Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

הפקולטה להנדסה
ע״ש איבי ואלדר פליישמן
אוניברסיטת תל-אביב

FPGA Digital Control System

Project Number: 2692

Project Report

Student: Ronen Dvorkin ID: 315203539

Student: Omri Meron ID: 316384981

Supervisor: Meir Alon

Project Carried Out at: Projects Lab Tel Aviv University

Contents

Acronym Table 3

Abstract.......... 4

1 Introduction 5

2 Theoretical background 6

3 Implementation 8

4 Simulation...... 17

5 Analysis of results 24

6 Conclusions and further work 32

7 Project Documentation 34

8 References... 34

List of figures

Figure 1: Block diagram 4

Figure 2: View of DDS IP core operation 6

Figure 3: The system 8

Figure 4: The system 8

Figure 5: Max-Min Calculatior State Machine 13

Figure 6: Dual-Port BRAM The system 14

Figure 7: Check Of $Reset$ Operation 17

Figure 8: Sweep Simulation- Check that Count and Multiplication are working 18

Figure 9: Sweep Simulation- Last Frequency To First Frequency, Initialization 19

Figure 10: Sweep Simulation- Second To Last Frequency To Last Frequency, Initialization 19

Figure 11: Max/Min, Reset and Write The system 21

Figure 12: Max/Min, New Frequency (New_Theta inserted) 22

Figure 13: V_{in} Values From The Bram, 2000 Addresses 24

Figure 14: The DAC's Output As Seen Within The Oscilloscope 25

Figure 15: Gain $\frac{V_{out}}{V_{in}}$ Of Shortened Red Pitaya, Without DUT 26

Figure 16: The DUT's Input Signal (Yellow) Generated From The FPGA Design, and the DUT's Output (Green) 28

Figure 17: The DUT's input sweep signal with $f \geq 1\ [Hz]$ (yellow) generated from The FPGA design, and the DUT's output (green) 28

Figure 18: ADC's V_{pp} Results For Short Circuit 29

Figure 19: The LPF RC Time Delay As Function Of Frequency 31

List of tables

Table 1: Acronym Table 3

Table 2: Address Ranges 15

Acronym Table

In the upcoming project documentation, we will employ various terms, which will be explained in the next acronym table:

Acronym	Full Form
FPGA	Field Programmable Gate Array
RTL	Register Transfer Level
HDL	Hardware Description Language
DUT	Device Under Test
LUT	Lookup Table
IP Block	Intellectual Property Block
DAC	Digital-To-Analog Converter
ADC	Analog-To-Digital Converter
DDS	Direct Digital Synthesizer
BRAM	Block Random Access Memory
PCB	Printed Circuit Board
LPF	Low Pass Filter

Table 1: Acronym Table

Abstract

Our project aimed to create a Digital Control System using an FPGA, utilizing the Red Pitaya STEM lab 125-14 FPGA development kit. This endeavor delved into various aspects of hardware, such as FPGA implementation, RTL design, timing considerations and simulations, alongside software and communication domains, including Linux interfaces, communication protocols, user-hardware interactions, and related areas.
The significance of control systems extends to various sectors including medicine, military applications, and other areas reliant on electronically controlled systems.
The outcome of our work is a digital system centered on an FPGA, designed for analyzing the Bode of a device under test (DUT), which could be any linear circuit of our choice, such as RLC circuit, and capable of controlling it through a PID controller. This functionality spans across a frequency sweep between $[1 - 10^6]$ $[Hz]$, and a voltage range of $[(-1) - 1]$ $[V]$.

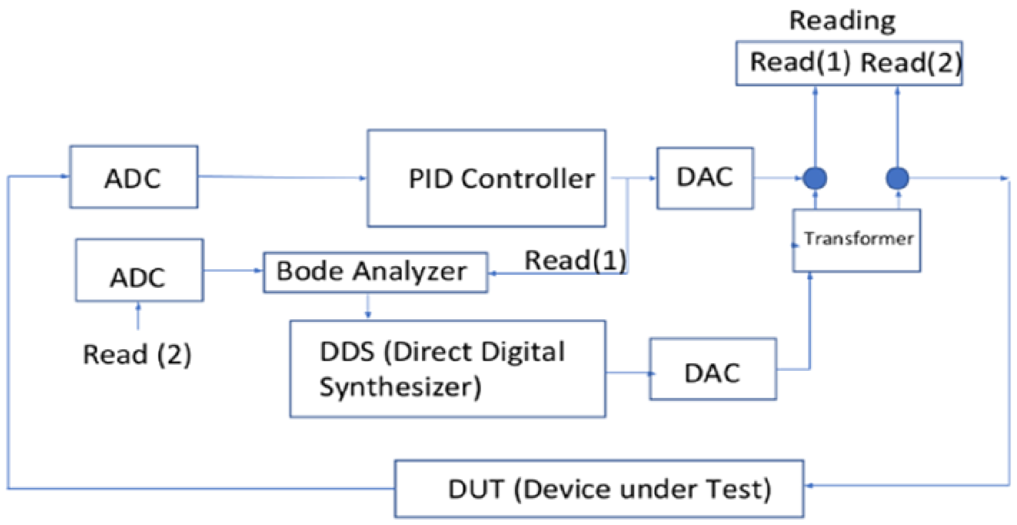


Figure 1: Block diagram

Due to time constraints, as approved by both project management and the project instructor, we have opted to solely focus on implementing the Gain part of the Bode plot.

1 Introduction

The project aims to develop a Digital Control System capable of two primary functions: analyzing input and output signals to determine Bode characteristics (including gain and phase), and utilizing a PID controller to regulate the DUT. These functionalities are required to operate within specific parameters, including a frequency sweep ranging from $1\ [Hz]$ to $1\ [MHz]$, and a voltage range of $(-1)\ [V]$ to $(1)\ [V]$.

In terms of implementation, the Red Pitaya board's extensive connectivity options and onboard peripherals streamline integration with external sensors, actuators, and other components essential for comprehensive Bode analysis and PID control. Its rich set of input/output interfaces, including analog and digital ports, Ethernet, and USB connections, facilitate seamless communication and data exchange with external devices, simplifying system setup and operation.

The project will be constructed from the following stages, The system operates by generating a frequency sweep that it applies to the DUT, simultaneously calculating and storing the amplitude of this swept signal (marked as V_{in}) within the FPGA’s BRAM. Subsequently, the signal passes through the DUT and returns to the FPGA, where it’s amplitude (marked as V_{out}) is computed, and this data is also stored in BRAM. The frequency sweep spans from $1\ [Hz]$ to $1\ [MHz]$ with an amplitude range of $(-1)\ [V]$ to $(1)\ [V]$. The analysis results, comprising the amplitudes of V_{in} and V_{out} are saved in text files, facilitation the creation of Bode gain diagrams through data plotting.

Bode analysis is widely utilized across various industries to assess signal behavior and characteristics. Fields such as audio engineering, amplifiers, radars, and medical equipment heavily rely on Bode analysis to obtain accurate amplification and gain. Our goal is to manage the system effectively, minimizing instabilities, malfunctions, and errors to ensure smooth operation, using a PID controller. This is crucial particularly in applications like medical equipment, where flawless functionality is essential.

2 Theoretical background

Measuring a transfer function via frequency response analysis is a powerful method for the design of electronic systems. In electronics, "gain" refers to the amplification or multiplication of a signal's amplitude or power as it passes through a device or circuit. It's a measure of how much a signal is boosted. Gain is typically expressed in decibels (dB) and can be positive or negative. The gain usually defined as the mean ratio of the signal amplitude at the output port to the amplitude at the input port:

$$(1)\ Gain = 20Log(\frac{V_{out}}{V_{in}})\ [dB]$$

Another important value is the knee frequency, knee frequency is the frequency at which the response deviates significantly from the flat or desired response in the context of frequency response curve. Therefore, in this frequency the curve exhibits a sharp change. The knee frequency plays a big role in designing of Filters, the knee frequency often marks the point at which the filter's attenuation begins to rapidly increase or its phase response starts to deviate significantly from its linear behavior. For example, as part of the testing, we built a LPF that consists from capacitor and resistor, his transfer function is:

$$(2)\ LPF = \frac{\frac{1}{RC}}{S + \left(\frac{1}{RC}\right)}$$

In that case the knee frequency is $= \frac{1}{RC}$, every frequency after that will be filtered.

DDS, is a very important component in many digital communication systems. His operation is defined like this:

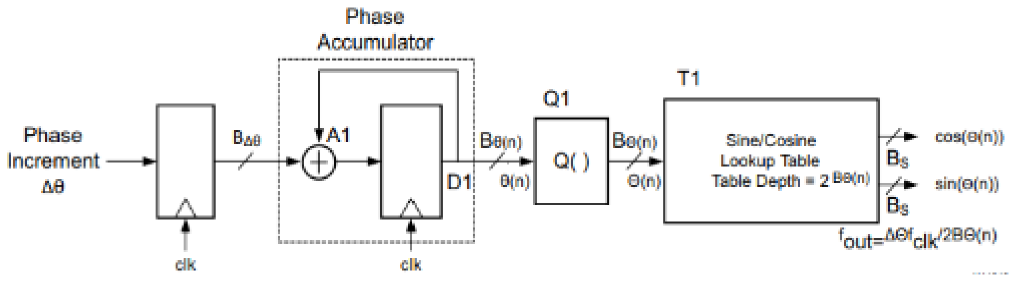


Figure 2: View Of DDS IP Core Operation

The DDS uses an addressing scheme with an appropriate lookup table to form samples of a frequency sinusoid. The lookup table stores uniformly spaced samples of a cosine or sine wave, those samples can be generated using a phase accumulator block, we use the lower 14-bits of the wave to get a voltage range of $(-1)\ [V] - (1)\ [V]$. This block accumulates phase values at a constant rate, it operates based on a phase increment value which determines the frequency of the output waveform. By manipulating the phase increment value, the DDS can generate waveforms of different frequencies, in such way we can get a frequency sweep.

In order to store the peak- to- peak amplitude values of each swept signal we used a Block Memory Generator IP in Dual Port Configuration (BRAM). The BRAM is a volatile memory, which means once you disconnect the system all the data it stores erases. This Dual Port configuration includes two ports which can be used for both reading and writing operations, simultaneously. The BRAM consists of cells that are one byte long, each has its own address. Larger cells (such as 32-bit) can be configured, in such a case, when operating in 32-bit mode with an 8-bit address, the addressing steps need to be multiplied by 4. Moreover, the BRAM provides flexibility in terms of organization, users have the freedom to configure the depth and width of the memory according to their requirements.

