UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Design and implementation of an audio band digitally tunable analogue filter

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Joan Francesc Serra Bou

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONIC ENGINEERING

Advisor: Jordi Cosp

Barcelona, July 2015

Title of the thesis: Design and implementation of an audio band digitally tunable analogue filter

Author: Joan Francesc Serra Bou

Advisor: Jordi Cosp

Abstract

A Digitally Tunable Analogue Filter with central frequency (ω_0) and quality factor (Q) control loops tuned to a reference sinusoidal signal frequency is designed. The system, implemented with discrete components mounted on a PCB, is based in a Continuous–Time filter formed by a band-pass filter where the resistors have been substituted by SPI-controlled digital potentiometers in order to allow a digital control of the analogue circuit. The system is implemented in a Master-Slave topology where the master includes both control loops to obtain the potentiometer parameters while the slave replicates master parameters to filter an arbitrary input signal. Both control loops are FPGA based allowing a real-time control. The main objective is to achieve a system capable of tuning the central frequency and the quality factor. In order to proof the system functionality and performance a PSPICE simulation, bode diagrams and THD measurements are presented in this document.

Resum

S'ha dissenyat un Filtre Analògic Digitalment Sintonitzable amb llaços de control de la freqüència central (ω_0) i del factor de qualitat (Q) sintonitzat a la freqüència d'una sinusoide de referència. El sistema, implementat amb components discrets muntat sobre PCB, està basat en un Filtre de temps continu format per un filtre passa-banda on els resistors s'han substituït per potenciòmetres digitals controlats per "SPI" per a permetre un control digital del circuit analògic. El sistema està implementat amb una topologia Mestre-Esclau on el mestre inclou ambdós llaços de control per a obtenir els paràmetres del potenciòmetres. L'esclau en canvi replica els paràmetres del mestre per a filtrar un senyal arbitrari d'entrada. Ambdós llaços de control es basen en "FPGA" permetent un control a temps real. L'objectiu principal és aconseguir un sistema capaç de sintonitzar la freqüència central i el factor de qualitat. Per a demostrar la funcionalitat i qualitat del sistema en aquest document es presenten una simulació en PSPICE, diagrames de Bode i mesures del THD.

Resumen

Se ha diseñado un Filtro Analógico Digitalmente Sintonizable con lazos de control de la frecuencia central (ω_0) i del factor de calidad (Q) sintonizado a la frecuencia de una sinusoide de referencia. El sistema, implementado con componentes discretos montado sobre PCB, se basa en un filtro de tiempo continuo formado por un filtro pasa-banda donde los resistores se han substituido por potenciómetros digitales controlados por "SPI" para permitir un control digital del circuito analógico. El sistema se ha implementado con una topología Maestro-Esclavo donde el maestro incluye los dos lazos de control para obtener los parámetros de los potenciómetros. El esclavo en cambio replica los parámetros del maestro para filtrar una señal arbitraria de entrada. El objetivo principal es conseguir un sistema capaz de sintonizar la frecuencia central y el factor de calidad. Para demostrar la funcionalidad y la calidad del sistema en este documento se presentan una simulación PSPICE, diagramas de BODE y medidas del THD.

Acknowledgments

I dedicate this thesis to my beloved family. Especially to my mother who has given me all the strength to complete this final thesis, as a finishing point of eight years of engineering study.

I would like to acknowledge the support of Jordi Cosp Vilella who gave me generous access to the facilities and equipment of the Department of Electronic Engineering of UPC, and also guided me through this thesis with great explanations and advices.

I am also grateful to Herminio Martínez-García for his assistance in the ORCAD simulation.

Table of contents

List of Figures

List of Tables

1. Introduction

This project is intended to study and implement an audio band DTAF (Digitally Tunable Analogue Filter) with central frequency (ω_0) and quality factor (Q) control loops tuned to a reference sinusoidal signal frequency. Then for different audio band frequencies of the reference signal the DTAF must automatically adapt its own central frequency to that new frequency.

In order to proof the concept the DTAF is fabricated with discrete components mounted on a PCB. To avoid complex fabrication process and others difficulties only through-hole packages are selected.

The DTAF is based in a CTF (Continuous-Time Filter) formed by a second order active band-pass filter where the resistors have been substituted by digital potentiometers in order to allow a digital control of the CTF. Such digital potentiometers are controlled by a FPGA trough 4-wire SPI communication.

Also, the DTAF is implemented in a Master-Slave topology. This technique allows to separate the control loops from the circuit that truly filters the arbitrary input signal. Then in the master filter, where the central frequency and quality factor control loops operate, the digital potentiometer parameters are obtained and then sent to the slave filter. So, the slave filter is not involved within the feedback loops and is dedicated to filter the input arbitrary signal.

Although the feedback signals are originated in the master filter board and some circuitry required for the control loops is placed also in the master filter, the control logic is completely implemented in the FPGA platform. Then both control loops are FPGA based allowing a real-time control. Also, in order to have a fast control all logic blocks have been designed in a concurrent way except some sequential blocks that control the SPI-communications.

The main objective is to achieve a DTAF system capable of tuning the central frequency and the quality factor. To verify the design a PSPICE simulation to obtain a bode diagram is presented. Finally, in order to proof the system functionality and performance a bode diagram, SPI-waveforms and THD experimental measurements are obtained from the fabricated DTAF and presented in this document.

1.1. Topology from previous works

The basic topology of the DTAF designed is a second order band-pass CTF with TQE (transimpedance Q-enhancement). The design detailed in figure 3 in [1], it is a project developed in the Department of Electronic Engineering of UPC. In this work only the CTF topology is reused.

1.2. Statement of purpose

The first objective of the work is to design and fabricate a DTAF with a frequency tunable range of at least one decade and focused around the decade of 100 Hz to 1 kHz, where the desired frequency range is from 300 Hz up to 10 kHz. While the second goal is to be able to tune the quality factor of the DTAF implemented.

Also, in [1] is shown that under some conditions chaotic behavioural can occur within a CTF with an automatic control. Then the third goal to reach within this project is to avoid such chaotic behavioural.

As the implemented control loops are based on a FPGA. Notice that to ensure a proper control, the FPGA resources must be used in a concurrent way whenever is possible. Because the fourth goal is to achieve a real-time and fast control. So, in this project there is no microcontroller or processor involved. Hence there is no software, only embedded hardware.

1.3. Work plan

The work load has been spread in several steps. From recognizing the whole system to implement every small detail.

a. Implement and simulate the proposed CTF with variable resistors

   The first task is to identify, implement and simulate the CTF with variable resistors. The CTF with variable resistors must be implemented and verified within laboratory facilities, to have a first impression of the circuit functionality, and then simulated to compare later the results with the DTAF complete system.

b. FPGA Platform analysis

   As the control loop for the tunable filter is FPGA based, an FPGA platform is needed. So, second task is to get used to the chosen FPGA platform, and identify each peripheral or resource required.

c. Implement SPI communication protocol for Digital Potentiometers Interfacing

   As the automatic control loops technology solution is based on digital potentiometers, and usually this components communicates by I2C or SPI, one of this protocols must be implemented within the FPGA system. In this work SPI is used because, is simpler to use than the I2C protocol and there are enough I/O.

d. Implement a controller for ADC Interfacing

   The ADC (Analogue to Digital Converter) that incorporates the FPGA platform needs a controller to interface with FPGA logic. Which must be implemented according to requirements of the tunable filter.

e. Implement a central frequency control loop by phase matching

   The first automatic control loop for the DTAF is a central frequency loop. In order to implement it, the phase mismatch from input and output of the filter must be first detected, and then reduced with a control action.

f. Implement a quality factor control loop by amplitude matching

   The second automatic control loop for the digitally tunable analogue filter is a quality factor loop. In order to implement it, amplitude mismatch from input and output of the filter must be detected, and then reduced with a control action.

g. PCB design and fabrication

h. Test the DTAF operation. And compare results with first task

1.4. Eventualities

Firstly the Spartan-3A Starter Kit platform was selected as it was available at the laboratory facilities, and had all the required elements to implement the DTAF control loops. But after some eventualities, FPGA platform was switched to Nexys4 Artix-7. Then an analysis about what both platforms include and how to work with them was needed.

The ADC (Analogue to Digital Converter) that incorporates the Spartan-3A Starter Kit uses the same communication protocol as the Digital potentiometers, so the code could have been reused despite of some differences. But as the whole FPGA platform was changed into the Nexys4 Artix-7, a brand new controller had to be implemented and more implementation and integration time was required.

