Semester Long Internship Report

On

Embedding Ngspice UI in eSim and testing out ATTINY Microcontrollers

Submitted by

Pranav P
MCA
College of Engineering Trivandrum

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

August 15, 2023

Acknowledgement

I wish to express my heartfelt gratitude and appreciation to the entire FOSSEE team for having me as a part of their Semester Long Internship program they offered. The experience and knowledge that I have gained during this internship period is hard to express in words. I have gained tremendous insights about various programming concepts and have met very talented and hard working people as a part of this internship.

I would like express my heartfelt appreciation to my mentors Sumanto Kar sir and Rahul Paknikar sir who have helped me throughout this internship with great patience and diligence. From them I have gained greater insight about eSim and also on how to write better and effective code. Working under them communicating various ideas about the project have helped me gain better grasp at communicating abstract concepts. They have instilled knowledge and confidence throughout this endeavor.

I am very grateful to the whole eSim community for providing me the help needed throughout this internship phase. My opportunity to be at IIT Bombay even for a brief amount of time and the opportunity to meet with several new highly motivated interns is considered as a blessing. I am very much confident that the skills that I have gained through this internship will continue to benefit me throughout my career. Once again, I thank the entire FOSSEE team for the unforgettable opportunity.

Contents

Chapter 1

Introduction

The FOSSEE (Free/Libre and Open Source Software for Education) project encourages the adoption of FLOSS technologies to improve educational quality in our country. It attempts to lessen educational institutions' reliance on proprietary software. It promotes the adoption of FLOSS tools through a variety of initiatives in order to guarantee that commercial software is replaced with equal FLOSS tools. It also creates new FLOSS tools and upgrades old ones to satisfy the needs of academics and research.[1]

The FOSSEE project is part of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Human Resource Development (MHRD), Government of India.

Ngspice[2] is a circuit simulator for mixed-level/mixed-signal circuits. It is built using three open source software packages: Spice3f5, Cider1b1, and Xspice. It's the open source replacement for these venerable programmes. Many changes, bug corrections, and enhancements have been made to the code, resulting in a robust and functional simulator. eSim also uses Ngspice to workout its simulations.

While running simulations on Ngspice, the output graphs associated with the simulation would be displayed by the Ngspice software by default in pop-up windows like in Figure 1.1 even when eSim had its own plotter. So, one of my task was to suppress those plots made by Ngspice. Another task assigned to me was to add some multithreading so that eSim would not freeze during simulation (which used to do). And also I replaced the usage of a seperate terminal emulator to run Ngspice by directly running it in the background and designed some UI to monitor its status from within eSim.

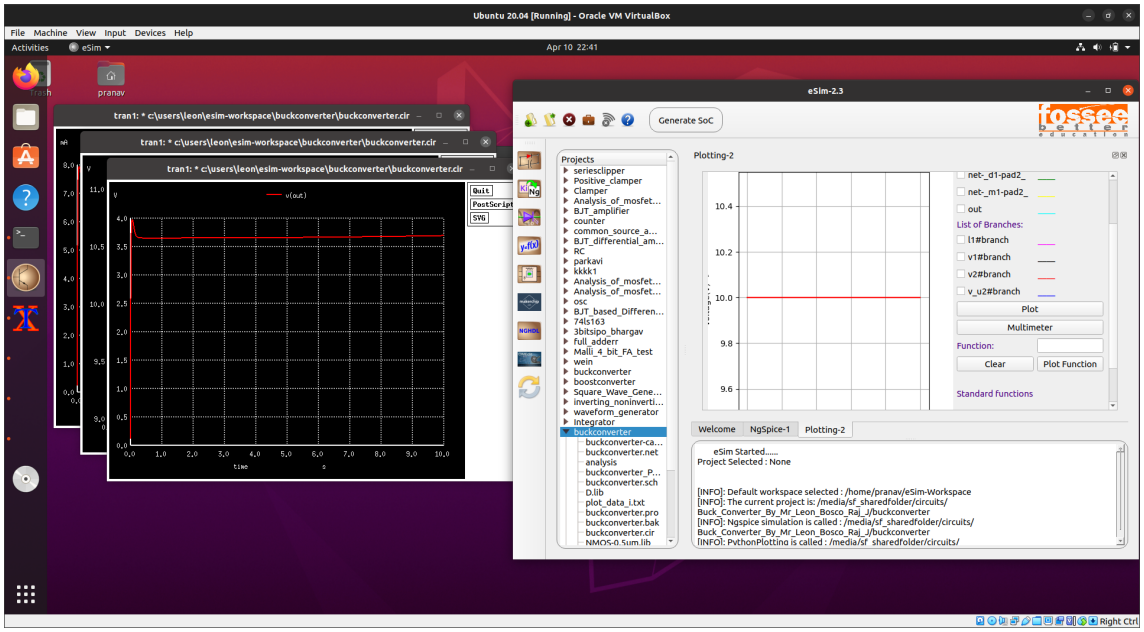

Figure 1.1: pop-up windows from Ngspice on left and eSim's built-in plotter on right

Chapter 2

Suppressing plots and sending Ngspice to background

Some of the major problems that eSim had in association with Ngspice were the pop-up of the plots after the completion of the simulation and the requirement of an external console (XTerm was used previously) to run Ngspice. While these were easy problems to fix, a thorough look at the Ngspice documentation, the existing eSim code and PyQt documentation was required to fix these problems.

2.1 Suppressing plots

One of the things discovered from the documentation of Ngspice was that the default execution of Ngspice was in interactive mode. Ngspice may run in either interactive or batch mode, with interactive being the default. This is a particularly helpful option since it allows you to quickly modify the kind and parameters of an analysis, allowing for faster convergence to a workable simulation.[3]

What struck me was the idea of batch mode. In batch mode the graphics plot output of Ngspice get's suppressed automatically with a warning message. So, simply using batch mode instead of using the interactive mode was sufficient to deal with that problem. To run in batch mode only the '-b' flag had to be added with the Ngspice call.