3 Implementation

Given that our project primarily focuses on creating and executing a Bode Gain Analyzer using an FPGA as its core, our implementation is rather straightforward. Our system consists of 3 main parts: a Red Pitaya FPGA development kit (Xilinx ZYNQ 7010 SoC), the DUT, Linux interface (sits on a Dual-Core ARM Cortex-A9 MPCore micro-processor) and a PC (where we've used MATLAB for analyzing the results).
A simplified block diagram illustrating the system can be seen within the next figure:

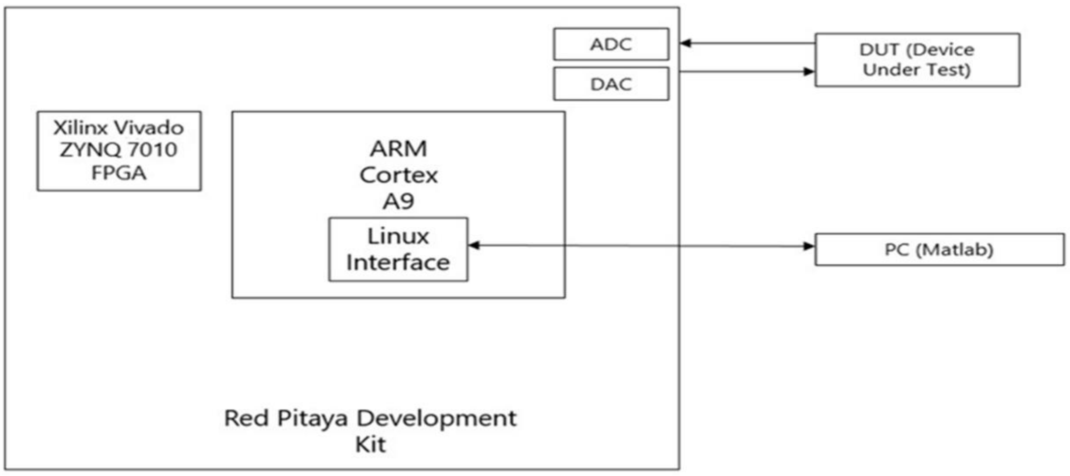


Figure 3: The system

Our design is described as follows:
1. **<u>FPGA design and implementation (Hardware Description):</u>**
In the figure below, we observe the complete FPGA project that we have assembled:

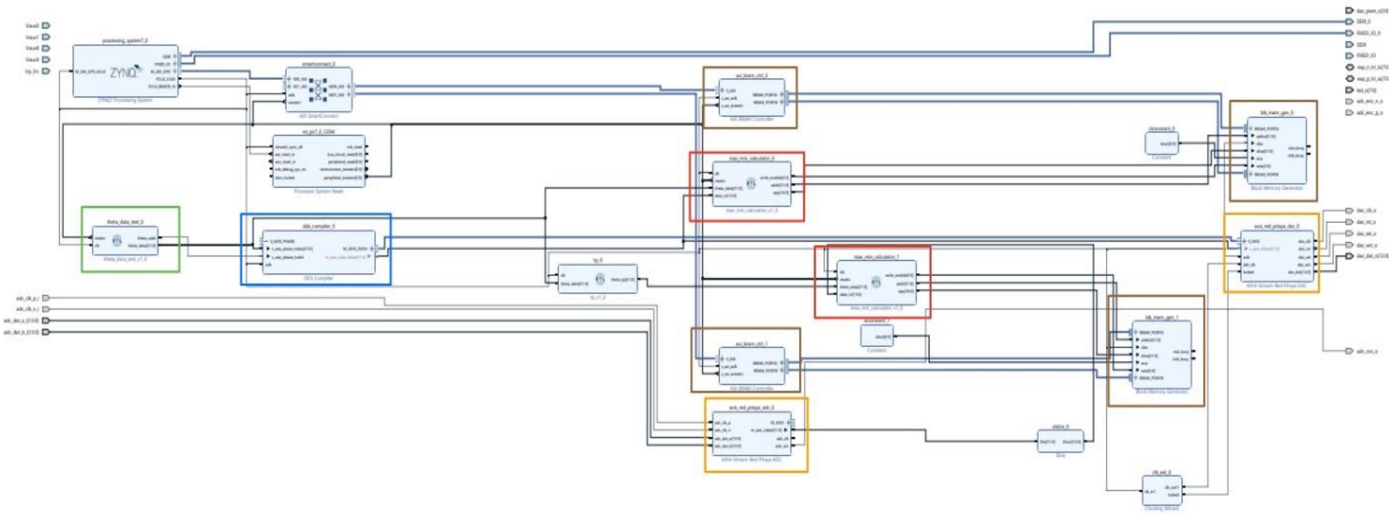


Figure 4: The System- Max/Min in red, Bram In brown, DDS in blue, Theta Data in green, ADC/DAC in yellow

This part includes 3 main objectives:

1.1 Frequency Sweep:

This include designing a block that generates a frequency sweep within the desired range ($1\ Hz - 1\ MHz$), spanning across 2000 frequencies.
The rationale behind selecting this frequency count is to ensure a dense sweep, as each DUT has a critical frequency where signals with equivalent or higher frequencies are filtered. This frequency is known as the "Knee Frequency" (see detailed explanation in the theoretical background). This Block consists of 2 main parts:

a. A block that generates a loop of the above frequency values in the 'θ' domain. The relationship between the 'θ' domain and the frequency domain depends on the DDS Compiler's configuration (detailed in section b).
This is a self-written RTL block (which we wrote ourselves) Inside the Vivado, called 'theta_data_test' block.
The DDS Compiler outputs a sample of the signal each clock cycle, namely each $\frac{1}{125\cdot10^6} = 8\ [nsec]$. The samples set to represent a sinusoidal signal of a certain frequency.
Consequently, the peak-to-peak amplitude calculation (Peak-to-Peak Amplitude Calculation section) has been done using the next formula:

$$(3)\ V_{pp} = V_{max} - V_{min}$$

When outputting a digital signal using the Red Pitaya's DAC it generates a sinusoidal signal of the next form:

$$(4)\ \sin(2\pi f t)$$

As a result, the peak-to-peak values can receive each value within the interval $[0,2]\ [V]$.
In order to optimize the sweeping time, and still maintain a decent level of accuracy, we've decided to sweep every frequency for exactly 1 period. This guarantees that both positive and negative samples are generated and that none are accidentally missed, and hence will result in a wrong peak-to-peak value.
As described earlier, we set the number of frequencies in the sweep to be 2000. Since the clock within our design is set to be with a rate of $125\ [MHz]$, in order to sweep each frequency for a full period, the block should wait a certain amount of time before changing the theta value (and by it, the appropriate frequency value). We'll mark this time by $t_{wait}(f)$. This time depends on which frequency the block is sweeping and on the clock rate.
Since the block design within the FPGA is a synchronous circuit, instead of calculating this time directly, we've calculated the number of clock cycles this time is equal to (marked by $count$).
According to the DDS configuration (detailed in section b) it holds that:

$$(5)\ count = \frac{2^{28}}{\theta}$$

This block counts the clock cycles up until it reaches a value of $\frac{2^{28}}{\theta}$. Afterwards, it initializes $count$ to 0, and advances θ by the appropriate step (meaning, advances the sweep to the next frequency).
According to the DDS configuration, and our 2000 frequency values, to receive the appropriate thetas it holds that:

$$(6)\begin{cases} \theta_{min} = \frac{2^{28}}{125\cdot 10^6} f_{min} = \frac{2^{28}}{125\cdot 10^6}\cdot 1 \simeq 2.147 \\ \theta_{step} = \frac{2^{28}}{125\cdot 10^6} f_{step} = \frac{2^{28}}{125\cdot 10^6}\cdot \frac{10^6-1}{1999} \simeq 1074.278 \\ \theta_{max} = \frac{2^{28}}{125\cdot 10^6} f_{max} = \frac{2^{28}}{125\cdot 10^6}\cdot 10^6 \simeq 2147483.6 \end{cases}$$

Since we need to insert the DDS an integer value we'll work with the next values:

$$(7)\begin{cases} \theta_{min} = 2 \\ \theta_{step} = 1074 \\ \theta_{max} = \theta_{min} + 1999\theta_{step} = 2 + 1999\cdot 1074 = 2146928 \end{cases}$$

These values correspond to the frequency interval of $[0.931\ Hz - 0.9997\ MHz]$ which is, by no doubt, close enough to the desired range.
To answer the question of how long this sweep will take, we've calculated the ideal total duration of the sweep (assuming instantaneous transitions within the FPGA blocks) using the following expression:

$$(8)\ t_{sweep} = \sum_{n=1}^{N_f} T_n(f)$$

Where $T_n(f)$ is the period time of the swept signal (that has a frequency f), N_f is the total number of frequencies within the sweep (which we set to be 2000).
This sum can also be written as follows:

$$(9)\ t_{sweep} = \sum_{n=1}^{N_f} \frac{1}{f_n} = \sum_{n=0}^{N_f-1} \frac{1}{f_0 + nf_{step}} = \sum_{n=0}^{N_f-1} \frac{1}{f_0 + \frac{f_{max}-f_{min}}{N_f-1}n}$$

Hence, the sweep duration is:

$$(10)\ t_{sweep} = \sum_{n=0}^{1999} \frac{1}{1+\frac{f_{max}-f_{min}}{1999}n} = \frac{1}{1} + \frac{1}{1+\frac{10^6-1}{1999}} + \cdots + \frac{1}{1+\frac{10^6-1}{1999}\cdot 1999} \simeq 1.016341\ [sec]$$

This is implemented within the 'theta_data_test' Verilog module.
The module also generates a valid bit that is '1' unless a *reset* is being asserted, since the DDS Compiler requires such an input in addition to the theta input.

b. DDS Compiler- This is an internal component (IP) that receives a value ('θ') that corresponds to a frequency f according to this equation:

$$(11)\ f = \frac{f_{clk}}{2^{B_{\theta(n)}}}\theta$$

Where:
f_{clk}- The clock frequency. In our design, we chose a clock with a frequency of 125 $[MHz]$, aligning it with the frequency of the Red Pitaya's ADC and DAC. This choice helps prevent clock skew within the design.
$B_{\theta(n)}$- The number of bits within the internal block's Phase accumulator. This internal component is responsible for generating the phase increment values used to control the output waveform's frequency. It accumulates phase values over time based on the desired frequency and clock rate. In our design its value is 14.
θ- This is the "phase increment" value, serving as the input to the DDS that determines the specific lookup table (LUT) used for generating the signal's samples. Each LUT corresponds to a distinct value of θ.

Note that to generate the desired sweep, we needed to make sure that θ is changed only after 1 period of the swept signal (as described in section a, using the counter).
Since our clock rate is constant, it results in a different number of samples for each swept signal, which can be calculated using the next formula:

$$(12)\ \#_{samples} = \frac{T(f)}{T_{clk}} = \frac{f_{clk}}{f}$$

This is exactly $count$, hence:

$$(13)\ count = \frac{f_{clk}}{f} = \frac{f_{clk}}{\frac{f_{clk}}{2^{B_{\theta(n)}}}\theta} = \frac{2^{B_{\theta(n)}}}{\theta} = \frac{2^{28}}{\theta}$$

Consequently, the lowest frequency will have the largest number of samples (125 million samples) and the highest frequency will have the least number of samples (125 samples).
We assume that 125 samples for one sinusoidal period is a sufficient number to receive an accurate peaks amplitude.

1.2 Peak-To-Peak Amplitude Calculation

This include designing a block that receives both the samples and the frequency value at each clock cycle and calculates the peak-to-peak amplitude, namely V_{pp}.
As mentioned earlier, it holds that:

$$(14)\ V_{pp} = V_{max} - V_{min}$$

Hence, we designed this block to find the maximum and minimum values within each samples set (associated with a single swept frequency). This block

is implemented within the 'max_min_calculator' Verilog module inside our FPGA design (which we wrote ourselves).
It has 4 inputs:
$data\_in$ $[13:0]$- These are the samples that are generated by the DDS compiler which is a 14-bit wide bus.
$theta\_data$ $[31:0]$- This is the θ value (the frequency value in the 'θ' domain) of the swept signal.
clk- The 125 $[MHz]$ clock signal.
$resetn$- An active low reset signal.
The block has 3 outputs:
$write\_enable$ $[3:0]$- A write enable signal for writing operation of the BRAM.
$addr$ $[31:0]$- Address value designates where the data should be stored in the corresponding BRAM cell.
vpp $[16:0]$- The calculated peak-to-peak amplitude value. As we mentioned earlier the peak-to-peak amplitude can have any value between $0-2$ $[V]$, but since $data\_in$ can be a signed value, vpp has also to be defined as a signed register. Moreover, since our samples are 14-bit wide, in a signed register (meaning integer in 2's complement representation) their values can be any integer in the interval $[(-8192)-(8191)]$. For instance, a value of (-8192) is corresponding to an amplitude of $\left(\frac{-8192}{8192}=-1\ [V]\right)$, and a value of 3333 is corresponding to an amplitude of $\frac{3333}{8191}\simeq 0.41$ $[V]$. Hence, vpp can have integer values between 0 and $8191-(-8192)=16383$. Here, for example, a value of $vpp=9900$ is corresponding to a peak-to-peak amplitude of $\frac{9900}{16383}\cdot 2\simeq$ 1.21 $[V]$.
First, the Block sets internal registers to be used for its operation.
Registers *current_max* and *current_min* are used to store the current maximal and minimal values of the current samples set.
Registers *theta_data_Q1* and *new_theta* are used for identifying a new swept frequency.
In each cycle there is a check whether a reset has been inserted. If so, *current_max* and *current _min* are initialized to (-8192) and 8191 respectively, and *write_enable* is being set to 0. This definition ensures that both *current_max* and *current _min* are ready to receive the subsequent data, as intended when a reset occurs, indicating that the data for the next cycle needs to be considered.
If a reset hasn't been inserted, the block's state is set as *next_state*.
Then, it checks whether a new frequency has been received (using registers *theta_data_Q1* and *new_theta*).
If so, multiple operations occurs:

1) Register *new_theta* is set to 1.
2) Register *max_reg* and *min_reg* receives the values of registers *current_max* and *current_min* respectively (the next samples

are of a new frequency so we've done with the current set of samples).