2. State of the art of the technology used or applied in this thesis

The DTAF system is formed by two main parts, the CTF and the control loops. About the CTF, there are several choices to consider as a basic element for the CTF. First of all are presented several technologies used in audio band DTAF systems. Then different investigations of tunable filters around MHz band are also reviewed.

Although the control loop can be either direct or indirect, most of the projects on the literature choose the indirect control.

Notice that DTAF is not a recent concept since there are patents about the idea of tunable filter system. In [2], a tunable bandpass filter system is patented. In this work the filter central frequency is tuned from the frequency of an input signal. Even there is quality factor adjustment.

2.1. Audio band Tunable Filters

One option to implement a DTAF system is to use Operational Transconductance Amplifiers (OTA). In [3], a band-pass filter with automatic gain control is proposed. To achieve the tunability of the filter they actuate directly on the transconductance at transistor level. In the paper it shown the capacity of tuning the central frequency from 70 Hz up to 4 kHz, and the quality factor it can be adapted from -2,5 dB up to 8 dB.

Others works consider to implement MOSFET resistive Circuit (MRC) in order to directly substitute the resistors. In this case OPAMPs are used instead of OTAs.

In [4], a fifth order Bessel filter is designed employing passive resistors and current-steering MOS transistors operating in the triode region. Where in order to reduce the distortion of the nonlinear devices a feedback loop is applied. So, to have low-distortion filter instead of substituting the resistor elements by MOSFETs they implement a structure formed by a passive resistor, a MOSFET and a capacitor. And is detailed that it is capable to reduce the THD. In this work the cutoff frequency of the implemented filter can be tuned from 2 kHz up to 35 kHz.

In the next paper, [5], the MOSFET-C it is used as a base element of the DTAF. The filter implemented method is indirect, so a master-slave topology is designed. Actually the architecture of proposed DTAF have one master for a central frequency loop which is a voltage-controlled oscillator (VCO), and a second master for a quality factor loop which is a voltage-controlled filter. Is detailed that the filter can tune the central frequency from 2 kHz up to 20 k Hz, and the quality factor from 1,78 up to 12,92.

Another design with MRC-C (MOSFET-C resistive circuit) is presented in [6]. This work has a similar DTAF architecture as the last paper mentioned, following a master-slave topology. Also it have a central frequency and a quality factor control loops while is based on a Transimpedance Q-Enhancement (TQE) structure. It is stated that the design has been fully implemented at full-custom level and bode diagram demonstrate a tunability from 2 kHz up to aprox. 18 kHz, and the possibility to tune several Q-set points.

In a different point of view a reference design, [7], presented by Texas Instruments implements a DTAF based on a Multiplier Digital-to-analogue Converter component, which contains two current-steering R-2R ladder DACs. So all the resistors involved in the filter are substituted by the MDAC. In the results they exhibit a frequency range of 2 Hz up to

28,76 KHz and a quality factor range from 0,46 up to 2,14. Even a third control loop is added in order to control the gain, having a gaing range from -53,31 dB up to 6 dB.

2.2. MHz Tunable Filters

Although the DTAF presented in this project is designed for audio band operation, similar techniques are used for filters working around MHz frequencies.

In [8], a wide tuning range is achieved by implementing triode-biased input MOSFETs, with an adaptive DC-blocking. A 4th-order Butterworth low-pass filter is proposed and demonstrates a tuning capability range from 0,5 MHz up to 12 MHz. This work it is also designed at transistor level.

On the other hand the next works consider to structure the tunable filter around a current conveyor.

In [9], a tunable second order bi-quad filter is designed using current conveyor based on a FPAA. The FPGA is used to test the digital hardware part while the FPAA is used to test the analogue part. As in such work is considered to embed all the filter system inside an IC chip the use of programmable resistor arrays is discarded. They justify that it consumes a large area and that cannot be easily fabricated within the same IC chip. Tuning the filter by changing the OTA trans-conductance gain, is also discarded because is detailed that biasing circuits are complicated and sensitive to process variation. Finally, the basic tunable element on this work is a digitally programmable current conveyor based on current division network. They conclude that the filter designed can tune a band-pass filter output cut-off frequency from 576 kHz to 4,78 MHz, and a low-pass filter output from 16,4 kHz to 11,6 MHz.

Another DTAF implemented with current conveyor is presented in [10]. More precisely the basic element used is defined as Digitally Controlled Differential Voltage Current Conveyor (DC-DVCC). In this case a HPF and LPF is designed also at transistor-level but in this case the tune is discrete, having only a 3-bit word to create only 8 different cutoff frequencies. Finally, they exhibit several simulation results with a range of 1 MHz up to 7 Hz for the HPF, and 140 kHz up to 1 MHz for the LPF.

Finally another design, [11], demonstrated on rapid-prototyping FPAA implements 55 digitally tunable OTAs in a Hexagonal Lattice, allowing intuitive mapping of the Gm-C filter cells. Although the frequency range achieved goes from 29 MHz up to 82 MHz it only allows 5 different steps.

3. Project development

Next are explained all the project steps, from the CTF proposed topology to the DTAF system results.

3.1. Proposed Continuous-Time Filter

3.1.1. Topology and component selection

The topology used for the CTF is shown in *Figure 1*. This circuit is a second order continuous-time filter called TQE (transimpedance Q-enhancement) [1]. It is formed by only two operational amplifiers, three capacitors and four resistors. As this circuit is used a base for the DTAF, the value component selection is done having in mind the desired frequency range tunablity, which should be at least one decade wide, in the audio band and centred around hundreds of Hertz frequencies.

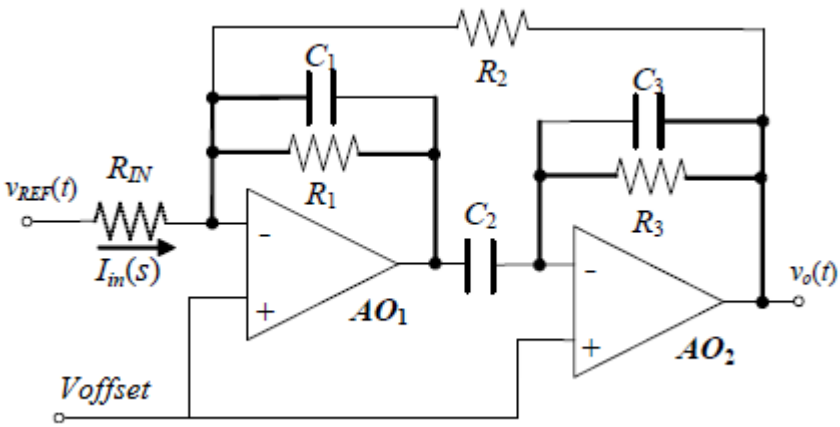


Figure 1. 2nd order band-pass filter TQE (transimpedance Q-enhancement) topology.

The first parameter to fix will be the central frequency (ω_0).Then as the circuit is a second order band-pass filter, and fixing $R_{in} = R_1 = R_3$ and $C_1 = C_2 = C_3$, it depends on the next formula:

$$f_0 = \frac{1}{2\pi R_{in} C} \qquad [1]$$

At this point the capacitor value and the variable resistor range have to be chosen depending on which range of frequencies are needed. On one hand, to work with frequencies around hundreds of Hertz a few µF capacitor must be chosen. But to reach higher frequencies (tents of kilo Hertz) a lower value, a few nF capacitor must be chosen instead. On the other hand, a 100 kΩ digital potentiometer range is supposed. Then a variation of the resistor would be possible from minimum total potentiometer resistor (R_{min}) to 100 kΩ (R_{max}), through a middle point (R_{middle}) of 50 kΩ.

As the proposed DTAF is intended for audio band, which means 20 Hz to 20 kHz, the capacitor is chosen to have an ideal frequency range operation from 234 Hz to 93,62 kHz. Then three calculations are done to verify that the capacitor value chosen adjusts.

$$f_{0(max)}|R_{min} = \frac{1}{2\pi \times 250\ \Omega \times 6.8\ nF} = 93{,}62\ kHz$$

$$f_0|R_{middle} = \frac{1}{2\pi \times 50\ k\Omega \times 6.8\ nF} = 468{,}10\ Hz$$

$$f_{0(min)}|R_{max} = \frac{1}{2\pi \times 100\ k\Omega \times 6.8\ nF} = 234{,}05\ Hz$$

With C1 = C2 = C3 = 6,8 nF and digital potentiometers of 100 kΩ the DTAF have an ideal middle point central frequency of 468,10 Hz. Notice that although the maximum frequency ideally reached with the values selected is outside the audio band, major of the variable resistor range is placed in the hundreds Hertz frequencies.

