 4-bit full adder using Verilog HDL. The code was modeled using the continuous assignment operator *always* block with a sensitivity list that includes all inputs. An always block is executed whenever any of the inputs in the list changes their values.

Designers have the ability to set up a top-level module for the purpose of testing, wherein they call the module through the use of instances. The module at the highest level is commonly known as the "Testbench". To enhance the functional verification of the circuit's logic, it is essential that the test code covers a comprehensive range of statements, branches, conditions, paths, triggers, and state machine states within the system. Verification engineers are required to generate a sufficient amount of input within the testing platform. The process involves stimulating and establishing a connection with the module being tested, followed by evaluating the performance of the module's output against expected outcomes. Verilog offers a specialized data structure designed for efficient verification, which can be validated through the use of random testing. This method is particularly valuable in the intricate process of verifying integrated circuit designs. In order to properly invoke a module, it is essential to establish the port connections in the sequence specified by the module declaration. The top-level verification module does not require calls by external entities; thus, it lacks both input and output ports.

2.3.2 VHDL

Very High Speed Integrated Circuit (VHSIC) hardware description language or VHDL is one of the powerful languages for describing electronic systems. It was invented by the USA’s department of defense in 1983 and later it was standardized by IEEE in 1987.

VHDL has been developed to address various requirements within the design process.

2.3.2.1 VHDL Constructs

A VHDL system in a .vhd file is described in 2 parts; the entity and the architecture. Inside the entity, the interfaces (the input and the outputs) of the system is described. The system behavior is described in the architecture. The package defines the functionality of VHDL, which includes operators, signal types, and functions. The packages are organized into a library. The standard package defined by IEEE includes the fundamental set of functionalities for VHDL. This standard package is then encompassed within a library called IEEE. The inclusion of libraries and packages in a VHDL file is typically specified at the beginning, prior to the declaration of the entity and architecture. The inclusion of other packages allows for the incorporation of supplementary features into VHDL. However, it is important to note that all packages are built upon the fundamental functionality outlined in the standard package[19].

```
library ieee;                  //Library declaration
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder_1_bit is          //Entity declaration.
  port(
      -- Inputs               //Declaring inputs
      A : in std_logic;
      B : in std_logic;
      Cin : in std_logic;
      -- Outputs              //Declaring outputs
      SUM : out std_logic;
      C_out : out std_logic
      );
end entity adder_1_bit;

architecture rtl of adder_1_bit is//Describing Architecture
begin
    process(A,B,Cin) is
    begin
        SUM   <= (A xor B xor Cin);
         C_out <= ((A xor B) and Cin) or (A and B);
    end process;
end architecture;
```

Program 2. *Example of simple adder design with Verilog*

Program 2 shows the VHDL code for designing a simple 1-bit adder. After the library declarations, entity is described where all the inputs and outputs are defined. In the architecture of adder_1_bit, the transfer function of the inputs to the outputs are written in the form of RTL code.

2.3.3 SystemVerilog

SystemVerilog is a Hardware Description Language (HDL), which is developed by Accellera as an extension of Verilog HDL in 2004, which is used to design, specification and verification language in semiconductor and system design industries.

Being a unified language, SystemVerilog can be used for abstract and detailed design specification, assertions specification, testbench verification, both on manual and automatic methodologies and in coverage of the design[20]. SystemVerilog Assertions are a form of verification code that assesses the conformity of a design to a given specification. They serve to confirm the intended behavior of the design and provide a corresponding message as feedback. If the design does not demonstrate the specified behavior, the assertion will fail, suggesting a flaw in the design's behavior.

It was originally developed as an extension to the IEEE 1364-2005 standard for the Verilog design language, however the two standards were combined into a single language in 2009. The primary objective of the initial SystemVerilog extensions was to offer verification engineers with enhanced tools operating at a higher level of abstraction, thereby enhancing productivity, reusability, and readability. In addition, there was various design specification techniques, including the introduction of novel data types, the utilization of packages, the expansion of port declarations, and the implementation of interfaces [21].

SystemVerilog constructs will be discussed in chapter 3 in details.

2.4 Electronic Design Automation

Electronic design automation (EDA) plays a pivotal role in driving technological advancements aimed at enhancing the quality of human life and facilitating daily usage. In the context of an electronic system modeled at the electronic system level (ESL), the process of EDA serves to automate the design and testing procedures. This involves verifying the accuracy of the ESL design in relation to the electronic system specifications, guiding the ESL design through multiple synthesis and verification stages, and ultimately conducting tests on the manufactured electronic system to ensure compliance with the electronic system's specifications and quality standards. The electronic system may also manifest as a printed circuit board (PCB) or an (IC. The integrated circuit has the capability to function as a SoC, an ASIC, or an FPGA [22].

EDA is a collection of hardware, software and services tool used in the entire manufacturing process of the semiconductor devices. The aim of these tools is to define, plan, design, implement, verify and subsequently manufacturing of the semiconductor devices or ICs. The advancement of EDA tools helps the IC manufacturers in increased productivity and gives many other advantages, such as,

- EDA tools are capable to handle this increased complexity.
- Helps in shortened time to market to maintain a competitive advantage.
- With the help of EDA tools, power consumption of ICs can be reduced. Also, the performance and space can be optimized with respect to the requirements.
- Incorporating with Artificial Intelligence/Machine Learning (AI/ML) tools, EDAs are currently used in design, simulation, verification, and emulation [23] [24].

EDAs are mainly a very complex and sophisticated domain of software where the Market share of the tools are distributed among the following aspects of design and manufacturing tools.

Before implementing a proposed circuit, *simulation* tools simulate its behavior based on its description. Typically, this description is written in a standard hardware description language, such as Verilog or VHDL. Simulation tools mimic the behavior of circuit elements in varying degrees of detail and execute a variety of operations to predict the circuit's behavior. The required level of detail is determined by the type of circuit being designed and its intended application. When processing a large quantity of input data, hardware approaches such as emulation and rapid prototyping are considered. This occurs when the operating system of a processor must be evaluated against real-world scenarios, such as video processing. Without a hardware-assisted strategy, the runtime for these cases can be challenging.

The purpose of *design* tools is to take a proposed circuit function and generate the set of circuit elements needed to realize that function from the required description. Choosing and connecting the appropriate circuit parts to provide the intended function may be a logical procedure throughout this assembly. One such example is *logic synthesis*. It can also be a physical procedure involving the placement and routing of geometric forms on silicon to create the circuit, which in general is known as place and route. It can also take the form of a designer-guided interactive procedure which is referred to as custom layout [24] .

Verification tools check if the final chip design is physically or logically implemented correctly and can provide the expected performance. There exist numerous processes that can be employed in this context. The process of physical verification involves the examination of interconnected geometries to ascertain whether their placement conforms to the manufacturing requirements of the fabrication facility. The complexity of these requirements has significantly increased, encompassing a multitude of rules that can exceed a quantity of 10,000. Verification can also be achieved by conducting a comparison between the implemented circuit and the original description, in order to ascertain that the former accurately represents the desired function. The process referred to as *Layout vs. Schematic*, or LVS, serves as an illustrative example of this phenomenon. Simulation technology can be employed in the functional verification process of a chip to assess the conformity of its actual behavior with the anticipated behavior. The effectiveness of these approaches is constrained by the extent to which the input stimulus is comprehensive. An alternative method involves algorithmically validating the performance of the circuit, eliminating the requirement for input stimulus. The aforementioned methodology is commonly referred to as equivalence checking and is a fundamental component within the field of formal verification [21].

3. SYSTEMVERILOG

As mentioned earlier, SystemVerilog can be used for abstract and detailed design specification, assertions specification, testbench verification, both on manual and automatic methodologies and in coverage of the design. The basic understanding of the constructs of System Verilog will be discussed in this chapter.

3.1 SystemVerilog Design and Verification Building Blocks

3.1.1 Design Elements

Design elements are the fundamental components used to model and construct a design or verification environment. These building blocks hold the declarations and procedural code that will be discussed in later sections of this thesis. The design elements in System Verilog are module, program, interface, checker, package, primitive and configuration[20].

3.1.2 Modules

The fundamental unit in SystemVerilog is the module. A module is encapsulated by the keywords module and endmodule. Modules are predominantly used for representing design blocks. They can also be used as containers for verification code and facilitate the interconnections between verification blocks and design blocks. Modules might contain the constructs like port definitions, data and constant declarations, class definitions, procedural blocks etc.

Procedural statements include behavioral code, in which programming statements such as if-else, case, or for loop structures are used to define the desired functionality. The statements typically consist of a sequential block that is bounded by the keywords "begin" and "end". Within this block, the statements are executed in a sequential manner, following the specified order. As a result, all the statements within the procedure function act syntactically as a single assignment. There exist two fundamental categories of procedures: initial and always. The initial procedures are executed only once and are typically used for the purpose of initializing variables. Always procedures define combinational and sequential logic, which is executed in response to events specified in the sensitivity list. The sensitivity list is declared by utilizing the character @[20].

3.1.3 Programs

The program construct is provided for replicating the testbench environment and is enclosed between the keywords program and endprogram. The primary use of the program construct is to define specific simulation execution semantics. On top of that, it acts as a distinct separator between the design and the testbench components. Also, by utilizing the clocking blocks, the program construct allows a race-free form of collaboration between the design and the testbench, thereby enabling the utilization of cycle- and transaction-level abstractions.

The program block acts in three distinct ways. Firstly, it serves as a starting point for the implementation of testbenches. Secondly, it allows to incorporate data, tasks, and functions throughout the program. Lastly, it provides a syntactic context for specifying scheduling within a reactive region set.

A program block may include data declarations, class definitions, subroutine definitions, object instances and initial or concluding procedures. It should not contain always, procedure, primitive, module or interface instances [25].

3.1.4 Interfaces

The interface construct, which is defined by the keywords interface...endinterface, serves as a container for facilitating communication between design blocks, as well as between design and verification blocks. This enables a seamless transition from an abstract system-level design to progressively more detailed representations, such as register-transfer and structural views of the design. The interface construct enables design reuse by encapsulating the communication between blocks.

Interfaces are basically a set of nets or variables. The interface can be connected to the interface ports of other instantiated modules, interfaces, and programs after being instantiated in a design[26].

3.1.5 Checkers

The checker is a representation of a verification block that includes assertions and modeling code. Checkers are designed to be used as building blocks for generating abstract auxiliary models used in formal verification or as verification library modules. It is defined within the keywords checker…endchecker.

