POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Develop a UVM Metric Driven Verification Environment for Mixed Signal Simulation

Supervisors

Prof. Maurizio MARTINA

Internship Tutors

Engr. Giacomo CAPPELLIN

Engr. Martino ZERBINI

Candidate

Roman ALI

April 2024

Summary

This thesis discusses the development of a Universal Verification Methodology (UVM) based Metric-Driven Verification (MDV) environment to efficiently and effectively verify Mixed-Signal Designs. The UVM provides a standardized framework for developing portable, reusable, and easy-to-maintain simulation environments. The thesis explains the structure of mixed signal environment, the challenges faced in the simulation of mixed-signal environment and the use of tools such as *Xcelium*, *Spectre*, and *Virtuoso* in the simulation process. The integration of UVM testbenches and MDV techniques with simulation tools is discussed, along with the importance of assertions and auto checks for analog and digital signals. The text also explores the use of modeling techniques such as Real Number Modeling (RNM) to create models for analog signal generators to reduce simulation times.

First part of the thesis involves the setup of a mixed signal simulation environment for Analog Mixed Signal (AMS) designs, which contains both analog and digital components. The Cadence tool *Virtuoso* is used to build the mixed signal simulation environment, which handles all the analog schematics, digital design files, and testbench files. The environment includes separate simulators for analog and digital parts of the design, and the simulation is run according to the mixed signal configurations chosen during the setup in the tool. The passages also discuss the structure of the mixed simulation environment and the settings that can be configured through the GUI tool. These settings include selecting simulator options, specifying connect modules, and including digital design files, libraries, and testbench files in the simulation environment. The steps involved in starting the netlisting and simulation process for the mixed signal simulation environment are also described, including setting up the transient analysis time and selecting interactive or batch mode for simulation. During the netlisting process, a unified netlist of all the analog and digital parts of the design is created, along with

scripts for all the settings and configurations selected. The simulation log window opens once the simulation starts, and the waveforms can be viewed in the *VIVA* waveform window at the end of the simulation.

The next step involves development of a command line-based environment for launching multiple simulations in a mixed simulation environment. The "runams" command is used to extract the unified netlist and simulation scripts required for simulation. The simulation scripts extracted by "runams" contain the settings and configurations chosen in the ADE explorer for the mixed simulation environment, which can be modified for distinct multiple simulations from the command line. The "runSimulation" script present in the netlist folder of all the multiple simulation directories is used to run the simulation by invoking XRUN with the "xrunArgs" arguments. The simulations are run in batch mode without GUI, and the progress and results of the simulations can be viewed in the "xrun.log" and "psf" folder, respectively. This approach streamlines the process and makes it faster to launch multiple simulations without invoking the GUI.

Secondly, the concept of analog assertions in SystemVerilog is discussed, which are used to check the correctness of a design by verifying that certain properties always hold true. We also discussed the two types of assertions, immediate and concurrent, and their applications in mixed-signal designs. We then focused on the complexities of checking the integrity of analog signals in a mixed simulation environment and the features that need to be checked to verify the correctness of the signal. We then discussed the implementation of several assertion checkers for analog signals, including the frequency checker, amplitude checker, high/low time checker, and rise/fall time checker. We also discussed the benefits of using analog assertions, such as providing an automatic way to verify the features of analog signals, being built in modules that can be reused multiple times in the same testbench for signals with similar features. By leveraging these analog assertion checkers, verification engineers can improve the efficiency and effectiveness of the verification process for analog circuits.

Third part of thesis discusses the implementation of signal generators using real number modeling (RNM). Signal generators are electronic devices that produce electrical waveforms at various frequencies and amplitudes. RNM is a mathematical technique used to represent and analyze physical systems using real numbers. The signal generators can

be used to provide input stimulus for analog systems when being verified. The generators include sine wave, triangular wave, and square wave, with various features such as frequency range, amplitude control, duty cycle control, rise/fall time control, and differential output. The generators are controlled by a virtual interface, and the testbench stimulus provided in the test class.

Lastly, the thesis discusses the methodology of metric driven verification (MDV) for verifying the functionality of an AMS design. MDV involves, firstly, defining a verification plan, based on specifications of the device, where all the features that need to be covered by assertions/covergroups are listed. Secondly, MDV involves defining set of metrics to measure the quality and completeness of the verification process, such as code coverage, functional coverage, and performance metrics based on the verification plan. The goal of MDV is to ensure that the design meets the required specifications and reduce the risk of design errors. The merging of metrics is also discussed, which involves combining multiple metrics to gain a more comprehensive results for the desired verification goals. Multiple simulations with distinct configurations are run in parallel to collect these metrics for verifying the DUT using randomized input stimuli. The coverages related to covergroups and analog assertions are discussed, and the multiple metrics from simulation runs are merged to achieve the desired coverages.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Martino Zerbini and Giacomo Cappellin for their invaluable guidance, support, and encouragement throughout my thesis. Their insightful comments, constructive feedback, and unwavering commitment have been instrumental in shaping this thesis. I am also grateful to them for providing me with the opportunity to conduct my thesis at STMicroelectronics.

I am also grateful to my unversity supervisor, Maurizio Martina, for his assistance and support in various aspects of my thesis. His contributions have been invaluable and has helped me to complete this thesis successfully.

I would like to extend my heartfelt thanks to my family for their unwavering support, love, and encouragement. Their constant support and motivation have kept me going during the challenging times.

Finally, I would like to thank all the participants who took part in my thesis and shared their valuable insights and experiences. Without their cooperation, this thesis would not have been possible.

Once again, I express my sincere gratitude to all those who have helped and supported me throughout my thesis journey.

Table of Contents

List of Figures

Listings

Acronyms

MDV
Metric-Driven Verification

UVM
Universal Verification Methodology

SV
SystemVerilog

SVA
SystemVerilog Assertion

PSL
Property Specification Language

RNM
Real-Number Modeling

AMS
Analog and Mixed Signal

GUI
Graphical User Interface

ASIC
Application-Specific Integrated Circuit

DUT
Design Under Test

RTL

Register Transfer Level

Chapter 1

Introduction

1.1 Background

Verification is a critical step in the development of any silicon chip or system, ensuring that it meets its functional requirements and specifications. The process involves testing and verifying the design at various stages of development, from the initial design phase to the final production stage. Silicon chips have increased in complexity over time, and traditional waveform checks have become obsolete. Therefore, there is a need to find new ways to verify these complex silicon chips as fast as possible efficiently and effectively.

The verification of digital systems has been widespread in the silicon industry for some time. It has become efficient and more automated in the industry. The goal of digital verification is to ensure that the design meets the functional and performance requirements specified by the design team. Verification is a time-consuming and resource-intensive process that often accounts for a significant portion of the overall design cycle.

On the other side of the scope, the verification process for analog chips typically involves a combination of simulation and testing. The simulation process uses software tools to model the behavior of the analog circuitry and predict its performance. Simulation tools can be used to verify the functionality of the circuit, analyze its performance under different operating conditions, and optimize the design for performance and power consumption.

The silicon chips in the industry are not always digital-only or analog-only. Many of the complex chips designed nowadays are Mixed Signal Integrated Circuits, which contains both analog and digital components. The design process is complex

and requires expertise in both analog and digital designs, as well as consideration of the interactions between the two types of circuitries. The design must ensure that the analog and digital signals interact but do not interfere with each other and that the circuit performs reliably and accurately.

The verification of analog-only circuits and verification of digital-only circuits is common in industry and there has been a lot of research to make them efficient reliable and fast. When it comes to mixed signal verification, it is a relatively new concept, and not a lot of work is done to find methodologies to verify the functionality of these mixed signal devices efficiently and reliably. The verification is not complete by just testing the analog components separately using analog-only methods and digital components separately by digital only methods. On the scope of whole device, the analog and digital components interact with each other and there should be a verification system which can handle both analog and digital components and the whole device as one unit.

Mixed Signal Environment with Metric-Driven Verification(MDV) provides an automated and systematic approach to verification of these mixed signal designs, allowing designers to quickly identify and fix errors in their designs.

1.2 Motivation and Scope

A Universal Verification Methodology (UVM) based MDV environment for mixed signal is the answer for this complex problem. The UVM methodology is widely used in digital verification and provides a standardized framework for developing verification environments. UVM provides a framework for creating simulation environment that is portable, reusable, and easy to maintain. By extending the UVM methodology to mixed-signal simulation, we aim to provide a more efficient and effective approach to verifying mixed-signal designs.

Mixed signal verification is a complex process, as it involves testing both analog and digital components of the system. Digital components are tested using digital simulation tools. In this project Cadence tools are used so *Xcelium* is being used as digital simulator, while the analog components are tested using analog simulation tools, which is *Spectre* in this project. The interaction between the analog and digital components is also tested using mixed signal simulation tools. This task is performed by another cadence tool Virtuoso. This tool act as a cockpit, handling the schematics, digital design files, simulator settings and simulations. Understanding how these tools run in collaboration is the key to ensure the correct behavior

of analog and digital simulations and their correct interaction with each other.

This mixed signal simulation environment is extensively studied in this thesis. We will discuss, how the UVM testbenches and MDV techniques are integrated with the simulation tools to verify the features and specifications of analog and digital components. The features of the tools used in the simulation will be discussed as well. We will also discuss, how we can use this environment to efficiently run the mixed signal simulation.

MDV involves running the simulation multiple times with different configurations and test classes to verify the mixed signal designs and then collecting the functional and covergroup coverages from these regression runs to ensure the correct behavior of these designs. The simulation environment should be robust modular and customizable to make user select between multiple configurations and testclasses as well as the number of simulations to run on the go. This is done by writing some scripts and integrating them with the UVM testbench and simulation tools which make user eligible to run multiple simulations with different specifications from command line making the concept of MDV realizable.

Assertions and autochecks for analog and digital signals are important components of this project to ensure correct behavior of mixed signal device. These assertions and design autochecks are also integrated in the mixed signal simulation environment and ensure the verification of the analog as well as digital signal in the whole device. Coverages from these assertions or covergroups are collected for multiple simulations using the coverage collector tool, *IMC* for this project. These coverages from multiple simulations are then merged together, using the same tool, to receive a more robust and complete verification of the device.

A mixed signal simulation even with UVM based environment takes significantly more time than a digital simulation. Modelling of the analog blocks in the design is the solution for this problem. Multiple modelling techniques are available to develop different levels of efficiency and accuracy compared to the original analog design. Real Number Modeling (RNM) in SystemVerilog(SV) and Modeling in Verilog-AMS are some examples. The analog blocks in the design can be replaced by the modeled counterparts to significantly reduce the simulation times. Some models for configurable signal generators are developed in RNM to explore this concept in this thesis.

This thesis aims to demonstrate the effectiveness of a UVM-based MDV environment for mixed-signal simulation in improving the quality and efficiency of the verification process incorporating the effectiveness of using modelling techniques for analog blocks in the simulation. By providing a standardized framework and

quantitative approach to verification, we can help designers identify and fix errors in their mixed-signal designs more quickly and accurately, ultimately leading to better quality products.

All the techniques described above are extensively studied in this thesis. The design choices made, the methodologies used, and the results obtained are reported as well.

1.3 Training

The first step involved getting familiar with the framework and the tools required for the project. Mixed signal verification is a complex topic, and a number of courses were taken to gather the fundamental concepts of the topics involved in the thesis. Cadence offers a range of online courses and training programs to help engineers and designers develop the skills they need to use Cadence tools effectively. The project is built on Cadence tools. The initial courses involve getting familiar with SV, UVM and Assertions and Auto checks. Then some courses were taken to get familiar with Mixed signal simulation platforms and MDV techniques. Some courses were also taken to get familiar with modelling techniques mainly RNM. The list of all the course taken during the thesis are listed in the appendix.

Chapter 2

Mixed Signal Simulation Environment

The Analog Mixed signal (AMS) designs are very complex, involving both analog and digital components working together in a single device. Simulation of these mixed signal designs is also very complex as we require separate simulators for analog and digital parts of the design. In this chapter we will discuss how the mixed signal verification environment is set up for simulation. *Virtuoso* is a Cadence GUI based tool which is used to build the analog only or mixed signal simulation environment. This tool is responsible for handling all the schematics and digital design files as well as testbench files. When Simulation is run, the tool invokes the simulators for analog and digital part of the simulation.

Figure 2.1 shows the general structure of the mixed simulation environment. Cadence tool Virtuoso has library, Cell and view configured by the user. Here we add all the analog schematics and libraries to the maestro view along with the testbench tb_top which drives these schematics. In the tool settings we provide the list of all the digital testbench files present. This includes testbench files, assertions files, UVM configuration files etc. The integration of UVM based testbench is not general. There are many mixed simulation environments built without UVM. The goal here is to integrate the digital UVM approach to a mixed signal design. In the digital testbench files we again see the tb_top. This tb_top is same as the one present in maestro view. The tb_top of the testbench is purely digital RTL, and it instantiates also the analog schematics. This way we can access both analog and digital signals from the mixed simulator. Along with digital testbench files we also provide the list of all the digital design files as well.