On the other hand, the resistor R_2 fixes the quality factor (Q) of the filter. To obtain a flat response the quality factor must be set to Q = 1, in which way the output filtered signal doesn't present any amplitude variation versus the input signal. The next formula states how to fix the Q parameter:

$$Q = \frac{1}{\left(2 - \frac{R_1}{R_2}\right)} \qquad [2]$$

Then for Q = 1 the resistor R_2 will have the same value as R_1.

In the results chapter, 4.1, gain and phase real values will be shown for the ideal DTAF.

3.1.2. OPAMP Selection

A suitable operational amplifier to be used within the DTAF could be any OPAMP intended for audio processing applications. The first requirement needed for the application is to have a quad OPAMP in order to reduce the PCB area and the number of components. It also must have a PDIP package for an easy rapid-prototyping. As it is intended for audio application a higher bandwidth is not a must, so an OPAMP with a few MHz bandwidth it is enough.

Two of the OPAMP are used as comparators, and their output is evaluated by the FPGA logic which finally operates at 100 MHz. So, in order to be able to read any output variation between each FPGA clock period a slew rate of 330 V/µs would be required. But as the comparator outputs, at the end, trigger the digital potentiometer block which requires at least 3 µs to operate there is no need to have such a fast slew rate OPAMP. Since there is no need to trigger each 50 ns a block that lasts 3 µs to operate, the desired slew rate OPAMP would be able to raise a voltage of 3,3 V with about 500 ns.

Another important fact for the OPAMP chose is that the power supply source is the same FPGA platform, so it is 3,3 V. Then it is better to have a rail-to-rail input/output OPAMP with a power supply range that accept such 3,3 V.

Finally the selected OPAMP should have a low Total Harmonic Distortion (THD) to minimize the THD of the DTAF.

Then, the selected OPAMP is the MCP6024, [12]. The MCP6024 is quad, can have a PDIP package, it has 10 MHz bandwidth, a 7V/µs slew rate, it is a rail-to-rail I/O, and has a THD+N of 0,00053% and a power supply range from 2,5 V to 5,5 V.

3.2. DTAF indirect control

The DTAF proposed have an indirect control, so it follows a master-slave scheme. There is a filter (slave) that receives the signal to be filtered as an input, while the output is the desired filtered signal. The slave filter does not have any controls loop involving his input or output, instead receives the control commands directly from a second filter (master) which has all controls loops that correctly tunes both the master and the slave filter. The master filter reconfigures itself depending on an external reference signal.

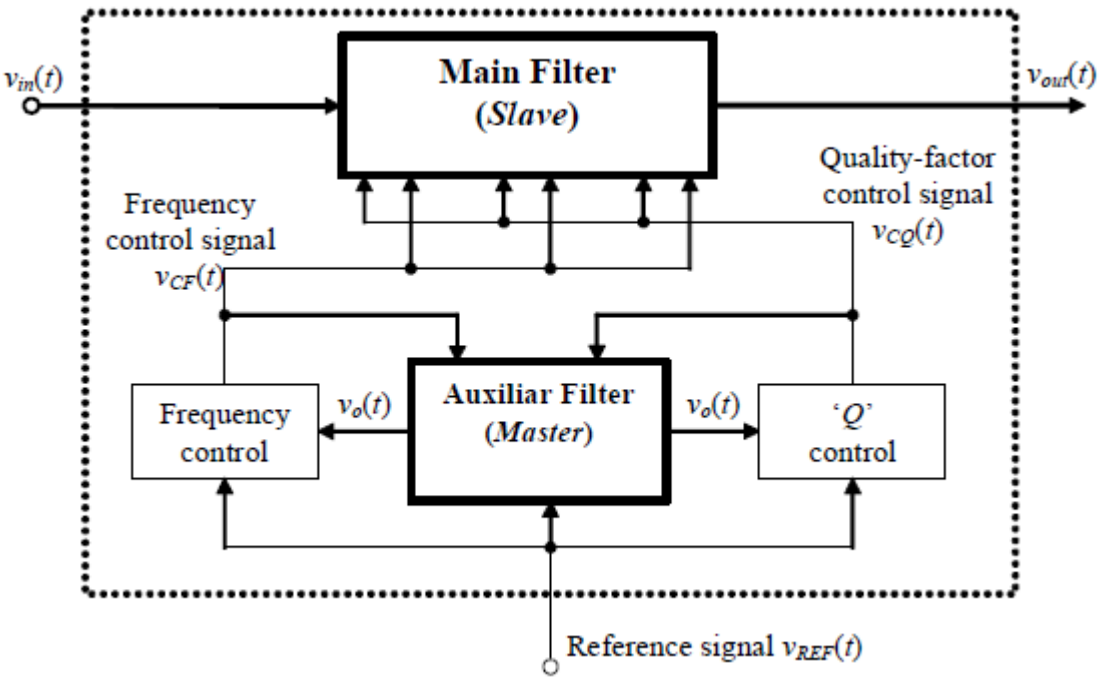


Figure 2. Master-slave scheme of CTF based tunable filter [1]

The CTF topology and components have been explained by now. But, as the objective of the project is to design a DTAF, it is needed to replace all resistors for variable resistors, and add detection and control blocks. Then, according to the above diagram, *Figure 2*, the complete DTAF will be formed by a master filter composed by a CTF with digital potentiometers and central frequency and quality factor control loops; and a slave filter also composed by CTF topology with digital potentiometers but without control loops.

As the slave filter design only implies a connection exercise, it will not be commented again during the project until the PCB design step, chapter 3.8.

3.2.1. Control action of central frequency and quality factor

The implementation of the variable resistor must allow a digital control. That is why a conventional potentiometer is not suitable. So, the already chosen digital potentiometer must be implemented as the variable resistors in the CTF topology, see *Figure 3* below.

Having controlled variable resistors within the filter means that it is possible to control the parameters where this resistors are involved. So, there is a control on the equations (1) and (2), which is a control of the central frequency and a control of the quality factor respectively.

Then the master filter is formed by a CTF topology with two automatic control loops. First of all there is a loop to control central frequency which involves the resistors R_{in}, R_1 and R_3. And secondly there is a loop to control the quality factor which involves only the resistor R_2.

So two action blocks will be required, one for **central frequency control**, and other one for **quality factor control**.

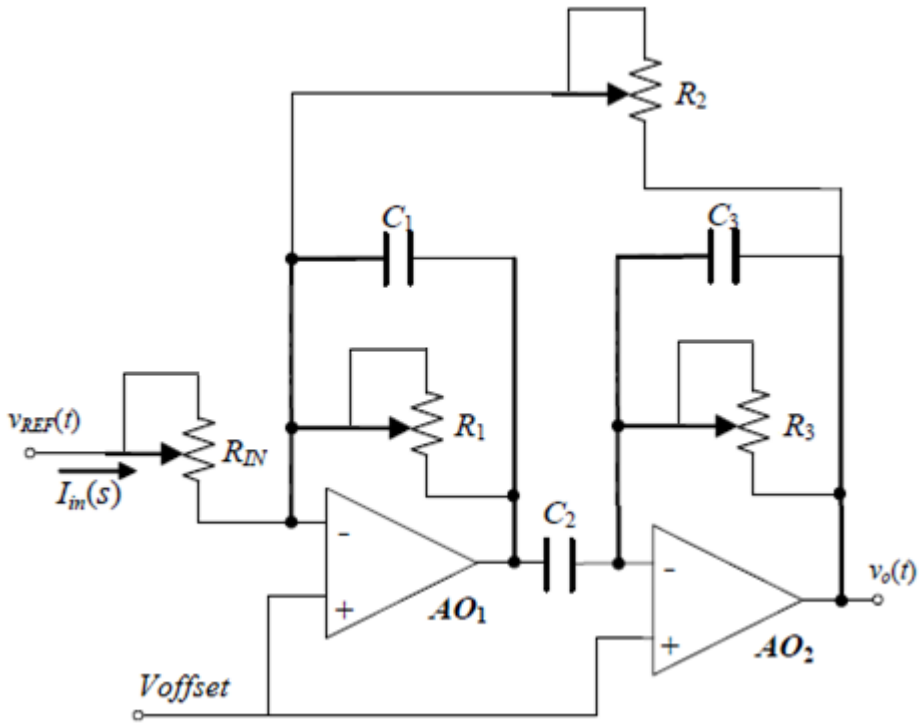


Figure 3. 2nd order band-pass filter potentiometer-based with TQE topology.