3) Register vpp is being calculated. It's being done by taking the difference between sign extended of both values of the registers *max_reg* and *min_reg*.
4) Register *write_enable* is being set to 1 because we intend to write the value of vpp to the BRAM in the subsequent clock cycle.
   Note that our design allows for a proper writing operation within 1 clock cycle, satisfying both the t_{hold} and t_{setup} conditions.
5) Address advancing- Address progression involves setting the initial address of this block to $0x0000\_0000$.
   The increment is by 4 because we've configured the BRAM to accommodate 32-bit values. Since each address represents 1 byte within the memory, advancing the address by 4 allows us to store the next 32 bits.
6) Both *current_max* and *current_min* registers are set to receive *data_in* because the next clock cycle will bring a new set of samples. This setting ensures that they discard their current values and only consider the new set of samples. Additionally, it prevents the mixing of data from different frequencies.

If the frequency remains the same, it sets *new_theta* and V_{pp} to 0.
This block's functionality is described using the following state machine:

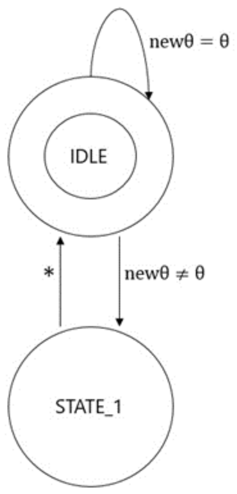


Figure 5: Max-Min Calculatior State Machine

This state machine is also included within the Verilog module.
The IDLE state, which is the default state, includes the *current_max* and *current_min* calculation and checks if a new frequency has arrived. If so, it goes

over to STATE_1 and set *write_enable* to 0 (meaning it stops performing the write operation).
STATE_1 performs as a 'dummy' state, meaning a buffer of 1 clock cycle. This state simply sets register *next_state* to be IDLE.
This block is designed to operate both on the data that is transited to the DAC (namely V_{in}) and the data that is transited from the ADC (namely V_{out}). This design effectively reduced the workload on each data line, resulting in considerable time savings.

1.3 Data Storing

In order to store the calculated vpp values we used 2 BRAMs, one for V_{in} values and another for V_{out}. We've configured the BRAMs to be in a Dual-Port Configuration, meaning each block will have 2 ports (A and B) which can be used for both writing and reading operations.
In our design, we set port A for writing, and port B for reading. Hence, the outputs of the module 'max_min_calculator' are connected only to port A. In more details, the output $addr$ $[31:0]$ is connected to port A's address input, the output $write\_enable$ $[3:0]$ is connected to port A's wea $[3:0]$ input and the output vpp $[16:0]$ is connected to port A's $dina$ input.
Moreover, in both BRAMs, the ena input of port A is fed with a constant value of 1.
This can be seen within the next figure:

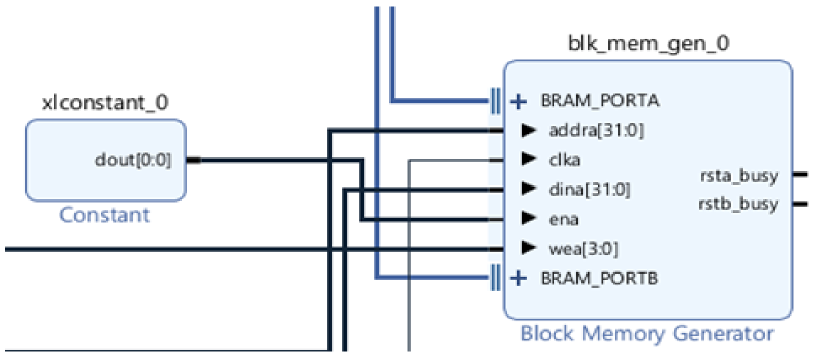


Figure 6: Dual-Port BRAM

Both BRAMs are connected to AXI BRAM Controller IPs, which are used to transfer their data to the AXI SmartConnect IP, which is a common block for both BRAMs. This block is connecting between the BRAM controllers and the ZYNQ-7 Processing system, so a connection between Red Pitaya's Linux Interface and the BRAMs is possible.
As mentioned earlier, we've decided to sweep 2000 frequencies. Consequently, each peak-to-peak amplitudes vector ($\boldsymbol{V_{in}}$ and $\boldsymbol{V_{out}}$) is 2000 values long. Meaning, each BRAM stores 2000 values. In other words, each BRAM has 2000 cells.

While working on this part of the project, we initially attempted to utilize BRAM with 16-bit wide cells. Despite successfully implementing this configuration, we found that working with 32-bit cells was much easier when exporting data from the BRAMs using C codes within the Linux interface. As per the Xilinx BRAM documentation, the address ranges should be configured as outlined in the following table:

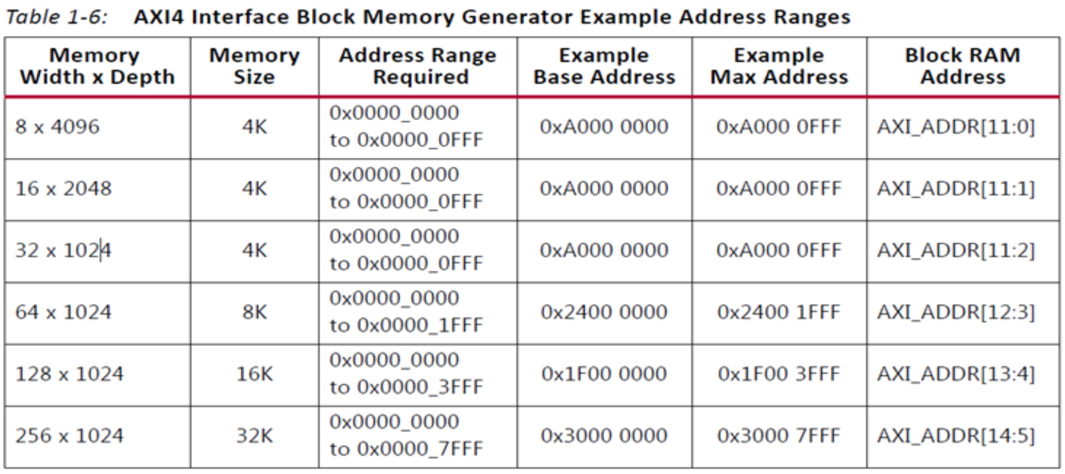


Table 1-6: **AXI4 Interface Block Memory Generator Example Address Ranges**

Memory Width x Depth	Memory Size	Address Range Required	Example Base Address	Example Max Address	Block RAM Address
8 x 4096	4K	0x0000_0000 to 0x0000_0FFF	0xA000 0000	0xA000 0FFF	AXI_ADDR[11:0]
16 x 2048	4K	0x0000_0000 to 0x0000_0FFF	0xA000 0000	0xA000 0FFF	AXI_ADDR[11:1]
32 x 1024	4K	0x0000_0000 to 0x0000_0FFF	0xA000 0000	0xA000 0FFF	AXI_ADDR[11:2]
64 x 1024	8K	0x0000_0000 to 0x0000_1FFF	0x2400 0000	0x2400 1FFF	AXI_ADDR[12:3]
128 x 1024	16K	0x0000_0000 to 0x0000_3FFF	0x1F00 0000	0x1F00 3FFF	AXI_ADDR[13:4]
256 x 1024	32K	0x0000_0000 to 0x0000_7FFF	0x3000 0000	0x3000 7FFF	AXI_ADDR[14:5]

Table 2: Address Ranges

Since we have 2 BRAM IPs we've decided to configure the BRAM for V_{in} to store its peak-to-peak amplitudes in the addresses *0x4000_0000, 0x4000_0004, … , 0x4000_1F3C*. These addresses equivalent to the numbers 0,4,…,7996, which is exactly 2000 addresses in steps of 4 (since $\frac{7996}{4} = 1999$).
In the same manner, we've decided to configure the BRAM for V_{out} to store its peak-to-peak amplitudes in the addresses *0x4200_0000, 0x4200_0004, … , 0x4200_1F3C*.
These addresses routing (between the value computed inside the module to the *0x4000* and *0x4200* addresses is done using the AXI SmartConnect IP and the appropriate settings inside the Address Editor within Vivado).

1.4 ADC/DAC

The DDS Compiler is utilized to generate a swept signal composed of digital samples representing a sinusoidal waveform, as explained earlier, This is V_{in}.
The DUT is assumed to be a linear circuit, such as RLC circuits, essentially an analog circuit.
Subsequently, these digital samples are converted into analog form via the Digital-to-Analog converter of the Red Pitaya device.
They are than fed to the DUT using the SMA connector and appropriate cables and wires. In the same manner, V_{out} is fed from the DUT to the Red Pitaya, through an SMA connector and an Analog-to-digital converter of the Red Pitaya device. Note that this ADC has a width of 14 bits and operates at a frequency of 125 $[MHz]$. As previously discussed, this is the rationale behind configuring the clock rate in our design.

2. **Exporting The Data (Hardware Description):**

As mentioned above, the peak-to-peak amplitudes of both V_{in} and V_{out} are stored in the appropriate BRAMs.
Given that BRAM is a volatile memory, the system requires access it during operation. This process involves executing C code that retrieves data from each BRAM and then writes the cell contents to a text file.
We designed the code to read the content of each cell and save it to a text file, with each line in the file representing one cell in the format '<Address$_i$>: <content$_i$>'. The address is expressed in hexadecimal, while the content is represented in binary. This process involves the creation of two separate C codes, one for V_{in} and another for V_{out} , resulting in the generation of two distinct text files. It's worth noting that each line in these files corresponds to a specific frequency. Additionally, these files are generated within the Red Pitaya's Linux environment
The C code operates as follows:

1) It initially creates a text file.
2) It maps the 1st address of the memory and writes it to the text file with ':'.
3) It reads the content inside the address and writes it to the text file. This is done by treating the content as an unsigned 32-bit integer (since the peak-to-peak amplitude is a non-negative number). To represent it in binary form, the program iterates from 0 to 31. In each iteration, it performs the operation *(value >> i) & 1* where i represents the current iteration index.
   This operation performs a logical shift by i positions and then performs a bitwise AND operation with the constant '1', effectively extracting the ith bit in each iteration.
4) After reading and writing each cell it iterates over to the next address and repeats the process.

3. **Analyzing And Plotting (Hardware Description):**

The generated text files are then manually transferred to the PC to analyze the results.
This can be accomplished using software of preference, in our case, we opted to utilize MATLAB for analyzing and visualizing the results.

4 Simulation

Within our design, we've implemented 2 RTL modules: 'theta_data_test' which is responsible for generating the logic needed to produce the 2000 frequencies required for system sweeping, and 'max_min_calculator' which handles the computation of the peak-to-peak amplitudes of the samples generated during the sweeping process. The remaining blocks in the design consist of passive components such as ADC and DAC converters, as well as BRAM. Therefore, we did not simulate these blocks.
Instead, we tested them using the Linux interface of the Red Pitaya device and the laboratory equipment available, including the Oscilloscope, as elaborated in Chapter 5.
Since both of these RTL blocks are Verilog modules, we've used the Vivado's simulation environment in order to simulate their operation.
The simulations that we've conducted are:
1. **'theta_data_test'**- To simulate this module, we've written a test bench that instantiates this block and visually presents the corresponding waveforms of the relevant registers and wires contained within it.
According to the detailed explanation we've provided for this block operation in section 3 (Implementation), we wish to see multiple behaviors throughout this simulation result:

a. A proper operation when a $reset$ signal is being asserted.
b. Correct progression of the counter ($count$) and its initialization at the proper moments.
c. Correct progression of the θ value (which corresponds to the frequency that is being swept) and its initialization to the first value, at the proper moment, and by that restarting the frequency sweep.