In the tool, there are a number of settings to configure the mixed simulation,

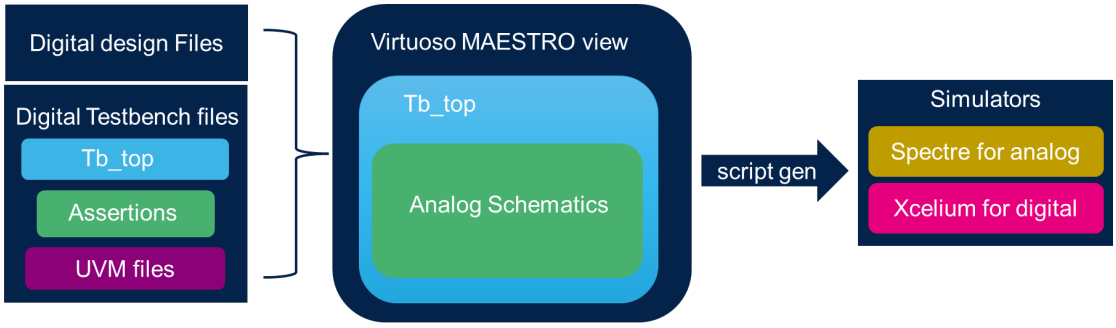

Figure 2.1: Mixed Signal Simulation Environment

which we will discuss shortly. Once the mixed simulation environment is configured, the GUI tool is commanded to start the netlisting and simulation routine. When the simulation is started, firstly, the unified netlist of digital and analog portions of the device is created. The tool also generates the scripts which contain the mixed simulation configurations we selected. All these files are passed to the XRUN command to start the simulation. This XRUN command invokes *Xcelium* and *Spectre*, the simulators for digital and analog parts of the device respectively. The simulation is run as per the mixed signal configurations/settings chosen during the setup in the tool.

The mixed simulation settings in the GUI and its overall structure are explained in detail in the next sections.

2.1 Mixed Simulation through GUI

The Mixed signal simulation for this project is built on Virtuoso tool. The Schematics of the design can be added to the hierarchy editor of the tool. Tool libraries and all the libraries required for the design are also added in the hierarchy editor. Hierarchy Editor consists of a Library which have Cells and these cells can have multiple View setups. These views may contain top level schematics, testbench for schematics or any other sublevel schematics. We can open a view i.e., Maestro view, and an ADE explorer window opens from where we can configure the simulation settings.

In the ADE explorer window, there is a list of all the signals which are chosen to be probed. Signals to be probed can be added or removed in this window. From ADE explorer window if we go to Setup→High Performance Simulation, we open a window where simulator options can be selected as shown in the Figure 2.2. Here we have option to choose between different *Spectre* simulator modes. We can also select multi-threading option and the number of threads to choose. The accuracy

of the simulator can also be selected for the whole design as well as for sub-blocks of the design schematics in the locally scoped section.

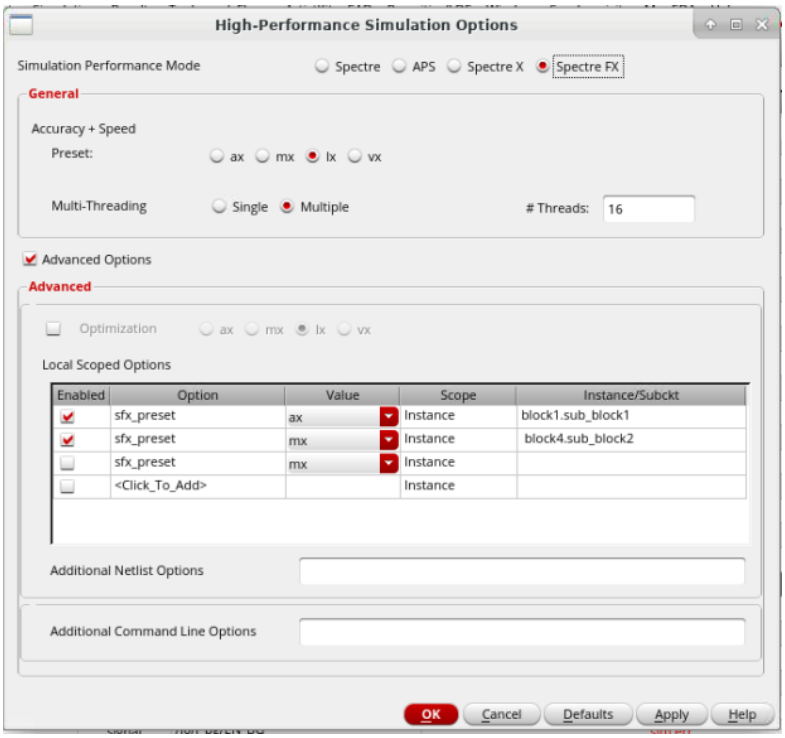

Figure 2.2: AMS Simulator Options (Image is courtesy of Cadence Design Systems)

From the ADE explorer, if we go to Setup→Connect Rules/IE setup, we open a window as shown in Figure 2.3. Here we can list all the instances in our design where digital signals are directly connected to an analog node or vice versa. These connect modules give the AMS simulator knowledge of how it must tackle this problem. Let's consider we have a clock signal in digital domain. It will operate between 1 and 0. But if we connect it directly to an analog node, the simulator must consider that clock signal as an analog signal, it will work the same way, but now it will have a voltage. In the connect modules we can specify the value of voltage it will be allotted when it is converted to an analog signal. This concept can be understood with this example in Figure 2.4. Here the clock is a digital signal but in analog domain. As we have defined the connect module rule for global as 1.8V, the clock is also allotted 1.8V. Specific rule can be defined for a particular area of the project or even a single node. Having this feature in the simulation, we can connect signals from different domains with each other, let it be digital, real, or analog, without worrying about the simulator giving errors.

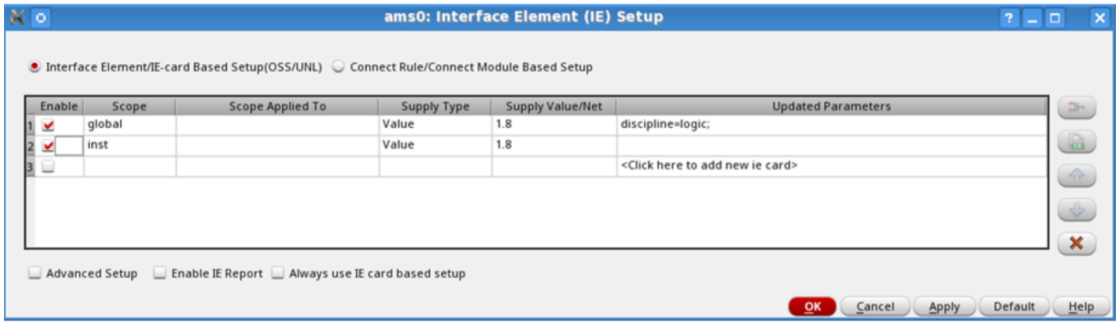

Figure 2.3: Connect Modules/IE Setup (Image is courtesy of Cadence Design Systems)

Figure 2.4: Connect Modules Example

From the ADE explorer, if we go to Simulation→options→AMS_simulator, we open a window as shown in Figure 2.5. Here in the Include Option Settings, we can include list of all the digital design files, digital libraries and all the digital testbench files to the Simulation environment. We can add timescale options in this tab as well which should be concurrent with timescale options in SV files. In the UVM tab we can specify which test class will be used for the simulation.

In the ADE explorer window, we can set up the transient analysis time of the simulation. It is the time of the analog part of the simulation. The digital simulation time is configured in the test class. This transient analysis time is essentially the simulation run time. The digital simulation will run according to the configuration in the test class, but it will stop as well once the transient analysis time finishes even if the digital simulation time is larger than transient analysis time. From the ADE window, going to Simulation→netlist and run options, open the simulation options window where we can select if the simulation should run in interactive mode or batch mode. Here flex matrix mode for simulation can also be selected to activate advance optimization for mixed simulation.

Once all the settings are configured, netlist and simulation can be started from ADE explorer. During the netlisting, the GUI creates a unified netlist of all the analog as well as digital parts of the design. It also creates scripts for all the settings selected and configurations done. All the scripts required for the simulation to

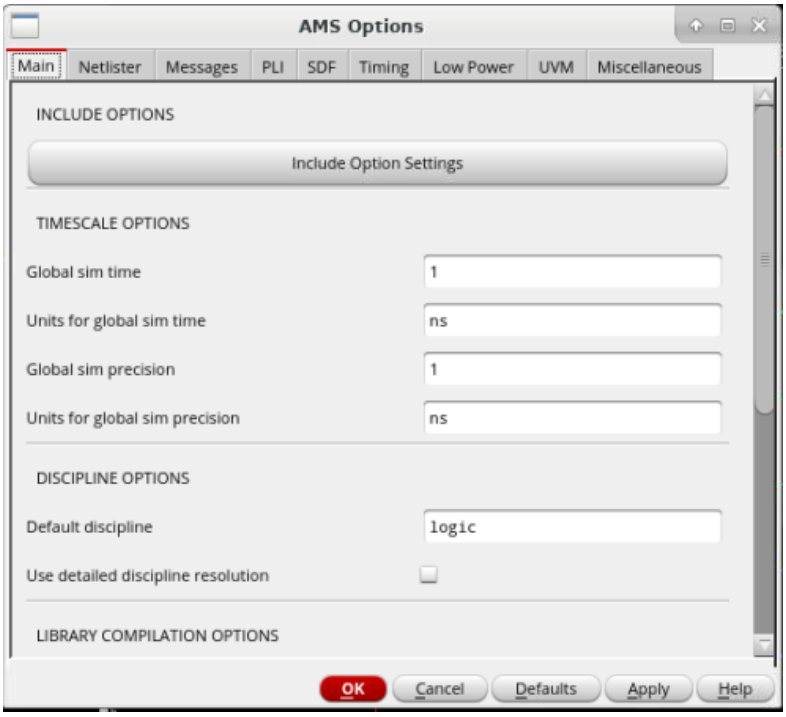

Figure 2.5: Including Digital Files (Image is courtesy of Cadence Design Systems)

run are also created and then Simulation is started. The simulation log window opens once the simulation starts and at the end of the simulation, we can view the waveforms in the VIVA waveform window as well. It has to be noted that the digital and analog design files are compiled when the unified netlist is created. The digital testbench files are only compiled once the simulation starts. This approach, of running the simulation with some pre-compiled RTL files (Digital Design files) and some RTL files (testbench files) which will be compiled after the simulation starts, is quite new in the industry.

To get a detailed understanding of how virtuoso is used to build mixed simulation environment, the cadence course "Mixed-Signal Simulations Using *Spectre* AMS Designer — AVUM" is very beneficial. A short overview of the course is present in Appendix A.5.

2.2 Mixed Simulation through Command Line

We have discussed in the section 2.1, how the mixed simulation environment is built using the ADE. This is not sufficient for the Metric Driven Verification(MDV)

approach we want to use for this project. We want to introduce MDV in this project by running multiple simulations with different configurations and test classes and then collecting coverage and cover group metrics and merging the results. ADE provides the ability to run multiple simulations for analog only projects. But ADE does not provide the ability to run multiple simulations for the project where we have UVM incorporated mixed simulation environment and want to run multiple simulations with different configurations. We want to bypass the ADE explorer (GUI) to run these simulations without GUI mode through command line to streamline the process and make it faster to launch multiple simulations.

We have built a process through which the GUI based mixed simulation environment can be converted to command line-based environment where multiple simulations can be launched at once. For this, we have written some scripts which can extract the unified netlist and simulation scripts. These scripts are normally generated by the tool when we start "netlist and simulation" from ADE explorer. But now we will not invoke ADE and generate these scripts from command line.

Figure 2.6 illustrates how the command line scripts are using the same Virtuoso Lib, cell, and view, used to configure the mixed simulation, to extract the unified netlist and other simulation scripts. After the netlisting and scripts extraction is done, the XRUN can also be invoked directly from command line by calling the runSimulation script, which was generated with all the other simulation files. So essentially, the GUI is not getting invoked. It does not do the netlisting and simulation. All of this is being done from command line without invoking the GUI.

2.2.1 Extracting Simulation Scripts through Command Line

“Runams” command provides support for netlisting and running an AMS simulation from the UNIX command line with the unified netlist-based flow. “Runams” is command-line equivalent to running the “create netlist” or “netlist and run simulation” on ADE GUI. “Runams” can also create netlist standalone from command line as well. When “runams” is invoked, it creates the “netlist.vams” file as well as all the AMS configuration files required for simulation, including “amsControlSpectre.scs”, “Probe.tcl”, “xrunArgs”, “runSimulation” to name a few, inside the netlist directory.