3.1.6 Primitives

Low-level logic gates and switches are modeled using the primitive building block. Numerous primitive types are included by default in SystemVerilog. User-defined primitives (UDPs) are a way for designers to extend the built-in primitives. The words primitive...endprimitive are followed by a UDP. Gate-level models, often known as timing-accurate digital circuits, can be modeled using the built-in and UDP components.

3.1.7 Packages

Modules, interfaces, programs, and checkers serve the purpose of establishing a localized name space for declarations. Identifiers that are declared within a module, interface, program, or checker are considered to be local to the respective scope in which they are declared. As such, these identifiers do not have any impact on or create conflicts with declarations made in other building blocks. Packages offer a designated area for declaring variables, functions, and other elements that can be accessed and utilized by other components within a system. Package declarations have the ability to be imported into various building blocks, such as other packages.

3.1.8 Configurations

SystemVerilog offers the possibility to define design configurations, which allow for the specification of the binding details between module instances and specific SystemVerilog source code. Configurations make use of libraries. A library is a compilation of various components such as modules, interfaces, programs, checkers, primitives, packages, and other configurations.

The files inside the library map serve the purpose of specifying the source code location for the cells that are contained within the libraries. The designation of the library map files is commonly indicated as invocation options for simulators or other software tools that parse SystemVerilog source code.

3.2 SystemVerilog Hierarchy Overview

A SystemVerilog hierarchy is constructed using the fundamental building blocks of modules, programs, interfaces, checkers, and primitives. One building block instantiating another building block results in hierarchy. A new level of hierarchy is generated when a module contains an instance of another module, interface, program, or checker. Connections to the ports of the instantiated module, interface, program, or checker are the main means of communication between levels of hierarchy.

The following is an example of a design utilizing a simple two levels of hierarchy. Module top_design holds an instance of module mux2to1, which allows to create two levels of hierarchy.

```
module top_design;          //Module with no ports
  logic in1, in2 , select;  //Variable declarations
  wire out1;                //Net Declarations
  //Module Instance
  mux2to1 m1(.a(in1), .b(in2), .sel(select), .y(out1);
endmodule: top_design


module mux2to1 (input wire a, b, sel,
                output logic y); // combined port and type declaration

  // netlist using built-in primitive instances
  not g1 (sel_n, sel);
  and g2 (a_s, a, sel_n);
  and g3 (b_s, b, sel);
  or g4 (y, a_s, b_s);
endmodule: mux2to1
```

Program 3. *Example of design hierarchy in SystemVerilog*

3.3 SystemVerilog for verification

SystemVerilog allows object-oriented programming for verification purposes on a higher level of abstraction. The object-oriented properties in SystemVerilog adhere closely to the principles of object-oriented programming, making the structure familiar to designers with prior experience in languages such as C++, Java, or other object-oriented programming languages. One of the primary advantages of object-oriented testbench design is the ability for the designer to declare intricate data types and integrate them with routines that manipulate the data. Instead of directly manipulating bits in the Device Under Test (DUT), these routines can be utilized to execute even intricate transactions without the need to consider the state of every bit during each clock cycle[26].

The primary component for constructing a high-level testbench in SystemVerilog is the class. The class combines data and routines within a unified code block. Routines in SystemVerilog can either be tasks or functions. Functions have return types. Functions possess the ability to accept input and produce output values, and they are executed without impeding the progression of the simulation time. From a simulation standpoint, functions promptly return their computed value. On the other hand, Tasks do not have a return value, but they are capable of blocking the simulation time during execution, thus possessing a temporal aspect. Tasks have the ability to include delays in order to synchronize the processing with a specific point in time, a signal value, or another event.

Classes are instantiated as objects. An object possesses both a type and a name, and its instantiation involves the initial creation of a variable of the class's type to serve as an object *handle*. Subsequently, the object is created and assigned to the variable through the utilization of the *constructor* function, commonly referred to as "new". The extends keyword allows for the derivation of new classes from existing base classes. The properties and methods that are explicitly stated in the base class can be accessed using the keyword "super". Child classes can also override the methods declared in base class.

```
class transaction;

  // Class properties
  bit [31:0] data;
  int id;

  //Class methods
  task update(bit [31:0] m_data, int m_id);
    data = m_data;
    id = m_id;
  endtask

  function void print(transaction tr);
    $display("Value of data = %0h and id = %0h", tr.data, tr.id);
   endfunction

 endclass

 module tb_top;
   initial begin
        transaction tr;    //variable of class data_type or class handle
        tr = new();        // memory is allotted for a variable or object.
     tr.update(5, 9);
     tr.print(tr);
   end
 endmodule
```

Program 4. *Example of a SystemVerilog class with routines.*

Program 4 shows an example of SystemVerilog class called transaction. The class contains a task and function with return type void. Both the task and function can be accessed through the handle defined in the tb_top module.

4. UNIVERSAL VERIFICATION METHODOLOGY

The Universal Verification Methodology (UVM) is an industry standard that enables re-use of verification environments and verification IP (VIP), which in turns helps in faster development and verification of a product. It is researched, developed, and standardized by Accellera, an association of EDA tool vendors and users such as Mentor Graphics, Cadence, Intel, AMD and Synopsys.

UVM is a System Verilog based verification methodology that is defined as a set of classes which uses System Verilog syntax and semantics. By offering an Application Programming Interface (API) framework that can be used across various projects, UVM's main goal is to assist businesses in creating modular, reusable, and scalable testbench structures [27].

4.1 UVM Testbench Basics

The UVM has a layered, object-oriented methodology to testbench development, making it possible to act as a "separation of concerns" among team members. To improve productivity and facilitate reuse, each component in a UVM testbench serves a specific purpose and has a well-defined interface to the rest of the testbench. When these components are combined to form a testbench, the result is a modular, reusable verification environment that allows the test writer to focus on the transaction level, depending on the functionality that needs to be verified, while the testbench architect dedicate themselves on how the test communicates with the Design Under Test (DUT)[16].

4.2 UVM Building Block

A UVM testbench's structure is made up of relatively small but simple components that are arranged hierarchically. The UVM environment is kept separated from the test. The tests contain information about how to test the DUT whereas the environment describes the connection of the DUT to the testbench.

The advantage of dividing the testbench into small components is that it simplifies testbench design and reuse. Reuse can be both horizontal and vertical, which means that, in addition to using the same components in different testbenches, it is also possible to combine the verification environments for different blocks to create subsystems, which can then be further combined to implement system level testing.

As UVM testbench is developed from dynamic objects, which has no existence in the memory before they are being created. That is why, for launching the simulation, it is necessary to have a static component [16]. This static component in UVM is a top-level System Verilog module which consists of pin connections to DUT and initiates the test, sets up the environment, and executes a series of transactions to the DUT.

The UVM testbench file hierarchy utilizes SystemVerilog packages. Packages are structural elements that unite associated declarations and definitions within a unified namespace, serving as a consistent compilation unit for the simulator. In order to gain access to the namespace and the base definitions, it is necessary to import the package. The use of packages enables the testbench developer to effectively structure the code and maintain consistent references to types and classes.

A package file should have all the class definition files that go with it. For a simple UVM testbench, all the definitions could be in a single package. For a large system-level testbench, however, the declarations could be split between multiple packages, with a separate package for each bus interface and a number of packages for different types of test sequences that contain all the declarations for running different tests [16]. Instead of declaring all of the classes directly in the package file, Mentor Graphics' coding standards define that each class should be declared in its own file, and all of the declaration files should be added to the package using a SystemVerilog include directive. With the include command, the compiler needs to put the whole contents of a source file inside another file. Only the include instructions for class declaration files should be in the package.

4.2.1 Objects and Components

The object acts as the basic component in a UVM testbench, with all objects derived from the uvm_object base class [27]. The primary function of the uvm_object base class is to establish the standard methods for fundamental operations, such as creation and printing, that are commonly used by every objects. Additionally, it highlights interfaces for identifying instances, for example, unique id. The fundamental objects in this context are data packages that are transmitted to the DUT. These packages are instantiated as sequences of packages, which are then used to generate test input.

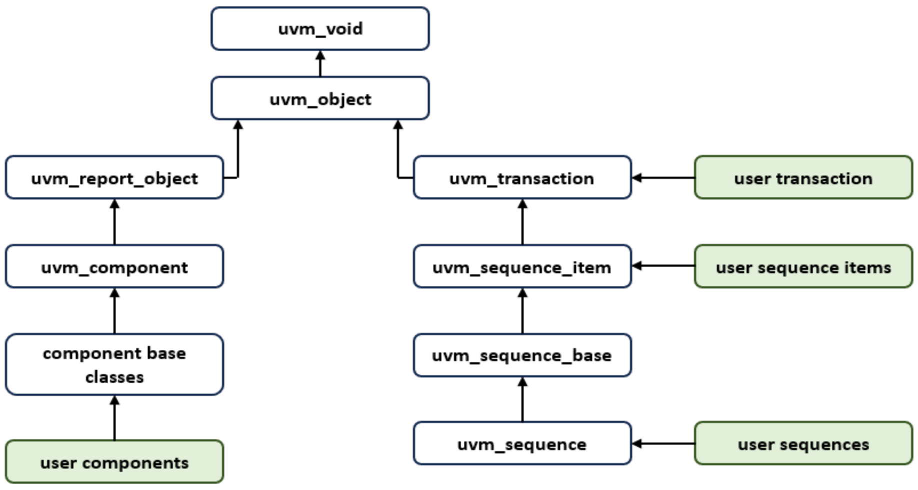


Figure 4. *UVM base class hierarchy [16]*

Figure 4 presents a hierarchical representation of the fundamental UVM classes. The uvm_object class is enhanced with a reporting interface to create the uvm_report_object class. The uvm_report_object class is then expanded to include the uvm_component class, which introduces the notion of hierarchy and the necessary properties for the creation and connection of components. The uvm_component class is expanded to include base classes for user-implementable components, as depicted in the figure. In addition, the uvm_object is further expanded upon in the context of transactions and sequence items, which collectively constitute a sequence of items that are subsequently transmitted to the DUT. The user has the ability to create the highlighted classes.

Given that all base classes for individual components are derived from the uvm_component, which includes all shared properties, the implementation of these classes is relatively straightforward. The hierarchical tree of the typical UVM components is depicted in *Figure 5*. The testbench designer has the ability to differentiate between components by inheriting them from their respective base class. This approach ensures that the components will inherit and utilize all the characteristics of the base class. Certain foundation classes, like uvm_monitor, serve as empty containers that do not introduce any additional functionality to classes derived from uvm_component. However, it is possible that functionality could be incorporated in subsequent versions of UVM.

