Semester's Project

Propagation Delay measurement of VHDL Components Using TCL Scripting

Viktoria Biliouri
Maria Ramirez Corrales

January 25, 2021

Contents

1 Introduction

In terms of this project, we implemented a process that measures the propagation delay of VHDL components, by creating a Python script for the insertion and execution of a TCL Script in Vivado Design Suite. After the execution of this Python script, Vivado processes the TCL script in batch mode, and the propagation delay returned by Vivado, is stored in a csv file, in order to be used for the timing analysis of the results.

Further details for this process, are given in the next chapters. First, we will analyze the construction of the TCL script that is used from Vivado. Secondly, the following information concerns the structure of the full Python script that is responsible for the initialization and the final steps of the process. Afterwards, we will see the plotting results, extracted after executing the script. And finally, we will propose a mathematical model for the calculation of the propagation delay of our VHDL component of interest, which is an vhd file, that described the behaviour of an adder.

In terms of this project, our work depends on a repository of vhd files created by the doctorate candidate Rémi Parrot. However, the most important part of our script is parametric, and with slight changes, it can be functional for the measurement of propagation delay of any vhd component, if the guidance analyzed in the last chapter, is followed.

The most critical test of the latency described in this report, is the analysis of a specific vhdl file, the Adder, so the last chapter is devoted to this exact vhdl code.

This project was implemented under the supervision of Professor Mikael Briday and the guidance of Rémi Parrot, within the curriculum of the Master 2 - Control and Robotics - Embedded Real-Time Systems, held by Ecole Centrale de Nantes.

2 TCL

2.1 General Information

TCL (Tool Command Language) is a language used to interface with a large variety of design tools like Vivado Design Suite. It is a very common language for industrial designers as it can interact with widely used industrial software.

It counts with some general commands to read and write files in the system or create folders and add files to it or to a project. It is also very flexible for the implementation of designs and access to object's properties. Another interesting characteristic is the ease it has to customize reports after the analysis of the designed systems. Therefore, those functionalities can easily be extended with some customized functions. Xilinx has created libraries with Vivado Design Suite commands that ease the labor of programmers.

It is a simple language as in an instruction, the first word identifies the command and the next words are the arguments. Moreover, commands can be nested within others in brackets.

This language is that powerful that just launching the TCL script in the command prompt, we can access to Vivado Desing Suite and execute a design without opening its graphical interface. This means, we can execute a TCL file in batch mode.

In Vivado we can execute TCL instructions in the TCL console, where we can also have access to the track of the TCL instructions that we manually executed in the graphical interface. This capacity made easier our work in the project. However, we used [1] and [2] to search for the functions that we additionally needed.

2.2 TCL script

Seen the variety of functionalities that TCL has by executing it Vivado, we created a TCL script to generate the critical path delay in a vhd object.

First, the script creates a new project in Vivado in the specified path folder, setting VHDL as main language. It adds the desired vhd file to the project and it sets it as the main source. Moreover, it adds the Arty-A7-35 constraints file as it is the used FPGA for the project.

Timing constraints cannot be set before having implemented at least once the whole design, so, the next instruction is launching the synthesis and the implementation of the vhd file. After executing both the synthesis and the implementation, some delay (some minutes) is added because if not, Vivado will execute next instructions without having finished them and this will lead to a fail in the execution.

Once those process are finished, it creates a new timing constraints file and it adds it to the project. Then, it configures the delay that we want to measure thanks to the inputs and outputs of the hvd module (instruction *set_max_delay*).

Now, we can measure the delay, so we can re-launch the synthesis and implementation of the project with the new file with timing constraints. It refreshes the design and erases the timing constraints file (because if not, the next execution of the script will fail). Finally, it exports the timing report (a csv file) where the desired data is stored with the name *SLACK*.

3 Python Scripting

All the functionalities analyzed into this report, are concentrated and managed by a Python script. The sub-processes that are integrated in our Python code, aim to build an independent tool, for the measurement of the propagation delay, based on a set of vhd files. Apart from the different kind of vhd components, we propose a way of alternating the current vhd files, in order to function with different number of bits in their input and output ports. To achieve this functionality, we insert to the script, a vhd template corresponding to the component we want to examine, and not inserting the exact vhd file. Thus, the user, can specify the number of input and output ports, so as to be able to examine the file in different cases, automatically.