"Runams" script streamlines the netlist and simulation process of mixed environment as it bypasses the GUI. Here "runams" is the command which dumps netlist and simulation scripts from the GUI Maestro View. The "cdslib" and "cdsinit" files are directly linked in the folder where runams script is present. This is done so that

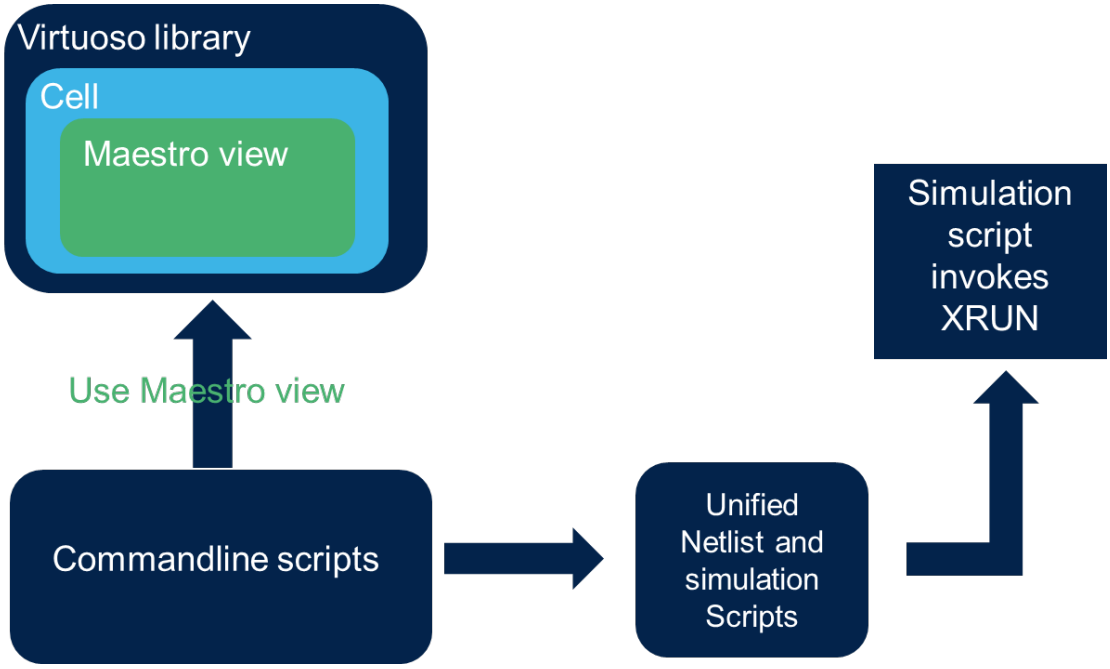

Figure 2.6: Extracting Simulation Scripts from command line

the scripts can be run from anywhere and not necessarily from the directory where mixed simulation environment is present. Furthermore, the GUI library, cell, and view, using which, the netlist and simulation scripts are extracted, are provided as arguments to runams command. Argument *-savescripts* is used to save all the scripts required for AMS simulation, and *-rundir* provides the directory where all the scripts will be stored. We can make this script and the scripts mentioned in the next section configurable and portable by creating a script with variables for these scripts. In this way only the script with variables can be modified without changing anything in the main scripts. Runams command and some of its arguments are mentioned in Listing 2.1:

Listing 2.1: Runams Command and its Arguments

```
#!/bin/sh
runams \
-cdslib ${cdslib} \
-lib ${library} \
-cell  ${cell} \
-view ${view} \
-netlist all \
-savescripts \
-rundir ${netlistdir} \
```

```
10 -log ${netlistdir}/runams.log
11
```

The runams command creates two subdirectories inside the directory mentioned in -rundir argument. These two subdirectories are “netlist” which contains the generated unified netlist and all the scripts for simulation and “psf” which contains the log files and results once the simulation is launch. Inside “netlist” directory there should be “netlist.vams” file which was created by "runams". Along with this file, there are scripts and auxiliary files to run the AMS simulation using XRUN i.e., “xrunArgs” file and “runSimulation” script. In the “psf” directory, there are log files which can be used to see the progress of the simulation if the simulation is not interactive. There are other results which include profiler log, waveforms dump, coverage reports etc. Figure 2.7 explains the directory structure after the netlist and simulation scripts extraction.

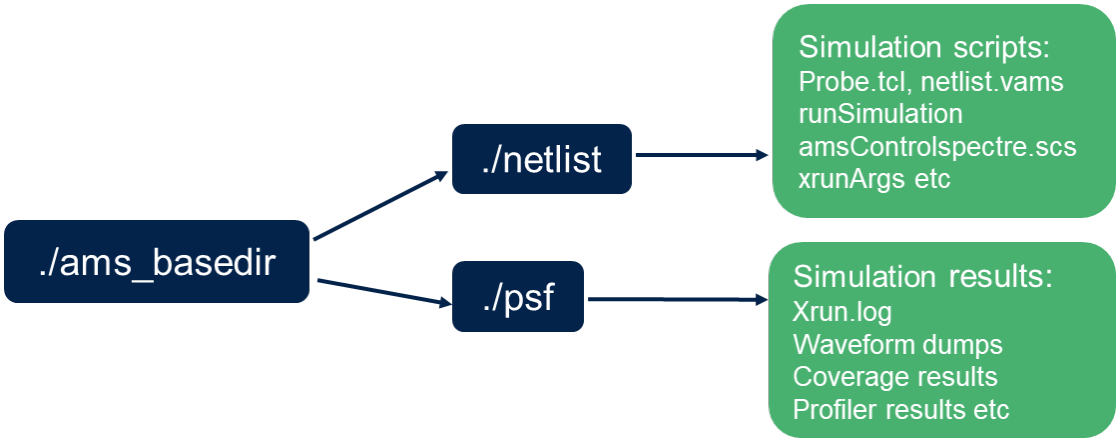

Figure 2.7: Directory Structure of Mixed Simulation

It should be noted that here we are just doing the netlisting part of the simulation using the runams command. Runams can perform the netlisting as well as simulation, essentially everything which can be done from the ADE explorer. But here we are only creating netlist and simulation scripts while not starting the simulation yet. The reason is that we want to make multiple copies of these netlist and simulation scripts, make changes to the scripts and run simulations from these multiple directories simultaneously.

2.2.2 Multiple Simulation Setup

Once we have extracted the unified netlist and simulation scripts from the command line using "runams" and have placed them in the directory mentioned by *-rundir*, we will consider this as parent directory. We can make a configurable number of multiple directories with the same “netlist” and “psf” subdirectories setup. Copy all the files from the netlist folder of the parent directory to the netlist folder of the newly created directories. We can make, in the new directories, soft links (instead of copying) of all the simulation files which we don’t need to change, to save up space in the server. The structure of the newly built directories is same as the structure of the base directory shown in Figure 2.7

Now we must see what is present inside these simulation scripts to understand how to make changes in the scripts for distinct multiple simulations. We have already discussed in the previous section that we are using the same mixed simulation environment which was set up in the ADE explorer. So, the simulation scripts, extracted by runams, contains the settings and configurations we chose and configured in the GUI while setting up the mixed simulation i.e., iecards, Simulator options, probe signals etc. When we ask ADE to netlist and run the simulation, it creates simulation scripts for all the settings and configurations we performed so that when simulation is running the simulator can understand the configurations through these scripts. In this case when the scripts are extracted using runams we still have all these scripts containing simulation settings/configurations we chose.

If we open these scripts, we will be able to see the configurations from the ADE reflected in them. We will look at some of these scripts and see what we can change in them to help our goal of multiple distinct simulations from command line. Listed below are some scripts created from the ADE explorer after configuring it, which is discussed in Section 2.1.

Probe.tcl

Probe.tcl is generated in the netlist folder after the extraction of scripts. In Section 2.1 we discussed that signals can be added in the ADE explorer window for probing and waveform dump. All those signals are present in this probe.tcl file. The simulator looks at this file and probes the corresponding signals before starting the simulation so that waveforms can be created for them. We can change these signals depending on the situation of multiple simulations. If we are targeting different area of the project in these distinct multiple simulations, the signals for only those respective area of the project can be added in probe.tcl and all other signals maybe removed to make the simulations run faster.

Figure 2.8 shows a typical probe.tcl file created after extracting the netlist and simulation scripts from command line. The probed signal can be voltage, current, real, digital or assertion etc. The full hierarchical path from the tb_top to the signal must be provided. In case of current signals -flow must be added before the signal path as well.

```
#
# Database settings
#
if { [info exists ::env(AMS_RESULTS_DIR) ] } { set AMS_RESULTS_DIR $env(AMS_RESULTS_DIR)} else {set AMS_RESULTS_DIR "../psf"}
database -open ams_database -into ${AMS_RESULTS_DIR} -default
#
# Probe settings
#
probe -create -emptyok -database ams_database {tb_top.dut.sig_analog}
probe -create -emptyok -database ams_database {tb_top.dut.I0.dig_clk}
probe -create -emptyok -database ams_database -flow {tb_top.dut.sig_current}
probe -create -emptyok -database ams_database {tb_top.V_ampchecker.range_check.range_check}
probe -create -emptyok -database ams_database {tb_top.clk_freq_checker.freq_check.freq_check}
```

Figure 2.8: Probe.tcl File

amsControlSpectre.scs

amsControlSpectre.scs file contains settings and configurations for *Spectre*(Simulator for analog part of the device). Figure 2.9 show the typical amsControlSpectre file extracted using runams. Here the analog simulation time can be changed mentioned as tran stop=6m, which was originally being configured from the ADE explorer window. We can also provide SimulatorOptions for *Spectre* from this file and set simulator preset for the subsections of the analog parts of the device (fxsimOpt1 etc.). These presets change the accuracy of the simulator for the specific part of the device making the simulation faster or slower.

```
simulator lang=spectre

global 0 vdd! gndsub! vdde3v3! en_ls! v_write_sc! OTP_VPP! OTP_VPP_INT! \
gnd1! vdde! VSSCOREDUMMY! vddefw! KOFF! TRIGGER! VDDECOREDUMMY! KOFF3V3!

simulatorOptions options temp=27 tnom=27 scale=1.0 scalem=1.0 reltol=1e-3 \
vabstol=1e-6 iabstol=1e-12 gmin=1e-12 rforce=1 maxnotes=5 maxwarns=5 \
digits=5 pivrel=1e-3 docheckLimit=no checkLimitDest=both
fxSimOpt1 options sfx_preset=ax inst=[ tb_top.dut.block1.sub_block1 ]
fxSimOpt2 options sfx_preset=mx inst=[ tb_top.dut.block4.sub_block2 ]

tran tran stop=6m write="spectre.ic"
```

Figure 2.9: amsControlSpectre.scs File

xrunArgs

This file contains all the argument provided to XRUN command which starts the simulation. This file has a lot of arguments configuring the mixed simulation. Only some of these arguments are mentioned in the Figure 2.10. We can see the *-profile* option present which enables the profiler, providing details about the Performance of the digital blocks in terms of how much time in percentage they are taking from the overall simulation. We can change the UVM test class for the simulation from here as well. Coverage arguments are also added to the xrun arguments to enable coverage for tb_top.

The files cds_globals, netlist.vams, probe.tcl, amsControlSpectre.scs etc. are added as well to xrun arguments. Argument *-run -exit* means that the simulation will be in batch mode and not interactive. To make the simulation interactive these arguments can be replaced with *-gui*. *Spectre* arguments are also present reflecting the settings we configured as per Figure 2.2. The settings can be changed from here now, instead of opening the window from ADE explorer. There are also many other arguments present in this file; the list of all the digital design and testbench files and the path addresses to look for them, the list of digital and analog libraries to add to the simulation, timescale settings etc.

```
 2 -profile
 3 -coverage all
 4 -covworkdir ../psf/coverage_results/$TNAME/
 5 -covdut tb_top
 6 -top cds_globals
 7 ./netlist.vams
 8 ./ie_card.scs
 9 ./amsControlSpectre.scs
10 -input ./probe.tcl
11 -run -exit
12 +UVM_TESTNAME=test_base_mixed
13 -spectre_args "+fx +spectre +preset=lx +mt=16"
```

Figure 2.10: xrunArgs File

It can be said that if the mixed simulation environment is configured once in GUI and if the unified netlist and simulation scripts are extracted using runams from command line setup, we can then reconfigure the environment directly from these scripts in the command line and completely skip the GUI mode for reconfigurations.

2.2.3 Running the Simulation

After the unified netlist and simulation scripts for mixed simulation are extracted and multiple simulations directories are set up, we can proceed to running the simulation by invoking XRUN. runSimulation is present in the netlist folder of all the multiple simulation directories. This script can be invoked to run the simulation. In this script, XRUN is called with the arguments present in xrunArgs. The figure 2.11 explains this. Here simdir are the multiple directories. A small bash script can go into each directory and invoke runSimulation for all the directories. In case of multiple simulations, they are always run in batch mode without gui.