The argument list is to be changed as follows

```
self.args = ['-b', '-r', netlist.replace(".cir.out", ".raw"), netlist]
```

2.2 Other possible ways

Another way in which this could have been done is by recompiling Ngspice using the *—without-x* flag. With this flag the plot command is disabled. This idea was omitted with thought in mind that the user of eSim might want to use Ngspice separately as well (That is not as a part of eSim). But in case of future needs this is a possible solution to lean into.

2.3 Sending Ngspice to the background

QProcess[4] was the answer for sending Ngspice right into the background. The Xterm was being launched in a QProcess itself. So, removing the arguments of Xterm with that of the Ngspice and replacing the Xterm command with the Ngspice command in the *QProcess.start* call sent Ngspice to the background.

Given below is a part of the code from NgspiceWidget.py which does the Ngspice calling.

```
self.args = ['-b', '-r', netlist.replace(".cir.out", ".raw"), netlist]
self.process = QtCore.QProcess(self)
self.process.setWorkingDirectory(self.projDir)
self.process.setProcessChannelMode(QtCore.QProcess.MergedChannels)
self.process.readyRead.connect(self.readyReadAll)

self.process.finished.connect(
    lambda exitCode, exitStatus:
    self.finishSimulation(exitCode, exitStatus, simEndSignal, False)
)
self.process.errorOccurred.connect(
    lambda: self.finishSimulation(None, None, simEndSignal, True))
self.process.start('ngspice', self.args)
```

Chapter 3

Fixing the eSim freezes during simulation problem

One of the most interesting problems that was to be tackled was the problem that eSim's UI would freeze during simulations. As soon as one would press the simulation button the eSim window will freeze and is not usable until the simulation is completed. Looking deep into the eSim codebase the problem was figured out to be an undesired result of some polling.

3.1 The requirement for the polling and its problem

When Ngspice runs a simulation, as soon as the simulation is completed, it would record its simulation values into two text files called plot_data_I.txt and plot_data_v.txt. So, how eSim used to detect the completion of a simulation was by repetitively checking whether those files were created. So, a polling method was implemented to do this.

Moreover, the *sleep* function from the *time* library was used for this process. PyQt graphical user interface (GUI) applications have a main thread of execution that runs the event loop and GUI. When started the *time.sleep* function on this thread, theGUI will be frozen until the task completes. The user will be unable to interact with the program during this period.

3.2 QTimer - the ineffecient way of doing it

The first idea I got while skimming through the PyQt documentation was one which dealt with the use of QTimer[5]. The QTimer class provides repetitive timers that works with a different thread. This means a separate thread will be doing the polling process. The main thread will be free for the proper execution of the even loop and the user will not experience any issues with the UI.

Even though this approach does work, polling is a very inefficient way of dealing with the problem. Apart from that, checking the created/updated time of the text files dealing with the plots does not seem to be a wise way of checking whether the problem has finished its execution.

3.3 Signals and slots

Signals are notifications that widgets generate when anything happens. That anything might be anything from pushing a button to modifying the wording of an input box or the text of the window. Signal receivers are referred to as slots in Qt. Any function (or method) in your program may be utilised as a slot in Python by simply attaching the signal to it. If the signal delivers data, the receiving function will also receive it. Many Qt widgets also have their own built-in slots, allowing you to connect Qt widgets directly.

Signals and slots[6] are a suitable alternative for our problem as the QProcess which runs the Ngspice process can send signals when its execution is finished and can indicate whether the process finished its execution successfully or some error had been occurred with the help of the exit code and exit status that the process returns along with the signal. This fixed the problem of polling and checking a file to see whether the simulation in finished.

The following code shows how all of the three problems discussed so far was fixed.

Part added in **frontEnd/Application.py**

```
class Application(QtWidgets.QMainWindow):
    """This class initializes all objects used in this file."""
    global project_name
    simulationEndSignal = QtCore.pyqtSignal(QtCore.QProcess.ExitStatus, int)

    def __init__(self, *args):
        """Initialize main Application window."""
        def __init__(self, *args):
        # Flag for mode of operation. Default is set to offline mode.
        self.online_flag = False

        # Set slot for simulation end signal to plot simulation data
        self.simulationEndSignal.connect(self.plotSimulationData)

        # Creating require Object
        self.obj_workspace = Workspace.Workspace()
        self.obj_Mainview = MainView()
        def help_project(self):
        print("Current Project is : ", self.obj_appconfig.current_project)
```

```
        self.obj_Mainview.obj_dockarea.usermanual()

    @QtCore.pyqtSlot(QtCore.QProcess.ExitStatus, int)
    def plotSimulationData(self, exitCode, exitStatus):
        """Enables interaction for new simulation and
           displays the plotter dock where graphs can be plotted.
        """
        self.ngspice.setEnabled(True)
        self.conversion.setEnabled(True)
        self.closeproj.setEnabled(True)
        self.wrkspce.setEnabled(True)

        if exitStatus == QtCore.QProcess.NormalExit and exitCode == 0:
            try:
                self.obj_Mainview.obj_dockarea.plottingEditor()
            except Exception as e:
                self.msg = QtWidgets.QErrorMessage()
                self.msg.setModal(True)
                self.msg.setWindowTitle("Error Message")
                self.msg.showMessage(
                    'Data could not be plotted. Please try again.'
                )
                self.msg.exec_()
                print("Exception Message:", str(e), traceback.format_exc())
                self.obj_appconfig.print_error('Exception Message : '
                                               + str(e))

    def open_ngspice(self):
        """This Function execute ngspice on current project."""
        projDir = self.obj_appconfig.current_project["ProjectName"]

        if projDir is not None:
            projName = os.path.basename(projDir)
            ngspiceNetlist = os.path.join(projDir, projName + ".cir.out")

            if not os.path.isfile(ngspiceNetlist):
                print(
                    "Netlist file (*.cir.out) not found."
                )
                self.msg = QtWidgets.QErrorMessage()
                self.msg.setModal(True)
                self.msg.setWindowTitle("Error Message")
                self.msg.showMessage(
                    'Netlist (*.cir.out) not found.'
                )
                self.msg.exec_()
```

```
                return

            self.obj_Mainview.obj_dockarea.ngspiceEditor(
                projName, ngspiceNetlist, self.simulationEndSignal)

            self.ngspice.setEnabled(False)
            self.conversion.setEnabled(False)
            self.closeproj.setEnabled(False)
            self.wrkspce.setEnabled(False)

        else:
            self.msg = QtWidgets.QErrorMessage()
```