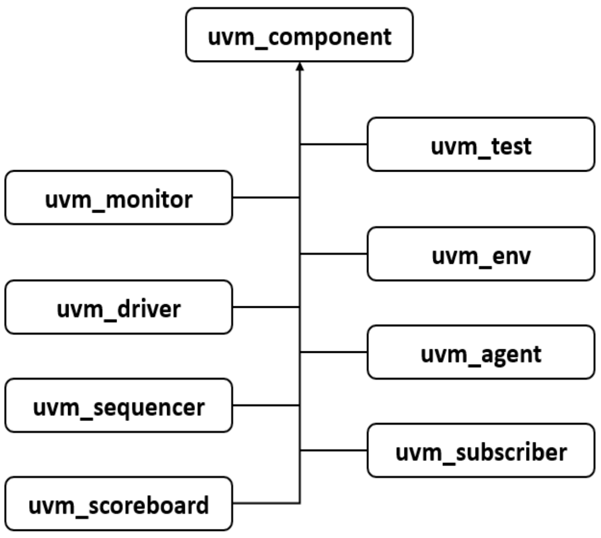


Figure 5. *UVM component class hierarchy[16]*

4.2.2 UVM Phasing

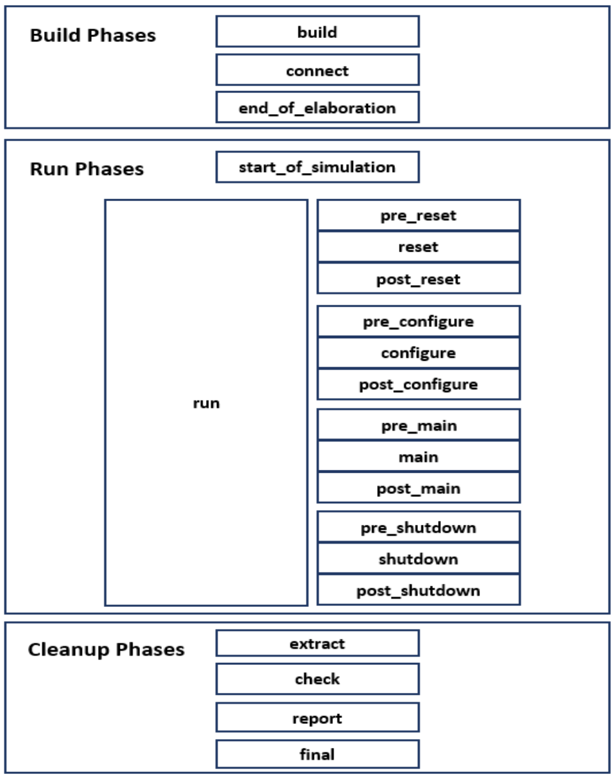


Figure 6. *UVM Phases [16]*

The process of simulating UVM consists of distinct phases. The chronological order of the UVM phases is depicted in *Figure 6*. There are a total of 21 simulation phases, which can be classified into three separate categories. At the start of the simulation, the build time phases are responsible for constructing the test environment. This is achieved by using the factory to build components, establishing connections between the Transaction Level Modeling(TLM) channels, and configuring all the components through the configuration database. The various stages of the build process do not require any simulation time.

The run-time phases are initiated, and the simulation time is used after the test environment has been built. The actual simulation in which the test case is executed for the DUT is carried out during the run-time stages. When a test is terminated, simulation time is no longer used, and cleanup phases gather and report the test case's results [25].

4.3 UVM Architecture

A Universal Verification Methodology (UVM) testbench's architectural framework is made up of user-defined components that derive from base classes offered by the UVM library. Every element of the testbench hierarchy has a given name and a parent. These components are created sequentially from top to bottom during the build phase using the UVM factory. *Figure 7* shows a block level UVM testbench in illustrated form. The UVM environment in figure *7* is used in many tests. The agents communicate between the two bus interfaces present in the sample scenario, together with elements that track test coverage and carry out functional checks on the DUT.

A simple UVM testbench was developed as a use case as part of this thesis. So, Understanding the UVM architecture is a crucial objective. In the following section, the difference between UVM tests and UVM environment is highlighted.

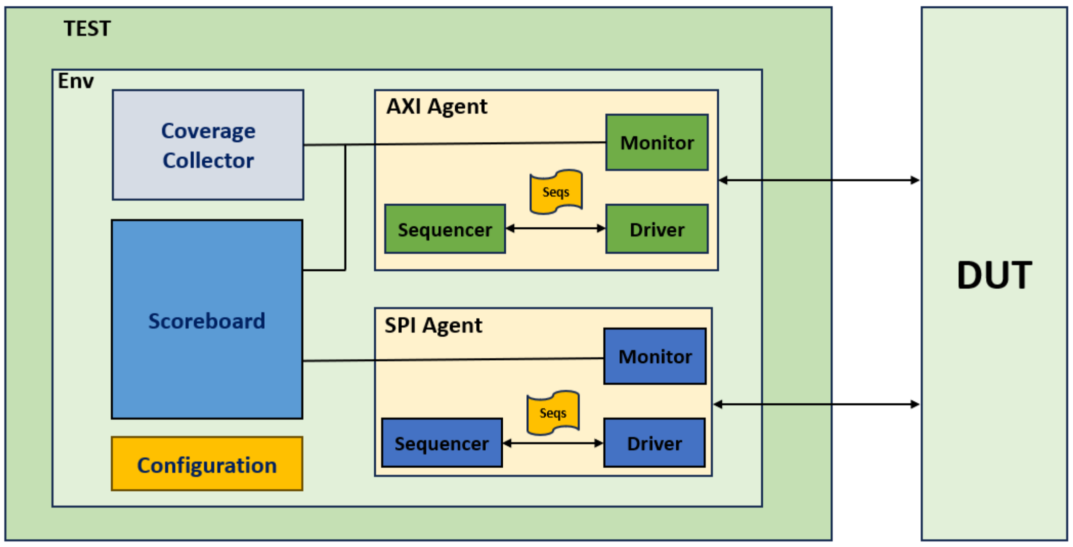


Figure 7. *Block Diagram of UVM Architecture [27] .*

4.3.1 UVM Environment

The *environment* refers to the structural framework of the testbench, that includes its physical architecture. The instantiation of components occurs in a hierarchical manner, with the possibility of multiple sublevel environments being contained within a single environment. In the context of system level testbenches, it is common to have a single top-level environment that creates multiple instances of environments for each individual block within the design. The use of identical block level environments can be employed for conducting block level testing prior to integration. The process of constructing a system-level testbench by utilizing block-level environments is commonly known as vertical reuse.

The environment consists of multiple *agents* that interact with the DUT, as well as *subscribers* who utilize the data supplied by these agents. The agent, as shown in figure 7, instantiate the components that control the stimulus flow, provide the DUT with input data, and track the signals as they travel between the testbench and the DUT. There is typically one agent for each interface that the DUT is connected to. The configuration database determines whether an agent will play an active or passive role: an active agent supplies stimulation data for the DUT, whilst a passive agent just keeps track of the transfers [27].

The typical components instantiated within the agent include a *sequencer*, *driver*, and *monitor* (As depicted in *Figure 7*). The sequencer works as a liaison that retrieves sequence objects from a list and regulates the progression of the sequence. The sequences are transmitted in the form of TLM transactions. The driver is responsible for receiving the TLM transactions from the sequencer and transmitting them to the DUT. Consequently, the driver converts the high-level transaction sequences into low-level activity within the DUT.

A *monitor* tracks the operations occurring within the DUT interface and samples its behavior. The pin-level activity within the DUT is transformed into TLM transactions and then transmitted for analysis. The monitor includes an analysis port that facilitates the transmission of TLM data to the testbench. In case of a passive agent, the sequencer and driver functionalities are disabled, leaving only the monitor operational. The information transmitted from the monitor undergoes analysis by *subscribers* that utilize analysis exports, establishing a connection to the analysis port within the monitor. Typically, these subscribers are located outside the agent within the environment. However, in more intricate designs, they may also be instantiated within the agent itself.

The *coverage collector* is a subscriber responsible for collecting data on functional coverage. It accomplishes this by collecting and examining all the transactions transmitted by the monitor and using this data to update counters within covergroups. Covergroups define the particular signals and conditions to be observed as coverpoints. The counter values associated with each coverpoint provide real-time insights into functional coverage, indicating both tested and untested scenarios.

The *scoreboard* is a vital component of the UVM that is responsible for evaluating the correct functionality of the Design Under Test DUT. The process involves the specification of a reference model and the subsequent comparison of the output generated by the DUT with the reference. In basic designs, the declaration of the reference model and comparator can be consolidated within a solitary component. However, it is also feasible to employ distinct components or even use an external model of the Device Under Test DUT as the reference [16].

4.3.2 UVM Tests

The *test* acts as the top-level component within a UVM testbench. The test is responsible for managing the construction and configuring the test environment, determining the sequence of stimuli to be used during the test, and directing the simulation process. Multiple tests can be conducted within a shared environment, each applying distinct sequences and configurations. It is a common practice to create a base test class that initializes the testing environment and performs the required configuration. Subsequently, this base test class is extended to include various test cases for the DUT.

Sequences are collections of objects that are transmitted to the sequencer within the agent. The *sequencer* methodically processes each item. The layering of sequences allows for the description of complex transactions involving multiple layers. A sequence at a higher level of abstraction has the ability to control transactions and command to sequences operating at a lower level that are closer to the hardware. In order to facilitate randomized testing, it is possible to assign variables in sequences to be randomized. Constraints can be implemented in order to restrict the randomization process to a particular range of values or to establish distributions that ensure a signal is predominantly set at a high level, approximately 95% of the time, while being set at a low level for the remaining duration [27].

5. OVERVIEW OF EDA PLAYGROUND

EDA Playground is an open-source web-browser based integrated development environment (IDE) that allows the electronic design and simulation of various HDLs such as SystemVerilog, Verilog, VHDL, and C++/SystemC. The primary objective of this initiative is to accelerate the learning of design and testbench development skills by facilitating code sharing and enhancing accessibility to simulators and libraries.

It was originally developed by Victor Lyuboslavsky as a part of Victor EDA, which was later acquired by Doulos in 2019.

The objective of this chapter is to demonstrate the extent of usability of EDA playground by applying design, simulation, verification, and synthesis with VHDL and SystemVerilog.

The work was mainly focused on two parts. Firstly, a simple VHDL design was created in the platform and a VHDL testbench was simulated using different simulators. Secondly, a simple SystemVerilog design was implemented, and the design was simulated using simulators and verified using the UVM. The focus of the latter work was to showcase the feasibility of implementing a complete UVM flow utilizing the EDA Playground platform.