Once the simulations start running, the progress of the simulations can be viewed in xrun.log from their respective "psf" folder. All the simulation results are also present here including coverage results, profiler results, waveform dumps etc.

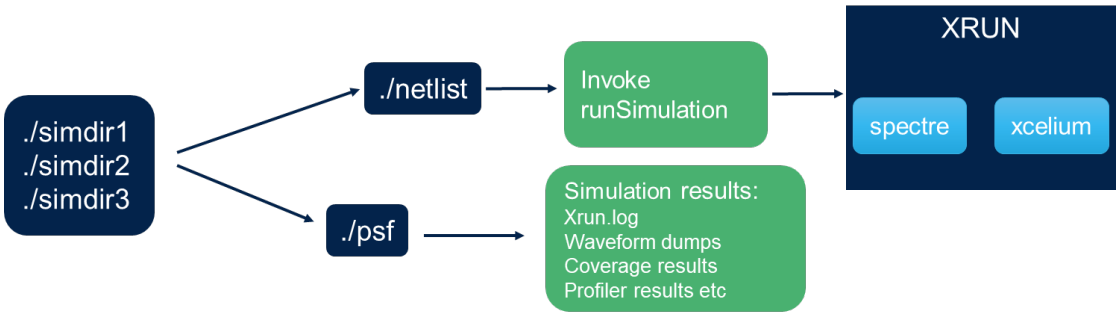

Figure 2.11: Running the Simulation

2.3 Mixed Simulation Setup: The Other Way

The Sections 2.1 and 2.2 explain how Mixed Simulation Environment is set up in GUI, and how we found a way to convert it to run multiple simulations and change the settings and configurations of the mixed signal environment on the go from command line. We had to follow this approach because we had the mixed simulation environment already built on GUI and then decided to transfer it to the command line-based environment. Also, the analog part of the device is present in schematics format and attached to GUI already.

There is a way to build the mixed simulation environment directly from command line and completely bypassing the GUI. As the analog part of the device is present in schematics form, it is not possible to use the approach for this project. The

approach only works if the analog parts of the project are present in spice files. The approach is explained in the course "Command-Line-Based Mixed-Signal Simulations with the Xcelium™ Use Model". A short introduction to the course in present in the Appendix A.6.

Chapter 3

Assertions for Analog and Mixed Signals

Assertions in SystemVerilog are a verification technique used to check the correctness of a design by verifying that certain properties always hold true. Assertions are used in silicon verification to detect errors, bugs, and other design issues early in the verification process, which helps to reduce the time and cost of verification.

SystemVerilog Assertion (SVA) is a language extension to SystemVerilog that provides a powerful and flexible way to write assertions. SVA allows designers to specify complex properties and constraints that must hold true in the design. SVA supports both immediate and concurrent assertions, which can be used to check for a wide range of properties, such as data integrity, timing constraints, and protocol compliance.

3.1 Types of Assertions

There are two types of assertions based on how they are used in the testbench. Immediate assertions and concurrent assertions. Immediate assertions are evaluated at the point where they are encountered in the code. They are typically used to check the value of a signal or variable at a specific point in time. Immediate assertions can be used with any type of simulation, including event-driven and cycle-based simulations. Immediate assertions can be written using the assert statement. If the expression evaluates to false, the assertion will trigger an error.

Listing 3.1: Immediate Assertion Example

```
    always@ (event) begin
        assert (sig_A > 1.5) else
            $error("sig_A is not greater than 1.5");
    end

```

This assertion will check that real signal sig_A is greater than 1.5 (If it is not, it will trigger an error) at the point where it is encountered in the code. This type of assertions is put inside an always/procedural block to make them get triggered based on an event in the code.

On the other hand, Concurrent assertions are continuously evaluated while the simulation is running. They are typically used to check the relationship between signals or variables over time. Concurrent assertions can only be used with clock-driven simulations. Concurrent assertions are written using SystemVerilog Assertion (SVA) or Property Specification Language (PSL), which allows you to specify complex temporal relationships between signals to check the integrity and relationship of signals over multiple clock cycles.

Listing 3.2: Concurrent Assertion Example

```
    property my_property;
        @(posedge clk) (sig_A) |=> (sig_B);
    endproperty

    assert property (my_property);

```

This assertion will continuously check at each positive clock edge that whenever sig_A goes high, sig_B also goes high at the next clock edge. If this relationship is ever violated, the assertion will trigger an error.

3.2 Features of Analog Signals

In the mixed simulation environment, we have signals in digital, analog, and real domain. Checking the integrity of signals in analog domain is a lot more complex in terms of writing assertions as analog signals are represented by continuous values that can vary over time, than the digital signals where there are only two main states 0 and 1. Analog signals can have many features which need to be checked to verify the correctness of the signal. Below are listed some features:

- Frequency of an analog signal (voltage/current).

- Amplitude of a signal (voltage/current) in a particular range.
- Peak-Peak amplitude.
- High/low time of a signal.
- Rise or Fall time of a signal.

Apart from these features there can be dependencies of analog signals on each other as well, just like we have in digital signals as shown in the listing 3.2. This type of dependency can be present in analog signals as well i.e. if sig_a goes above a certain amplitude, only then check the frequency of sig_b. We are going to use immediate assertions as there is no clock synchronicity present in analog domain. Because of this lack of synchronicity, incorporating these types of dependencies is a complex task as well.

While we are working with analog signals, there are two states of signals present on a particular node i.e. Voltage or Current. When we want to look at an analog signal, we usually look for its node in hierarchy from tb_top. While calling that node in mixed signal testbench, we need to identify whether we want to check the current property of the signal or the voltage property of the signal.

This problem is countered by the $cds_get_analog_value, a SV function provided by Cadence that allows to retrieve the analog value of a signal at a specific point in time during a mixed-signal simulation. The syntax of the $cds_get_analog_value is listed below:

Listing 3.3: $cds_get_analog_value Example

```
1    real V_real;
2    always #(10) begin
3         V_real = $cds_get_analog_value("hierarchical_path_to_node",
    "potential");
4    end
```

In this example, the function is used to retrieve the voltage signal at node mentioned in first argument. The code samples the retrieved value in V_real variable of type real. The function is triggered every 10 time unit. The first argument is the hierarchical path to the node under consideration. The object referred to by hierarchical path must be owned by the analog solver. The second argument provides the state of the analog signal which needs to be retrieved. The fetch process can be customized to access any of the following quantities associated with an analog signal: “potential”, “flow”, “power”, “parameter”. Once the analog value is retrieved in a real variable, it can then be passed to an assertion module

for verification.

It's important to note that $cds_get_analog_value is a Cadence-specific function and may not be available in other simulation tools. Additionally, the function can only be used in a mixed-signal simulation, where you have both analog and digital signals in your design and an analog solver is present. If you are working with a purely digital design or a different simulator, you would use other methods to retrieve the value of analog signals.

3.3 Assertion Checkers for Analog Signals

We have discussed some features of analog signals in the previous section, which will be kept in mind while designing the assertions checkers for them. If we can cover these features in assertion checkers, we ensure to verify the integrity of analog signals in every way possible. The assertion checkers must provide an automatic way to verify the features of analog signals. They should be reusable modules for signals with similar features. Now we will discuss how assertions checkers are implemented to answer the above-mentioned questions.

3.3.1 Frequency Checker

The complexity of verifying the frequency of an analog signal comes with the fact that the value of frequency is not inherently present in the signal, unlike the amplitude which is the direct property of the signal. We need some extra block of code to calculate the realtime frequency of the analog signal. Figure 3.1 shows the approach used to calculate the frequency of the signal. The time is calculated from the midpoint upward crossing to the next midpoint upward crossing. The SV implemented of this approach is discussed later in this section.

To verify the frequency of an analog signal, an immediate assertion can be used which can compare the realtime calculated frequency with the frequency provided by the specification. This type of assertion is mentioned in the listing 3.4.

Listing 3.4: Frequency Checker Assert Block

```
1  `timescale 1ns/1ps
2  module frequency_checker(in);
3
4      input var real in;                    // input real value signal
5
```

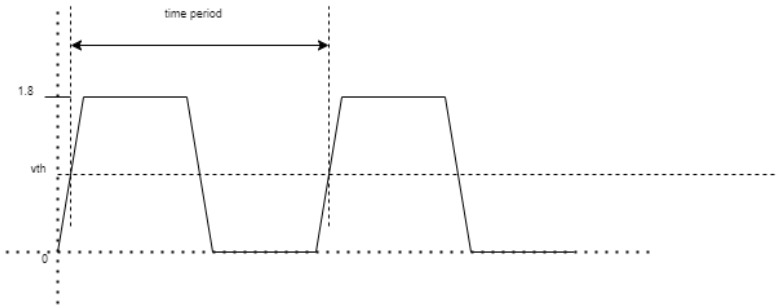

Figure 3.1: Frequency calculation of a Signal

```
6       parameter real fin=20e6;                    //Frequency provided
    by the specification in Hz
7       parameter real Vhi=0.8;                   //Voltage high
    amplitude of the input signal
8       parameter real Vlo=0;                   //Voltage low amplitude
    of input signal
9       parameter real freq_tol_hi=0.5e6;               //freq
    tolerance parameter at higher end in Hz
10      parameter real freq_tol_lo=0.5e6;             //freq_tolerance
     parameter at lower end in Hz
11
12     ....
13
14     always_comb  begin
15      freq_check: assert ((out_freq <= (fin+freq_tol_hi)) && (
    out_freq>= (fin-freq_tol_lo)))
16                  else  $warning ("frequency %f is out of range at time
     %f", out_freq,$realtime);
17     end
18
19     endmodule
```

Here the parameters of the specification are provided along with the input signal “in”. Here the frequency checker verifies whether the frequency of the input signal is 20 MHz. The assertion checks if the "out_freq", which is the realtime calculated frequency, is according to the specification. If the frequency of the input signal is not according to the specification the assertion produces an error.

We are also incorporating tolerance in this assertion because analog signals always have some amount of noise or overshoots and incorporating these tolerances are important to make sure how much uncertainty or noise can the device withstand. These tolerances are also included in assertions to make sure we don’t have any false negative assertion failures, because the analog signals can have noise, and

they can oscillate above or below the actual value mentioned in specification. The tolerances specify how much noise can be tolerated for the signal. The tolerances of +/-0.5MHz is provided in the parameters for this purpose.

Calculating the realtime frequency of analog signal takes some additional block of code which is mentioned in listing 3.5. The mid-point (upward zero crossing) of the amplitude of signal is determined, and the time period is calculated from this event to the next occurrence of this event. The value of time period is flushed again to calculate the time period for the next cycle. This time period is then divided over 1 to calculate the realtime "out_freq". Timescale is taken into consideration while calculating the realtime frequency. Once the "out_freq" is determined, it can be used by assert to verify if this calculated frequency meets the specification.

Listing 3.5: Realtime Frequency Calculation Block

```
1   initial  begin
2       Vth=((Vhi+Vlo)/2.0);    //Calculating the center voltage to
    use as a thershold.
3       end
4
5   //Frequency calculation block starts here.
6
7   always @(in) begin
8       if ($realtime==T0) V0=in;                                    //
    value updated in zero time
9       else begin
10          if (V0<Vth && in>=Vth) begin                             //
    upward crossing detected
11              Tnew = T0 + ($realtime-T0)*(Vth-V0)/(in-V0);         //
    interp crossing time
12              if (Tup>0) Per = Tnew-Tup;
13              Tup = Tnew;
14          end
15          T0=$realtime;
16          V0=in;
17      end
18  end
19  //report freq when period available:
20  assign out_freq = (Per==0)? 0.0 : 1/(Per*1e-9);          //
    Calculated output frequency.
```

We will look at various types of dependencies of one signal to another. Type of dependencies are implemented based of specification of signals in different type of assertion checkers. But the concept can be interchanged on any of the implemented assertions. To implement the dependency of one signal on another, the enabling or disabling assertions technique in SV can be used which involve $asserton and

$assertoff. The example is mentioned in listing 3.6. Here the assertion freq_check is off from start and will turn on after #5000 unit time which is 5us. This type of dependency involves the case where we know that the signal will only be mature after a certain time has passed in the simulation. In this way the assertion will only turn on when the signal is usable by the device to save simulation time.

Listing 3.6: $sserton $assertoff Implementation

```
1    $assertoff (0,freq_check);
2    #5000
3    $asserton (1,freq_check);
```

The $asserton and $assertoff functions have two arguments. The first one signifies the level at which the function should be performed. For '0', the function is applied to the assertions in that module and all the assertions of same name in the modules called inside it. For '1', the function is applied to only the assertions of that module. The second argument specifies the name of assertion for which the function in applied.