When the measurement of the propagation delay has finished, we proceed with an analysis of the results, by saving them to a csv file, and plotting them. The next paragraphs, contain more particular information, concerning the construction of the script. In Figure 1, you can see a general scheme with the flow of the procedure:

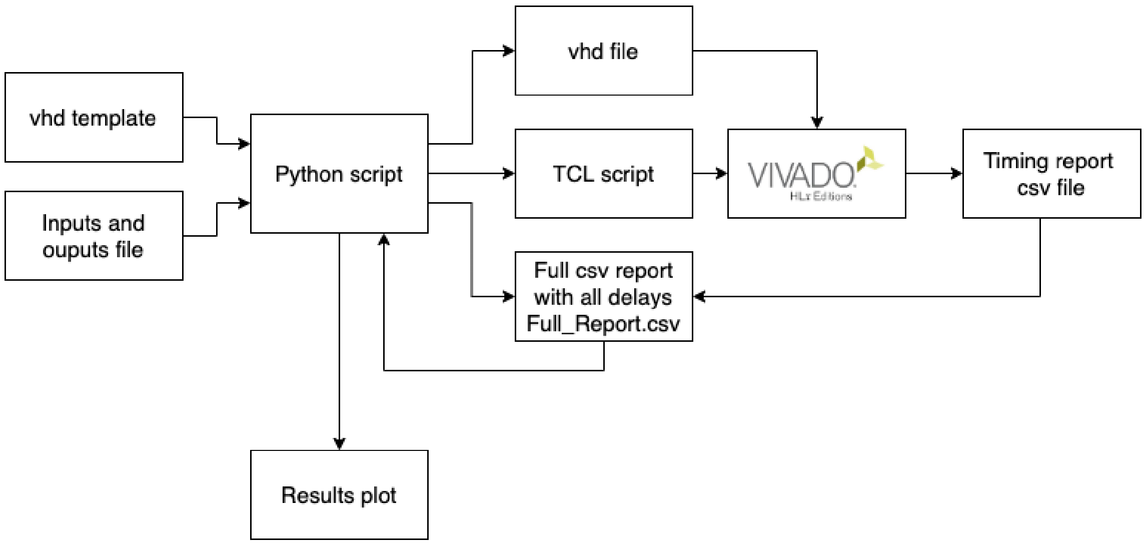

Figure 1: General scheme of the project

3.1 VHDL templates

For the execution of our script, we need to insert as arguments, a vhd file (or template), and the location for the storage of the files created by Vivado. As we chose to automatically change the bit width of the I/O ports, we are not able to use ready vhd components. For this reason, we needed to construct VHDL templates based on each specific vhd component, which contains understandable for our script, keywords, that should be modified in order to cover the needs of each execution. These keywords, describe the values that we want to change and they are introduced by the character "@" before and after the word of interest. Except for the character "@", we introduced some keys, which demonstrate to the Python code, the correspondence of the values that we want to add, with the ones that we want to replace.

For instance, for the adder VHDL component, we have two inputs (*in1* and *in2*), and one output (*out1*), and we want to insert the bit width of these ports to the template, by replacing the keywords. The template for the n-bits adder is shown below:

```
entity Add_3 is
  Port (
        in1 : in STD_LOGIC_VECTOR (@in1_minus@ DOWNTO 0);
        in2 : in STD_LOGIC_VECTOR (@in2_minus@ DOWNTO 0);
        out1 : out STD_LOGIC_VECTOR (@out1_minus@ DOWNTO 0)
        );
end Add_3;

architecture Behavioral of Add_3 is

begin
  out1 <= std_logic_vector(0 + signed(in1) + signed(in2));
end Behavioral;
```

In the example shown above, we can see that our keywords are *@in1_minus@*, *@in2_minus@*, and *@out1_minus@*, which signify the number of bits for each port minus 1 (as if we have 8 bits for a port, the values it can get is from 0 to 7).

The keywords of each template depend on the vhd file that we want finally to be created and executed by Vivado, so the different keywords can have slight differences in each template.

After the insertion of the template, as arguments of the script, we need to realize the modifications mentioned before, by constructing a function that aims to the creation of a new vhd file, based on the template and on the number of inputs/outputs we want. This function is called *change_vhd_ports*, and it takes as inputs, the number of bits for the different inputs and outputs, and it produces a new vhd file, whose number of bits is also shown on its name. For example, if we want an adder for 8-bits inputs, the new VHDL file created, will have the name *Add_3-8.vhd.* Although this method of storage, does not seem to be important, it is proved to be useful for the next steps, and for this reason the function returns this exact name to the *main.* Our next question is, how does the user specify the desired number of bits for the I/O ports?