5.1 Application of EDA Playground

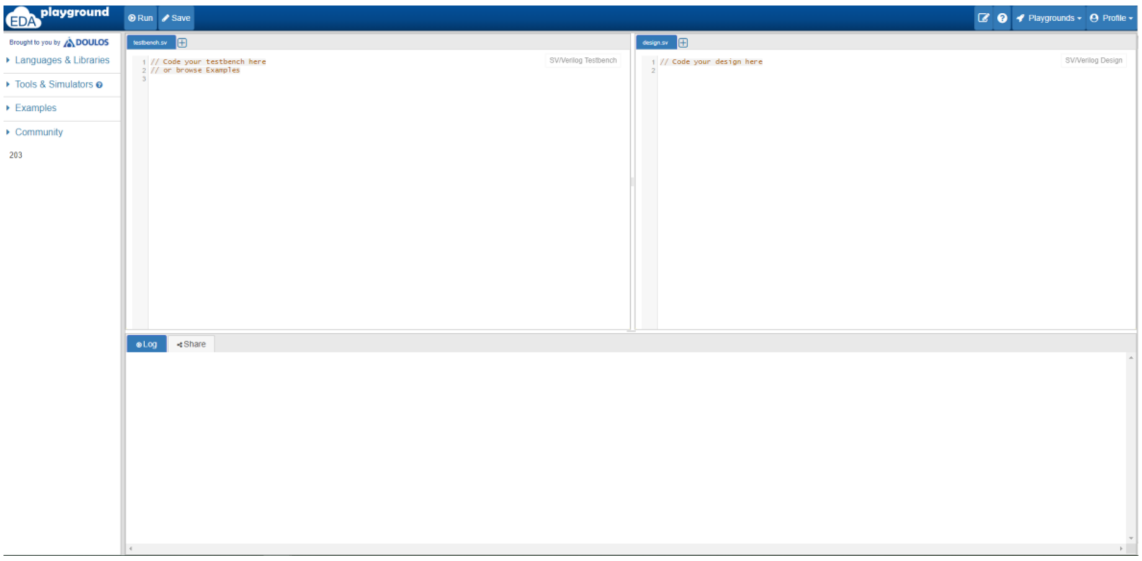
EDA Playground has the potential to be used for various applications. Primarily, this platform can serve as an introductory educational tool or for the purpose of prototyping, as it offers support for design and testbench in multiple programming languages, as well as a diverse range of simulators. Additionally, it provides support for various verification frameworks such as UVM, SVUnit, Verilog, or Python, which are frequently used by engineers. Moreover, numerous engineers have reported that it has been used as an assessment tool for evaluating the coding and debugging abilities of interview candidates.

5.2 User Interface

EDA Playground has a straightforward user interface. To access the application, users can direct their browsers to the domain address: https://www.edaplayground.com/. Users can continue to the interface without logging in, but it is recommended to register and log in as it allows some additional features like sharing code and saving workspace.

Figure 8 illustrates the interface of EDA Playground. it shows the presence of two windows. The window located on the right side is designated for design purposes, while the window situated on the left side is intended for testbench functions. The user has the capability to include a maximum of ten files using the "+" icon, which is accessible in both the design and testbench windows. Located on the far-left side of the interface, one can find the selection panel which includes language options, tools and simulators, examples, as well as community sharing features. The log can be found in the lower section of the screen. The profile and saved playground options are in the upper right corner of the interface.

After the completion of the design and testbench, the user is required to choose the preferred simulator and initiate the execution process by selecting the run button. The user interface includes checkboxes that allow for the opening of EPWave (a tool used for visualizing waveforms) and for visualizing the netlist generated after synthesis. Once these boxes are selected, the waveform and netlist will be displayed in a new tab.



```
// Code your testbench here
// or browse Examples
```

```
// Code your design here
```

Figure 8. *User Interface of EDA Playground*

5.3 Languages and Libraries

EDA Playground allows users to design and test with several languages, which include: VHDL, SystemVerilog, C++/SystemC, python, perl, and others. It also allows users to use methodologies like UVM 1.2 and OVM 2.1.2.

5.4 Tools and Simulators

Within the user interface (UI) of the EDA Playground, there exists a variety of simulation tools situated on the left-hand side. These tools include both commercial and open-source simulators, providing users with a diverse range of options for their EDA needs. Individuals have the ability to choose simulation tools based on their specific requirements. In addition to the aforementioned tools, a limited number of synthesis tools are also accessible for users who wish to assess the designed design's utilization. In addition to simulators and synthesizers, there are also available compilers for C++ and Python programming languages. All simulators within the EDA Playground are integrated with EPWave, a debugging tool that enables users to visualize waveforms.

5.4.1 Commercial Simulators

The available commercial simulators on this platform include Aldec Riviera Pro 2022.04, Cadence Xcelium 20.09, Mentor Questa 2021.3, and Synopsys VCS 2021.09. Each simulator possesses distinct advantages, allowing users to choose the most suitable option based on their individual requirements.

5.4.2 Synthesis Tools

Commercial synthesis tools that can be used with EDA Playground include Mentor Precision 2021.1 and Aldec SyntHESer 2022.05. Along with these, this EDA platform also incorporates opensource synthesis tools like Yosys 0.9.0 and VTR 7.0. There is no target platform for these synthesis tools as these are all generic and available for educational purposes.

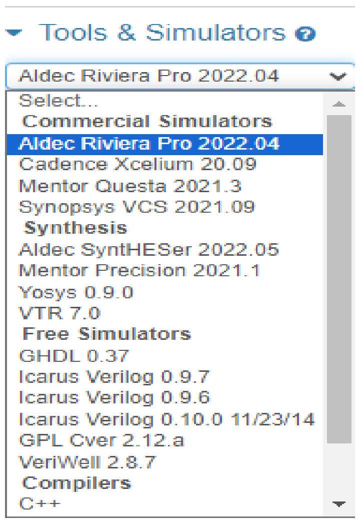


Figure 9. *Tools and Simulators in EDA Playground*

5.5 EPWave

EPwave is a designated wave viewing tool developed by Doulos which is incorporated with EDA Playground. During the course of this thesis, designs were created with VHDL, Verilog and SystemVerilog. All the designs' testbenches were visualized with the help of EPWave. It will open up in the web browser automatically after the simulation is completed if the "Open EPWave after run" checkbox is clicked. It was also ensured that the testbench code incorporated suitable function calls for the purpose of generating a file with a *.vcd extension.

```
initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0, tb_top);
end
```

Program 5. *Code for dumping filename and variables*

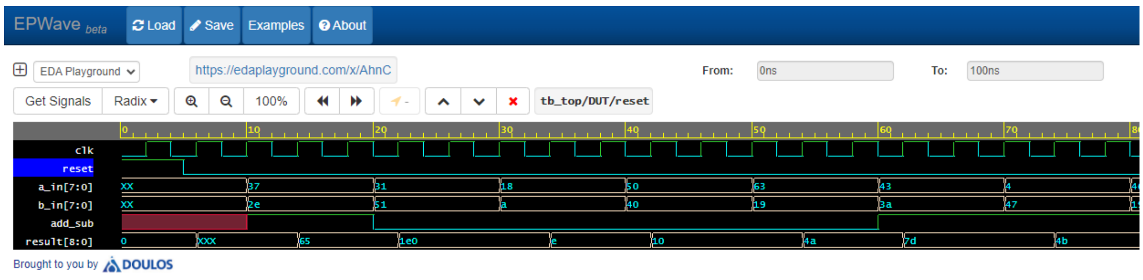


Figure 10. *Waveform Viewer EPWave*

Signals can be added to the waveform using the "Get Signals" button, as shown in *Figure 10*, and then appending the signals one after another. The Radix is only limited to Binary and Hexadecimal and can be selected through the use of the "Radix" button. Users can zoom in or out, move forward or backward with the help of buttons at the top of the screen. This allows for easier debug of the design. The signal path is visible in a window at the side of the buttons. It is also possible to share wave through EPWave.

While EPWave offers a convenient solution for viewing webform, it does have certain limitations in terms of its interface. The current interface lacks the functionality to enable simultaneous selection of multiple signals for the purpose of dragging and dropping them for alignments. In order to align the signals, it was necessary to first select the appropriate option, followed by using of the up and down buttons. In the event of deletions, they had to be performed on an individual basis. One additional constraint of EPWave relates to the absence of flexibility in selecting the decimal Radix.

5.6 USE CASE 1: Design and Simulation With VHDL

The usability of EDA Playground was started with a simple VHDL adder design. In the design pane, a simple VHDL code for adder was written and in the testbench pane, code for the testbench was written.

For this specific design, selection of library was not required. Then the top entity name was specified, which in this case was the testbench entity name. For this run, all of the simulators were used individually, and they provided more or less the same output. As mentioned earlier, EPWave opened automatically, and the waveform was analysed for the correctness of the design.

The Mentor Precision 2021.1 and Aldec SyntHESer 2022.05. were then used for synthesis. Both required specific files with .do extensions that were customized for the required synthesis tools. Along with the gate level netlists, Aldec SyntHESer 2022.05 was able to produce critical route diagrams. It was clear that this particular architecture required 2 LUTs, 5 wires, 5 wire bits, and 2 cells. Mentor Precision 2021.1, on the other hand, was able to create the netlist. Due to the compact size of the design, there were few differences and similar device usage. Yosys and VTR does not support VHDL, so these steps were not proceeded with.

Next a 4 bit 4-to-1 MUX was designed, and the similar steps were followed.

Figure 11 shows the correctness of the design behaviour of adder using EDA Playground.