Figure 3.2 shows the waveform of frequency assertion implementation. Here, the assertion is off for the first 5us. And once the assertion starts, it is verifying the frequency of the signal. The green portion in the assertion waveform represents assertions pass.

Figure 3.2: Frequency Assertion Example

3.3.2 Amplitude Checker

Amplitude checker includes assertion which checks if the voltage/current level of the signal is within range mentioned in the specification. The amplitude checker is much easier to implement than the frequency checker because the signal inherently contains the amplitude property and signal amplitude can directly be compared to the amplitude of specification. This assertion can be used in a very broad application. Apart from range checking, the assertion, once enabled, can also check if the signal always remains above or below a certain amplitude level. These types of application can be extended by slightly modifying the assertion.

An immediate assertion is used for the verification of amplitude of analog signals written in an always_comb block. It is mentioned is listing 3.7. The assertion checks that the input signal 'in' should be within range of the amplitudes provided as the parameters to the assertion module. If it is not within range the assertion produces an error message. Tolerance of +/-1% is also given in the parameters and implemented in the assertion as per the specification.

Listing 3.7: Amplitude Checker Assert Block

```
1    parameter real range_tol = 0.01;
2    parameter real Vhi=0.4;                  //Voltage high amplitude of
     the input signal
3    parameter real Vlo=-0.4;                  //Voltage low amplitude of
     input signal
4
5    ...
6
7    always_comb  begin                  //assertion check block
8      range_check:  assert ((in <= Vhi *(1 + range_tol)) && (in >=
     Vlo * (1 - range_tol)))
9                 else if (in > Vhi) $warning ("FAIL, signal is too
     high: %f", in);
10                else $warning ("FAIL, signal is too low: %f", in);
11   end
```

Like previous assertion, here as well, the dependencies of one signal on another is involved. The assertion should wait for some event to happen before starting. When an enabling signal goes high, wait for a time delay, and then start the assertions. This type of dependency was a requirement of the specification for this signal, but it can be implemented with any other type of assertion.

Listing 3.8: Amplitude Assertion Enable Block

```
1    parameter real Vsig_high = 1.8;
2    parameter real tran_delay = 100000;
3    int count=1;
4
5    initial begin
6      $assertoff (0,range_check);
7    end
8
9    always@(enable_sig)  begin                  //assertion enable block
10       if (enable_sig>=Vsig_high*0.99 && count == 1) begin
11               #(tran_delay)
12               $asserton (1,range_check);
13               count = 0;
14       end
```

```
15      end
```

Listing 3.8 shows the implementation of this dependency. Here the parameters for enabling signal and time delay are passed to the module. Assertion enable block turns the assertion on, once the enabling signal goes high.

Figure 3.3 shows the waveform of Amplitude assertion implementation. Here, the assertion is off till the enable_sig goes high. Once the enable_sig is high, the assertion waits for tran_delay time of 100us and then turns on. The assertion checks if the signal_in is within the range mentioned in the specification. Here the assertion is passed, after it gets enabled, for the whole simulation time.

Figure 3.3: Amplitude Assertion Example

3.3.3 High/Low Time Checker

This assertion checks if the high time of the signal is as per the specification. Similar to Frequency checker, the realtime calculation of high time of the signal is required. Extra block of code is required to calculate this time before writing assertion. High time is calculated from the point when the signal goes higher than the midpoint of the amplitude, to the point when the signal drops back to below the midpoint of the amplitude. This assumption can be modified based on what amplitude should be considered as a high signal. Figure 3.4 has the waveform showcasing this approach.

Listing 3.9 shows the SV implementation of this approach. Here the start time is sampled when the signal goes above the midpoint and finish time is sampled when the signal goes back below the midpoint. Here a token system is also introduced which lets the code sample the upward and downward time only once per clock cycle. This approach can also be modified to calculate the Low time of the signal instead of the high time. This block is triggered whenever there is a change in the input signal. Once the high time is calculated, it is assigned to out_time.

Listing 3.9: High Time Calculation Block

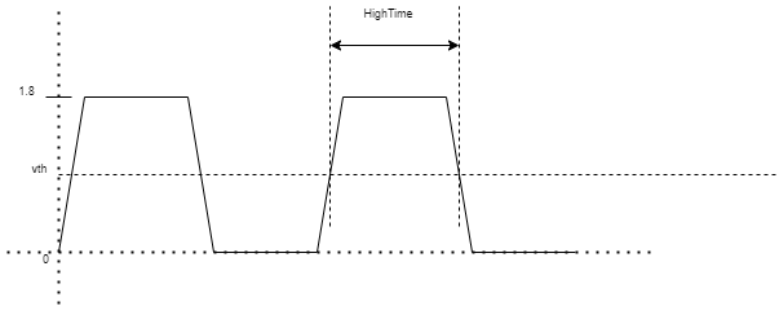

Figure 3.4: High Time of a signal

```
1     initial  begin
2          Vth=((Vhi+Vlo)/2.0);     //Calculating the center voltage to
      use as a thershold.
3     end
4     //High time calculation block starts here.
5     always @(in) begin
6               if (up == 0 && in>=Vth) begin                          //
      upward crossing detected
7                    Tup = $realtime;       // sample the start time
8                    down = 0;
9                    up = 1;
10              end
11              if (down ==0 && in<=Vth) begin         //downward crossing
      detected
12                   Tdown = ($realtime - Tup);       // sample the finish
      time and calculate
13                   down = 1;
14                   up = 0;
15              end
16    end
17    //report time when available:
18    assign out_time = Tdown;         //Calculated total high time.
```

Once the High time is calculated, a simple assertion can check if the high time of the signal is less than the value mentioned in the specification. Listing 3.10 shows the assertion. Here the parameter Tcheck represents the high time 1.5us from the specification. The assertion should verify that the high time of the signal does not go higher than this value. For this, Tcheck is compared with the out_time calculated from the input signal. Assertion produces an error message if the high time of signal goes higher than the specification.

Listing 3.10: High Time Assertion Block

```
1
2     parameter real Tcheck=1500;
3
4     ...
5
6     //Assertion module block starts here
7     always_comb  begin
8           hightime_check: assert (out_time <= (Tcheck*1.01))
9                 else  $warning ("High time %f is not correct at time
    %f", out_time,$realtime);
10    end
```

Figure 3.5 shows the waveform of High time assertion implementation. Here, the assertion is on from start and checking if the high time of the signal is less than 1.4us, what's mentioned in the specification. Here the assertion is passed for the whole simulation time.

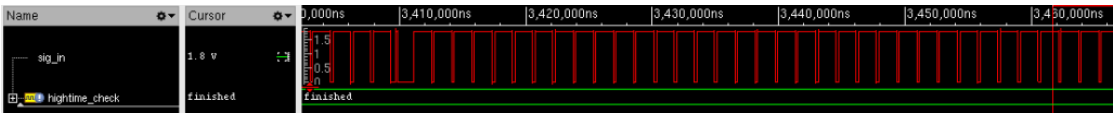

Figure 3.5: High Time Assertion Example

3.3.4 Rise/Fall Time Checker

This assertion checks if the rise or fall time of a signal is as per the specification. Here again, we need some extra block of code to calculate the real-time rise and fall time of the signal and then verify it by comparing it to the specification. Calculating rise/fall time of the signal in realtime is the most complex part of all the assertions implemented. While the signal is low, the start time of for the rise time calculation is keep on updating, we sample the start time once the signal starts rising. Once the signal reaches its high value the final time is sampled. The difference of the two times is defined as the rise time of the signal. The listing 3.11 describes the implementation of realtime rise time calculation.

Listing 3.11: Rise Time calculation Block

```
1
2     real Tf,Ti,rise_time_calc;
3     real Vhi_lo_per, Vhi_hi_per;
4     real vamp_tol = 0.01;
5     real count,count1;
6     real tsam=0.02;
```

```
7
8     initial begin
9         Vhi_lo_per= Vlo - vamp_tol*Vlo;
10        Vhi_hi_per= Vhi - vamp_tol*Vhi;
11    end
12
13    ...
14
15    always #(tsam) begin
16    //for risetime;
17      if((in>=Vlo) && (in<=Vhi_lo_per)) begin
18        count = 1;
19      end
20
21      else if (count == 1) begin
22        Ti = $realtime;
23        count = 2;
24      end
25
26      if((in<=Vhi) && (in>=Vhi_hi_per) && (count == 2)) begin
27        Tf = $realtime;
28        count = 0;
29        rise_time_calc = (Tf - Ti)* 1e-9;
30      end
31    end
```

The tsam is the frequency at which the signal will be monitored. The signal is monitored to check its status, whether the signal is high, low, or rising. If the signal is low, nothing happens. Once the signal starts rising the initial time is sampled. When the signal reaches high value, the final time is sampled. The rise time is calculated by taking the difference of the two times. Tsam can be decreased to increase the accuracy of the rise time calculation and can be increased to decrease the simulation time at the cost of less accuracy. Vhi_lo_per is the value of the minimum change in the signal which is significant by taking the tolerance of the signal into account. This value is essential to find the point where the signal is just starting to rise and when the signal is just going to be at high value.

Fall time can also be calculated similarly by swapping the high and low voltage values. In that case the calculation will start from the high value and end at the low value of the signal.

Once the rise time is calculated, a simple assertion can check if the rise time of the signal is equal to the value mentioned in the specification. Listing 3.12 shows the implementation of this assertion. Here the parameter rise_time_user_in represents the rise time from the specification. The assertion should verify that the

calculated rise time of the signal is equal to this value. For this, rise_time_user_in is compared with the rise_time_calc calculated from the input signal. Assertion produces an error if the high time of signal goes higher than the specification.

Listing 3.12: Rise Time Assertion Block

```
1
2     parameter real rise_time_user_in=5e-9;     //User supplied rise
    time input parameter to be used against the calculated value in
    seconds(sec)
3     parameter real rise_time_tol=0.01;         //Tolerance value over
    calculated rise time. 0.01 represent 1% tolerance.
4
5     ...
6
7     always_comb  begin                        //Rise time assertion check
8
9         rise_time_check:  assert ((rise_time_calc >= (
    rise_time_user_in*(1-rise_time_tol))) && (rise_time_calc <= (
    rise_time_user_in*(1+rise_time_tol))))
10
11  end
```

Figure 3.6 shows the waveform of Rise assertion implementation. Here, the assertion is on from start and checking if the rise time of the signal is equal to what's mentioned in the specification. Here the assertion is passed for the whole simulation time.

Figure 3.6: Rise Time Assertion Example

3.4 Benefits of Analog Assertions

Looking at waveforms to verify if the signal is working correctly for long simulation runs is very tedious and time-consuming. The assertions are not very complex to design and provide an automatic way to verify the features of analog signals. One of the main benefits of these assertions is that they are built on SV basic functions and can be used in any simulator or any type of environment. We just need to take care of extracting the current or voltage properties from the analog node as this is done differently in each simulator. Another benefit of the assertions is that they

are built in modules and can be reused multiple times in the same testbench for signals with similar features.

While discussing the assertions in the previous section, waveforms are attached to visualize the working of the assertions. This is not required at all, as the assertions do not require waveform dumping to work. They work independently of waveform probing. The benefit it brings is that, if the sole purpose of storing the waveform was to visualize the assertions, then it can be skipped in favor of much faster simulation runs.

Chapter 4

Signal Generators Modelling

Signal generators are electronic devices that produce electrical waveforms, such as sine waves, square waves, and triangle waves, at various frequencies and amplitudes. Signal generators can be either analog or digital, and they typically have various features such as variable duty cycle, voltage offset, differential output etc. Analog Signal generators are complex systems. To simulate complex circuits, the software program must perform a very extensive set of calculations. These circuits can be difficult and time-consuming to simulate without models.

Real Number Modelling (RNM) is a mathematical technique used to represent and analyze physical systems using real numbers. In the context of signal generators, real number modelling is used to represent the electrical waveforms produced by the generator using mathematical functions that are defined using real numbers. The signal generator models when built using RNM offer a simplified circuit representation of these analog circuits that may be simulated much faster compared to analog counterparts.

These signal generators can be used to provide input stimulus for analog systems when these systems are being verified standalone or as combined. This way we do not need to use the complex analog signal generators which take a lot of simulation time, instead we use the models of signal generators for this purpose to speed up the simulation time as much as possible. We used the ramp generator to check the output signal of a linear driver.

In this section we will discuss the techniques used to build these signal generators with RNM and the specifications and features required for these signal generators to have. Triangular, sine and square wave generators are modelled in this thesis with variety of features.

4.1 Specification and Features

The signal generators should produce these types of waves: sine wave, square wave, and differential sine and triangle waves. Signal Generators should have configurable parameters such as these:

- Frequency range: Signal generators should allow users to adjust the frequency of the wave, ranging from a few Hertz to several MHz
- Amplitude control: Signal generators should allow users to adjust the amplitude of the output waveform to a desired level.
- Amplitude offset control: Signal generators should allow users to adjust the amplitude offset of the output waveform.
- Duty Cycle Control: Signal generators should allow users to adjust the duty cycle of the waveform. Except for sine waves.
- Rise/Fall time control: Signal generator should allow user to adjust the rise fall time of the waveform in case of square wave generator.
- Differential Output: The signal generators should provide differential output in case of sine wave and triangular wave.