3.2.2. Phase and amplitude mismatch detection

In any control system to properly apply a control action, first the detection of an error must occur. That is why each of the loops needs a detection block. But, to understand which parameter errors have to be detected is necessary to understand the nature of the circuit. So, one important fact of a tuned filter circuit is that the input signal is phase matched with the output signal. It is shown in the next image, *Figure 4b*:

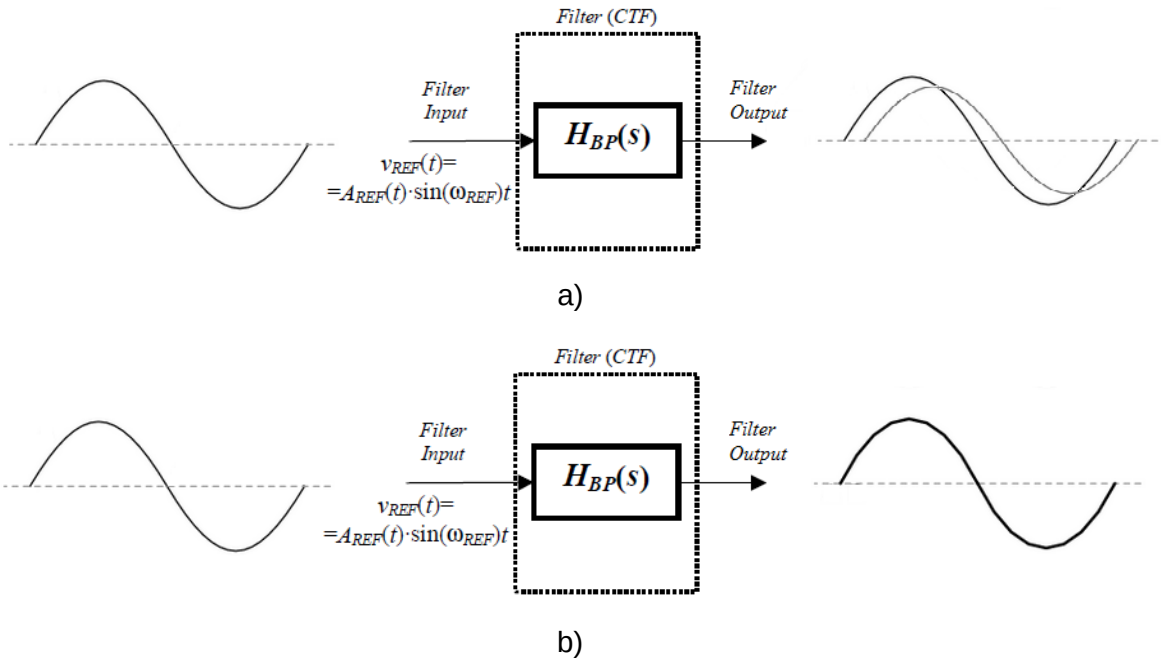


Figure 4. a) Not tuned filter generates a phase and amplitude mismatch. b) Tuned filter generates phase and amplitude matched output signal.

That only occurs when the filter is tuned, so when the frequency of the input signal matches the central frequency of the filter. If the filter is not tuned a phase mismatch will occur. Then a circuit that can detect a phase mismatching between the input signal and the output signal, would generate the error needed to properly control the central frequency control loop.

Then, the central frequency loop will need a **phase mismatching detector** to operate between the input and the output of the master CTF.

In the same way, if the filter is active and is configured to have a gain, G = 1, when the filter is tuned, the gain at the output is the same as in the input. But when the input signal frequency it is higher or lower than the central frequency an amplitude mismatch occur between the input signal and the output signal (*Figure 4a*). Then a circuit that can detect that mismatch would be suitable to generate an error signal to properly control the quality factor loop.

Then, the quality factor loop will need an **amplitude mismatching detector** between the input and the output of the master CTF.

3.2.3. Master filter

Once the filter topology is selected and the detection and action blocks are used for closing the control loops, a control system with two loops is formed which will became the Master filter. So the whole CTF with automatic tuning loops system, working as the Master Filter, is shown in the next diagram block, *Figure 5*:

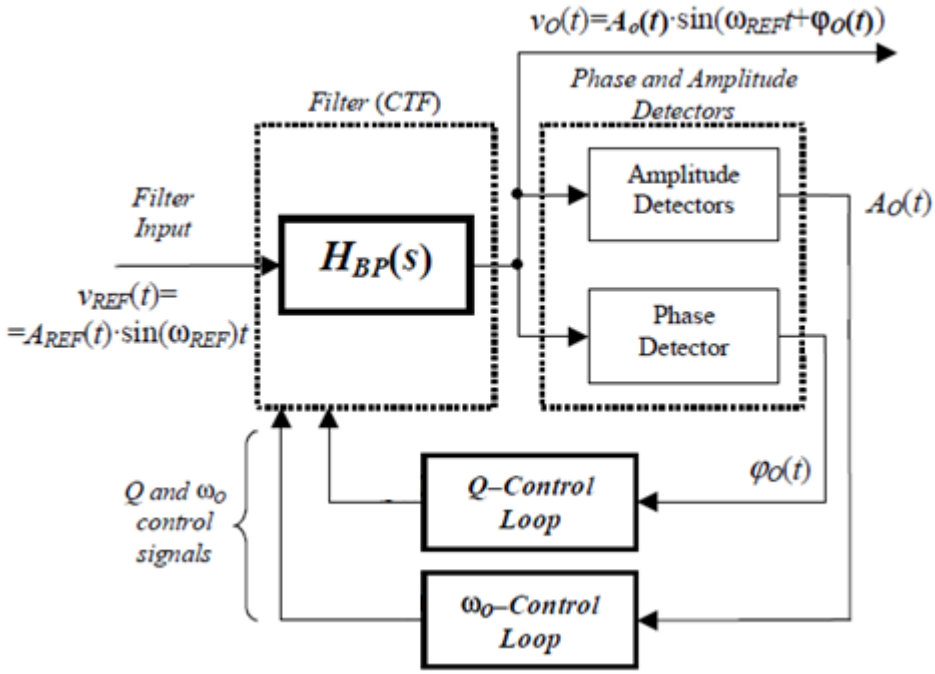


Figure 5. Master filter block diagram

3.3. FPGA Platform Analysis

This project does not require a large amount of logic gates to operate, because there are no complex arithmetic counts, there are only a few I/O signals to control, and only few peripherals are used. Most important is the speed achieved within the control loop which is the reason why an FPGA controller is desired.

During this project two FPGA boards have been used due to some eventualities. At the beginning a Spartan-3A Starter Kit was used because was available at the laboratory facilities and fulfils all the requirements to operate within the proposed DTAF. But when almost all he design was implemented on the Spartan-3A an eventuality (see 3.3.4) made impossible to fit all implementation within the FPGA. That is why it was decided to switch to a newer Nexys4 Artix-7 FPGA board.

3.3.1. Spartan-3A Starter Kit analysis

The Spartan-3A Starter Kit, [13], is a board from Xilinx based on Spartan-3A FPGA. Spartan-3A is a 90 nm process technology FPGA chip which was released around 2007. The exact reference of the FPGA is Xilinx 700K-gate XCS700A. Xilinx presents this FPGA family as perfect fit on high-volume, cost-sensitive, I/O-intensive electronic applications. Although the Spartan-3A family is rather a low-end family product from Xilinx, it is completely suitable for the project application. In the next table, *Table 1* (data obtained from Xilinx), there is a comparison of the bottom to top families of Xilinx, including the Spartan-3A. Most important of this table is to notice that the Spartan families does not have the XADC core. This means that all boards with a Spartan FPGA will require an external ADC chip if desired.