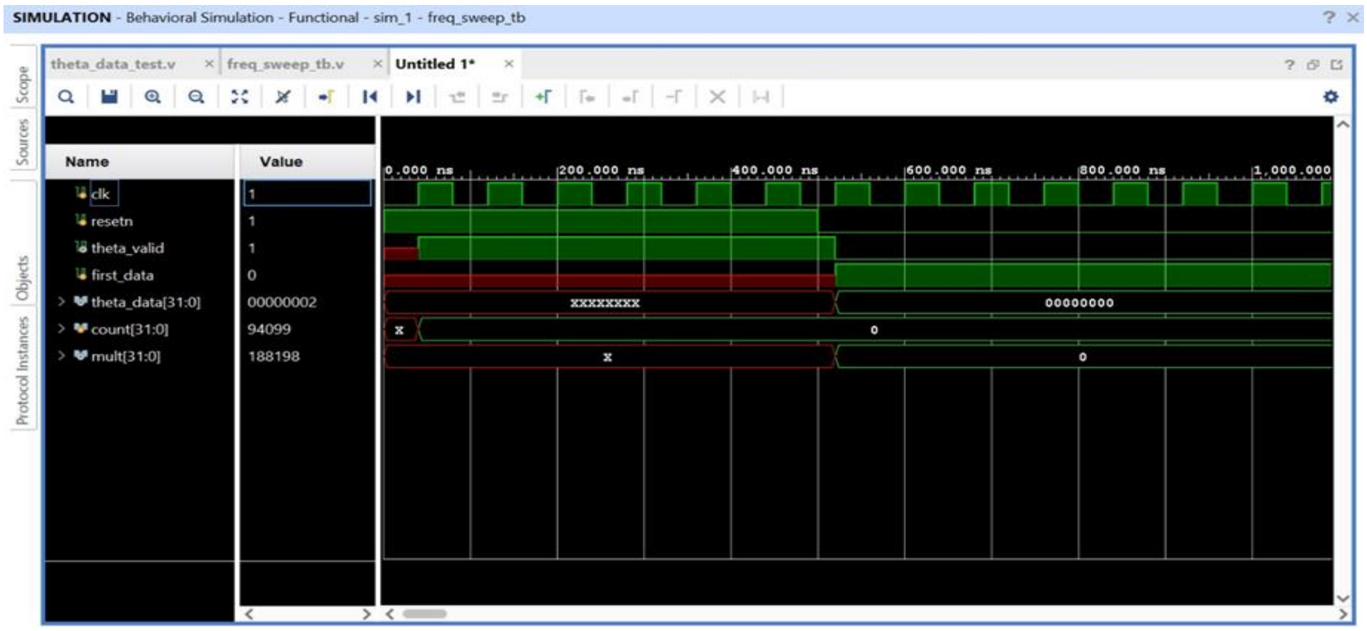


Figure 7: Check Of $Reset$ Operation

This waveform (figure 7) shows that the $reset$ operation we've defined indeed works as expected:
First, it's indeed active low. Second, on the positive edge of the clock, right after the reset signal is asserted (when it receives a value of 0) the signal $first\_data$ is high. This signal indicates that the next $theta$ will be the 1st one, since a reset has occurred.
Moreover, up until the $reset$ is deactivated we've defined θ to be 0 so no samples will be generated. Similarly, we've defined $count$ to be 0 (since there are no samples to count), and the multiplication value $mult$ (which is defined as $count \cdot \theta$) is indeed 0.
Overall, this simulation's result ensures a proper operation of reset mode. This reset mode is only for a case that something goes wrong and we need to restart the system.

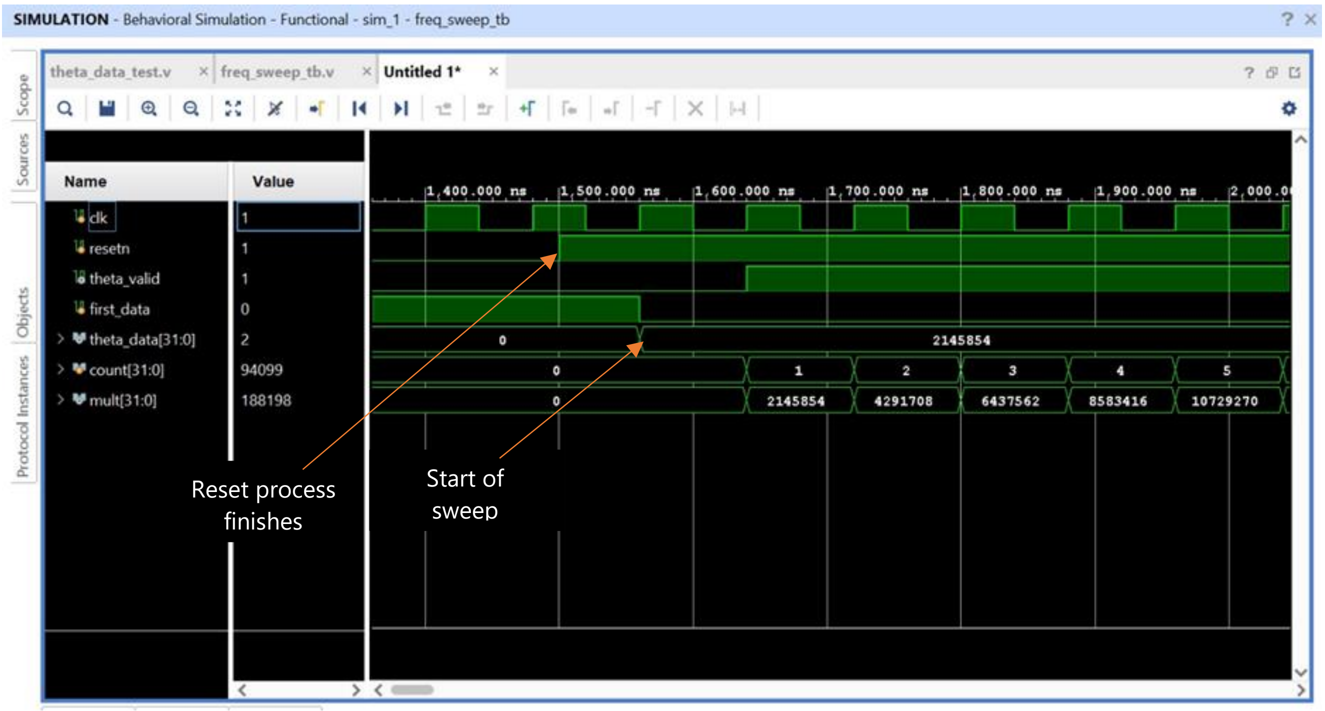


Figure 8: Sweep Simulation- Check that Count and Multiplication are working

It can be seen (figure 8) that after the reset process finishes, θ is assigned a value to start its sweeping operation.
Ideally, this value should be 2. However, due to its association with an extremely low frequency, the simulation would require a substantial amount of time (and computer memory) to execute. Therefore, we chose to assign it the value 2145854, which represents the frequency just before the last one.
It can also be seen that the counter is indeed counting the samples every clock cycle and the multiplication value is indeed correct.

Another simulation result is:

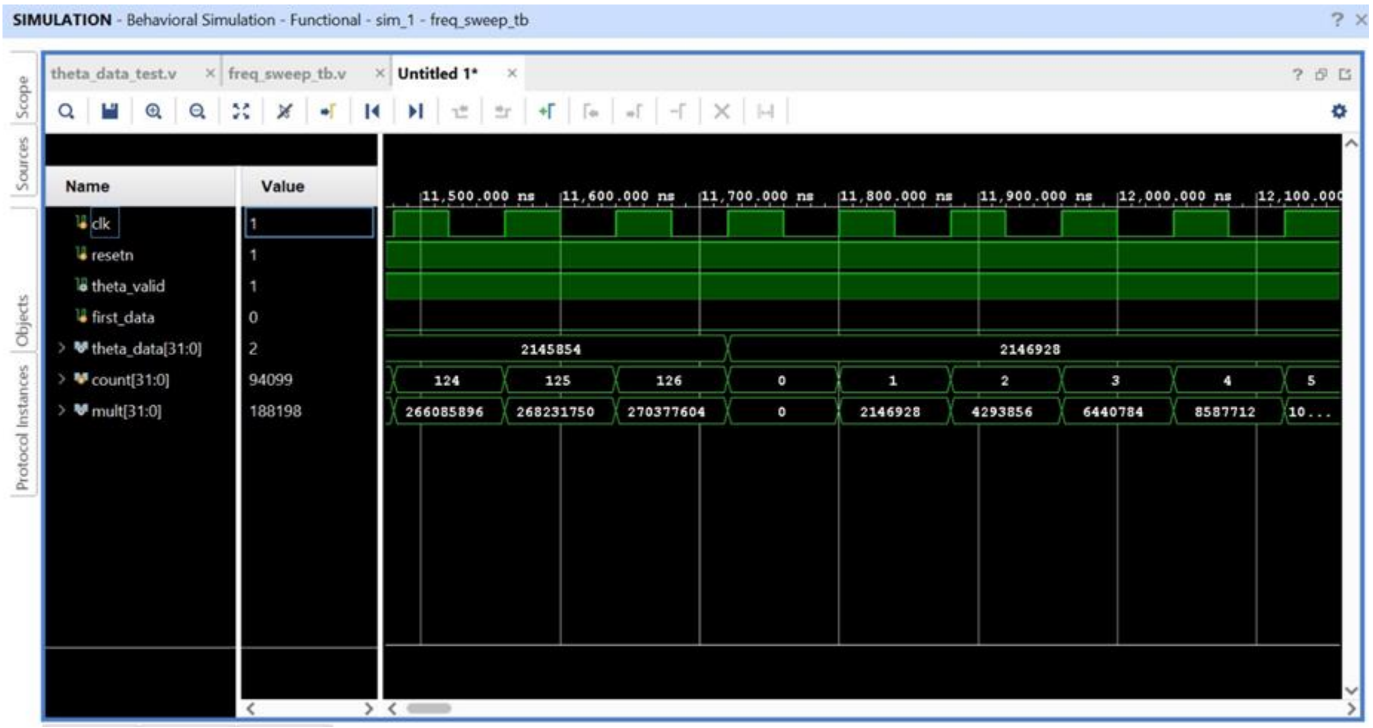


Figure 9: Sweep Simulation- Last Frequency To First Frequency, Initialization

Here (figure 9), it can be seen that once the multiplication value is equal to (or greater than) $\theta \cdot count$ (in our case $2145854 \cdot 126 = 270377604 \geq 2^{28} = 268435456$), θ is advancing by 4 and $count$ and $mult$ are initialized accordingly. This means that we are at the limit and want to start a new count.

Another simulation result is:

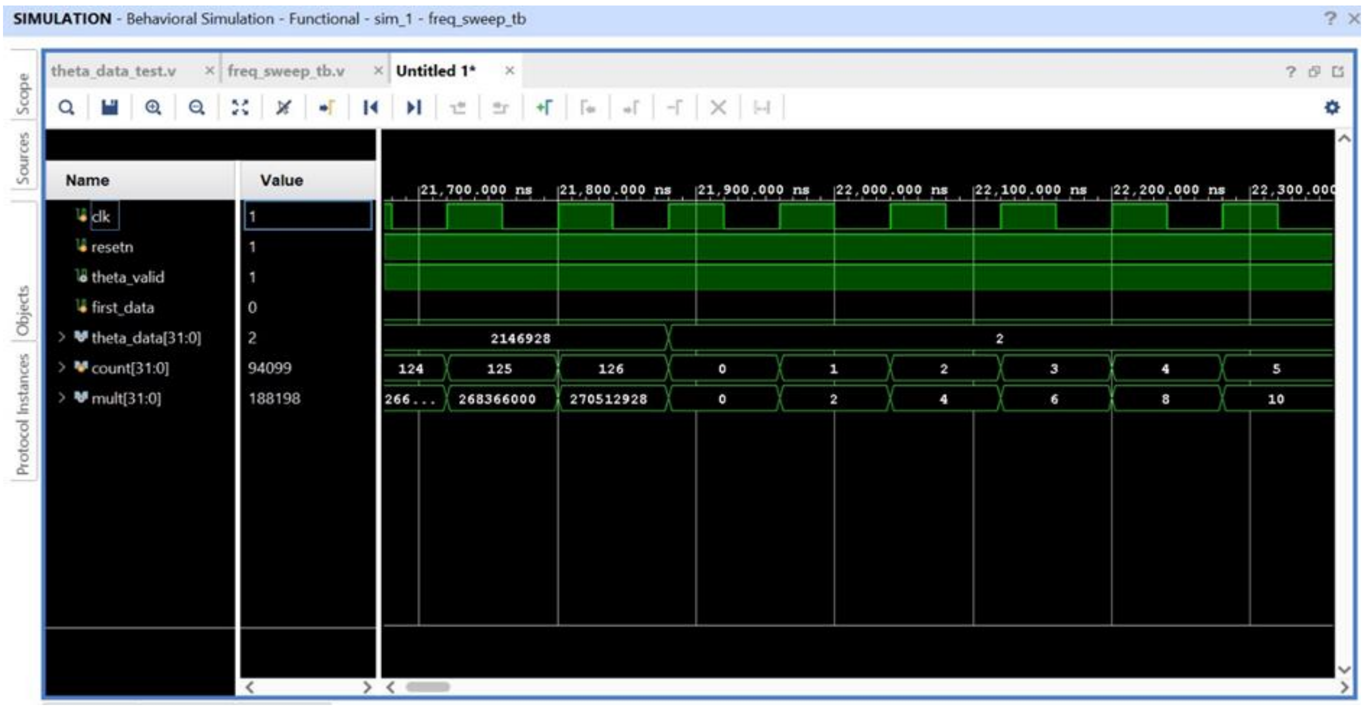


Figure 10: Sweep Simulation- Second To Last Frequency To Last Frequency, Initialization

Here (figure 10), too, θ is advancing back to the initial value (2) once the multiplication condition has been met.
Note: the condition for advancing θ is when $count \geq \frac{2^{28}}{\theta}$. We chose to employ multiplication rather than division within the module's implementation because it is simpler to implement (in the hardware) and takes at most 1 cycle to complete a calculation. On the contrary, division is more complex to implement and requires more than 1 cycle to complete a calculation, potentially introducing inherent delays in our design and leading to timing issues.
These simulation’s results confirm the proper operation of “theta_data_test” module.