All the signal generators are built with the same name scheme for input and outputs so that they can be reused and replaced with one another whenever required. Input signals include "fin" for frequency, "voff" for offset voltage, "vpp" for peak-to-peak voltage, "duty" for duty cycle input and "trf" for rise/fall time. The output signal is "vout" for all modules but is "voutn" and "voutp" for differential sine wave and triangular wave generator.

The signal generators are controlled by a virtual interface which is instantiated and configured through the testbench. Input stimulus for the signal generators is controlled by the testbench is a test class.

4.2 Implementation of Signal Generators

Now we will discuss the technique and approach used for the implementation of signal generators using RNM.

4.2.1 Sine Wave generator

Sine wave generator is one of the easiest generators to implement in SV real-number modelling. As SV supports sine function, we can really understand complexity difference of designing the sine generator in completely analog environment to designing sine wave generator in RNM where everything can be done by just one mathematical function. We just have to incorporate the voltage offset variable to the equation and, then we get the fully customizable sine wave as per the specification described above.

Before looking at the sine function, we will discuss listing 4.1 first. Here a "phase" is being calculated based on the "tsam" and frequency input to the module. Time "tsam" is the sampling time in the modelling technique. This time should be very, very small compared to the maximum frequency we are going to achieve in using the model. If the "tsam" is very close to the maximum frequency, the accuracy of the signal generator will be compromised. This way for each frequency cycle the "phase" start from 0 and increments a small amount after every "tsam" and when the frequency cycle reaches its end the "phase" also approaches 1. When the new frequency cycle arrives, the "phase" is reset to zero as well. If "tsam" is 1000 times smaller than the maximum frequency then at maximum frequency "phase" will take 1000 steps to reach from 0→1, making the "phase" steps smoother.

Listing 4.1: Phase Calculation

```
1    always #(tsam) begin
2        phase = phase + freq*1000000*(tsam*1e-9);
3        if (phase>1) phase = phase - 1;
4    end
```

This approach of calculating "phase" helps in determining where in the frequency cycle we are actually present i.e., if the "phase" is 0.5 this means we are in the midpoint of a frequency cycle. This helps when changing duty cycles and when designing some peculiar waveforms like triangular waves which do not have a built-in function in SV. The "phase" is visualized in Figure 4.1. Here, as the input frequency changed midway, the "phase" calculation has modified itself as well.

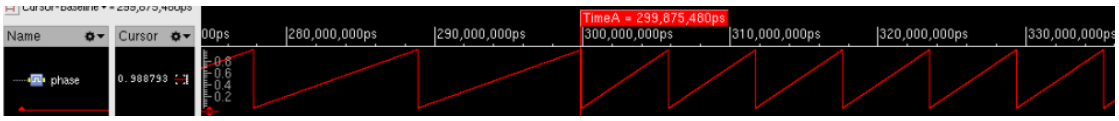

Figure 4.1: Phase Waveform

Once the phase is calculated, it can be given as argument to sin function along

with other design variables. Here voltage offset is added to the sin function and phase multiplied with 2*pi is applied as its argument. Voltage peak to peak is also multiplied with the sin function. The sine function implementation is mentioned in listing 4.2.

Listing 4.2: Sine Wave Function

```
    `timescale 1ns/1ps
    `define M_TWO_PI 6.28318530718
    // get sine function from C math library:
    import "DPI" pure function real sin (input real rTheta);
    module vco_sin (sin_if.sin_mod dif);

    ...

        assign dif.vout = dif.voff + dif.vpp/2*$sin(`M_TWO_PI*phase);

    endmodule
```

The sine wave generator module is controlled through a virtual interface. The testbench stimulus provided in the test class and the interface are mentioned in listing 4.3.

Listing 4.3: Sine Wave Interface and Stimulus

```
interface:

    `timescale 1ns/1ps
    interface sin_if;
    real fin, voff, vpp, vout;

     modport sin_mod ( output vout,
                       input  fin, voff, vpp );
    endinterface : sin_if
    ..........................
testclass:
class test_base extends uvm_test;

    virtual sin_if sin_vif;

    ....

    sin_vif.voff = 0.2;
    sin_vif.vpp = 0.8;
    sin_vif.fin = 22;  #500      // freq in MHz
    sin_vif.fin = 37;  #400
    sin_vif.vpp = 0.6;
```

```
23      sin_vif.fin = 7;   #600
24      ....
25
26  endclass
```

The waveform generated can be seen in Figure 4.2. The waveform is working in accordance with the input stimulus applied. Initially, the frequency is 22 MHz, "vpp" is 0.8V and "Voff" is 0.2V. Then, frequency is increased to 37 MHz and then again it is decreased to 7 MHz along with decrease in "vpp" to 0.6V.

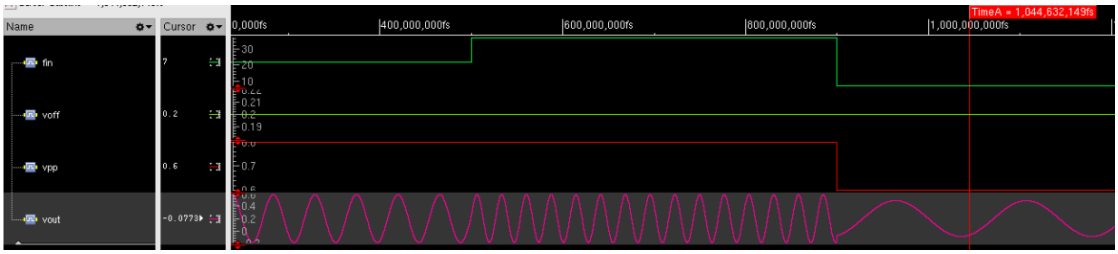

Figure 4.2: Sine Waveform

In the case of differential output of sine wave, there is only a slight change in the overall approach. We can generate two waveforms of half the voltages opposite to each other. For this, two "sin" functions can be use as mentioned in listing 4.4. Similarly, interface is changed as well and after providing similar stimulus for differential wave, we get the result as shown in the figure 4.3.

Listing 4.4: Differential Sine Wave Code

```
1
2       assign difd.voutp = difd.voff + difd.vpp/4*$sin(`M_TWO_PI*phase);
3       assign difd.voutn = difd.voff - difd.vpp/4*$sin(`M_TWO_PI*phase);
4
5
```

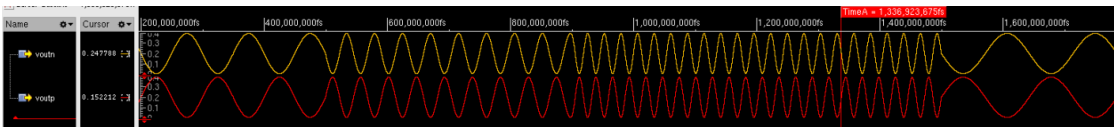

Figure 4.3: Differential Sine Waveform

4.2.2 Triangular Wave Generator

The triangular waveform has a linear rise and fall time, with the peak voltage occurring at the midpoint of the waveform. The triangular wave generator we designed has a variable frequency, variable voltage peak-peak, variable voltage offset and variable duty cycle. The output of the wave generator is a differential two pin output. Variable duty cycle and differential output are the two new parameters for triangular wave generator compared to sine wave generator. We are still using the same approach of first calculating the "phase" of the signal from the input frequency. This way we can figure out where exactly should the peak of the wave be, based on variable duty cycle.

Listing 4.5 shows the first step for the implementation of triangular wave generator. Here the always block is sensitive to all the input variables. Whenever any of the variable changes this always block will get triggered. Here the "freq", "dt", "vtp", "vtn" are updated whenever any variable change essentially reinitializing all of them with the new values. The "phase" gets reinitialized as well. The "vtp" and "vtn" are also both initialized with the peak negative and positive amplitude values, respectively, whenever this always block is triggered.

Listing 4.5: Triangular Wave Initializer

```
1
2 always @(dif.fin, dif.duty, dif.vpp, dif.voff) begin
3     freq <= dif.fin;
4     dt <= dif.duty;
5     vtp <= -dif.vpp/4 + dif.voff;
6     vtn <= dif.vpp/4 + dif.voff;
7     phase = 0;
8  end
```

Listing 4.4 shows the second part of the implementation of the triangular wave generator. Here the always block is triggered every "tsam" time. This is the sampling time which can be set in the module and should be at least 1000 times smaller than the maximum frequency required to be achieved. This will cause the module to always work accurately. The "phase" is calculated in the same way as explained in the previous section. The "phase" goes from 0→1 linearly in one frequency cycle time. We can use this "phase" to change the duty cycle of the wave as well.

As shown in the listing, when the "phase" is below the duty cycle, the "vtp" is minimum and is incremented a small amount. The increment amount is calculated based on the total voltage peak-peak, "tsam" and the frequency. This increment is

calculated with the concept that if the signal starts from zero it should reach the maximum value coinciding with the "phase" reaching the duty cycle time. Once the "phase" is above the duty cycle time, the "vtp" is now decreased a small amount each time the block is triggered. This small amount is also calculated based on the fact that the signal should go back to zero as the "phase" reaches 1 and the next cycle has to start. This way signal goes from minimum to maximum in the duty cycle time and goes back to minimum in the other time in once frequency cycle. The same case is true for "vtn" but with opposite concept as the signal should go from maximum to minimum and then back to maximum in the same frequency cycle.

Listing 4.6: Triangular Wave Generation Block

```
always #(tsam) begin
    phase = phase + freq*1000000*(tsam*1e-9);
    if (phase>1)
       phase = phase - 1;

    if (phase < dt) begin
        vtp <= vtp + (dif.vpp/2 * freq*1000000*(tsam*1e-9) /dt);
        vtn <= vtn - (dif.vpp/2 * freq*1000000*(tsam*1e-9) /dt);
     end

    else begin
        vtp <= vtp - (dif.vpp/2 * freq*1000000*(tsam*1e-9) /(1-dt));
        vtn <= vtn + (dif.vpp/2 * freq*1000000*(tsam*1e-9) /(1-dt));
     end

end

assign dif.voutp =  vtp;
assign dif.voutn =  vtn;
```

The triangular wave generator module is controlled through a virtual interface. The testbench stimulus provided in the test class and the interface are mentioned in listing 4.7.

Listing 4.7: Triangular Wave Interface and Stimulus

```
interface:

    `timescale 1ns/1ps
    Interface tri_if;
        real fin, voff, vpp, voutn, voutp, duty;

        modport tri_mod ( output voutn, voutp,
                          input  fin, voff, vpp, duty );
```

```
    endinterface : tri_if
    ..........................
testclass:
class test_base extends uvm_test;

    virtual tri_if tri_vif;

    ....

    tri_vif.fin = 10;        // freq in MHz
    tri_vif.voff = 0.2;
    tri_vif.vpp = 0.8;
    tri_vif.duty = 0.25;
    #500
    tri_vif.fin = 22;
    #400
    tri_vif.fin = 37;
    tri_vif.duty = 0.7;
    #600
    tri_vif.fin = 7;
    tri_vif.duty = 0.5;
    ....

endclass
```

The waveform generated can be seen in Figure 4.4. The effect of change in the duty cycle, and frequency can be seen. Voltage peak to peak is 0.8V for whole simulation. Voltage offset of 0.2V is also applied. The differential output triangular waveforms are represented by "voutn", "voutp".

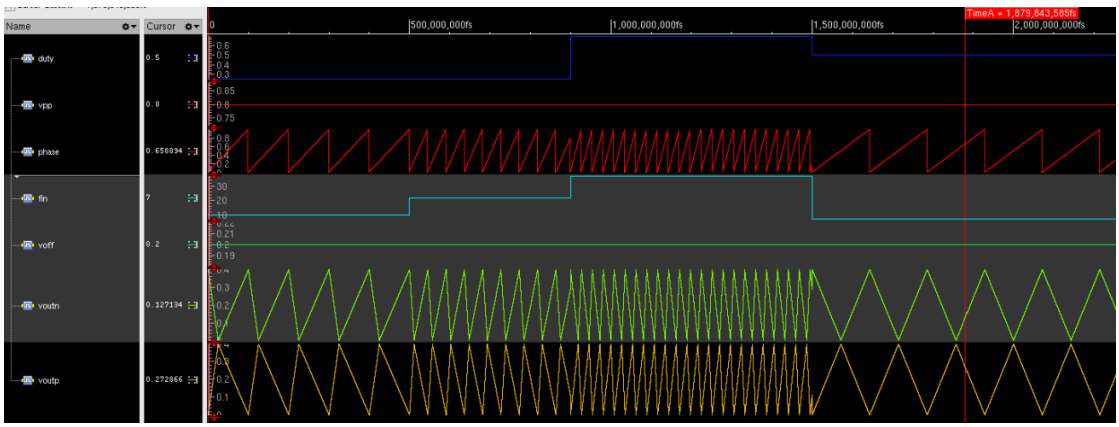

Figure 4.4: Triangular Waveform

4.2.3 Square Wave Generator

A square wave is a type of waveform that alternates between two voltage levels, typically a high voltage level and a low voltage level, with a fixed period and duty cycle. We want to model the square wave generator with features such as, variable duty cycle, variable frequency, variable voltage peak-peak, variable voltage offset and variable rise and fall time of square wave. Incorporating variable rise and fall time in the square wave generator is the most complex task here.