ngspiceSimulation/NgspiceWidget.py

```
import os
from PyQt5 import QtWidgets, QtCore
from configuration.Appconfig import Appconfig
from frontEnd import TerminalUi


# This Class creates NgSpice Window
class NgspiceWidget(QtWidgets.QWidget):

    def __init__(self, netlist, simEndSignal):
        """
        - Creates constructor for NgspiceWidget class.
        - Creates NgspiceWindow and runs the process
        - Calls the logs the ngspice process, returns
          it's simulation status and calls the plotter
        - Checks whether it is Linux and runs gaw
        :param netlist: The file .cir.out file that
            contains the instructions.
        :type netlist: str
        :param simEndSignal: A signal that will be emitted to Application class
            for enabling simulation interaction and plotting data if the
            simulation is successful
        :type simEndSignal: PyQt Signal
        """
        QtWidgets.QWidget.__init__(self)
        self.obj_appconfig = Appconfig()
        self.projDir = self.obj_appconfig.current_project["ProjectName"]
        self.args = ['-b', '-r', netlist.replace(".cir.out", ".raw"), netlist]
        print("Argument to ngspice: ", self.args)
```

```
        self.process = QtCore.QProcess(self)
        self.terminalUi = TerminalUi.TerminalUi(self.process, self.args)
        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.terminalUi)

        self.process.setWorkingDirectory(self.projDir)
        self.process.setProcessChannelMode(QtCore.QProcess.MergedChannels)
        self.process.readyRead.connect(self.readyReadAll)
        self.process.finished.connect(
            lambda exitCode, exitStatus:
            self.finishSimulation(exitCode, exitStatus, simEndSignal, False)
        )
        self.process.errorOccurred.connect(
            lambda: self.finishSimulation(None, None, simEndSignal, True))
        self.process.start('ngspice', self.args)

        self.obj_appconfig.process_obj.append(self.process)
        print(self.obj_appconfig.proc_dict)
        (
            self.obj_appconfig.proc_dict
            [self.obj_appconfig.current_project['ProjectName']].append(
                self.process.pid())
        )

        if os.name != "nt":      # Linux OS
            self.gawProcess = QtCore.QProcess(self)
            self.gawCommand = "gaw " + netlist.replace(".cir.out", ".raw")
            self.gawProcess.start('sh', ['-c', self.gawCommand])
            print(self.gawCommand)

    @QtCore.pyqtSlot()
    def readyReadAll(self):
        """Outputs the ngspice process standard output and standard error
        to :class:`TerminalUi.TerminalUi` console
        """
        self.terminalUi.simulationConsole.insertPlainText(
            str(self.process.readAllStandardOutput().data(), encoding='utf-8')
        )

        stderror = str(self.process.readAllStandardError().data(),
                       encoding='utf-8')

        # Suppressing the Ngspice PrinterOnly error that batch mode throws
        stderror = '\n'.join([errLine for errLine in stderror.split('\n')
                              if ('PrinterOnly' not in errLine and
                              'viewport for graphics' not in errLine)])
```

```
        self.terminalUi.simulationConsole.insertPlainText(stderror)

    def finishSimulation(self, exitCode, exitStatus,
                         simEndSignal, hasErrorOccurred):
        """This function is intended to run when the Ngspice
        simulation finishes. It singals to the function that generates
        the plots and also writes in the appropriate status of the
        simulation (Whether it was a success or not).

        :param exitCode: The exit code signal of the QProcess
            that runs ngspice
        :type exitCode: int
        :param exitStatus: The exit status signal of the
            qprocess that runs ngspice
        :type exitStatus: class:`QtCore.QProcess.ExitStatus`
        :param simEndSignal: A signal passed from constructor
            for enabling simulation interaction and plotting data if the
            simulation is successful
        :type simEndSignal: PyQt Signal
        """

        # Canceling simulation triggers both finished and
        # errorOccurred signals...need to skip finished signal in this case.
        if not hasErrorOccurred and self.terminalUi.simulationCancelled:
            return

        # Stop progressbar from running after simulation is completed
        self.terminalUi.progressBar.setMaximum(100)
        self.terminalUi.progressBar.setProperty("value", 100)
        self.terminalUi.cancelSimulationButton.setEnabled(False)
        self.terminalUi.redoSimulationButton.setEnabled(True)

        if exitCode is None:
            exitCode = self.process.exitCode()

        errorType = self.process.error()
        if errorType < 3:   # 0, 1, 2 ==> failed to start, crashed, timedout
            exitStatus = QtCore.QProcess.CrashExit
        elif exitStatus is None:
            exitStatus = self.process.exitStatus()

        if self.terminalUi.simulationCancelled:
            msg = QtWidgets.QMessageBox()
            msg.setModal(True)
            msg.setIcon(QtWidgets.QMessageBox.Warning)
```

```
        msg.setWindowTitle("Warning Message")
        msg.setText("Simulation was cancelled.")
        msg.setStandardButtons(QtWidgets.QMessageBox.Ok)
        msg.exec()

    elif exitStatus == QtCore.QProcess.NormalExit and exitCode == 0 \
            and errorType == QtCore.QProcess.UnknownError:
        # Redo-simulation does not set correct exit status and code.
        # So, need to check the error type ==>
        #   UnknownError along with NormalExit seems successful simulation

        successFormat = '<span style="color:#00ff00; font-size:26px;">\
                    {} \
                    </span>'
        self.terminalUi.simulationConsole.append(
            successFormat.format("Simulation Completed Successfully!"))

    else:
        failedFormat = '<span style="color:#ff3333; font-size:26px;"> \
                    {} \
                    </span>'
        self.terminalUi.simulationConsole.append(
            failedFormat.format("Simulation Failed!"))

        errMsg = 'Simulation '
        if errorType == QtCore.QProcess.FailedToStart:
            errMsg += 'failed to start. ' + \
                      'Ensure that eSim is installed correctly.'
        elif errorType == QtCore.QProcess.Crashed:
            errMsg += 'crashed. Try again later.'
        elif errorType == QtCore.QProcess.Timedout:
            errMsg += ' has timed out. Try to reduce the ' + \
                      ' simulation time or the simulation step interval.'
        else:
            errMsg += ' could not complete. Try again later.'

        msg = QtWidgets.QErrorMessage()
        msg.setModal(True)
        msg.setWindowTitle("Error Message")
        msg.showMessage(errMsg)
        msg.exec()

    self.terminalUi.simulationConsole.verticalScrollBar().setValue(
        self.terminalUi.simulationConsole.verticalScrollBar().maximum()
    )
```

```
simEndSignal.emit(exitStatus, exitCode)
```