Features	Spartan-3A	Spartan-6	Artix-7	Kintex-7	Virtex-7
Logic Cells (K)	25	147	215	478	1,955
Block RAM (Mb)	0.5	4.8	13	34	68
DSP Slices	-	180	740	1,920	3,600
Transceiver Count	-	8	16	32	96
Analog Mixed Signal (AMS)/XADC	-	-	XADC	XADC	XADC
I/O Pins	512	576	500	500	1,200
I/O Voltage	1.2V – 3.3V	1.2V – 3.3V	1.2V – 3.3V	1.2V – 3.3V	1.2V – 3.3V
Process Technology	90 nm	45 nm	28 nm	28 nm	28 nm

Table 1. Xilinx Families comparison

The Spartan-3A FPGA board includes common peripherals as several types of memories, two-line 16-character LCD screen, PS/2 port, VGA port, 10/100 Ethernet PHY, two DB-9 RS-232 ports, four-output SPI-based Digital-to-Analog Converter (DAC), two-input, SPI-based ADC with programmable gain amplifier, three 6-pin Digilent Peripheral Modules (PMOD), eight discrete LEDs, four push-button switches, four slide switches and a 100-pin Hirose FX2 expansion connector.

Despite all the peripherals, what is required for the project application is:

- ADC to read two analogue channels.
- Clock speed enough to achieve a maximum of 1 Msps through the ADC.
- Enough input/outputs pins to drive two digital potentiometer SPI buses, and acquire the feedback signals. Which makes a total of 10 digital I/O.
- A reset button and several leds that will help in the ADC controller development

Figure 6. Front view of the Xilinx Spartan-3A Starter Kit Board

The Spartan-3A FPGA board includes a whole external analogue capture circuit. Which includes a programmable pre-amplifier LTC6912-1 that scales the incoming analogue signals, and a dual 14-bits ADC LTC1407A-1. Both components are SPI configured or controlled by the FPGA. The amplifier SPI-communication interface is relatively slow, supporting only about 10 MHz clock frequency, while the ADC is capable of 50 MHz clock signals achieving a 1,5 Msps.

The family can operate in a wide clock frequency range (5 MHz to over 320 MHz), but the Spartan-3A Starter Kit has a 50 MHz oscillator crystal as a global clock, which is fast enough to drive the analogue capture circuit. Although the board also has an Auxiliary clock input with an 8-DIP socket with a 133 MHz oscillator, and a SMA connector for external clock connectivity they were not used.

For I/O connectivity PMODs are the fastest way, but instead a FX2 Breadboard, [16], was used as it was within the laboratory facilities. This extension board allows to easily connect to 40 I/O pin ports of the FPGA. The same connector includes ground and power pins to easily power components that may be inserted in the breadboard. In the next image, *Figure 7*, there is a picture of the FX2 Breadboard from Digilent, and a diagram of how all connectors are distributed. The board also have four female and four male PMODs ports. So the board itself can be also used with FPGA boards that does not have a FX2 connector.

On this board is where the DTAF detailed in chapter 3.1 has been wired and tested during all the project steps, prior to the building of the PCB design.

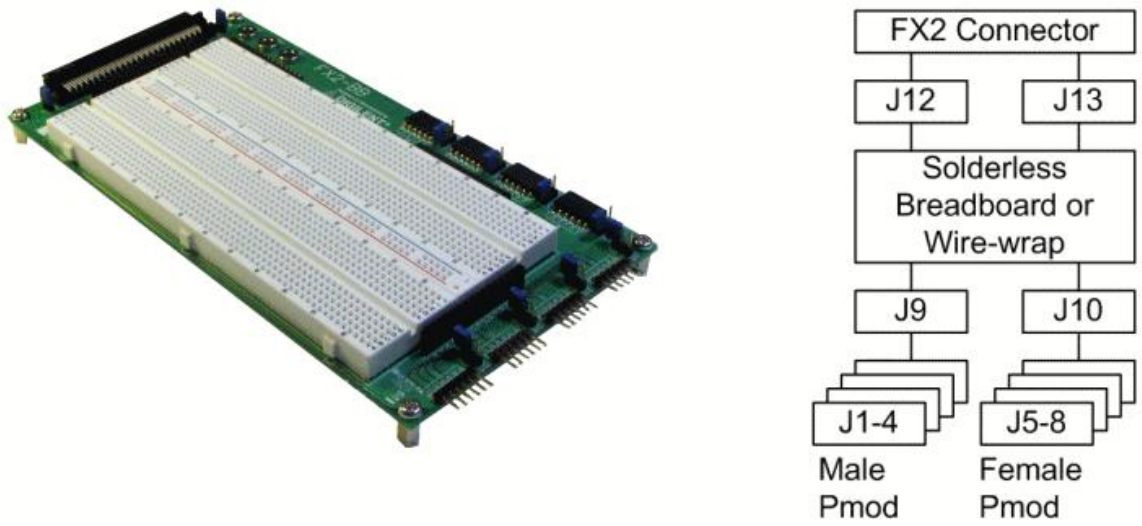


Figure 7. FX2 Breadboard and connector's diagram

3.3.2. Xilinx toolchain used and first implementations

The software used is part of the Xilinx development tools. On one hand the ISE 14.2 Project Navigator it is used to write all VHDL code, compile it and simulate. While on the other hand the software iMPACT it is used to program the FPGA. At the last part of the project, some informatics problems occurred with iMPACT software all functions to program FPGA were done in Adept 2 software instead, which is a software from the vendor of the Nexys4 board, Digilent Inc.

First designs implemented in the Spartan-3A Starter kit were basic code to turn on and turn off the on board leds, and read the slide switches or push buttons. Which means to generate a file "XXX.vhd" with the corresponding vhdl and also an "XXX.ucf" file which contains all the I/O pins that will be used within the FGPA. The "XXX.ucf" file is usually provided by the vendor as a starting point where all the possible connections to the peripherals on board should be stated. Also some reference designs with some peripheral control or with embedded soft cores are usually presented. In 0 support documentation is found.

In order to check that the VHDL code works as expected a Simulation tool is used. In this project the simulator used have been the embedded within the ISE environment, ISim. There are two ways to simulate FPGA systems. The most quick and basic way is to create a script file with all instructions to generate waveforms that will stimulate the logic that we have designed. It is a really fast option, but can be tedious when large scale I/O or models are implemented. The other way is to create a testbench. To use a testbench is necessary to create a new vhdl file that instantiates the Device Under Test (DUT), which is the logic we want to stimulate. So the testbench it is a VHDL entity upper in hierarchy that has connections directly to the I/O pins of the DUT and also has some VHDL code that generates the waveforms we want to use to stimulate the DUT.

In this project the script option is used, because the system implementation has been growing in a large number of steps. So it was required to change the simulation script quite fast. It is not useful to make a whole testbench system if the architecture of the system changes continuously.

About FPGA configuration, FPGA boards are usually provided with several programming modes. It may be possible to load an image of the program file into a Flash or PROM memory that during the power-up the configuration is loaded to the FPGA. The Flash memory option is not supported on the Spartan-3A, but in the Spartan-3AN. Another way is to download the configuration directly to the FPGA via JTAG. To avoid the JTAG cable in low-cost boards, boards implement an embedded USB-based programming logic, so a common USB cable is enough to program an FPGA. In this project there is no need of storing an image within the memories in the FPGA board, so the USB cable is used to configure all code modifications directly to the FPGA.

3.3.3. Nexys4 Artix-7 Analysis

The second FPGA platform used within the project is the Nexys4 Artix-7 FPGA board, from the vendor Digilent Inc. As the surname itself says is based on the Artix-7 family. The previous Table 1, shows that the Artix-7 family has many more logic cells than the Spartan-3A family. It is important to note that this family has a 28 nm process technology, which is quite smaller in comparison with the 90 nm of the Spartan-3A. While current Spartan is Spartan-6 which have a 45 nm instead. The Artix-7 family is optimized for high performance logic, and offers more capacity, higher performance, and more resources than Spartan family. The Artix-7 implemented in the Nexys4 part number is XC7A100T-1CSG324C.