We are using the same approach as the triangular wave generator. Initializing the variables with respect to change in input parameters and then modifying the output based on triggering another block after every sampling time. Listing 4.8 shows the initialization block of the module. The initial block initializes the "v" to a specific value. When any of the inputs change, the always block is triggered because of the sensitivities. The new value of "inc" is calculated based on any change in the input values. The value of frequency and duty cycle is updated as well. The "inc" variable is the unit step increase or decrease in voltage for the rise and fall of the signal. This "inc" is calculated based on the frequency of the signal, sampling time "tsam" chosen in the module, rise and fall time "trf" given as input parameter and voltage peak-peak given as the input. This "inc" is calculated in a way to make sure that the signal reaches from minimum to maximum value at exactly rise time specified in the input parameters. Similarly, the fall time of the signal will also hold true to this parameter.

Listing 4.8: Square Wave Initializing Block

```
1  initial begin
2    v = -dif.vpp/2 + dif.voff;
3  end
4
5  always @(dif.fin, dif.duty, dif.vpp, dif.voff, dif.trf) begin
           // when input changes
6    freq = dif.fin;
7    dt = dif.duty;
8    inc = ((dif.vpp) * dif.fin*1000000*(tsam*1e-9) /(dif.trf * 1e-9 *
     dif.fin*1000000));
9  end
```

Once the initialization of variables is done, we can proceed to the signal generation block of the module. Listing 4.9 represents the generation block of the square wave generator module. Here the always block is triggered every "tsam" time. This is the local variable of the module and sampling time chosen should be 1000 times smaller than the highest frequency expected to be achieved by this module to make sure the accuracy of the wave generation is not compromised.

The "phase" is calculated in the same way as mentioned in the previous sections. The "phase" is used to make the duty cycle of the waveform variable. We initialized the signal, in the initial block, to be at a minimum value when we start the generator. When the "phase" is below the duty cycle, the signal is checked if it is at the minimum level or not. If the signal is already at the minimum value nothing happens. When the "phase" goes above the duty cycle, the signal is checked whether it is below the maximum level or not. In this case, it is, because at the moment the signal is at the minimum. The if statement holds true, and the signal gets incremented every sampling time. It is determined already that the "inc" is added in a way that the signal will reach its maximum value exactly at the rise time mentioned in the input parameter. Once the signal reaches maximum, the if statement now holds false and the signal stops getting further incrementation.

When the cycle ends and the "phase" goes back to zero, now again, the "phase" is less than duty cycle. Now, the voltage is at the maximum, so the if statement holds true and the signal is the decrement the "inc" amount every sampling time and signal reaches the minimum value in exactly the fall time mentioned in the input. In this module, the rise and fall time of the signal is same and there is only one variable for both times.

At the end the variable "v" is assigned to the output variable vout.

Listing 4.9: Square Wave Generation Block

```
always #(tsam) begin
  phase = phase + freq*1000000*(tsam*1e-9);
  if (phase>1) phase = phase - 1;

  if (phase < dt) begin
    if(v > (-(dif.vpp/2) + dif.voff + inc))  begin
      v <= v - inc;
    end
  end

  if (phase >= (1-dt)) begin
    if(v < ((dif.vpp/2) + dif.voff - inc))  begin
      v <= v + inc;
    end
  end
end

assign dif.vout = v;

```

The virtual interface for the square wave generator and the input stimulus is mentioned in the listing 4.10. The generator is controlled by input stimulus from a test class by instantiating the virtual interface there. Initial frequency is 10 MHz, rise and fall time is 5ns, voltage offset is 0.2V, voltage peak-peak is 0.8V and the duty cycle is 0.4.

Listing 4.10: Square Wave Interface and Stimulus

```
1    interface:
2
3        `timescale 1ns/1ps
4        interface sq_if;
5            real fin, voff, vpp, vout, duty, trf;
6            modport sq_mod ( output vout,
7                             input  fin, voff, vpp, duty, trf );
8        endinterface : sq_if
9        ..........................
10   testclass:
11   class test_base extends uvm_test;
12
13       virtual sq_if sq_vif;
14
15       ....
16
17       sq_vif.fin = 10;          // freq in MHz
18       sq_vif.trf = 5;           // time in ns
19       sq_vif.voff = 0.2;        // voltage in V
20       sq_vif.vpp = 0.8;
21       sq_vif.duty = 0.4;
22       #300
23       sq_vif.fin = 22;
24       sq_vif.vpp = 0.6; #300
25       #300
26       sq_vif.fin = 15;
27       sq_vif.duty = 0.75;
28       #600
29       sq_vif.fin = 7;
30       sq_vif.vpp = 0.8;
31       sq_vif.duty = 0.5;
32       ....
33
34   endclass
35
```

The waveform generated by the square wave generator can be seen in Figure 4.5. The effect of change in the duty cycle, the frequency and the peak-peak voltage can be seen. Voltage peak to peak is initially 0.8V, then 0.6V, and then again 0.8V for whole simulation. Rise and fall time of 5ns is applied. Voltage offset of 0.2V is

also applied. The output square waveform is represented by "vout".

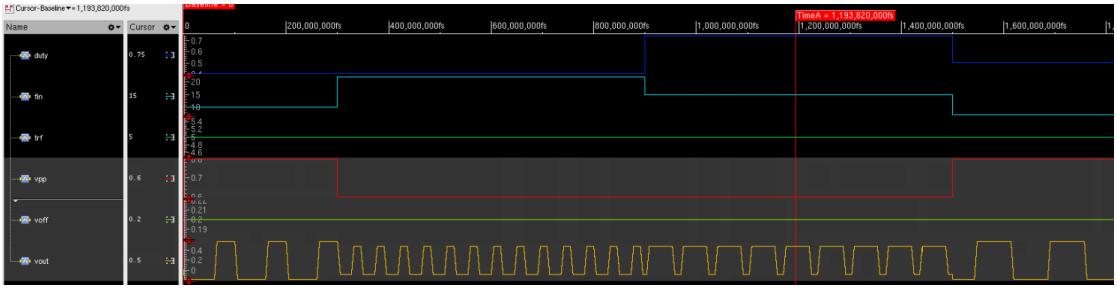

Figure 4.5: Square Waveform

Chapter 5

Metric-Driven Verification

Metric-driven verification (MDV) for Application-Specific Integrated Circuit (ASIC) is a methodology used to verify the functionality and performance of an ASIC design. It involves defining a set of metrics or parameters that are used to measure the quality and completeness of the verification process.

The metrics can include things like code coverage, functional coverage, performance metrics, covergroups coverage, and other parameters that are relevant to the design. These metrics are tracked and analyzed throughout the verification process to ensure that the design meets the required specifications.

The goal of MDV is to ensure that the design is thoroughly tested and meets all the required specifications before it is released for production. This approach helps to reduce the risk of design errors and ensures that the ASIC meets the required performance and power consumption targets.

Merging of metrics in MDV is the process of combining multiple metrics to gain a more comprehensive understanding of the verification process. This can be done by combining different types of metrics, such as code coverage, functional coverage, and performance metrics, to get a more complete picture of the verification process.

We have discussed in the previous sections that mixed signal designs are complex. It requires several internal registers and inputs of the device to be configured before the simulation can start. This configuration of internal registers is done using UVM backdoor configuration. UVM backdoor configuration is a technique used in verification to access and modify internal signals and registers of a Design Under Test (DUT) directly, bypassing the normal interface of the DUT. This technique is used to test and debug the DUT more efficiently and effectively. Once the device is configured for one stimulus, it runs with same configuration for the whole simulation time.

When we want to verify the DUT for multiple configurations and stimulus, we need to run multiple simulations with these distinct configurations. This is the reason we configured the command-line simulation environment, so that multiple simulations with distinct configurations can be run in parallel. We collect metrics from these distinct simulations. The assertions provide functional coverage metrics and randomized configuration stimuli provide covergroup coverage metrics. Once these metrics from the multiple simulations are acquired, they can be merged to achieve the required verification goals.

The merging of these metrics is an important concept in mixed signal simulation. In digital verification, the DUT is synchronous to the clock and randomized stimulus can be provided at each clock edge. This results in a single functional coverage containing the result of thousands of randomized stimuli. But in case of mixed simulation environment, we generate a random stimulus for the configuration and then run the whole simulation with that one configuration. So, many simulation runs are required for verifying the device with multiple randomized configurations to get the desired result of verifying the DUT for multiple randomized input stimuli. We randomized the configuration only once at the start for simulation because the device settings have too many dependencies. But for another devices it's possible to randomize more stimulus per run and then merge all the runs to increase coverage.

In the next section, we will discuss coverages related to randomized input stimuli and the coverages related to assertions. For the first case, we will randomize some internal registers and collect their coverages by writing some covergroups. For the second case, we need to use the assertions discussed in the previous chapter and implement them for some analog signals of the DUT and collect the functional coverages using them. We will then merge the multiple metrics from the simulation runs for both cases.

5.1 Randomized Stimulus Coverage

This step involves randomizing internal registers of the device before the start of the simulation and observing the effect on the output based on that. We are targeting a specific part of the device for this implementation. Two configuration registers were chosen to be randomized. Both registers are 3 bits.

The steps involve initializing the registers with "rand" for randomization, writing

covergroups for the two registers and their corresponding constraints and instantiating the covergroup inside the configuration class. For this project, the covergroup is placed inside the configuration class. The object of this class is used to store the register's values. The covergroup can be placed in a separate class as well and that class maybe called in the configuration class or the test class. The listing 5.1 shows the implementation of covergroup.

Listing 5.1: Covergroup Configuration

```
class test_base_cfg extends uvm_object;

    ...

    constraint reg_2_cnst {reg_2>0; reg_2<7;};

    covergroup thesis_covergroup;
        option.per_instance = 1;
        clk_div_cp: coverpoint reg_1;
        duty_prog_cp: coverpoint reg_2
         {
           bins range[1]= {1,2,3,4,5,6};}
    endgroup

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

    function void post_randomize();
        $display("post randomization done");

        thesis_covergroup.sample();

    endfunction : post_randomize
endclass

```

Here the names of these registers are changed for secrecy. The constraint represents that the register 2 only has values from 1 to 6. Its corresponding "bins" are also mentioned. The "bins" are constructs which allow every value inside to have its separate bin and coverage profile. The object of this class is instantiated and randomized in the test class using "randomize()" function of the SV. Once the "randomize()" is called and the randomization is done, the post_randomize() is also called and the coverage of the covergroup is collected.

Figure 5.1 shows the overall coverage collected from only one simulation run.

The covergroup coverage for register 1 can be seen below. The randomized stimulus generated for register 1 in this simulation run is 3. Register 2 has a constraint and multiple bins, so the coverage for it is 100%.

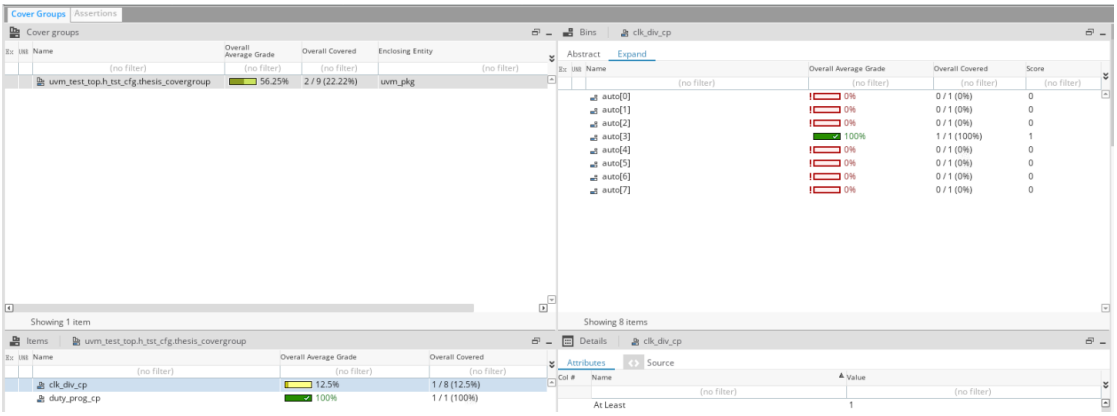

Figure 5.1: Covergroup Coverage for Single Simulation Run

Figure 5.2 shows the combined coverage collected by merging the results of five simulation runs. The covergroup coverage for register 1 has increased representing all the values which has been tested. In the five simulation runs, 4 values have been randomnly generated for inputs with one being generated twice. With more simulation runs we can generate all the possible input stimulus and achieve 100% covergroup coverage.