2. **‘max_min_calculator’**- To simulate this module, we’ve written a test bench that instantiates this block and visually presents the corresponding waveforms of the relevant registers and wires contained within it.
According to the detailed explanation we’ve provided for this block operation in section 3 (Implementation), we wish to see multiple behaviors throughout this simulation result:

a. A proper operation when a $reset$ signal is being asserted.
b. Ensure the address register progresses correctly as additional frequencies are received.
c. Proper behavior of new_theta and $write\_enable$ registers.
This will ensure correct timing of writing operation.
d. Correct calculation of peak-to-peak amplitude vpp by finding the correct maximum and minimum values.

We've provided the next simulation's results:

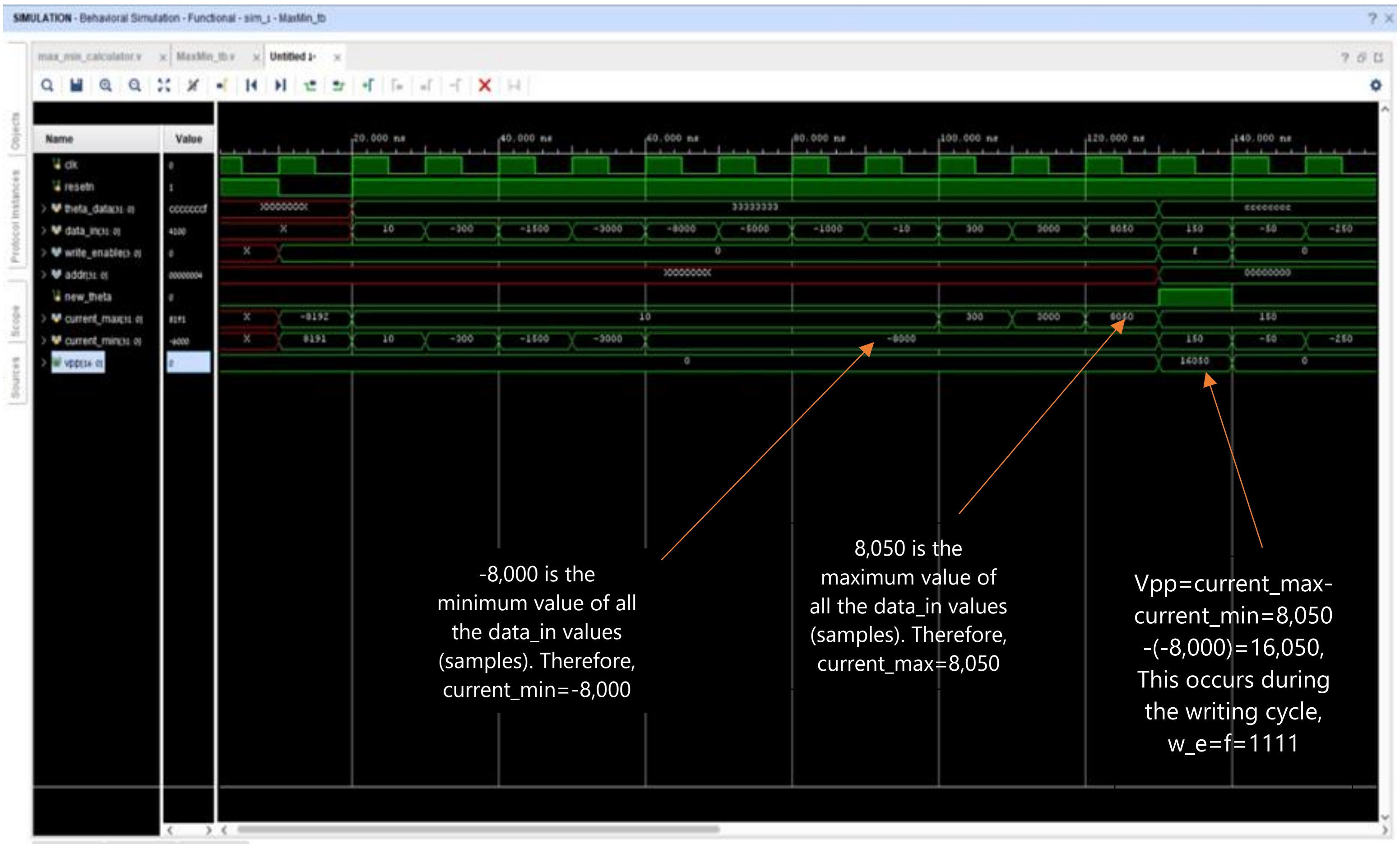

SIMULATION - Behavioral Simulation - Functional - sim_1 - MaxMin_tb
Name
Value
clk
resetn
theta_data
data_in
write_enable
addr
new_theta
current_max
current_min
vpp
-8,000 is the minimum value of all the data_in values (samples). Therefore, current_min=-8,000
8,050 is the maximum value of all the data_in values (samples). Therefore, current_max=8,050
Vpp=current_max-current_min=8,050 -(-8,000)=16,050, This occurs during the writing cycle, w_e=f=1111

Figure 11: Max/Min, Reset and Write

This waveform (figure 11) shows that the $reset$ operation we've defined indeed works as expected:
First, it's indeed active low.
Second, while the $reset$ signal is active, the maximum and minimum registers ($current\_max$ and $current\_min$) are initialized to (-8192) and 8191 respectively, so in the first clock cycle after the reset, the first sample will be taken into account by both registers.
This can be seen on the positive edge of the clock right after the reset signal is asserted (when it receives a value of 0) both registers indeed receive the first sample (the value of $data\_in = 10$).
We set randomly values for $theta\_data$(sweep frequencies) just to simulate cases, the value of the frequency itself does not affect this block's operation.
Within each $theta\_data$ value we randomly chose values for $data\_in$, 14-bit signed integer numbers, such that they'll represent samples of a sine wave.
We chose both negative and positive values to ensure correct operation.
Indeed, we see that along a specific value of $theta\_data$ (meaning, different samples ($data\_in$) of the same frequency) the registers $current\_max$ and

current_min properly identifies the maximum and minimum values, they start with a value of 10, and change according to the inserted *data_in*.
Once, a new value of *theta_data* arrives, meaning a new frequency will be swept, that means we've finished to sweep the "old" frequency.
So, the block needs to calculate the peak-to-peak amplitude using the values of *current_max* and *current_min* and set *write_enable* high.
As explained in section 3 (Implementation), the block uses a register *new_theta* that helps to identify when a new value of *theta_data* arrives.
So, the block set *write_enable* high when the register *new_theta* receive the value 1, and simultaneously calculates *vpp* to receive the peak-to-peak amplitude by taking the difference between *current_max* and *current_min* using sign extension as explained in section 3.
This occurs once a clock cycle after the frequency has been changed, which is the 1st clock cycle of the new value of *theta_data*.
This can be seen clearly within the simulation's result above.
It's important to note that *vpp* is being calculated during the new frequency's 1st clock cycle, which during that cycle it still addressing the *current_max* and *current_min* values of the previous frequency.
In this manner, *vpp* receives the correct value during this clock cycle, which ensures a correct value's writing operation to the BRAM.
Also, while the writing occurs, *current_max* and *current_min* receive 1st sample of the new frequency unconditionally, to ensure no data is being loss during the calculations.
In addition, only when the writing is performed, the address register *addr* is being advanced. To see that we've provided another frequency value (and samples) within the next print:

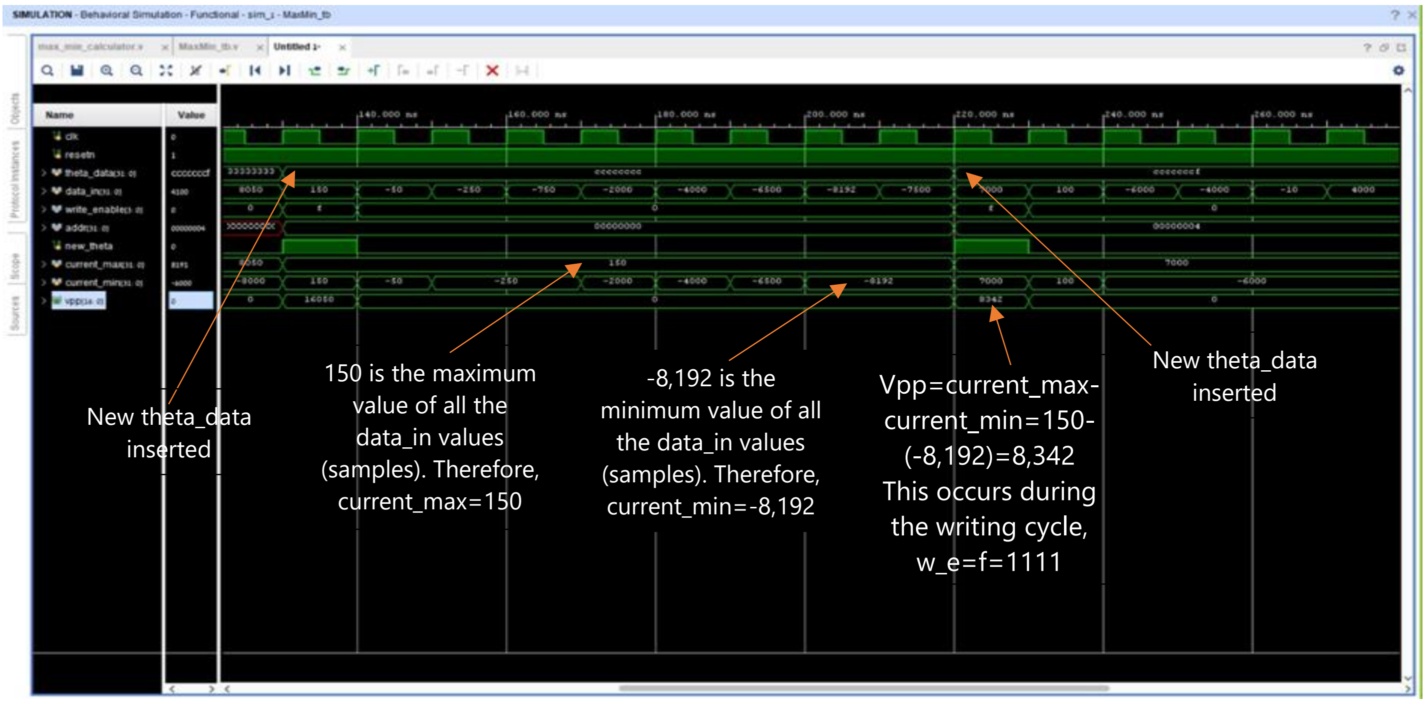
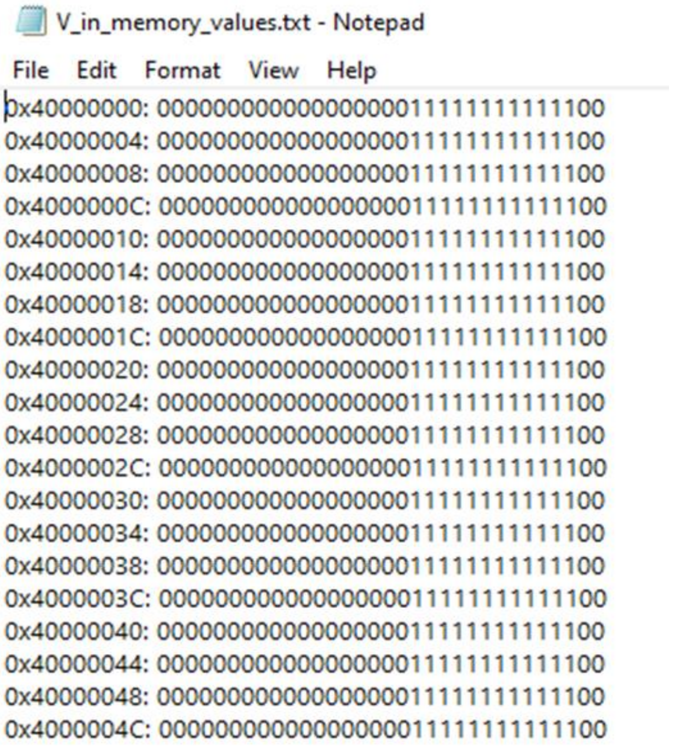