During simulation, in the UI, some buttons like open schematic, convert KiCad to Ngspice, Simulate etc. are disabled to ensure lesser faults during the simulation. This also is present in the above given codes.

Chapter 4

UI to monitor simulations

Now since we have removed the role of the xterm to run our ngspice there should be a way to display the messages sent by Ngspice to the user. For that purpose and to control some aspects of the Ngspice simulation a UI was written.

4.1 How output is taken from the process

The following line of code can be used to obtain a string containing what the Ngspice process returns to the standard input.

```
str(self.process.readAllStandardOutput().data(), encoding='utf-8')
```

The following line of code can be used to obtain a string containing what the Ngspice process returns to the standard error.

```
str(self.process.readAllStandardError().data(), encoding='utf-8')
```

Now these strings can be piped to our custom UI where these information can be displayed.

4.2 Code for building the UI

frontEnd/TerminalUi.py

```
from PyQt5 import QtCore, QtGui, QtWidgets, uic
import os


class TerminalUi(QtWidgets.QMainWindow):
    """This is a class that represents the GUI required to provide
    details regarding the ngspice simulation. This GUI consists of
    a progress bar, a console window which displays the log of the
    simulation and button required for re-simulation and cancellation
    of the simulation"""
    def __init__(self, qProcess, args):
```

```
        """The constructor of the TerminalUi class
        param: qProcess: a PyQt QProcess that runs ngspice
        type: qProcess: :class:`QtCore.QProcess`
        param: args: arguments to be passed on to the ngspice call
        type: args: list
        """
        super(TerminalUi, self).__init__()

        # Other variables
        self.darkColor = True
        self.qProcess = qProcess
        self.args = args
        self.iconDir = "../../images"

        # Load the ui file
        uic.loadUi("TerminalUi.ui", self)

        # Define Our Widgets
        self.progressBar = self.findChild(
            QtWidgets.QProgressBar,
            "progressBar"
        )
        self.simulationConsole = self.findChild(
            QtWidgets.QTextEdit,
            "simulationConsole"
        )

        self.lightDarkModeButton = self.findChild(
            QtWidgets.QPushButton,
            "lightDarkModeButton"
        )
        self.cancelSimulationButton = self.findChild(
            QtWidgets.QPushButton,
            "cancelSimulationButton"
        )
        self.cancelSimulationButton.setEnabled(True)

        self.redoSimulationButton = self.findChild(
            QtWidgets.QPushButton,
            "redoSimulationButton"
        )
        self.redoSimulationButton.setEnabled(False)

        # Add functionalities to Widgets
        self.lightDarkModeButton.setIcon(
            QtGui.QIcon(
```

```
                os.path.join(
                    self.iconDir,
                    'light_mode.png'
                )
            )
        )
        self.lightDarkModeButton.clicked.connect(self.changeColor)
        self.cancelSimulationButton.clicked.connect(self.cancelSimulation)
        self.redoSimulationButton.clicked.connect(self.redoSimulation)

        self.simulationCancelled = False
        self.show()

    def cancelSimulation(self):
        """This function cancels the ongoing ngspice simulation.
        """
        self.cancelSimulationButton.setEnabled(False)
        self.redoSimulationButton.setEnabled(True)

        if (self.qProcess.state() == QtCore.QProcess.NotRunning):
            return

        self.simulationCancelled = True
        self.qProcess.kill()

        # To show progressBar completed
        self.progressBar.setMaximum(100)
        self.progressBar.setProperty("value", 100)

        cancelFormat = '<span style="color:#FF8624; font-size:26px;">{}</span>'
        self.simulationConsole.append(
            cancelFormat.format("Simulation Cancelled!"))
        self.simulationConsole.verticalScrollBar().setValue(
            self.simulationConsole.verticalScrollBar().maximum()
        )

    def redoSimulation(self):
        """This function reruns the ngspice simulation
        """
        self.cancelSimulationButton.setEnabled(True)
        self.redoSimulationButton.setEnabled(False)

        if (self.qProcess.state() != QtCore.QProcess.NotRunning):
            return

        # To make the progressbar running
```

```
        self.progressBar.setMaximum(0)
        self.progressBar.setProperty("value", -1)

        self.simulationConsole.setText("")
        self.simulationCancelled = False

        self.qProcess.start('ngspice', self.args)

    def changeColor(self):
        """Toggles the :class:`Ui_Form` console between dark mode
                        and light mode
        """
        if self.darkColor is True:
            self.simulationConsole.setStyleSheet("QTextEdit {\n \
                background-color: white;\n \
                color: black;\n \
            }")
            self.lightDarkModeButton.setIcon(
                QtGui.QIcon(
                    os.path.join(
                        self.iconDir,
                        "dark_mode.png"
                        )
                    )
                )
            self.darkColor = False
        else:
            self.simulationConsole.setStyleSheet("QTextEdit {\n \
                background-color: rgb(36, 31, 49);\n \
                color: white;\n \
            }")
            self.lightDarkModeButton.setIcon(
                QtGui.QIcon(
                    os.path.join(
                        self.iconDir,
                        "light_mode.png"
                        )
                    )
                )
            self.darkColor = True
```