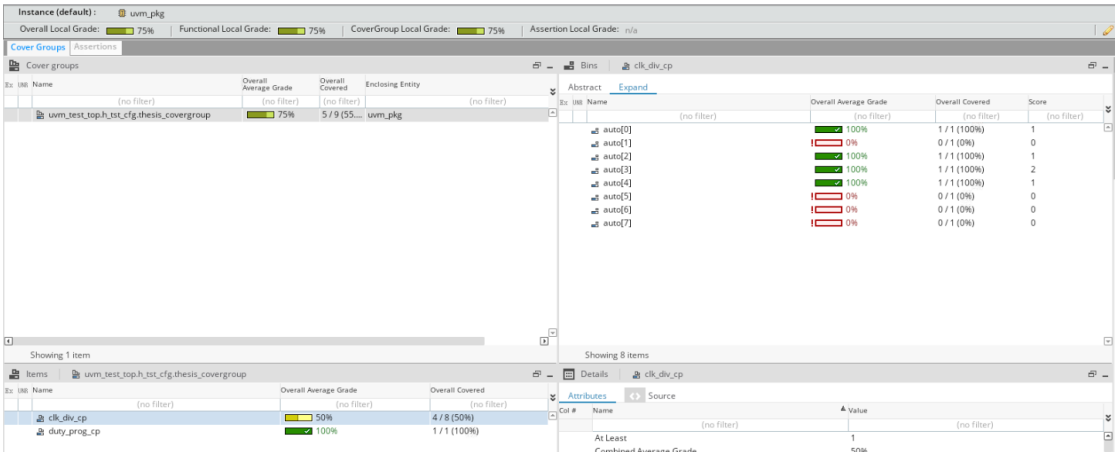

Figure 5.2: Covergroup Coverage for Five Simulation Runs

5.2 Assertion based Functional Coverage

The analog assertions discussed in the previous chapter are used to verify some signals of a specific part the device. The assertion modules are called in the tb_top module. The analog signals are converted to real signals using "cds_get_analog_value()" function. And the real values are passed to assertions as inputs. The Assertions can be placed in another module as well which can take all the signals as input. That other module can act as a wrapper around the assertions so that the tb_top module can be as clean as possible.

The analog assertions are used for 9 signals, and some assertions are used multiple times emphasizing the reusability of the modules. A variety of features of analog signals are verified including frequency of the signal, current and voltage levels range of the signals, high time of the signal etc.

Figure 5.3 shows the functional and code coverage metrics collected from a single run. Here some signal names are not shown for secrecy reasons. The type of assertion checkers used for each signal are also shown. The dark green bars in the figure show that all assertions are hitting 100% functional coverage and the light green bars show that the combined functional and code coverages of some assertions are not 100%. The data on the right side of the figure represent the coverage of overall testbench including some other assertions not shown here.

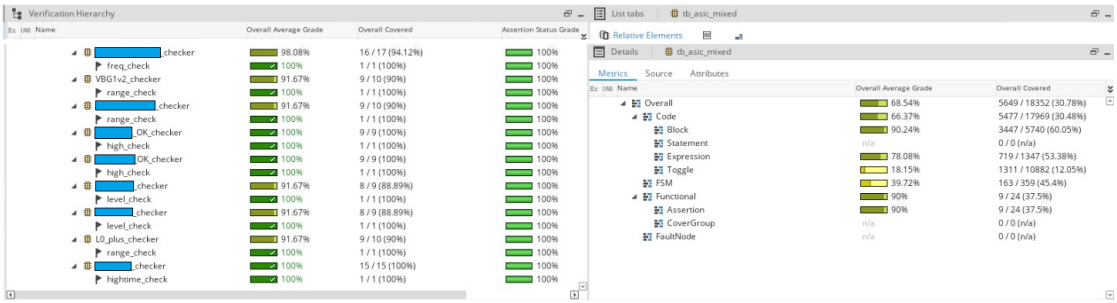

Figure 5.3: Assertion Coverage Result for Single Run

Now if we merge the metrics results of multiple simulation runs, we will be able to get a better coverage result. Figure 5.4 shows the functional and code coverages collected by merging metrics results from 3 simulation runs. Here, the overall coverage of the testbench mentioned on the right side of the figure has improved and the code coverage of some assertions has gone to 100% as well, which was not the case in the previous figure. By combining the metrics of just 3 simulation runs we can increase the overall coverage of the device.

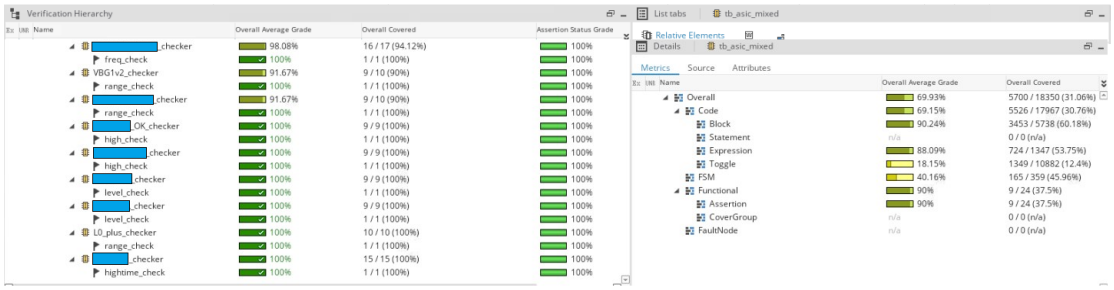

Figure 5.4: Assertion Coverage Result by Merging Metrics From Multiple Runs

The mixed signal simulations take a long time to complete. If we want to increase the code coverage with just one simulation, we will have to run the simulation for a longer time. And it will take a large amount of real-world time in terms of weeks for it to complete. But by running multiple short simulation runs in parallel we can achieve the same code coverage results in the real-world time of just one short simulation. This shows the benefit and importance of MDV and the metrics merge feature. Running hundreds of simulations in parallel and merging their metrics together can increase the efficiency and speed with which we can achieve the verification goals.

Summarizing the whole verification process of the mixed signal designs, we started from a short specification of a particular part of the device, we wrote a validation plan, where we identified the list of all the features of that part of the device to verify. Then, we wrote assertions and functional covergroups according to the validation plan, implemented MDV concepts in the environment, in order to check if we verified completely all the features of the validation plan.

Chapter 6

Conclusion

In conclusion, this thesis presents a comprehensive approach to efficiently and effectively verify mixed-signal designs using a Universal Verification Methodology (UVM) based Metric-Driven Verification (MDV) environment which includes analog assertions, and real number modeling (RNM) for signal generators.

The thesis discusses the challenges faced in simulation of mixed-signal environments and the use of Cadence tools such as ***Xcelium***, ***Spectre***, and ***Virtuoso*** in the configuration and simulation process of mixed signal designs. The integration of UVM testbenches and MDV techniques with simulation tools is also explored.

Furthermore, the thesis discusses the implementation of several assertion checkers for analog signals, including the frequency checker, amplitude checker, high/low time checker, and rise/fall time checker, and the benefits of using analog assertions. The thesis also presents the implementation of signal generators using RNM, which can provide input stimulus for analog systems during verification.

Lastly, the methodology of MDV is discussed, which involves defining a set of metrics to measure the quality and completeness of the verification process, and the merging of metrics to achieve the desired coverages. By leveraging these techniques, verification engineers can improve the efficiency and effectiveness of the verification process for mixed-signal designs and reduce the risk of design errors.

Overall, this thesis provides valuable insights into the development of efficient and effective verification environments for mixed-signal designs.

Appendix A

Training

The list and description of courses taken during the thesis.

A.1 Course: SystemVerilog for Design and Verification

The course provides a comprehensive introduction to SystemVerilog, is flexible and self-paced, and is taught by experienced instructors. By completing this course, learners will be well-equipped to create effective digital designs and verification environments for their projects. The modules include an introduction to SystemVerilog, data types and operators, control structures, functions and tasks, and object-oriented programming. The course also covers advanced topics such as constrained random testing, coverage-driven verification, and assertions.

A.2 Course: Essential SystemVerilog for UVM

The course covers the basics of SystemVerilog, a hardware description and verification language, and how it can be used in conjunction with UVM to create effective verification environments. One of the key benefits of the Essential SystemVerilog for UVM course is that it provides a comprehensive introduction to both SystemVerilog and UVM, making it ideal for engineers and designers who are new to these technologies. The course is also designed to be flexible and self-paced, allowing learners to study at their own pace and on their own schedule.

A.3 Course: SystemVerilog Assertions

The course offered by Cadence is designed to provide engineers and designers with the knowledge and skills they need to effectively use SystemVerilog Assertions (SVA) in their digital design and verification projects. SystemVerilog Assertions (SVA) Course covers Basic and advanced SVA constructs including sequences, properties, assertions, temporal operators, quantifiers, and hierarchical references. It explains how to use these constructs to specify complex behavior and to reuse SVA code. The course also covers how to use SVA in simulation to verify the behavior of a digital circuit. It explains how to write SVA code in a testbench, how to enable SVA in simulation, and how to debug SVA errors.

A.4 Course: Real Modeling with SystemVerilog

Real Modeling with SystemVerilog (RM-SV) course provides a comprehensive introduction to SystemVerilog modeling, making it ideal for engineers and designers who are new to this technology. The course provides an overview of RM-SV and its capabilities. It covers the syntax and semantics of RM-SV, and how it can be used to model real-world systems. It covers the data types and arrays in RM-SV, including integers, reals, and strings; the tasks and functions in RM-SV, including input/output tasks, blocking and non-blocking assignments, and delays; the classes and objects in RM-SV, including inheritance, polymorphism, and encapsulation; the randomization features in RM-SV, including random variables, distributions, and constraints. Overall, the Real Modeling with SystemVerilog course is an excellent resource for engineers and designers who want to develop their skills in digital modeling using SystemVerilog.

A.5 Course: Mixed-Signal Simulations Using Spectre AMS Designer

The course offered by Cadence is designed to provide engineers and designers with the knowledge and skills they need to effectively use Spectre AMS Designer for mixed-signal simulations. The course covers the basics of Spectre AMS Designer, a simulation tool used for mixed-signal designs. The course provides an overview of Spectre AMS Designer and its capabilities. It covers the features of Spectre AMS Designer, and how it can be used to simulate mixed-signal circuits. The course covers the topics such as, building a mixed-signal simulation, analog simulation, digital simulation, mixed-signal simulation, advanced simulation techniques, The course also covers advanced topics such as noise analysis, Monte Carlo simulations, and statistical analysis.

A.6 Course: Command-Line-Based Mixed-Signal Simulations with the Xcelium™ Use Model

The course covers the basics of the Xcelium™ Use Model, a simulation tool used for mixed-signal designs, and how it can be used to create effective mixed-signal simulations. The modules include an introduction to the Xcelium™ Use Model, mixed-signal simulation basics, analog behavioral modeling, and digital behavioral modeling. The course also covers advanced topics such as noise analysis, Monte Carlo simulations, and statistical analysis. This course covers everything explained by the previous course but using command-line based simulation with Xcelium™ use Model.

A.7 Course: Design Checks and Asserts

The course provides an overview of design checks and asserts and their role in the design process. It covers the types of checks and asserts that can be performed, and how they can be used to improve the quality of the design. The course covers how to set up design checks and asserts in vSPECTRE. It explains how to create design rules, how to set up the rule checks, and how to run the checks on the design. It also covers how to analyze the results of design checks in vSPECTRE. Furthermore, it explains how to interpret the results, how to identify design issues, and how to fix the issues.

A.8 Course: Foundations of Metric Driven Verification

The Course provides an overview of metric-driven verification and its role in the verification process. It covers the benefits of metric-driven verification, and how it can be used to improve the quality of the verification process. The course covers how to set up metrics in the verification environment. It explains how to define metrics, how to collect data, and how to analyze the data. It also covers how to analyze the results of metrics in the verification environment. Furthermore, it explains how to interpret the results, how to identify verification issues, and how to fix the issues. It gives an overview of coverage and its role in the verification process. It covers the types of coverage that can be used, how they can be used to improve the quality of the verification process and how to write coverage models in the verification environment. Furthermore, it explains the syntax and semantics of coverage models, and how to use them to check the behavior of the design. The last section covers how to debug coverage issues in the verification environment. It

explains how to identify coverage gaps, how to isolate the cause of the gap, and how to fix the issue.