3.2 Inputs/Outputs

The response to the above question is given in this paragraph. The script is designed to search the information concerning the inputs' and the outputs' bit width, in an external text file called *ports.txt.* The user should provide the script with this file, in order to be able to open it and obtain the necessary information. The format of the *ports.txt*, is a specific format, containing the name of the template written in VHDL, all the input and outputs, analytically named.

After the recognition of this file, we have defined a small process that trucks the number of the inputs and outputs provided, and the number of their bits. The results are stored in variables which are inserted as arguments in the function that is responsible to compose the new vhd file (*change_vhd_ports*). For example, when the *ports.txt* consists of the text below

```
Add_3.vhd input: {in1[0] in1[1] in1[2] in1[3]
                  in2[0] in2[1] in2[2] in2[3]}
          output: {out1[0] out1[1] out1[2] out1[3]}
```

our code will:

- find the name of the examined file (*Add_3*)
- recognize the input and output ports (*in1, in2, out1*)
- and finally, count the number of their bits (4 for *in1*, 4 for *in2* and 4 for *out1*)

Attention should be paid in the correct form of the VHDL file. In other words, we should take into consideration which is the relation between the number of bits for inputs-outputs, in order to produce a synthesizable and proper vhd component. For instance, the results of the execution of the adder are valid only if the inputs and outputs ports are of the same bit number. Otherwise either we will have a failure in the execution of Vivado (synthesis failure), which can understand syntax mistakes, or we will have faulty results. In both cases, we should determine the problem we want to check and not randomly testing numbers,

3.3 TCL generation

Once the template is generated, we call the *generateTclFile* function, which is responsible for creating the TCL file named *new_script.tcl*, with the functionality described in the previous chapter. If this TCL script already exists, its contents are overwritten, as we want it to be alterated in accordance to the needs of each execution. After the generation of the TCL, we source Vivado in batch mode, with input the *new_script.tcl*, and we are able to see in our console the *log* of the execution of the Vivado process (*synthesis, implementation, place and routing*, etc).

Additionally, the *new_script.tcl*, exists in our local folder, after the end of the script's run. We decided to leave it produced (originally for testing purposes), but also for giving the opportunity to the user to use it or test it locally into Vivado (TCL console provided in Vivado Design Suite), in case one's using the Vivado GUI for the modification of the vhd components.

3.4 Script's output

As long as the above-mentioned procedures are finished, we have one last mission: storing the results in a proper way, to be easily used by the user. In addition to this, we added an extra feature to our code which is that after each execution, we collect all the data produced by both the last and of all the previous executions (regarding the component we want to examine) and we plot them. In that way, we are able to comprehend the behavior, and be in the position to decide, whether the propagation delay results can be modeled or not. But lets analyze the final part of our project step by step.

First of all, as mentioned above, the timing results, obtained by the execution of Vivado for a VHDL file, are stored into a *csv* file, called *delay.csv*. This report contains, among others, the value of *SLACK*, which is a negative number that represents the value of the delay that lead to the missing of the timing constraints we set by the tcl script.

When the *SLACK* value has been obtained, we create a new *csv* file that contains the results of all the executions of the script. This file is called *Final_Report.csv*, it is created if it does not exist, but if it does exist, its contents are preserved and not overwritten among the different executions. In that way, we have stored all the values of the latency of different VHDL components. The first data written to the report is the name of the VHDL file, used in Vivado, with the index of the number of the input bit number, and then the value of the latency.

Before the storage of a new value in the *Final_Report.csv*, there happens a check in the report, concerning a previous execution of the exact same VHDL file. If these data already exist, we replace the delay value with the new one, otherwise, the data of the current file are stored in a new line in the end of the report.

Finally, we have defined a last function, called *plot_results*, it takes as input the *Final_Report.csv* and the name of the vhd component, and plots the result of delay obtained in the current execution, as well as all the previous data for the same component. In other words, it creates a plot with the different delays obtained when running the same template with different bit size input and output ports, the x-axis contain the bits' number and the y-axis the delay value.

The next chapter is devoted to the study of the results of one specific component, the n-bit Adder.