4.3 Components in the simulation monitor

The UI contains primary a console to display the messages sent by the Ngspice process during its execution. There is a progress bar showing whether the execution is being held or if it Is finished. Then there is a Resimulate button which when pressed would re-run the simulation. Cancel Simulation button is to be toggled during the simulation process if one wants to stop the simulation in between. Then there is a button to toggle between light and dark mode which changes the colour of the console accordingly.

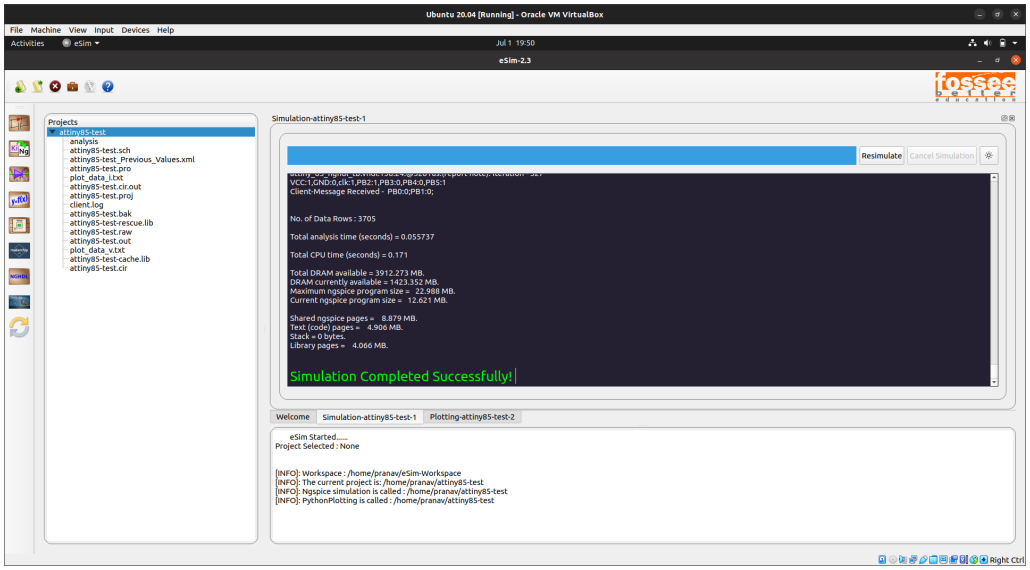

Figure 4.1: The designed UI to monitor Ngspice simulation

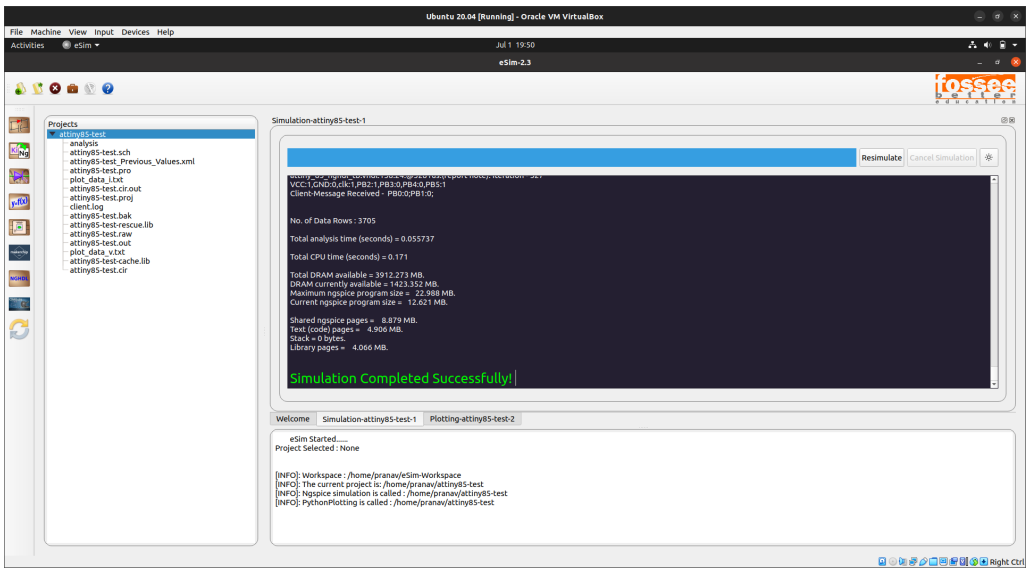

Figure 4.2: The designed UI to monitor Ngspice simulation

4.4 Some additional UI changes

As a part of making each and every tab in the docker clearer a new naming convention was also brought.