SIMULATION - Behavioral Simulation - Functional - sim_1 - MaxMin_tb
max_min_calculator.v
MaxMin_tb.v
Untitled 1
Name
Value
clk
resetn
theta_data[31:0]
data_in[31:0]
write_enable[3:0]
addr[31:0]
new_theta
current_max[31:0]
current_min[31:0]
vpp[16:0]
New theta_data inserted
150 is the maximum value of all the data_in values (samples). Therefore, current_max=150
-8,192 is the minimum value of all the data_in values (samples). Therefore, current_min=-8,192
Vpp=current_max-current_min=150-(-8,192)=8,342 This occurs during the writing cycle, w_e=f=1111
New theta_data inserted

Figure 12: Max/Min, New Frequency (New_Theta inserted)

It can be seen (figure 12) that, again, the correct values are being calculated and that the writing occurs for 1 clock cycle after every frequency has been sampled completely while advancing the address value by 4 ,as explained in section 3 (Implementation).
These simulation's results confirm the proper operation of 'max_min_calculator' module.

5 Analysis of results

As explained in the previous sections, once the frequency sweep has been finished, the 2 BRAMs contains 2000 values (each) of the peak-to-peak amplitudes of V_{in} and V_{out}.
Using the Red Pitaya's Linux interface, after each run, we ran the 2 C codes that output these BRAMs content into 2 text files, in the configuration: '<Address$_i$>: <content$_i$>' (as explained in section 3).
To validate it, we first looked at the text file of the peak-to-peak amplitudes of V_{in}:



```
0x40000000: 00000000000000000011111111111100
0x40000004: 00000000000000000011111111111100
0x40000008: 00000000000000000011111111111100
0x4000000C: 00000000000000000011111111111100
0x40000010: 00000000000000000011111111111100
0x40000014: 00000000000000000011111111111100
0x40000018: 00000000000000000011111111111100
0x4000001C: 00000000000000000011111111111100
0x40000020: 00000000000000000011111111111100
0x40000024: 00000000000000000011111111111100
0x40000028: 00000000000000000011111111111100
0x4000002C: 00000000000000000011111111111100
0x40000030: 00000000000000000011111111111100
0x40000034: 00000000000000000011111111111100
0x40000038: 00000000000000000011111111111100
0x4000003C: 00000000000000000011111111111100
0x40000040: 00000000000000000011111111111100
0x40000044: 00000000000000000011111111111100
0x40000048: 00000000000000000011111111111100
0x4000004C: 00000000000000000011111111111100
```

Figure 13: V_{in} Values From The Bram, 2000 Addresses

First it can be seen that the addresses are correct (jumps of 4), meaning that the BRAM indeed stores the values in the correct addresses.
Second, we see that the first peak-to-peak amplitudes (taking into account only the 17 LSBs, due to the calculation of the sign extension) are correspond to a value of $\sum_{i=2}^{13} 2^i = 16380$, hence the peak-to-peak amplitude in volts is $\frac{16380}{8191-(-8192)} \cdot 2 = \frac{5460}{5461} \cdot 2 \simeq 1.999\ [V]$. This is a tolerable result which can be explained by the fact that even though each signals' period has a decent number of samples, it's still finite, which can cause this quantization error. We still received a very close result to the one we expected ($2\ [V]$), only 0.02% error, hence negligible. To validate our design, and to check it, we first took the DAC output (OUT1 SMA Connector of the Red Pitaya's PCB). This port outputs the swept signal from the PCB.

Connecting it to the Oscilloscope within the lab, we received the next image:

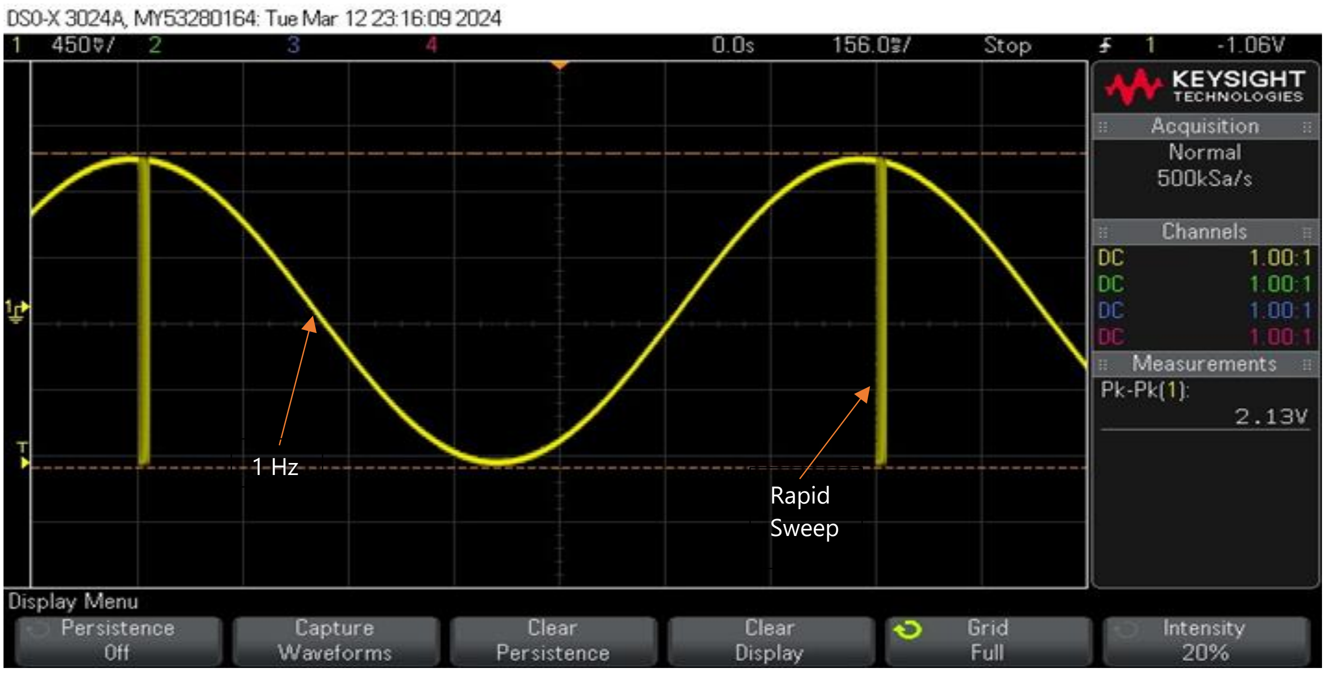


Figure 14: The DAC's Output As Seen Within The Oscilloscope

The 1 $[Hz]$ signal initially appears alongside what may seem like noise for a short period (figure 14).
However, this is not noise but rather the rapid sweep generated by the system, which is significantly faster than the sweep of the 1 $[Hz]$ signal.
It's important to note that the total sweep duration is approximately 1.016 $[sec]$, whereas the sweep of the 1 $[Hz]$ signal alone lasts for 1 second.
This indicates that the first signal occupies roughly 98.4% of the entire sweep duration.
In addition, it can be seen from this print, that the entire sweep is ~7 grid slots long, each lasting 156 $[msec]$. Meaning that:

$$(15)\ t_{sweep} \approx 7 \cdot 0.156 = 1.092\ [sec]$$

This duration is slightly longer than the calculated sweeping duration (section 1.1 in the Implementation). However, given that our system and equipment are not perfect, this outcome is reasonable and expected.
Our result should be a Bode Gain diagram. Meaning we want a plot of gain vs frequency. As mentioned, we chose to plot the results using MATLAB.
We used 2 MATLAB scripts. One that takes the text file and writes each peak-to-peak value to a corresponding csv file in decimal representation.
The other code reads the 2 csv files to 2 vectors: of V_{in} and V_{out}.
Note that the code also normalizes them to be a values between $[0-2]$ $[V]$ since this is the peak-to-peak range.
Afterwards, it computes the gain values in decibels according to the formula:

$$(16)\ gain_{dB} = 20\log_{10}\left(\frac{V_{out}}{V_{in}}\right)$$

This vector is our 'y axis'. Our 'x axis' vector is the frequencies that the system is sweeping.
Our system frequency interval is $[0.931\ Hz - 0.9997\ MHz]$, hence the code also generates a vector of these 2000 frequencies, namely:

$$(17)\ \vec{f} = 0.931, 0.931 + \frac{0.997 \cdot 10^6 - 0.931}{1999}, \ldots, 0.997 \cdot 10^6$$

As we saw earlier, the results for V_{in} are correct and received properly from the BRAM.
For 1st attempt to examine V_{out} results we shortened OUT1 and IN1 of the Red Pitaya's PCB using an SMA-to-SMA cable. This shorts between the DAC's output to the ADC's input. With this configuration we expect to see a constant graph with $gain_{dB} = 0$ at all frequencies, since no filter is connected.
After configuring the system as such, we received within the MATLAB the next figure:

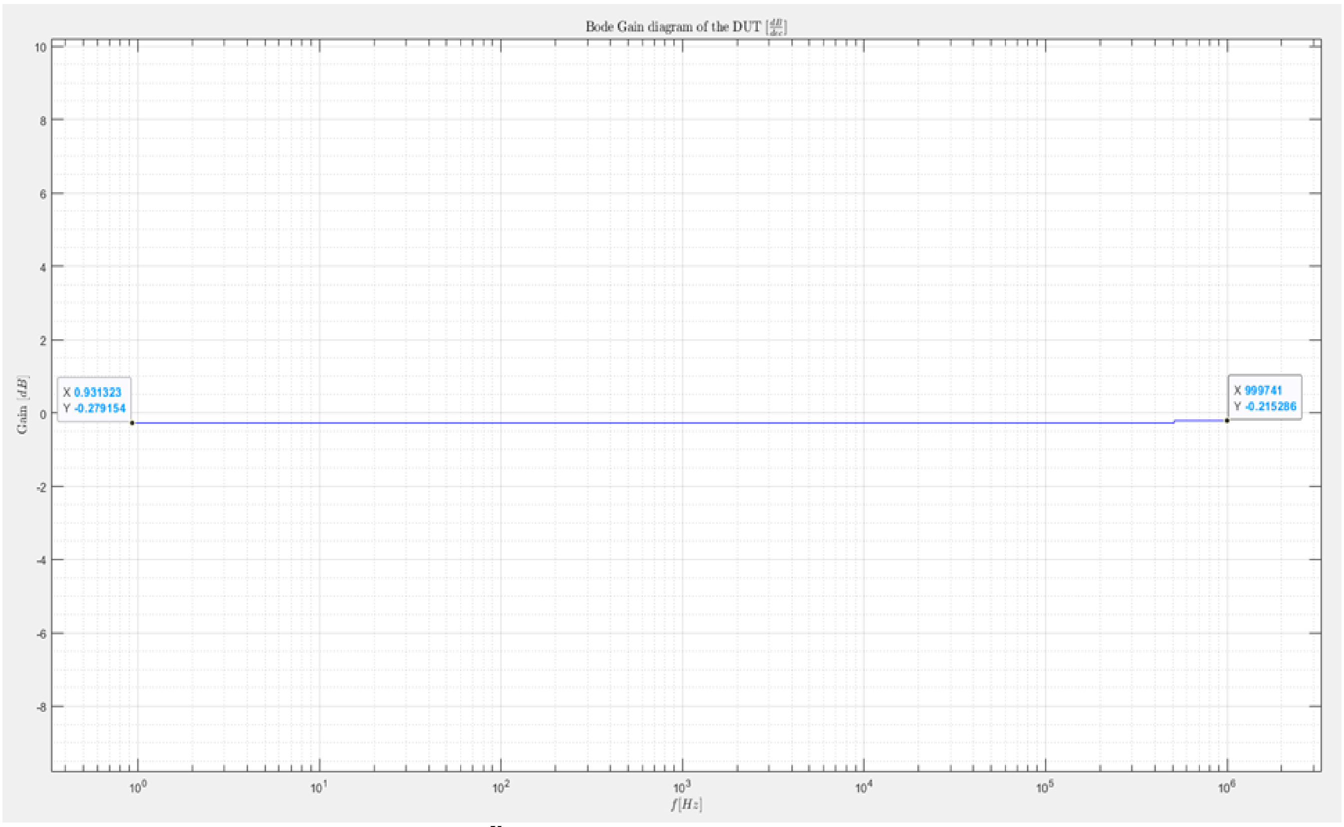


Figure 15: Gain $\frac{V_{out}}{V_{in}}$ Of Shortened Red Pitaya, Without DUT

We observe an "almost" constant line (figure 15).
Across most frequencies, the gain remains around$\sim -0.28\ [dB]$, while for higher frequencies, it hovers around$\sim -0.22\ [dB]$.
Firstly, it's important to recognize that these values are quite small and nearly approach the desired and expected $0\ [dB]$ value. The variations can be attributed to the non-ideal characteristics of the system, such as the small capacitance and resistance of the SMA-to-SMA Cable which leads to its

inherent gain loss, as well as similar effects present in the SMA connectors and the PCB itself.
Furthermore, the variation in gain values observed for higher frequencies can be attributed to the lower number of samples collected in these frequency ranges, leading to quantization errors as previously discussed. This outcome is expected and consistent with the behavior of the system.
Our Next step in examining and testing the system was to plug in a DUT.
To start, we opted for a simple LPF configured through a straightforward series connection of a resistor and a capacitor. We chose these values:

$$\begin{cases} R = 1\ [k\Omega] \\ C = 10\ [nF] \end{cases}$$

This LPF configuration has the known transmission function:

$$(18)\ H(s) = \frac{V_{out}(s)}{V_{in}(s)} = \frac{\frac{1}{RC}}{s + \frac{1}{RC}}$$

Where $s = j\omega$ and ω is the radian frequency, namely $\omega = 2\pi f$.
For our DUT this function will be:

$$(19)\ H_{DUT}(s) = \frac{100{,}000}{s + 100{,}000}$$

The expected knee frequency is:

$$(20)\ f_{-3dB} = \frac{100{,}000}{2\pi} \simeq 15.915\ [kHz]$$

That is to say that we expect this DUT to significantly filter out signals with frequency that is greater than $15.915\ [kHz]$.
We built this filter using the above components and a breadboard, and first connected it to an Oscilloscope to validate its operation, and received the next image:

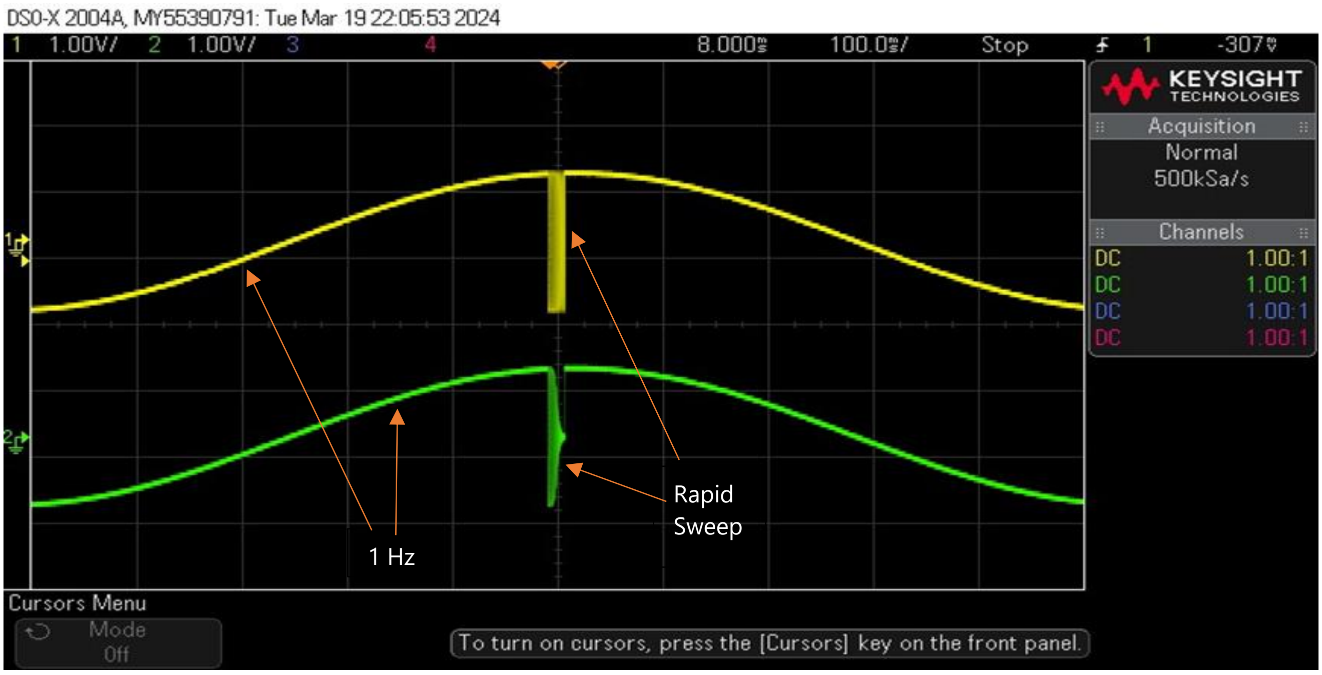

DSO-X 2004A, MY55390791: Tue Mar 19 22:05:53 2024
1 1.00V/ 2 1.00V/ 3 4 8.000s 100.0s/ Stop 1 -307mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
500kSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 1.00:1
Rapid Sweep
1 Hz
Cursors Menu
Mode Off
To turn on cursors, press the [Cursors] key on the front panel.

Figure 16: The DUT's Input Signal (Yellow) Generated From The FPGA Design, and the DUT's Output (Green)

Here (figure 16), again, we can see the $1\ [Hz]$ signal and the rest of the swept frequencies.
For convenience, we zoomed in at the swept frequencies that are above $1\ [Hz]$:

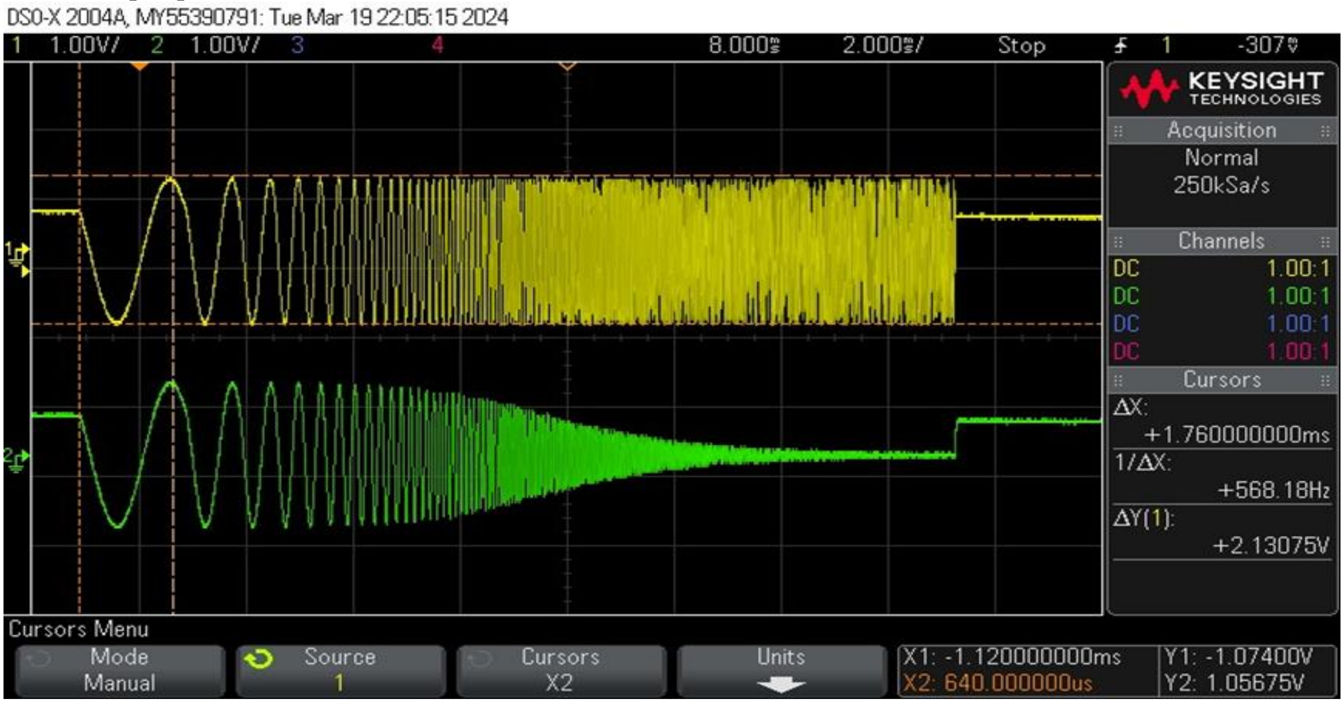
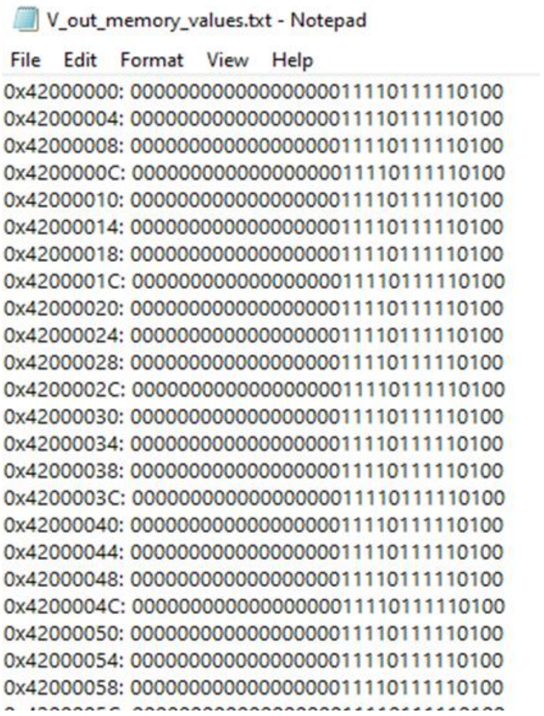


Figure 17: The DUT's input sweep signal with $f \geq 1\ [Hz]$ (yellow) generated from The FPGA design, and the DUT's output (green)

Here (figure 17) the expected LPF behavior can be seen clearly. Initially, the lower frequencies pass through the filter with minimal impact (can be seen clearly between the cursors, representing the 2nd frequency that is being swept), but as the sweep progresses, the higher frequencies are being filtered. This result clearly demonstrates the expected and correct behavior of the frequency sweep generated within the FPGA.