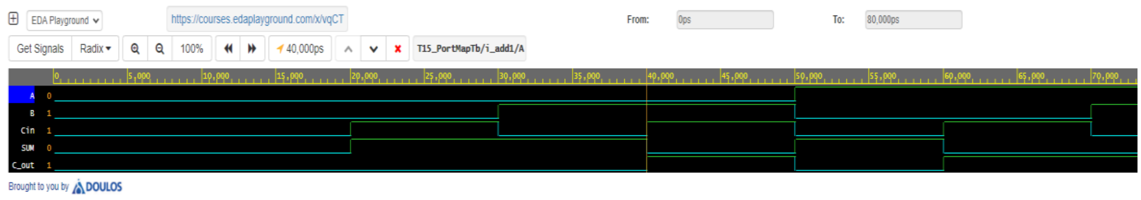
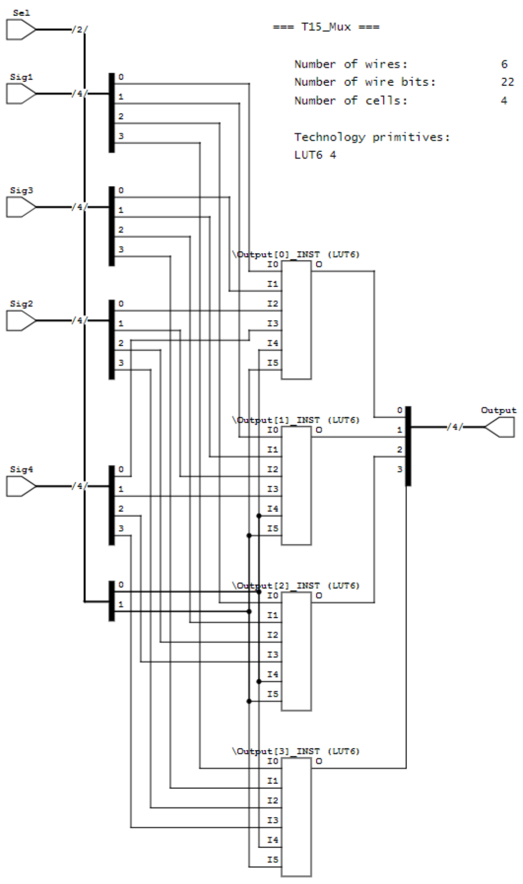


Figure 11. *Waveform for 4to1 MUX*

Figure 12 shows the critical route diagram for the 4to1 MUX. As per the report generated, it was utilizing 6 wires, 22 wire bits and 4 cells. The number of technology primitive was 4 LUTs.

The codes for the design and testbench of 4-to-1 MUX, and the simplified netlists are recorded in the **Appendix A**.



```
=== T15_Mux ===

Number of wires:          6
Number of wire bits:     22
Number of cells:          4

Technology primitives:
LUT6 4
```

Figure 12. *Routing Diagram for 4to1 MUX generated by Aldec SyntHESer*

5.7 USE CASE 2: UVM Verification of SV Design

For the next case, an adder/subtractor was designed using SystemVerilog. The main purpose of this experiment was to try and verify this design using UVM. For this reason, Design Language was selected to SystemVerilog from the left-hand side pane. UVM 1.2 was selected as verification methodology.

The architecture of this use case replicates in *figure 13*.

The design of the test environment started with declaring the interface which will aid in communicating to the DUT and the test environment components.

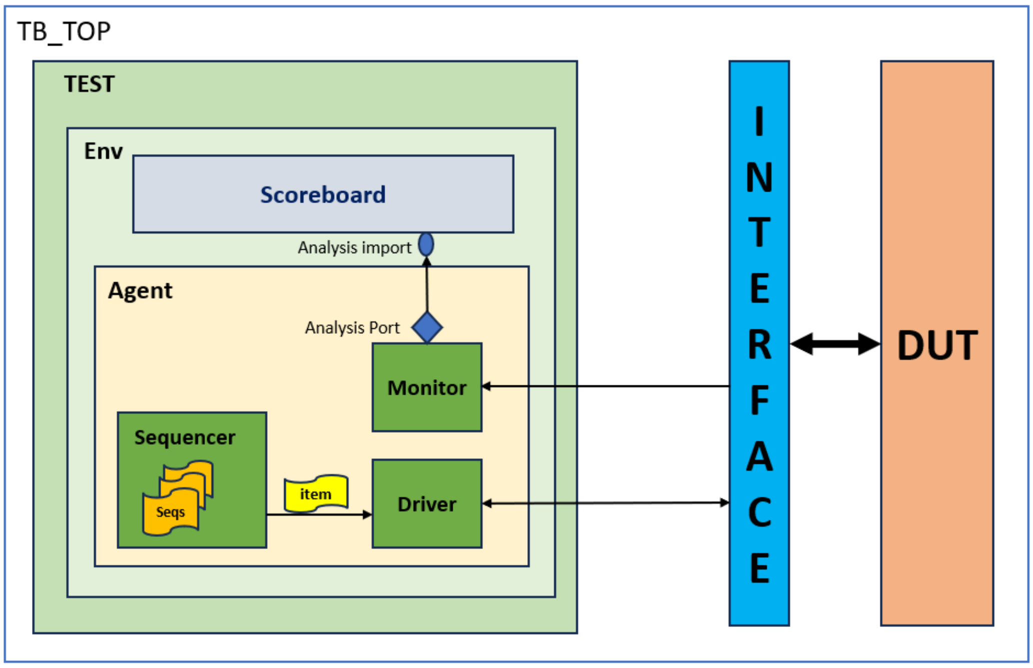


Figure 13. *UVM architecture for the use case 2*

Then, a sequence_item, extending from uvm_sequence_item was defined. Sequence_item defines the transaction object which the driver needs to drive. These objects were declared as random variables. Also, the objects were constrained and printed from the sequence_item. These sequence_item transaction objects act as the fundamental entity within the environment, facilitating the initiation of new transactions and the capture of transactions from the DUT. After that, a base_sequence was designed which was extending from a parameterized uvm_sequence. The parameter for uvm_sequence was the sequence_item. After that, a basic sequencer was designed extending from the uvm_sequencer. The sequence generates stimuli and transmits them to the driver through a sequencer, whereas the sequencer acts as a connector between the sequence and the driver.

After the sequencer was placed, a driver class extending from the uvm_driver was created. In the build phase of this class, the interface was connected to the driver through the configuration database. The example code is as follows in program 5:

```
if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
    `uvm_fatal(get_type_name(), "Not set at top level");
```

Program 6. *Getting the virtual interface through configuration database*

In the run phase, the driver requested to get the sequence_item objects through the sequence_item_port and then ran it to the virtual interface to the DUT. The example code is as follows:

```
// Driver to the DUT
seq_item_port.get_next_item(req);
vif.input1 <= req.input1;
vif.input 2 <= req.input2;
vif.add_sub <= req.add_sub;
seq_item_port.item_done();
```

Program 7. *Getting sequence item objects and driving the DUT through virtual interface*

After the driver class, a monitor class was created. At the beginning of the monitor class an uvm_analysis_port was defined. The monitor was connected to the virtual interface through the configuration database, similarly as it was shown in program 5. But now the virtual interface objects are passing the data to the monitor sequence items. The monitor items are then written into the analysis port. The example code is as follows:

```
class add_sub_monitor extends uvm_monitor;
//define analysis port
   uvm_analysis_port #(add_sub_seq_item) item_collect_port;
   .
   .
   .
   //inside run phase
   //getting sequence item objects
   mon_item.add_sub = vif.add_sub;
   mon_item.ip1 = vif.ip1;
   mon_item.ip2 = vif.ip2;
   @(posedge vif.clk);
   mon_item.out = vif.out;
   //writing analysis port
   item_collect_port.write(mon_item);
endclass
```

Program 8. *Getting virtual interface objects and writing them to analysis port*

The writing of the sequence objects was necessary so that the scoreboard can fetch it through the analysis import. Next, the scoreboard class was created, and the analysis import was defined. The sequence items obtained through the analysis import was stored in a queue. In the run phase of the scoreboard, a reference function to the DUT was created and using the objects from the sequence item, the expected value and the actual value was compared. In case of a mismatch, a UVM_ERROR was generated.

Before the scoreboard, an agent class was created so that the driver, monitor and sequencer instances can be hold in a container. In the connect phase of the agent stage, the connection of the driver and the sequencer was made through seq_item_port of driver instance and seq_item_import of sequencer instance.

As the scoreboard and agent was in place, it was necessary to connect the monitor analysis port to scoreboard analysis export, which was yet to be made. Thus, the env class was created and in the connect phase of env class, the connection was made. Program 8 shows the driver-sequencer and monitor-scoreboard connections.

```
//Example of connection between driver and sequencer in agent class
drv.seq_item_port.connect(seqr.seq_item_export);
//Example of connection between monitor and scoreboard in env class
agt.mon.item_collect_port.connect(sb.item_collect_export);
```

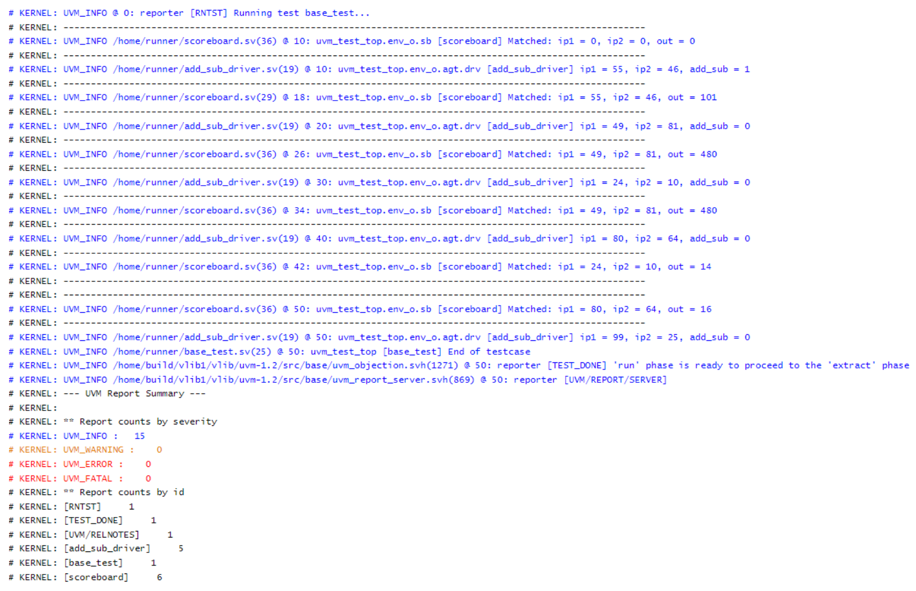
Program 9. *Example of TLM analysis port/export*

Then a base test was created which would start the sequence from its run_phase. It was also possible to control the number of transactions we needed to verify the DUT.