Figure 4.3: The designed UI to monitor Ngspice simulation

4.5 An additional point for Windows users

Note: In the Windows operating system, for the console in the GUI to display the output properly, one must recompile Ngspice with the *–without-x* flag.

Chapter 5

Verification of ATtiny 85 microcontroller

There were some extra tasks which I was assigned with. One of them was to verify the 'Implementation of GUI Interface and Simulation of Multiple Instance of Attiny Microcontrollers'[7] which was a work done by a former FOSSEE intern named Vatsal Patel. What was to be done here was to recompile nghdl with an Attiny microcontroller support. The one I chose was the Attiny 85 Microcontroller. After the compilation I had to draw a circuit with the Attiny microcontroller instance in it and then feed in the .hex file generated using the source code related with the concerned project and running the simulation once after the conversion from KiCad to Ngspice.

5.1 Recompiling NGHDL

The appropriate files required for the recompilation was provided by Rahul Sir. The following files were to be placed at the given locations

1. attiny_85_nghdl.zip $\Longrightarrow$ /nghdl-simulator/src/xspice/icm/ghdl/ (NGHDL files)
2. modpath.lst $\Longrightarrow$ /nghdl-simulator/src/xspice/icm/ghdl/ (NGHDL file)
3. eSim_Nghdl.lib $\Longrightarrow$ /usr/share/kicad/library/ (schematic symbol)
4. attiny_85_nghdl.xml $\Longrightarrow$ eSim-2.3/library/modelParamXML/Nghdl/ (required for passing params)

After placing the above files, you need to update the paths in all the files */nghdl-simulator/src/xspice/icm/ghdl/attiny_85_nghdl* according to your machine. Finally, go to */nghdl-simulator/release/src/xspice/icm/*, open a terminal and run: **make && make install** to install the microcontroller.

5.2 Testing the microcontroller

I was provided with a test circuit for testing. The .hex file was also provided and I too generated one using the Arduino IDE. The results are shown below