4 Adder Analysis

The last part of this project is the study of one specific vhd component, which is the VHDL code of an adder. The template we created for the adder, permits the modification of the number of inputs, in order to examine its behavior, regarding its performance. The restriction, in order for an adder to be functional, is that the number of bits of the two inputs and of the output, should be constantly the same. For this reason, when adding or removing bits in the *ports.txt*, we should do it to all of the three ports of the VHDL code.

4.1 Plotting Results

The first step to comprehend and analyze the performance results, is to plot our data. Each execution of the adder, adds an extra point into its diagram, so, the more numerous the results are, the more accurate our conclusion could be. Thus, we executed the Python script, with the adder template as input, and with the bit-number firstly from 1 to 32 bits and secondly from 1 to 50 bits.

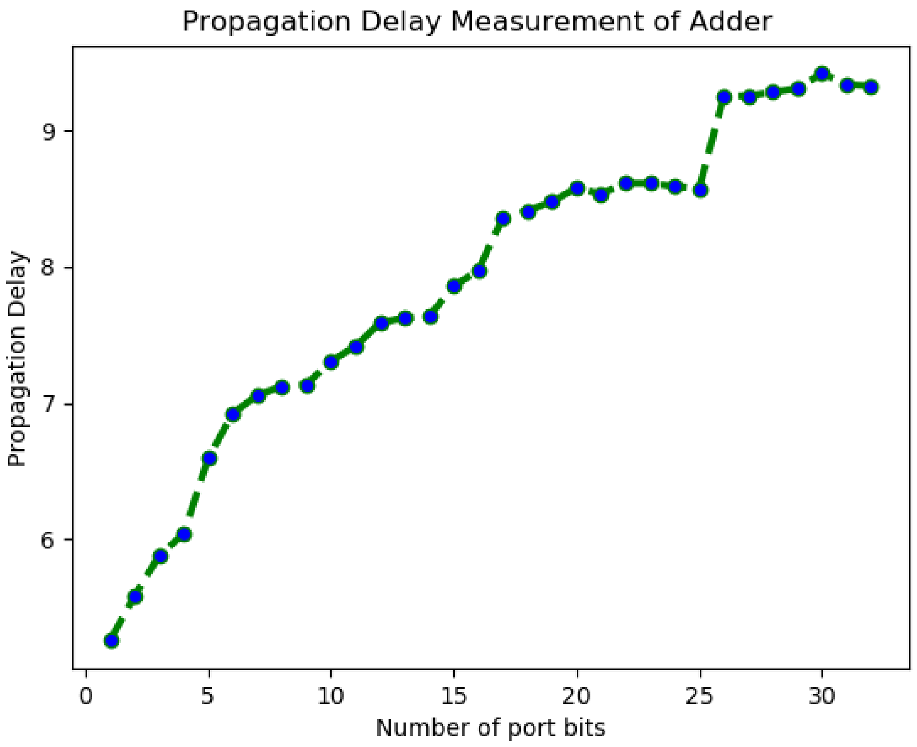

Figure 2: Delay for 1 to 32 bits

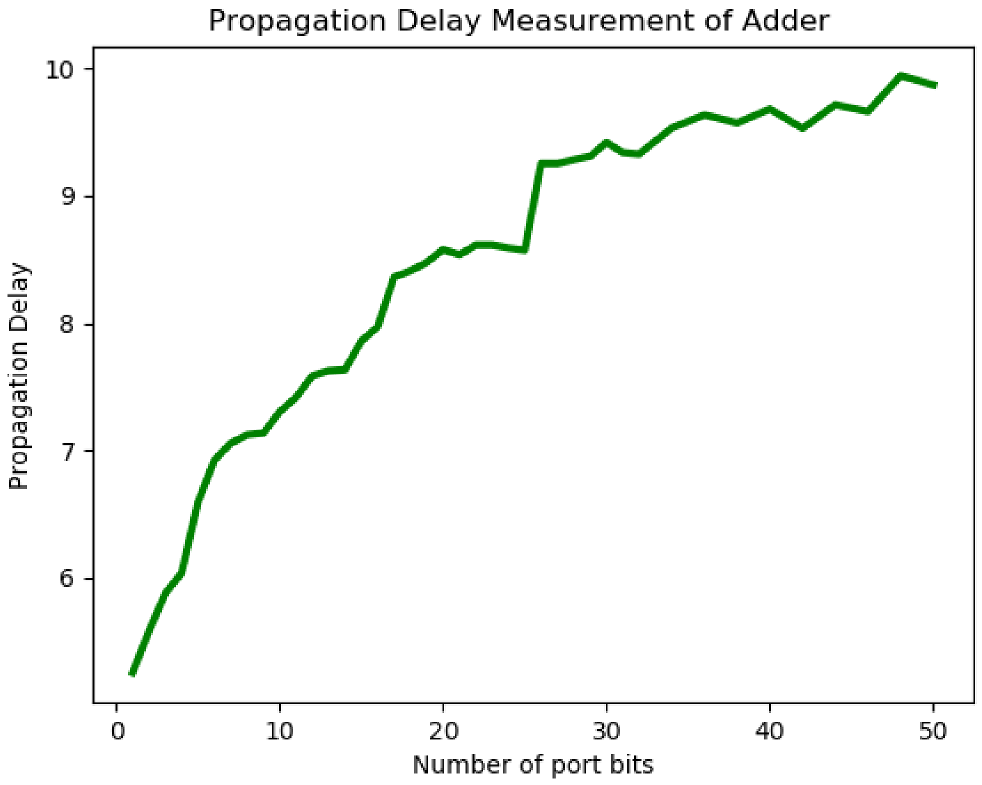

Figure 3: Delay for 1 to 50 bits