Lastly, the testbench top file was written. It included the uvm_macros, and all the classes mentioned before. In this module, the clock and reset were initialized. The DUT was connected to the interface and configuration database was set, as shown in program 10. Once all the classes were in place, the base test was called to run. Lastly, the variables were dumped for the waves to be viewed. Then using all the simulators, one by one, the test was allowed to run. The logs and waveforms showed that the DUT and the test was working as expected.

```
// set interface in config_db
uvm_config_db#(virtual add_if)::set(uvm_root::get(), "*", "vif", vif);
```

Program 10. *Setting the virtual interface through configuration database*



```
# KERNEL: UVM_INFO @ 0: reporter [RNTST] Running test base_test...
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(36) @ 10: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 0, ip2 = 0, out = 0
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/add_sub_driver.sv(19) @ 10: uvm_test_top.env_o.agt.drv [add_sub_driver] ip1 = 55, ip2 = 46, add_sub = 1
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(29) @ 18: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 55, ip2 = 46, out = 101
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/add_sub_driver.sv(19) @ 20: uvm_test_top.env_o.agt.drv [add_sub_driver] ip1 = 49, ip2 = 81, add_sub = 0
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(36) @ 26: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 49, ip2 = 81, out = 480
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/add_sub_driver.sv(19) @ 30: uvm_test_top.env_o.agt.drv [add_sub_driver] ip1 = 24, ip2 = 10, add_sub = 0
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(36) @ 34: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 49, ip2 = 81, out = 480
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/add_sub_driver.sv(19) @ 40: uvm_test_top.env_o.agt.drv [add_sub_driver] ip1 = 80, ip2 = 64, add_sub = 0
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(36) @ 42: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 24, ip2 = 10, out = 14
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(36) @ 50: uvm_test_top.env_o.sb [scoreboard] Matched: ip1 = 80, ip2 = 64, out = 16
# KERNEL: -------------------------------------------------------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/add_sub_driver.sv(19) @ 50: uvm_test_top.env_o.agt.drv [add_sub_driver] ip1 = 99, ip2 = 25, add_sub = 0
# KERNEL: UVM_INFO /home/runner/base_test.sv(25) @ 50: uvm_test_top [base_test] End of testcase
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_objection.svh(1271) @ 50: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(869) @ 50: reporter [UVM/REPORT/SERVER]
# KERNEL: --- UVM Report Summary ---
# KERNEL:
# KERNEL: ** Report counts by severity
# KERNEL: UVM_INFO :   15
# KERNEL: UVM_WARNING :    0
# KERNEL: UVM_ERROR :    0
# KERNEL: UVM_FATAL :    0
# KERNEL: ** Report counts by id
# KERNEL: [RNTST]    1
# KERNEL: [TEST_DONE]    1
# KERNEL: [UVM/RELNOTES]    1
# KERNEL: [add_sub_driver]    5
# KERNEL: [base_test]    1
# KERNEL: [scoreboard]    6
```

Figure 14. *Log from the UVM test run.*

Finally, for the use this use case, all the synthesis tools were used to get simplified netlist. A Yosys diagram after the synthesis is shown in *figure 15*. Yosys is a tool designed for the purpose of conducting logical synthesis and generating a netlist. It uses ABC: A System for Sequential Synthesis and Verification for the purpose of synthesizing a sample cell library. This tool shows the simplified implementation diagram after the synthesis. Yosys in EDA Playground exclusively processes code within the Design pane. The code contained within the Testbench pane will not be executed. The selections of UVM/OVM methodology and libraries are also disregarded.

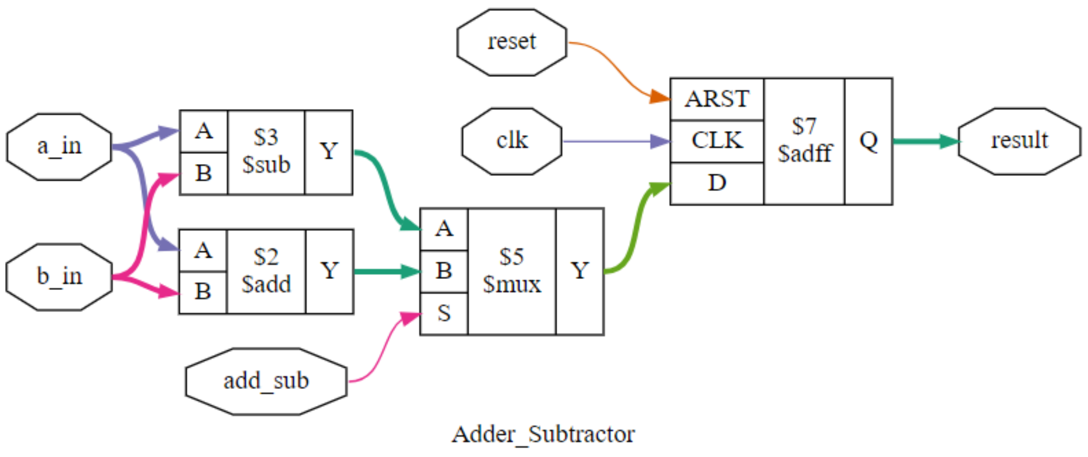


Figure 15. *Yosys simplified implementation diagram after synthesis for the adder subtractor*

5.8 Other Features

Apart from the features mentioned above, EDA Playground allow users to save and share their code in their respective accounts. If the code is shared as published, it will be visible in the example tab where users can access all the examples available. Users can also copy a published code in their own workspace and edit if is necessary. For a public code, anyone with the link can open the code.

Also, there is an EDA Playground community where users can ask questions, collaborate, and provide solutions.

6. CONCLUSION

Designing a digital system is complex and involves multiple stages. Among these stages, modelling, designing, and verifying a chip takes about half the time of the entire process. On the other hand, verification cannot keep up with the design growth, and design cannot keep up with the technology growth due to lack of competent manpower in the field of digital system design.

The absence of proficient personnel in this domain can be attributed to two primary factors: the accessibility of open-source software programming tools and the absence of open-source EDA tools. Students worldwide who lack access to EDA tools via their educational institutions have the opportunity to independently acquire knowledge in programming languages such as C++, Python, Java, or other similar platforms that are readily accessible. EDA Playground offers a solution in this regard.

The objective of the thesis was to showcase the open-source EDA platform, EDA Playground. EDA Playground provides a wide range of languages, tools and resources that can be used for the digital design and aspire digital engineers.

As it was seen in the previous chapter, EDA Playground allows users to code in multiple HDLs, allow them to test the design using multiple commercial and free simulators, and helps them debug the failures in design using EPwave and logs. Users can synthesize their codes using commercial synthesizers to check for device utilization and circuit diagram.

Another significant application of EDA Playground involves the integration of UVM 1.2. Prospective users can acquire knowledge about this methodology through educational resources, including tools and examples that are made available for their benefit. The ability for users to copy a sample workspace into their own workspace significantly facilitates the process of modifying and comprehending specific sections of the code, thereby promoting critical thinking.

The thesis presents a comprehensive range of use cases that demonstrate the diverse functionalities offered by EDA Playground. Throughout the course of this thesis, the various resources available on EDA Playground were utilized and assessed for their functionality. Despite certain limitations, the platform is a powerful tool designed to enhance the capabilities of digital system engineers.

REFERENCES

[1] S. Ramachandran, *Digital VLSI systems design: a design manual for implementation of projects on FPGAs and ASICs using Verilog*. Springer Science & Business Media, 2007.

[2] S. Bhunia and M. Tehranipoor, “Chapter 2 - A Quick Overview of Electronic Hardware,” in *Hardware Security*, S. Bhunia and M. Tehranipoor, Eds., Morgan Kaufmann, 2019, pp. 23–45. doi: 10.1016/B978-0-12-812477-2.00007-1.

[3] H. D. Foster, “Trends in functional verification: a 2014 industry study,” *Proceedings of the 52nd Annual Design Automation Conference*. Association for Computing Machinery, San Francisco, California, p. Article 48, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2744921

[4] D. Rittman, “Nanometer Physical Verification,” Jul. 2011.

[5] A. Anzaldua Jr., “Why Physical Verification Is Only Getting Tougher With Advanced Nodes,” Apr. 14, 2021. https://www.allaboutcircuits.com/news/why-physical-verification-is-only-getting-tougher-with-advanced-nodes/ (accessed Aug. 21, 2023).

[6] D. Abdulkareem, “Introductory Concepts: Fundamentals of Digital Systems.” College of Engineering/ Dept. of Computer Engineering, University of Baghdad. Accessed: Aug. 08, 2023. [Online]. Available: https://coeng.uobaghdad.edu.iq/wp-content/uploads/sites/3/2021/09/

[7] R. Oshana, “Chapter 2 - Overview of Real-time and Embedded Systems,” in *DSP for Embedded and Real-Time Systems*, R. Oshana, Ed., Oxford: Newnes, 2012, pp. 15–27. doi: 10.1016/B978-0-12-386535-9.00002-0.

[8] “A Beginner’s Guide to Digital Signal Processing (DSP).” https://www.analog.com/en/design-center/landing-pages/001/beginners-guide-to-dsp.html (accessed Aug. 08, 2023).

[9] G. Martin and H. Chang, “System-on-Chip design,” in *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*, Oct. 2001, pp. 12–17. doi: 10.1109/ICASIC.2001.982487.

[10] M. J. Flynn and W. Luk, *Computer system design: system-on-chip*. John Wiley & Sons, 2011.

[11] C. M. Maxfield, “Chapter 2 - FPGA Architectures,” in *FPGAs: Instant Access*, C. M. Maxfield, Ed., Burlington: Newnes, 2008, pp. 13–48. doi: 10.1016/B978-0-7506-8974-8.00002-8.

[12] M. Ferdjallah, *Introduction to digital systems : modeling, synthesis, and simulation using VHDL*, 1st edition. Hoboken, New Jersey: Wiley, 2011.

[13] C. M. Maxfield, “Chapter 17 - Application-Specific Integrated Circuits (ASICs),” in *Bebop to the Boolean Boogie (Third Edition)*, C. M. Maxfield, Ed., Boston: Newnes, 2009, pp. 235–249. doi: 10.1016/B978-1-85617-507-4.00017-6.

[14] S. Lammi, “How to become a System-on-Chip design expert?,” Aug. 30, 2022. https://blogs.tuni.fi/cs/teaching/how-to-become-a-system-on-chip-design-expert/

[15] Harry. J.M. Veendrick, *Nanometer CMOS ICs From Basics to ASICs*, 2nd ed. 2017. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-47597-4.

[16] M. Horn, H. van der Schoot, G. Allan, and M. Peryer, “Universal Verification Methodology UVM Cookbook.” Siemens Digital Industries Software. [Online]. Available: https://verificationacademy.com/cookbook

[17] Samir Palnitkar, *Verilog® HDL: A Guide to Digital Design and Synthesis, Second Edition*. Pearson, 2003.

[18] J. Cavanagh, *Verilog HDL design examples*, First edition. Boca Raton, FL: CRC Press, 2017. doi: 10.1201/b22315.