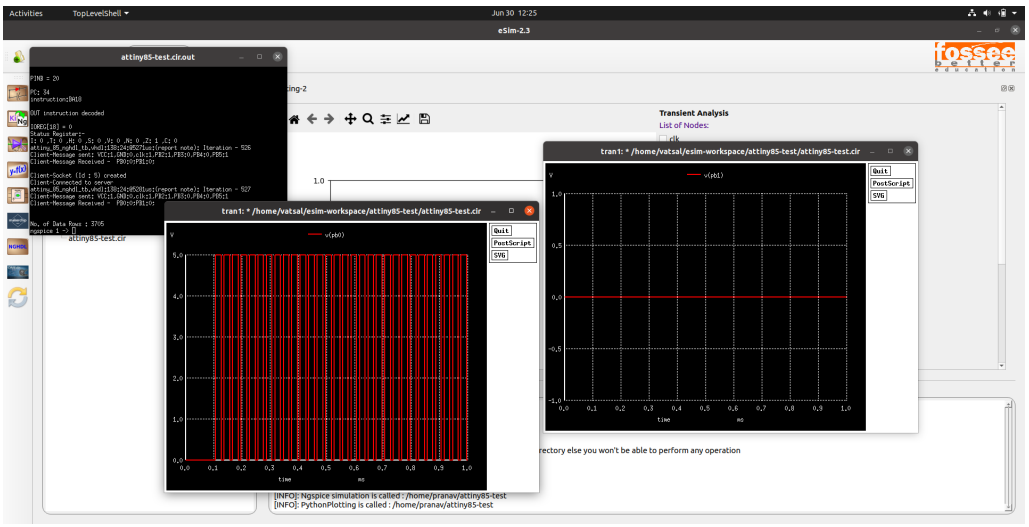

Figure 5.1: Multiplexer using Attiny 85

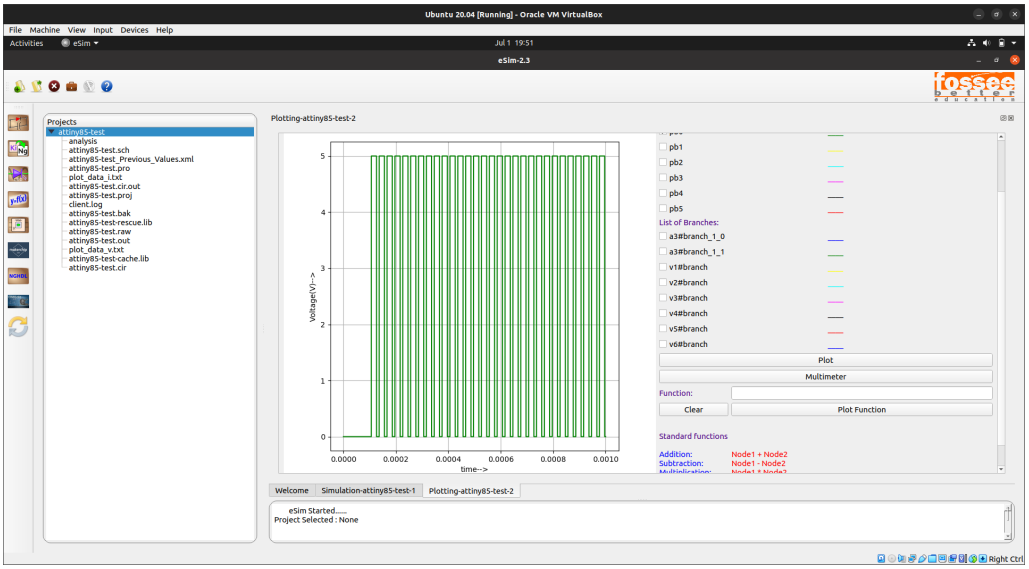

Figure 5.2: Multiplexer using Attiny 85

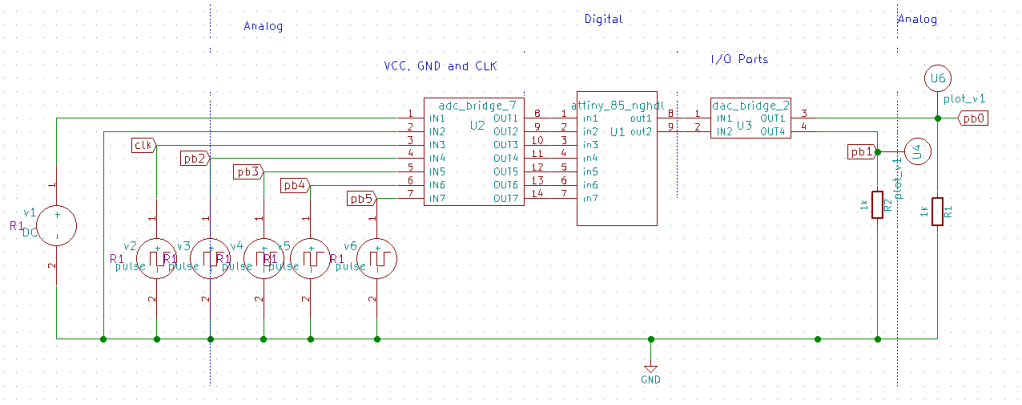

Figure 5.3: Multiplexer circuit with Attiny 85

5.3 Problem faced with Windows

With Windows, I was unable to successfully compile the microcontroller. Even after editing a lot of the makedefs and finally being able to compile it, the Attiny 85 instance was not being identified by NGHDL.

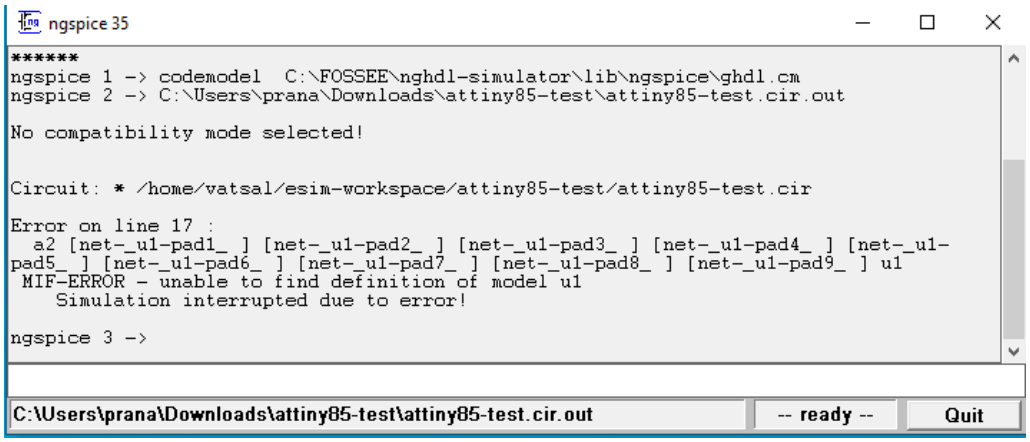

Figure 5.4: Error in Windows

Chapter 6

Verifying circuits

6.1 Initial set of circuits

During the beginning of the internship, I was asked to verify whether a list of circuits listed out by K.J.Somaiya College of Engineering , Vidyavihar, were able to be simulated using eSim. Most of the circuits listed by them were already present in the Completed Circuit Simulations in the eSim website[8]. I almost verified around 20 circuits.