4.2 Modeling

From the above graphs, we can see that the results have a constant rising tendency, but they also show a complexity to be modeled. However, according to the mathematical method of regression and some tests to a calculation machine, we concluded to two mathematical equations that can model the results for the *Adder* vhd component.

First, and the simplest, we have noticed a linear equation that shows similarities in the behavior of the points that we have collected. This linear model is described by the equation:

$$f(x) = 0.104103x + 5.85007$$

where f(x) is the delay and x the number of bits.

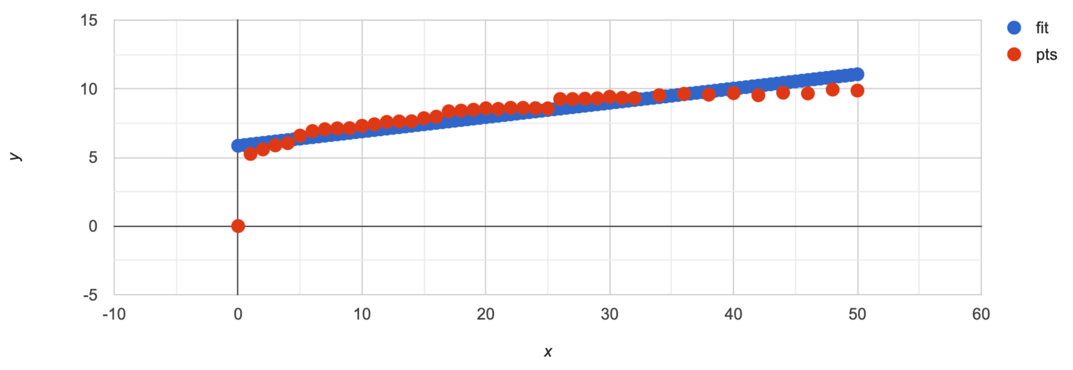

Figure 4: Linear Model of the Delay

Additionally, the calculation of a more complex model, seems to represent our data with greater accuracy, but yet not completely properly. The second model we propose is described of the above equation:

$$f(x) = 1.33019 ln(1.7833x - 2.37093) + 3.87062$$

where f(x) is the delay and x the number of bits.

This model has derived from a digital calculation, using a specific calculation machine, for the modeling of graph results.