[19] H. Kaeslin, *Top-down digital VLSI design : from architectures to gate-level circuits and FPGAS*, 1st edition. Place of publication not identified: Morgan Kaufmann is an imprint of Elsevier, 2015.

[20] “IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,” *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[21] J. Bergeron, *Writing testbenches using SystemVerilog*. Springer Science & Business Media, 2007.

[22] L.-T. Wang, Y.-W. Chang, and K.-T. (Tim) Cheng, Eds., “The Morgan Kaufmann Series in Systems on Silicon,” in *Electronic Design Automation*, Boston: Morgan Kaufmann, 2009, p. iii. doi: 10.1016/B978-0-12-374364-0.50001-1.

[23] “Electronic Design Automation (EDA).” https://www.cadence.com/en_US/home/explore/what-is-electronic-design-automation.html (accessed Jul. 09, 2023).

[24] “What is EDA (Electronic Design Automation)?” https://www.synopsys.com/glossary/what-is-electronic-design-automation.html

[25] A. Oinonen, “Implementation of SystemVerilog and UVM Training,” 2017.

[26] C. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 2. Aufl. New York, NY: Springer-Verlag, 2008.

[27] “Universal Verification Methodology (UVM) 1.2 User’s Guide.” Accellera Systems Initiative (Accellera), Oct. 08, 2015. [Online]. Available: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

APPENDIX A: DESIGN AND TESTBENCH CODES FOR 4-TO-1 MUX IN VHDL

```
//Design of 4 bit 4-to-1 MUX
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity T15_Mux is
port(
    -- Inputs
    Sig1 : in unsigned(3 downto 0);
    Sig2 : in unsigned(3 downto 0);
    Sig3 : in unsigned(3 downto 0);
    Sig4 : in unsigned(3 downto 0);

    Sel  : in unsigned(1 downto 0);

    -- Outputs
    Output : out unsigned(3 downto 0));
end entity;

architecture rtl of T15_Mux is
begin
  process(Sel, Sig1, Sig2, Sig3, Sig4) is
  begin
   case Sel is
     when "00" =>
       Output <= Sig1;
     when "01" =>
       Output <= Sig2;
     when "10" =>
       Output <= Sig3;
     when "11" =>
       Output <= Sig4;
     when others => -- 'U', 'X', '-', etc.
       Output <= (others => 'X');
    case;
  end process;
end architecture;
```

Program 11. *Design of 4 bit 4-to-1 MUX*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity T15_PortMapTb is
end entity;

architecture sim of T15_PortMapTb is
    signal Sig1 : unsigned(7 downto 0) := x"AA";
    signal Sig2 : unsigned(7 downto 0) := x"BB";
    signal Sig3 : unsigned(7 downto 0) := x"CC";
    signal Sig4 : unsigned(7 downto 0) := x"DD";

    signal Sel : unsigned(1 downto 0) := (others => '0');

    signal Output : unsigned(7 downto 0);

  begin

    -- An instance of T15_Mux with architecture rtl
    i_Mux1 : entity work.T15_Mux(rtl) port map(
        Sel    => Sel,
        Sig1   => Sig1,
        Sig2   => Sig2,
        Sig3   => Sig3,
        Sig4   => Sig4,
        Output => Output);

    -- Testbench process
    process is
    begin
        wait for 10 ns;
        Sel <= Sel + 1;
        wait for 10 ns;
        Sel <= Sel + 1;
        wait for 10 ns;
        Sel <= Sel + 1;
        wait for 10 ns;
        Sel <= Sel + 1;
        wait for 10 ns;
        Sel <= "UU";
        wait;
    end process;
end architecture;
```

Program 12. *Testbench for simulation of 4 bit 4-to-1 MUX*

```
// Verilog description for cell T15_Mux,
// Tue Aug 22 04:20:34 2023
//
// Precision RTL Synthesis, 64-bit 2021.2.0.8//


module T15_Mux ( Sig1, Sig2, Sig3, Sig4, Sel, Output ) ;

   input [7:0]Sig1 ;
   input [7:0]Sig2 ;
   input [7:0]Sig3 ;
   input [7:0]Sig4 ;
   input [1:0]Sel ;
   output [7:0]Output ;

   wire [7:0]Sig1_int;
   wire [7:0]Sig2_int;
   wire [7:0]Sig3_int;
   wire [7:0]Sig4_int;
   wire [1:0]Sel_int;
   wire  nx5780z1,  nx6777z1,  nx7774z1,  nx8771z1,  nx9768z1,  nx10765z1,
nx11762z1,
   nx12759z1;



   OBUF \Output_obuf(0) (.O (Output[0]), .I (nx12759z1)) ;
   OBUF \Output_obuf(1) (.O (Output[1]), .I (nx11762z1)) ;
   OBUF \Output_obuf(2) (.O (Output[2]), .I (nx10765z1)) ;
   OBUF \Output_obuf(3) (.O (Output[3]), .I (nx9768z1)) ;
   OBUF \Output_obuf(4) (.O (Output[4]), .I (nx8771z1)) ;
   OBUF \Output_obuf(5) (.O (Output[5]), .I (nx7774z1)) ;
   OBUF \Output_obuf(6) (.O (Output[6]), .I (nx6777z1)) ;
   OBUF \Output_obuf(7) (.O (Output[7]), .I (nx5780z1)) ;
   IBUF \Sel_ibuf(0) (.O (Sel_int[0]), .I (Sel[0])) ;
   IBUF \Sel_ibuf(1) (.O (Sel_int[1]), .I (Sel[1])) ;
   IBUF \Sig4_ibuf(0) (.O (Sig4_int[0]), .I (Sig4[0])) ;
   IBUF \Sig4_ibuf(1) (.O (Sig4_int[1]), .I (Sig4[1])) ;
   IBUF \Sig4_ibuf(2) (.O (Sig4_int[2]), .I (Sig4[2])) ;
   IBUF \Sig4_ibuf(3) (.O (Sig4_int[3]), .I (Sig4[3])) ;
   IBUF \Sig4_ibuf(4) (.O (Sig4_int[4]), .I (Sig4[4])) ;
   IBUF \Sig4_ibuf(5) (.O (Sig4_int[5]), .I (Sig4[5])) ;
   IBUF \Sig4_ibuf(6) (.O (Sig4_int[6]), .I (Sig4[6])) ;
   IBUF \Sig4_ibuf(7) (.O (Sig4_int[7]), .I (Sig4[7])) ;
   IBUF \Sig3_ibuf(0) (.O (Sig3_int[0]), .I (Sig3[0])) ;
   IBUF \Sig3_ibuf(1) (.O (Sig3_int[1]), .I (Sig3[1])) ;
   IBUF \Sig3_ibuf(2) (.O (Sig3_int[2]), .I (Sig3[2])) ;
   IBUF \Sig3_ibuf(3) (.O (Sig3_int[3]), .I (Sig3[3])) ;
   IBUF \Sig3_ibuf(4) (.O (Sig3_int[4]), .I (Sig3[4])) ;
   IBUF \Sig3_ibuf(5) (.O (Sig3_int[5]), .I (Sig3[5])) ;
   IBUF \Sig3_ibuf(6) (.O (Sig3_int[6]), .I (Sig3[6])) ;
   IBUF \Sig3_ibuf(7) (.O (Sig3_int[7]), .I (Sig3[7])) ;
   IBUF \Sig2_ibuf(0) (.O (Sig2_int[0]), .I (Sig2[0])) ;
   IBUF \Sig2_ibuf(1) (.O (Sig2_int[1]), .I (Sig2[1])) ;
   IBUF \Sig2_ibuf(2) (.O (Sig2_int[2]), .I (Sig2[2])) ;
   IBUF \Sig2_ibuf(3) (.O (Sig2_int[3]), .I (Sig2[3])) ;
   IBUF \Sig2_ibuf(4) (.O (Sig2_int[4]), .I (Sig2[4])) ;
   IBUF \Sig2_ibuf(5) (.O (Sig2_int[5]), .I (Sig2[5])) ;
   IBUF \Sig2_ibuf(6) (.O (Sig2_int[6]), .I (Sig2[6])) ;
```

```
  IBUF \Sig2_ibuf(7) (.O (Sig2_int[7]), .I (Sig2[7])) ;
  IBUF \Sig1_ibuf(0) (.O (Sig1_int[0]), .I (Sig1[0])) ;
  IBUF \Sig1_ibuf(1) (.O (Sig1_int[1]), .I (Sig1[1])) ;
  IBUF \Sig1_ibuf(2) (.O (Sig1_int[2]), .I (Sig1[2])) ;
  IBUF \Sig1_ibuf(3) (.O (Sig1_int[3]), .I (Sig1[3])) ;
  IBUF \Sig1_ibuf(4) (.O (Sig1_int[4]), .I (Sig1[4])) ;
  IBUF \Sig1_ibuf(5) (.O (Sig1_int[5]), .I (Sig1[5])) ;
  IBUF \Sig1_ibuf(6) (.O (Sig1_int[6]), .I (Sig1[6])) ;
  IBUF \Sig1_ibuf(7) (.O (Sig1_int[7]), .I (Sig1[7])) ;
  LUT6 ix5780z45004 (.O (nx5780z1), .I0 (Sig1_int[7]), .I1 (Sig2_int[7]),
.I2 (
  Sig3_int[7]), .I3 (Sig4_int[7]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix5780z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix6777z45004 (.O (nx6777z1), .I0 (Sig1_int[6]), .I1 (Sig2_int[6]),
.I2 (
  Sig3_int[6]), .I3 (Sig4_int[6]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix6777z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix7774z45004 (.O (nx7774z1), .I0 (Sig1_int[5]), .I1 (Sig2_int[5]),
.I2 (
  Sig3_int[5]), .I3 (Sig4_int[5]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix7774z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix8771z45004 (.O (nx8771z1), .I0 (Sig1_int[4]), .I1 (Sig2_int[4]),
.I2 (
  Sig3_int[4]), .I3 (Sig4_int[4]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix8771z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix9768z45004 (.O (nx9768z1), .I0 (Sig1_int[3]), .I1 (Sig2_int[3]),
.I2 (
  Sig3_int[3]), .I3 (Sig4_int[3]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix9768z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix10765z45004 (.O (nx10765z1), .I0 (Sig1_int[2]), .I1 (Sig2_int[2]),
.I2 (
  Sig3_int[2]), .I3 (Sig4_int[2]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix10765z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix11762z45004 (.O (nx11762z1), .I0 (Sig1_int[1]), .I1 (Sig2_int[1]),
.I2 (
  Sig3_int[1]), .I3 (Sig4_int[1]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix11762z45004.INIT = 64'hFF00CCCCF0F0AAAA;
  LUT6 ix12759z45004 (.O (nx12759z1), .I0 (Sig1_int[0]), .I1 (Sig2_int[0]),
.I2 (
  Sig3_int[0]), .I3 (Sig4_int[0]), .I4 (Sel_int[1]), .I5 (Sel_int[0])) ;
  defparam ix12759z45004.INIT = 64'hFF00CCCCF0F0AAAA;
endmodule
```

Program 13. *Simplified Netlist generated through Mentor Precision 2021.1 for 4-to-1 MUX*

APPENDIX B: SV AND UVM IMPLEMENTATION CODES FOR ADDER-SUBTRACTOR