The next result is analyzing the values of V_{out}, which are the values of the ADC's output.
The ADC of the Red Pitaya contains 32 bits, the 16 MSBs are of SMA connector "IN1" and the 16 LSBs are of SMA connector "IN2", these are referred as channels. In each channel the 3 left bits are identical, meaning that effectively each channel is 14 bits, with a sign extension to 16 bits.
We chose one of the channels arbitrary.
First, as explained earlier, we observed the values received for short circuit:



```
0x42000000: 00000000000000000011110111110100
0x42000004: 00000000000000000011110111110100
0x42000008: 00000000000000000011110111110100
0x4200000C: 00000000000000000011110111110100
0x42000010: 00000000000000000011110111110100
0x42000014: 00000000000000000011110111110100
0x42000018: 00000000000000000011110111110100
0x4200001C: 00000000000000000011110111110100
0x42000020: 00000000000000000011110111110100
0x42000024: 00000000000000000011110111110100
0x42000028: 00000000000000000011110111110100
0x4200002C: 00000000000000000011110111110100
0x42000030: 00000000000000000011110111110100
0x42000034: 00000000000000000011110111110100
0x42000038: 00000000000000000011110111110100
0x4200003C: 00000000000000000011110111110100
0x42000040: 00000000000000000011110111110100
0x42000044: 00000000000000000011110111110100
0x42000048: 00000000000000000011110111110100
0x4200004C: 00000000000000000011110111110100
0x42000050: 00000000000000000011110111110100
0x42000054: 00000000000000000011110111110100
0x42000058: 00000000000000000011110111110100
```

Figure 18: ADC's V_{pp} Results For Short Circuit

Here we see the $V_{out_{pp}}$ results (as explained earlier, the 1st address is $0x4200\_0000$). These values are basically the integer value $15{,}860$.
Meaning, correspond to a $V_{out_{pp}} = \frac{15860}{16383} \cdot 2 \simeq 1.936\ [V]$.
This result holds with the configuration we've done, a short circuit, and as explained earlier, we received a value that is very close to the ideal value $2\ [V]$ due to the system's lack of ideality and the parasitic capacitance and resistance of the cable. Hence, we received a constant graph at figure 15, with a minor degradation (from 0 to $\sim -0.28\ [dB]$).
This result is expected and logical considering the above explanation.

Afterwards, we tried doing the same with the simple RC LPF as the DUT (which it's behavior we explained earlier).
In the V_{pp} values that we've read from the BRAM using the text file, we received unexpected values which correspond to very low voltage values, even in lower (and unfiltered) frequencies.
We've examined it for a situation of Open Circuit, meaning where we have not connected any cable nor device to the SMA connector.
In this case, we received very low values, around 300, that corresponds to a voltage peak-to peak levels of $\frac{300}{16383} \cdot 2 \simeq 0.037\ [V]$.
These values are expected and logical. These are the noise values that are driven to the SMA connector from the non-ideal environment (atmospheric noise).
Hence, our 1st conclusion is that the ADC and the way we are reading values from it is correct.
The factor that we should have, and tried to, consider is the propagation delay within the DUT itself. To do so, we'll need to calculate this time delay given our DUT, which is RC LPF.
It's important to note that the system's purpose was not to characterize the device's behavior, but rather to compute its gain at different frequencies independent of the device.

In this DUT, the time delay depends on the frequency and can be calculated using the phase difference between the output sinusoidal wave and the input sinusoidal wave. Meaning:

$$(21)\ t_d = \varphi \frac{T}{2\pi} = \varphi \frac{1}{f}$$

For this RC LPF, the phase difference φ, can be calculated by:

$$(22)\ \varphi = arg\big(H(s)\big) = arg\left(\frac{\frac{1}{RC}}{s + \frac{1}{RC}}\right) = \tan^{-1}\left(\frac{-\omega \frac{1}{RC}}{\frac{1}{R^2C^2}}\right) =$$
$$= \tan^{-1}(-\omega RC) = -\tan^{-1}(2\pi f RC)$$

The minus sign is irrelevant for the size of the time delay, only the size.
Meaning, the time delay as a function of frequency is:

$$(23)\ t_d = \varphi \frac{1}{f} = \frac{\tan^{-1}(2\pi f RC)}{f}$$

For the device we chose we'll receive:

$$(24)\ t_d = \frac{\tan^{-1}(2\pi f \cdot 10^3 \cdot 10 \cdot 10^{-9})}{f} = \frac{\tan^{-1}(2\pi f \cdot 10^{-5})}{f}$$

Plotting this function in our frequency sweep range will give:

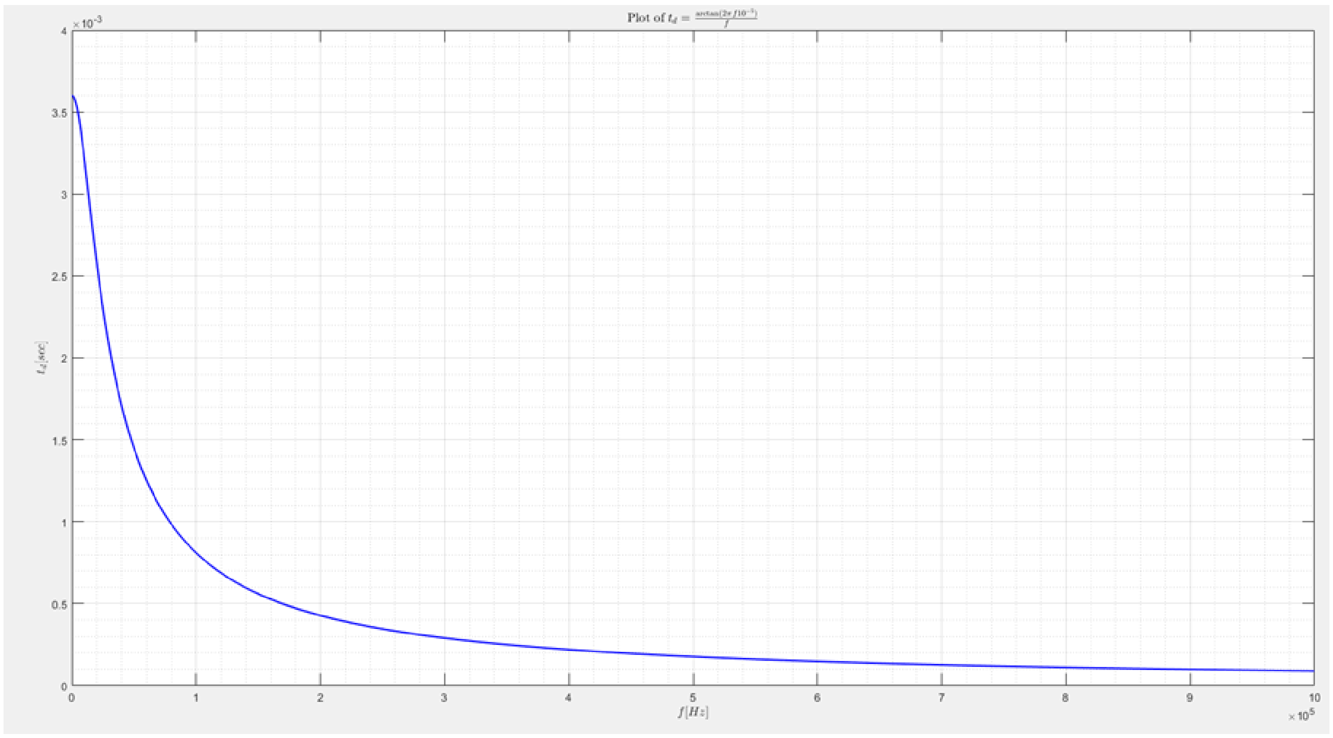


Figure 19: The LPF RC Time Delay As Function Of Frequency

This function implementation within Verilog is rather complex.
Also, as it can be seen from the graph, this delay behavior is not constant, making it challenging to find a suitable approximation that would simplify implementation in RTL (Verilog).
Note: there is a Verilog function, called \$atan2(x,y)\$, however, it cannot be utilized for synthesis purposes, as it is solely intended for simulation.
Moreover, even if we did manage to solve this issue, the solution would only apply to this specific DUT. Using another DUT will have us to calculate t_p and implement it in a different manner.
Due to the reasons discussed above, we were not able to produce results for this DUT, and due to time considerations of the semester, we were unable to resolve this issue in a timely manner.

6 Conclusions and further work

Conclusions:

1. The frequency sweep performs as intended, even accounting for inherent gain reductions (such as board loss, connector, and cable losses, etc). These reductions can be explained by the lack of ideality of the system's components.

2. The sweeping time is optimized which results in a fast result with minimal loss of accuracy.

3. The peak-to-peak calculation (using $max\_min\_calculator$) performed as expected and yielded conclusive and accurate results.

4. The use of BRAM blocks using the Red Pitaya's Linux Interface performed as intended. In addition, using the C scripts to read the values within the BRAM works as intended.

5. The total numbers of frequencies to be swept is set to be 2000. While this setting allows for a dense sweep and maintains memory efficiency, it also introduces an inherent resolution error due to the final number of steps. Consequently, the accuracy of knee frequency identification is limited.

6. The use of Red Pitaya's converters (DAC and ADC) works as intended, which saves the need to implement them manually and money.

7. The system works as expected for the basic case of short circuit. Also, for open circuit, the noise coming to the system is negligible.

8. The DUT delay timing issue was not resolved due to time constraints of the semester and to its high implementation complexity.

For the future of this project, we suggest the following:

1. Resolving the propagation delay issue. To do so, we suggest the following options:
   a. Implementing a block that calculates the propagation delay of the DUT, then adding it to our implementation such that this issue will be resolved.

   b. Reimplementing the peak-to-peak calculation (namely $max\_min\_calculator$) of the output signal (which is passing through the ADC) such that it won't need to take the propagation delay into account.

2. Implementing a Phase Shift Detector and adding it to this project in order to have a complete Bode Analysis System. The method to do so is to count clocks.

3. Implementing the system with a changing sweeping limit while maintaining the same 2000 steps within the sweep. For example, implement the system such after each sweep, f_{max} is decreasing by 10% of its initial value and f_{min} is increasing by 10%.
   This implementation resolves the inherent resolution error but requires extensive time for calculations, and a bigger cost of hardware.

4. Implementing a system that doesn't rely on the DDS Compiler, for example, by utilizing a signal generator (commonly available in most laboratories within the faculty). This approach will require a redesign of the peak-to-peak calculation but will save the use of the DDS Compiler block. This approach could potentially conserve FPGA resources by reducing the need for extensive blocks like the DDS Compiler. Additionally, it may lead to improved timing as the FPGA would not be responsible for generating the swept signals independently.

7 Project Documentation

The Project is saved within a compressed ZIP file that contains the following:

1. Folder named "4_frequency_counter".
   Note: The ADC block we've used is inside of Anton Potočnik "frequency_counter" project. Therefore, altering the project's name could result in internal issues.
   Moreover, the project might consist of blocks, written by Anton for this frequency counter project, which we've not used.
2. 2 C files, for reading the values stored in the BRAMs of V_{in} and V_{out}, they are named "Reads_2000_values_32bit.c" and "Reads_2000_values_32bit_Vout.c" respectively.
   These files should be inside the Red Pitaya's Linux Interface (can be done using the WinSCP software).
3. 2 Matlab files:
   a. "Text_to_CSV.m"- this file takes the text files of V_{in} and V_{out} and creates CSV files with the V_{pp} values in decimal representation.
   b. "Bode_gain.m"- this file reads both V_{in} and V_{out} CSV files, calculates and plots the gain of the system, using the known frequency sweep range, in logarithmic scale.
4. A PDF containing this document.

The ZIP file is saved in the next GitHub repository:

8 References

Here are some examples of references, and how they should be included.

Links:
[1] Red Pitaya's User's Manual:
Red Pitaya's User's Manual
[2] Xilinx Zynq7 Documentations:
Xilinx Zynq7 Documentations
[3] Anton Potočnik Research website:
FPGA » Anton Potočnik - research website (antonpotocnik.com)
This site contains projects for beginners in Red Pitaya, which we used for the Red Pitaya's ADC implementation.
[4] ASIC World Website:
Verilog Tutorial (asic-world.com)
Contains detailed explanations for Verilog RTL.

Academic Courses:
[1] Linear Circuits and Systems, School of EE, Tel-Aviv University.
[2] Digital Electronic Circuits, School of EE, Tel-Aviv University.