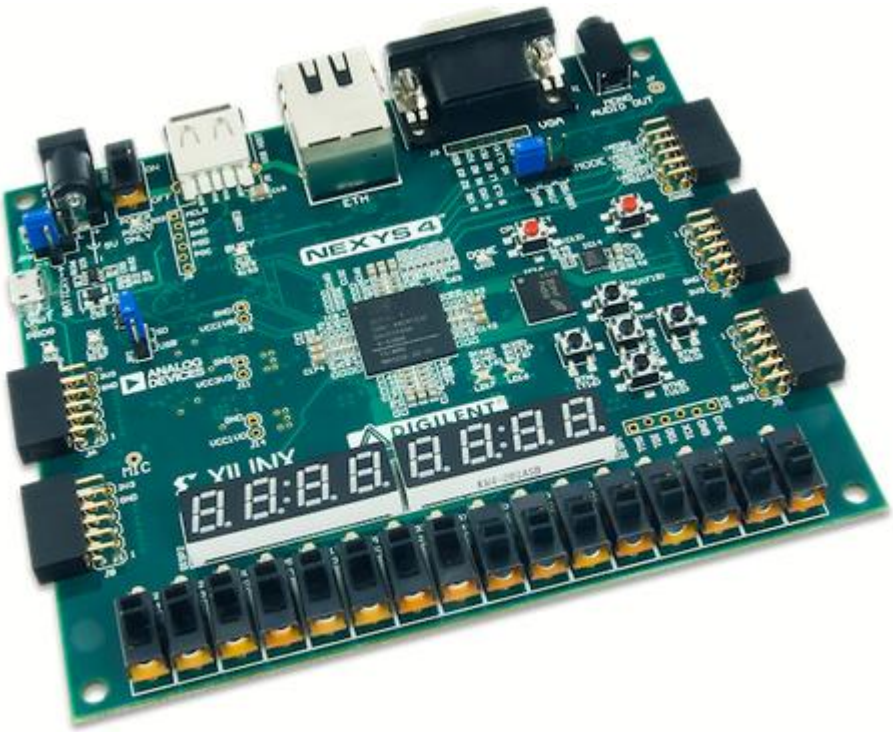


Figure 8. Nexys4 FPGA board

The Nexys4 FPGA board includes common peripherals as several types of memories, 16 slide switches, 16 LEDs, USB-UART bridge, a 3-axis accelerometer, two tri-color LEDs, temperature sensor, audio output, 8-digit seven segment displays, MicroSD card connector, MEMs microphone, 10/100 Ethernet PHY, USB Host, on-chip ADC (XADC), four I/O PMODs ports, and a PMOD ports with connections to 4 XADC channels.

From all this peripherals the only used within this project are the XADC PMOD, two PMOD ports for I/O signals, all the user LEDs to quickly verify ADC values read and a button to be used as reset.

In this board the XADC embedded core has a dual 12-bit ADC with a maximum sample rate of 1 Msps. It has the advantage that is embedded within the FPGA hence is easier to configure, but it can achieve 0,5 Msps less than the external ADC on the Spartan-3A board. In any case, 1 Msps is still enough for the required application. Also this board has no programmable gain amplifier, which means that the conversion range of the ADC is not selectable.

The Nexys4 has a single clock oscillator of 100 MHz which doubles the frequency of the clock used in the Spartan-3A.

The Nexys4 Board has not any FX2 connector, so it is not possible to attach the breadboard used before by this connection. Instead some PMODs connectors are used. The board has four I/O PMODs, JA, JB, JC and JD, and a XADC PMOD called JXADC. Then JXADC PMOD is used to connect the two analogue signals from one PMOD of the breadboard. And two I/O PMODs are used to connect the 10 digitals signals required.

3.3.4. Switch from Spartan-3A to Artix-7

Switching from Spartan-3A into Nexys4 is not evident, but as the FPGA are from the same vendor is not rather complicated. It is important to notice that not all components in the Artix-7 family have compatibility with ISE WebPACK tool, which is limited to the Artix-7 XC7A100T and the XC/A200T. As the Artix-7 within the Nexys4 board have this compatibility, there is no need of changing the design tool. If the FPGA chip wouldn't have this compatibility it would have been necessary to buy an ISE Design Suite license increasing the project budget or update into the new Vivado Design Suite. The problem of doing a project migration to the Vivado Design Suite is that extra time will be needed to adapt to the new environment. For example even the "XXX.ucf" file system does not exist anymore, which is changed to an "XXX.xdc" fle.

Furthermore, the difference between both platforms which causes major changes in the project are the peripherals changes. All the "XXX.ucf" file needs to be completely different. In this case as the board is from the vendor Digilent Inc. all the documentation and reference "XXX.ucf" or reference design are in [14], under the title Support Documents.

Also, thanks to the reference design it is possible to quickly understand how to drive some common peripherals such as the LEDs or the slide switches. So this reference design is taken as a starting point to implement all reusable code from the last FPGA platform.

The fact that the new board has a double frequency clock is positive, but a redesign of the main counter which works as clock divider it is needed.

But the main drawback is that in the Artix-7 the ADC is embedded inside the FPGA chip, instead of being an external chip. So, all the ADC SPI based controller designed for Spartan-3A Starter Kit is not useful anymore. Also the ADC maximum sample rate has drop from 1,5 Msps into 1 Msps.

The advantage of having an embedded ADC is that the eventuality originated in the Spartan-3A board is solved.

The eventuality that appeared while working with the Spartan-3A Starter kit is that, sometimes after a modification on the VHDL code which could be compiled and programmed in the FPGA, with no new warnings, the FPGA did not run. So, every time this happened output signals were not working as properly, as well all signals were stuck at the reset state.

Despite of the eventuality, with the Spartan-3A board was possible to implement controllers for DAC, ADC, and Digital Potentiometers. Even the implementation of the central frequency loop controller was all completed. But the integration of the central frequency control loop and quality factor loop working both together was impossible to do. For that reason the quality factor loop and the integration were not ended in this platform, but in the Nexys4.

One cause could be that the ADC and pre-gain amplifier controller were using the same SPI Bus signals as some of the memory chips available within the board, and then it could end to a wrong configuration of the FPGA, that the program software never detected. But that is unusual because the FPGA was always programmed in direct mode via USB interface.

That is why it was decided to switch to the new Nexys4 Artix-7 board.

3.4. Digital Potentiometers

A really important element of the DTAF design is the digital potentiometer. So, a whole VHDL controller has been designed to operate with it. And although finally 100 kΩ potentiometer are chosen and there is no need of concatenating two potentiometers, the designed controller allows to automatically change from one potentiometer to another changing a few code lines. Concatenating two potentiometers was quite useful in early steps were only 10 kΩ potentiometers were available, as the 100 kΩ potentiometers were bought during the PCB fabrication step.

3.4.1. MCP4251 Dual Potentiometers

The digital potentiometer chip used within this project is the microchip MCP4251, [20]. It is an 8-bit dual SPI digital potentiometer with volatile memory. The family chip can have a total wiper resistance of 5 kΩ, 10 kΩ, 50 kΩ or 100 kΩ between terminals PxA and PxB. There are some models in the family that have non-volatile memory, which allows chip to keep the last configured value even when power is off, which may be useful in some critical applications, but not in this research project.

Although the used chip has a volatile memory, during the power on reset (POR) stage it configures itself to a mid-scale value. So, as the used chip has a total resistance between terminals PxA and PxB of 100 kΩ, in the POR it configures itself to an approx., 50 kΩ value.

Basically, the digital potentiometer has a register per each wiper within the chip. In this case there are two wipers, then there are two registers to be controlled. Each register has an 8-bit word, which means 256 possible values. When a 00h is written to the control register, potentiometer is at zero scale, so wiper points to PxB. In opposite when register is set to FFh, potentiometer is at full-scale, so wiper points to PxA.

Common SPI devices are controlled by write/read commands addressed to each of the registers defined within the chip. In this 16-bit commands a data word is written or read depending on the command. Sometimes, there are more basic commands involved, which don't need to write/read any data word, so they transfer the minimum amount of bits. In this case the potentiometers also accept increment or decrement commands which are 8-bit commands.

3.4.1.1. Pinout

The next pinout, *Figure 9*, is the corresponding to the PDIP encapsulate, which is the same as the used chip. It has 14 pins, 6 pin for the potentiometers connections, 2 pin for power connection and 6 control pins. As the chip is driven directly from FPGA platform, the VSS and VDD are both connected to the FPGA platform, to ground and 3,3 V respectively. The write protection ($\overline{\mathrm{WP}}$) and shutdown ($\overline{\mathrm{SHDN}}$) pins are not used in this project, so the pins are connected to VDD directly at the PCB.