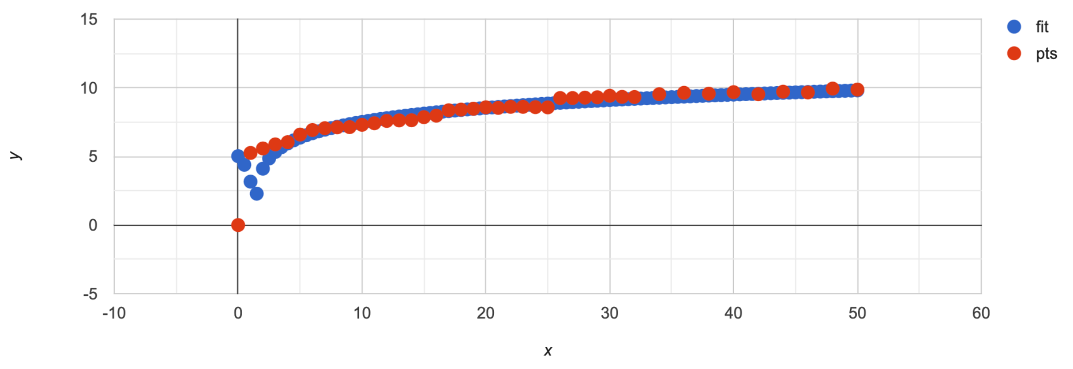

Figure 5: Logarithmic Model of the Delay

5 Execution Manual

The following information, is a guide in order to execute our project for the measurement of the propagation delay of the proposed VHDL files, or for the transformation of the script in order to cover your specific needs.

Contents:

- *pd_script.py*: the full process of the TCL script creation and the extraction of the propagation delay results.
- *ports.txt* : the input and the output ports of the vhd components we want to examine. You should pay attention in the path of this file. We recommend that it is stored in the same location as *pd_script.py*
- *VHDL_SOURCE_FILES* folder: contains all the VHDL templates of the vhd components whose propagation delay we want to measure (most important the *Add_3.vhd*)
- *Final_Report.csv* : You should use this file only if you want to plot the Adder-3 results for different bit number directly. Otherwise, it is not necessary - the script will create a new one.

In order to execute the Python script:

- Modify the input and output ports (add or reduce input and output bits) in the *ports.txt* as desired (the current input and output ports correspond to our measurement needs).
- Source Vivado from the comand prompt, using the command:
  - for Linux: $ `source /opt/Xilinx/Vivado/20XX.X/settings64.sh`, where XX.X is your Vivado edition.
  - for Windows: $ `C:/Xilinx/Vivado/20XX.X/settings64.bat`, the path can be modified depending on your Vivado installation folder. XX.X is your Vivado edition, as well.
- put the correct arguments in the script's execution command:
  - `argv[1]` : the full path of your vhd file
  - `argv[2]` : the path where you want to store the Vivado Project. Attention should be paid in the path given in `argv[2]`. We recommend that this path is the same with the Python script's full path.
- make sure you have installed the *matplotlib* library for Python.

After the execution:

First of all, the timing results of the file you want to check, will be plotted.

You will notice a new folder *projectMV* which contains the Vivado Project (*.xpr*) and all the files created by Vivado.

- the `projectMV\projectMV.srcs\reports\delay_info.csv` is the timing report

Also there will be in your local folder the TCL script named *new_script.tcl* which runs on Vivado.

In the folder with the *VHDL_SOURCE_FILES* you will notice a new vhd file, created from the vhd template according to the number of inputs' and outputs' bits defined in the *ports.txt*. This vhd file is the one that is executed by Vivado and its propagation delay is outputed in *Final_report.csv*. The number added to the name of the vhd file, declares the number of bits in the input of the vhd file.

Finally, you will see in your local folder the *Final_Report.csv* file, which contains the *SLACK* value of all the executions with different VHDL files. If a specific vhd file is executed more than once with the same number of inputs and outputs, the *SLACK* value is replaced.

6 Conclusion

To summarize, in this project we have implemented a parametric Python script that generates the necessary files to trigger Vivado Design Suite, measure the propagation delay of a VHDL file, after failing in the specified timing constraints, output the results, and finally plot and compare the current results with the ones of previous executions.

In the end, we proposed two approaches to model the propagation delay, in regard with the number of input and output bits, for a specific vhd file, which implements an adder of two numbers and outputs the results. The results of our modeling is that the delay values do not follow a standard norm, and for this reason it is difficult to be modeled.

Our future work could be first to transform the Python script to be a totally parametric tool for the measurement of the propagation delay of any hardware component executed in Vivado, with the addition of verilog hardware development language as a choice. Finally, we could conduct further research to the behaviour of the simple adder, by getting more information, of the addition of more bits in the input and output ports, in order to conclude to a more accurate and exact model.

7 Bibliography

[1] Vivado Design Suite User Guide. Using Tcl scripting.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug894-vivado-tcl-scripting.pdf

[2] Vivado Design Suite Tcl. Command Reference Guide.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug835-vivado-tcl-commands.pdf