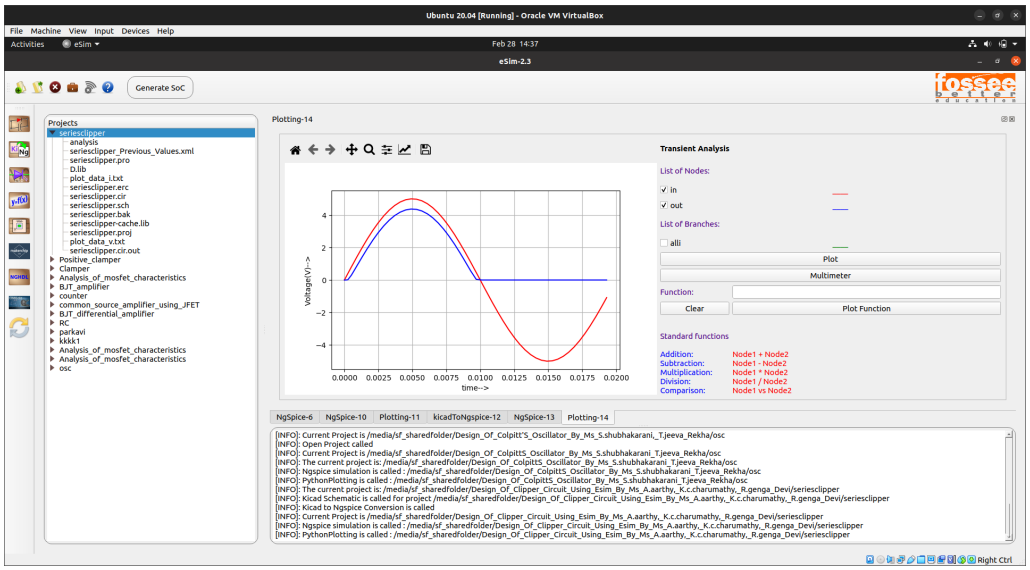

Figure 6.1: Series Clipper

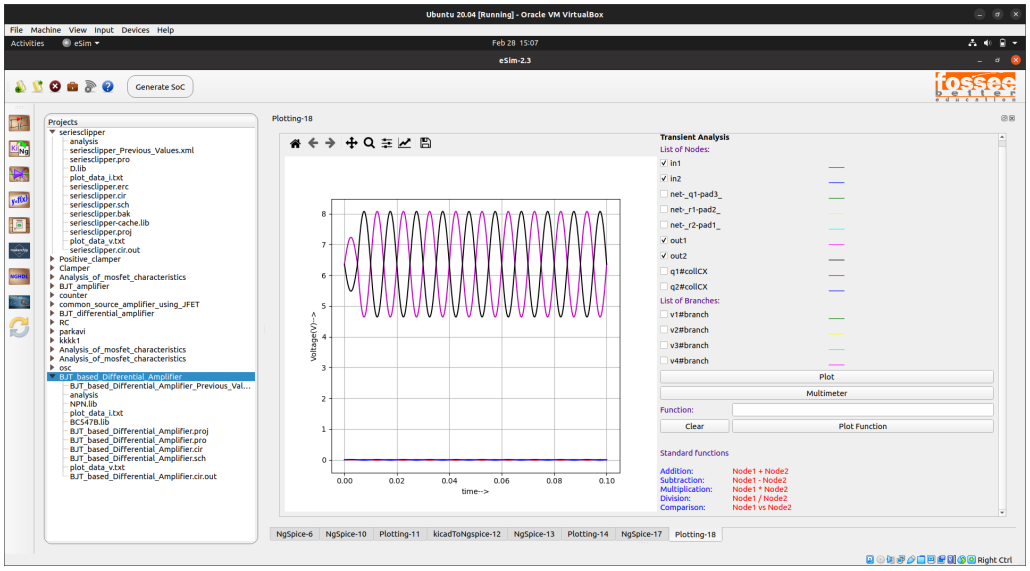

Figure 6.2: BJT based Differential Amplifier

6.2 Arduino on cloud circuits

Another small task assigned during the time I was at IIT Bombay along with the Summer fellows was to verify a series of circuits which is to be built with the arduino UNO using the cloud platform developed by the Arduino on Cloud team. The feedback that we provided were used by the Arduino on Cloud team to fix further bugs and to prepare documentations.

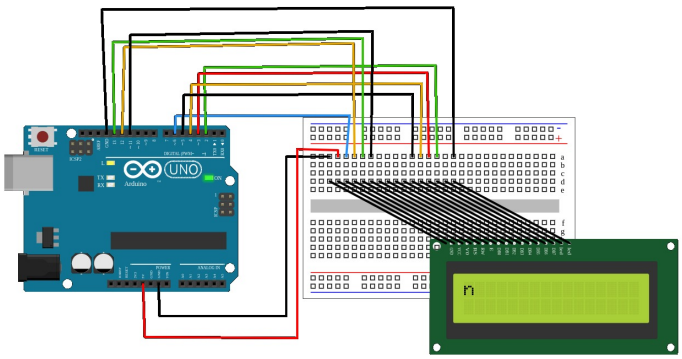

Figure 6.3: LCD displaying scrolling text

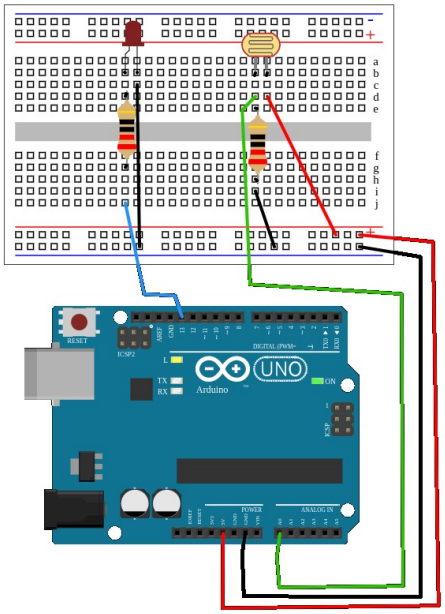

Figure 6.4: Streetlight that switches on and off based on sunlight

Chapter 7

Conclusion and Future Scope

I was fortunate enough to finish the issues related to Ngspice simulations. I tested the simulations with multiple circuits with multiple parameters making sure that things would not break apart. The lack of knowledge of using a debugger caught me here because I had to test out all of those caes manually. It would have been a whole lot easier with the testing part if I was comfortable with automated testing. Though my task with the microcontroller verification was not completed successfully.

Also, there are some areas where improvements can be made. One such area is the documentations section. A lot of the code present in the eSim code base is not written in a manner that is compatible with the Sphinx docstrings[9]. Another one is the fact that we cannot run multiple Ngspice simulations at the same time. For the time being I have blocked such actions because of potential crashes. But if someone could come up with a good idea to run multiple simulations at the same time then that would be a great addition. Another thing that has to be done is to package the Windows version of eSim with an Ngspice compiled with the *–without-x* flag.

Bibliography

[1] FOSSEE official website
URL: https://fossee.in/about

[2] Ngspice official website
URL: https://ngspice.sourceforge.io

[3] NGSpice: Interactive Quick Reference
URL:https://tomwwolf.wordpress.com/modeling-simulation/ngspice-interactive-reference/

[4] QProcess reference
URL: https://doc.qt.io/qtforpython-5/PySide2/QtCore/QProcess.html

[5] QTimer reference
URL: https://doc.qt.io/qtforpython-5/PySide2/QtCore/QTimer.html

[6] Slots & Signals reference
URL:https://www.pythonguis.com/tutorials/pyqt-signals-slots-events

[7] Implementation of GUI Interface and Simulation of Multiple Instance for Attiny Microcontroller
URL: https://static.fossee.in/fossee/FOSSEE-Summer-Fellowship2022/eSim/FSF_2022_VATSAL.pdf

[8] Completed Circuit Simulations
URL:https://esim.fossee.in/circuit-simulation-project/completed-circuits

[9] Sphinx docstring documentation
URL:https://sphinx-rtd-tutorial.readthedocs.io/en/latest/docstrings.html