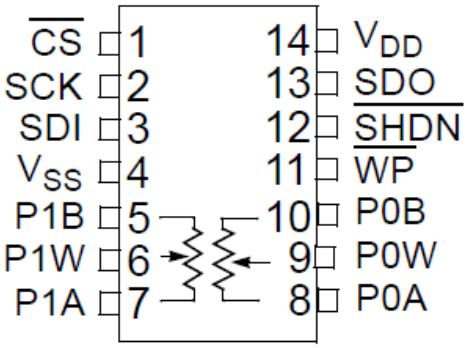


Figure 9. MCP4251 PDIP pinout, from [20]

The resistance pins are connected in rheostat mode, then pins P1A and P0A are not used. So, when the register count is ascending from 00h to FFh corresponds to an ascending from zero-scale to full-scale. The remaining pins $\overline{CS}$, SCK, SDI and SDO are used to communicate through SPI.

3.4.1.2. Single potentiometer

In the next image, *Figure 10,* there is an example connection of resistor R_{in} using the digital potentiometer as explained. In this case only one potentiometer is used, so the total resistance is 100 kΩ and a POR resistance of 50 kΩ. The total number of steps is 257.

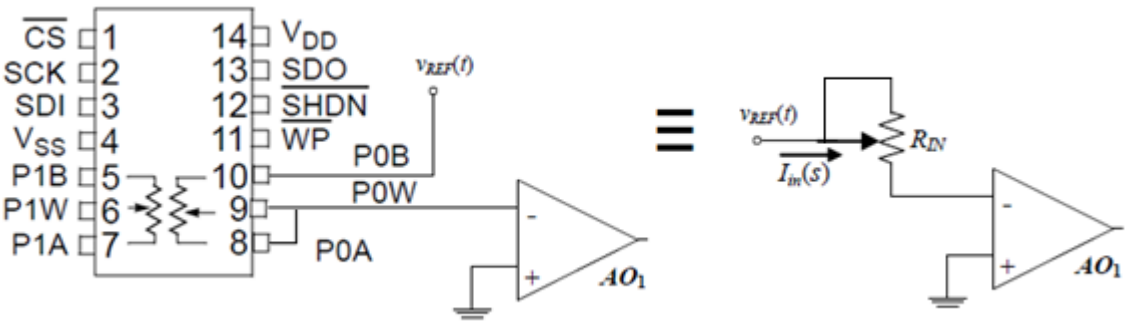


Figure 10. Single potentiometer connection. With digital potentiometer (left), with standard potentiometer (right)

3.4.1.3. Double potentiometer

In the early steps, in the laboratory five 10 kΩ digital potentiometers chips were available, four chip were used to substitute each resistor in the design, and fifth one was used for testing the resistor value. Instead of having single potentiometer configurations with variable resistors of 10 kΩ, double potentiometers configurations (20 kΩ) were done. With 100 kΩ potentiometers, the total digitally controlled resistance with only one MCP4251 device would be 200 kΩ using this technique. The next image, *Figure 11*, is the connection realized in the digital potentiometers. With this configuration a total of 512 steps are possible achieving 20 kΩ, and the POR resistance value is 10 kΩ.

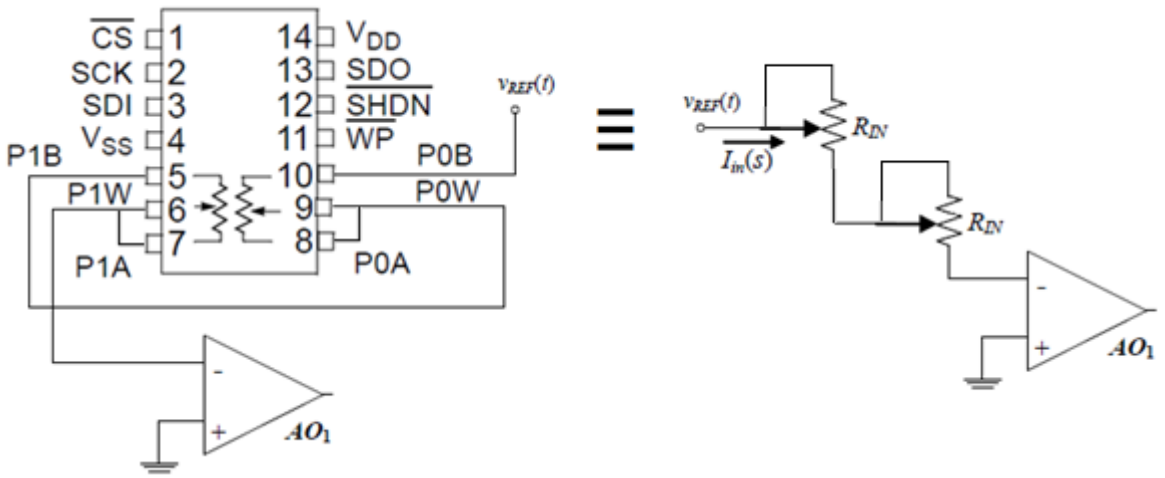


Figure 11. Double potentiometer connection. With digital potentiometer (left), with standard potentiometer (right)

To enable this configuration the VHDL controller needs to be re-configured. While in single potentiometer configuration the increment or decrement command are enough, in double potentiometer configuration a read command has to be added. Then the VHDL controller is capable of knowing the exact value of the wiper pointer. As the whole chip has the same SPI bus it is possible to monitor each of the wiper pointer easily. In the figure below, *Figure 12*, there is a block diagram of the MCP4251 chip. The diagram helps to understand the relation of the pins to each function. It also appears the wiper registers used in write/read and increment/decrement commands.

The increment or decrement commands change from the single potentiometer configuration in that the address of the register is changed depending the case. From zero-scale to mid-scale, the increment or decrement operations are addressed to wiper 0 register allowing a sweep from 0 Ω to 10 kΩ. But when the read operation returns a full-scale of wiper 0 register, then the increment or decrement operations are addressed to wiper 1 register. Then it allows to increment even more the total resistance from 10 kΩ up to 20 kΩ. To return to wiper 0, a zero-scale read in wiper 1 is required.

To avoid errors, another condition must be added. When incrementing wiper 0 to full-scale so the address is changed to wiper 1, the controller could realize a read operation on wiper 1. If the value returned would be zero-scale the controller would change again the address to wiper 0, not allowing the total potentiometer to increment further 10 kΩ. That is why an ascending condition is added.

In the same way, when decrementing wiper 1 to zero-scale so the address is changed to wiper 0, the controller could perform a read operation on wiper 0. If the value returned would be full-scale the controller would change again the address to wiper 1, not alllowing the total potentiometer to decrement under 10 kΩ. That is why a descendent condition is added.

The ascending and descending conditions are just a check on which was the last address used. Then, when ascending and a full-scale on wiper 0 is detected, the address is only changed to wiper 1 if the last address used was also on wiper 0. This prevent the controller to change from wiper 0 again to wiper 1 when descending.

Also, during the descending operation if a zero-scale is detected on wiper 1, the address will be changed to wiper 0 only if the last address was also on wiper 1. This will prevent the controller to change address from wiper 1 again to wiper 0 when ascending is performed.

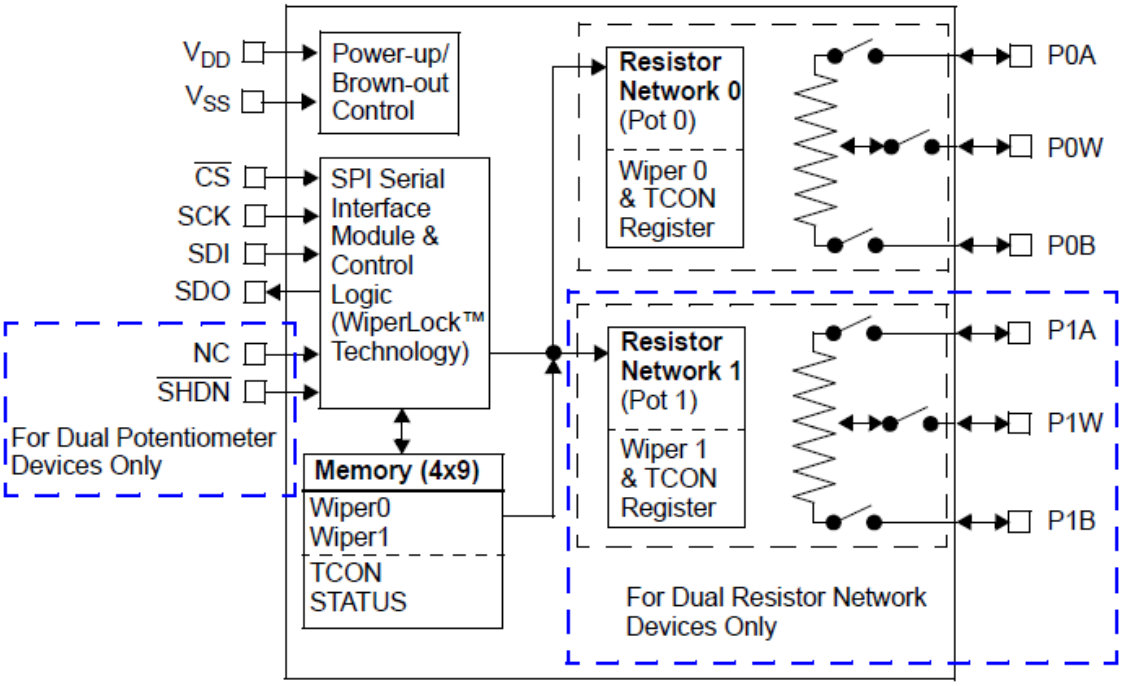


Figure 12. MCP4251 block diagram, from [20]