```
// Simple adder/subtractor module
module Adder_Subtractor(
     input           clk,
     input           reset,
     input [7:0]     a_in,
     input [7:0]     b_in,
     // if this is 1, add; else subtract
     input           add_sub,
     output reg [8:0] result
);

  always @ (posedge clk or posedge reset)
    if (reset)
      result <=0;
    else
      begin
        if (add_sub)
          result <= a_in + b_in;
        else
          result <= a_in - b_in;
         end
endmodule
```

Program 14. *SystemVerilog Design of Adder-Subtractor*

```
//-----------------------------------------
// TB_TOP file
//-----------------------------------------
`include "uvm_macros.svh"
package my_pkg;
  import uvm_pkg::*;
  `include "add_sub_seq_item.sv"
  `include "add_sub_base_seq.sv"
  `include "add_sub_sequencer.sv"
  `include "add_sub_driver.sv"
  `include "add_sub_monitor.sv"
  `include "add_sub_agent.sv"
  `include "scoreboard.sv"
  `include "add_sub_env.sv"
  `include "base_test.sv"
endpackage: my_pkg

`include "interface.sv"

module tb_top;
  import uvm_pkg::*;
  import my_pkg::*;
  bit clk;
  bit reset;
  always #2 clk = ~clk;

  initial begin
    reset = 1;
    #5;
    reset = 0;
  end

  add_if vif(clk, reset);

   Adder_Subtractor DUT(
                        .clk(vif.clk),
                        .reset(vif.reset),
                        .a_in(vif.ip1),
                        .b_in(vif.ip2),
                        .add_sub(vif.add_sub),
                        .result(vif.out)
   );

   initial begin
     // set interface in config_db
     uvm_config_db#(virtual add_if)::set(uvm_root::get(), "*", "vif", vif);
   end

  initial begin
    run_test("base_test");
  end

  initial begin
    // Dump waves
    $dumpvars(0, tb_top);
  end
endmodule
```

Program 15. *Top file of the testbench for adder-subtractor*

```
//----------------------------------------
// Interface for the adder/subtractor DUT
//----------------------------------------
interface add_if(input clk, input reset);
  logic [7:0] ip1;
  logic [7:0] ip2;
  logic       add_sub;
  logic [8:0] out;
endinterface: add_if
```

Program 16. *Virtual Interface for Adder-Subtractor*

```
//----------------------------------------
// Sequence Item
//----------------------------------------
class add_sub_seq_item extends uvm_sequence_item;
  rand bit [7:0] ip1, ip2;
  rand bit add_sub;
  bit reset;
  bit [8:0] out;

  function new(string name = "add_sub_seq_item");
    super.new(name);
  endfunction

  `uvm_object_utils_begin(add_sub_seq_item)
```

```
    `uvm_field_int(ip1,UVM_ALL_ON)
    `uvm_field_int(ip2,UVM_ALL_ON)
    `uvm_field_int(add_sub,UVM_ALL_ON)
  `uvm_object_utils_end

  constraint ip_c {ip1 < 100; ip2 < 100;}
endclass
```

Program 17. *Adder Subtractor sequence item*

```
//----------------------------------------
// Sequence
//----------------------------------------
class add_sub_base_seq extends uvm_sequence#(add_sub_seq_item);
  add_sub_seq_item req;
  `uvm_object_utils(add_sub_base_seq)

  function new (string name = "add_sub_base_seq");
    super.new(name);
  endfunction

  task body();
    `uvm_info(get_type_name(), "Base seq: Inside Body", UVM_LOW);
    `uvm_do(req);
  endtask
endclass
```

Program 18. *Adder Subtractor base sequence*

```
//----------------------------------------
// Sequencer
//----------------------------------------
class add_sub_sequencer extends uvm_sequencer#(add_sub_seq_item);
  `uvm_component_utils(add_sub_sequencer)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction
endclass
```

Program 19. *Adder-Subtractor sequencer*

```
//----------------------------------------
// Monitor
//----------------------------------------
class add_sub_monitor extends uvm_monitor;
  virtual add_if vif;
  uvm_analysis_port #(add_sub_seq_item) item_collect_port;
  add_sub_seq_item mon_item;
  `uvm_component_utils(add_sub_monitor)

  function new(string name = "add_sub_monitor", uvm_component parent = null);
    super.new(name, parent);
    item_collect_port = new("item_collect_port", this);
    mon_item = new();
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
      `uvm_fatal(get_type_name(), "Not set at top level");
  endfunction

  task run_phase (uvm_phase phase);
    forever begin
      wait(!vif.reset);
      @(posedge vif.clk);
      mon_item.add_sub = vif.add_sub;
      mon_item.ip1 = vif.ip1;
      mon_item.ip2 = vif.ip2;
      `uvm_info(get_type_name, $sformatf("ip1 = %0d, ip2 = %0d, add_sub
=%0d", mon_item.ip1, mon_item.ip2, mon_item.add_sub), UVM_HIGH);
      @(posedge vif.clk);
      mon_item.out = vif.out;
      item_collect_port.write(mon_item);
    end
  endtask
endclass
```

Program 20. *Adder-Subtractor monitor*

```
//----------------------------------------
// Driver
//----------------------------------------
class add_sub_driver extends uvm_driver#(add_sub_seq_item);
  virtual add_if vif;
  `uvm_component_utils(add_sub_driver)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
      `uvm_fatal(get_type_name(), "Not set at top level");
  endfunction

  task run_phase (uvm_phase phase);
    forever begin
      // Driver to the DUT
      seq_item_port.get_next_item(req);
      `uvm_info(get_type_name, $sformatf("ip1 = %0d, ip2 = %0d, add_sub
=%0d", req.ip1, req.ip2, req.add_sub), UVM_LOW);
      vif.ip1 <= req.ip1;
      vif.ip2 <= req.ip2;
      vif.add_sub <= req.add_sub;
      seq_item_port.item_done();
    end
  endtask
endclass
```

Program 21. *Adder-Subtractor driver*

```
//---------------------------------------
// Agent
//---------------------------------------
class add_sub_agent extends uvm_agent;
  `uvm_component_utils(add_sub_agent)
  add_sub_driver drv;
  add_sub_sequencer seqr;
  add_sub_monitor mon;

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

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

    if(get_is_active == UVM_ACTIVE) begin
      drv = add_sub_driver::type_id::create("drv", this);
      seqr = add_sub_sequencer::type_id::create("seqr", this);
    end
    mon = add_sub_monitor::type_id::create("mon", this);
   endfunction

   function void connect_phase(uvm_phase phase);
    if(get_is_active == UVM_ACTIVE) begin
      drv.seq_item_port.connect(seqr.seq_item_export);
    end
  endfunction
endclass
```

Program 22. *Adder-Subtractor agent*

```
//---------------------------------------
// Scoreboard
//---------------------------------------
class scoreboard extends uvm_scoreboard;
  uvm_analysis_imp #(add_sub_seq_item, scoreboard) item_collect_export;
  add_sub_seq_item item_q[$];
  `uvm_component_utils(scoreboard)

  function new(string name = "scoreboard", uvm_component parent = null);
    super.new(name, parent);
    item_collect_export = new("item_collect_export", this);
  endfunction

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

  function void write(add_sub_seq_item req);
    item_q.push_back(req);
  endfunction

  task run_phase (uvm_phase phase);
    add_sub_seq_item sb_item;
    forever begin
      wait(item_q.size > 0);
      if(item_q.size > 0) begin
        sb_item = item_q.pop_front();
        $display("------------------------------------------------------");
        if(sb_item.add_sub == 1) begin
          if(sb_item.ip1 + sb_item.ip2 == sb_item.out) begin
            `uvm_info(get_type_name, $sformatf("Matched: ip1 = %0d, ip2 = 
%0d, out = %0d", sb_item.ip1, sb_item.ip2, sb_item.out),UVM_MEDIUM);
          end
          else begin
            `uvm_error(get_name, $sformatf("NOT matched: ip1 = %0d, ip2 = 
%0d, out = %0d", sb_item.ip1, sb_item.ip2, sb_item.out));
          end
        end else begin
          if(sb_item.ip1 - sb_item.ip2 == sb_item.out) begin
            `uvm_info(get_type_name, $sformatf("Matched: ip1 = %0d, ip2 = 
%0d, out = %0d", sb_item.ip1, sb_item.ip2, sb_item.out),UVM_MEDIUM);
          end
          else begin
            `uvm_error(get_name, $sformatf("NOT matched: ip1 = %0d, ip2 = 
%0d, out = %0d", sb_item.ip1, sb_item.ip2, sb_item.out));
          end
        end
        $display("------------------------------------------------------");
      end
     end
  endtask

endclass
```

Program 23. *Adder-Subtractor scoreboard*

```
//-------------------------------------
// Environment
//-------------------------------------
class add_sub_env extends uvm_env;
  `uvm_component_utils(add_sub_env)
  add_sub_agent agt;
  scoreboard sb;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = add_sub_agent::type_id::create("agt", this);
    sb = scoreboard::type_id::create("sb", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    agt.mon.item_collect_port.connect(sb.item_collect_export);
  endfunction
endclass
```

Program 24. *Adder-Subtractor environment*

```
//----------------------------------------
// Base Test
//----------------------------------------
class base_test extends uvm_test;
  add_sub_env env_o;
  add_sub_base_seq bseq;
  `uvm_component_utils(base_test)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_o = add_sub_env::type_id::create("env_o", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    bseq = add_sub_base_seq::type_id::create("bseq");

    repeat(10) begin
      #10;
      bseq.start(env_o.agt.seqr);
    end

    phase.drop_objection(this);
    `uvm_info(get_type_name, "End of testcase", UVM_LOW);
  endtask
endclass
```

Program 25. *Adder-Subtractor base test*