3.4.2. SPI communication protocol

In order to perform a tunable control action within the DTAF, the digital potentiometers are used. The digital potentiometers are used to increment or decrement their resistor value. Then the VHDL controller needs to communicate with the digital potentiometer, in order to send commands or read data from them. The protocol used to communicate is the Serial Peripheral Interface (SPI), where the FPGA is the master and the potentiometers are slaves.

3.4.2.1. SPI Signals

The signals used to communicate, by SPI, between digital potentiometer and FPGA are:

- $\overline{CS}$: This signal is the serial interface's chip select. When it is at low value the digital potentiometer accept serial commands. When at high value it does not accept new commands.
- SCK: Is the SPI clock signal. SPI's rising edge will determine when a bit is valid to send.
- SDI: This signal is the serial interface Serial Data In.
- SDO: This signal is the serial interface Serial Data Out. Only in Double potentiometer.

Then the master SDO is connected to the slave SDI and the slave SDO is connected to the master SDI. The signals $\overline{CS}$ and SCK are just connected to their respective in the master and the slave. See *Figure 13* below.

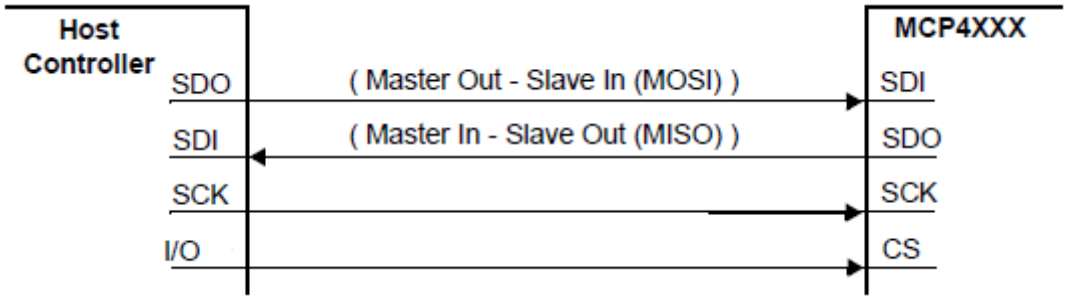


Figure 13. SPI connection between FPGA and MCP4251; from [20]

As MCP4251 has a limit of 10 MHz SPI frequency of operation the FPGA controller must not over pass it while communicating to this device.

3.4.2.2. SPI communication format

There are four standard SPI modes, but the MCP4251 only supports two of them which are mode 0,0 and mode 1,1 while the mode used in this project is the mode 0,0. In this mode data is clocked in on the slave SDI pin on the rising edge of SCK and clocked out on the slave SDO pin on the falling edge of SCK.

The possible MCP4251 commands are: write, read, increment and decrement. The write and read commands are 16-bit commands, while increment and decrement are only 8-bit commands. In this project the write operation is not used. The command bit strings protocol is formed by a 4 memory address bits, 2 command bits and 2 or 10 data bits depending on it is a 16 or 8 bit string. The bit string is MSB first as it is shown in *Figure 14*.

The address values used are the corresponding to the wipers. To communicate with wiper 0 and wiper 1 registers the address values will be 0h and 1h respectively. Then the command bits are set to ‘01’ for increment, ‘10’ for decrement and to ‘11’ for read commands.

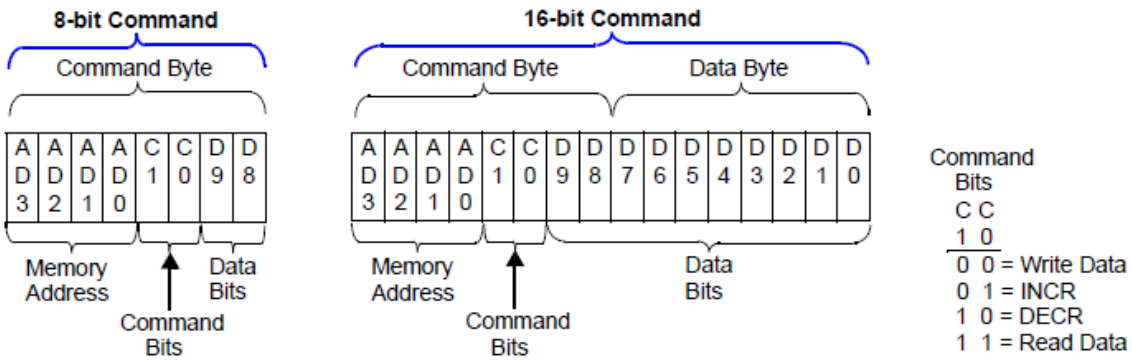


Figure 14. SPI commands format; from [20]

3.4.2.3. SPI used commands

Next picture, *Figure 15*, is a 16-bit read command waveform in mode 0,0. The steady state for $\overline{CS}$ is high value. While the steady state for SCK is a low value. The FPGA host must send through host SDO a string of four bits with the address selected followed by a string of two high bits to select the read data command. If the string its well send by the host, the MCP4251 slave will then send a string through slave SDO pin with one non-important bit and an 8-bit data. Then first bit received by the host it is discarded and the next 8-bit data must be stored.

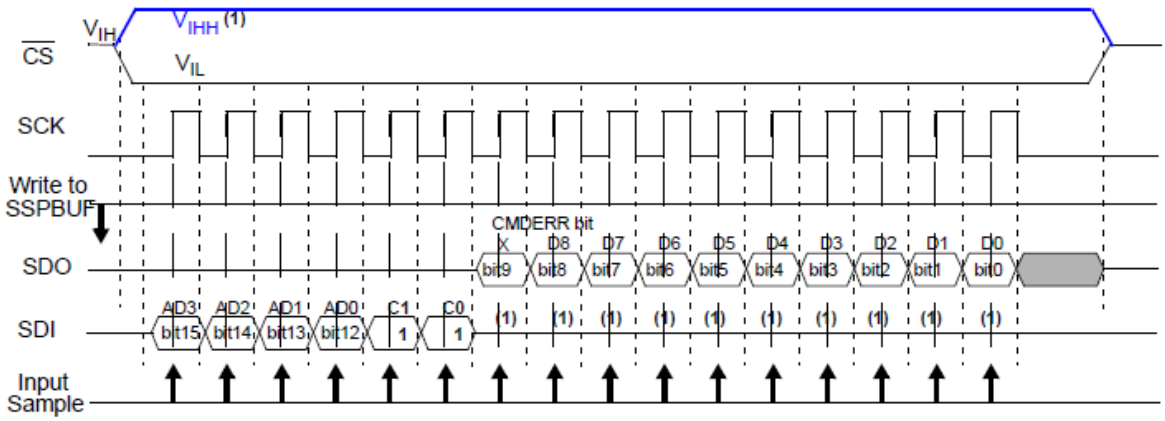


Figure 15. 16-bit SPI read command, mode 0,0; from [20]

In this project this command it is used by the FPGA controller to know the position of the wipers 0 and 1 registers.

The increment and decrement commands are 8-bit only, see *Figure 16*. Both commands has the same bit string excepting the command bits. In an increment or decrement operation the two last bits send will be zeros, as they are not important values. This commands will modify the value of the wiper addressed by adding or subtracting a position. The FPGA controller uses this commands the increment or decrement the resistance values on the digital potentiometer.

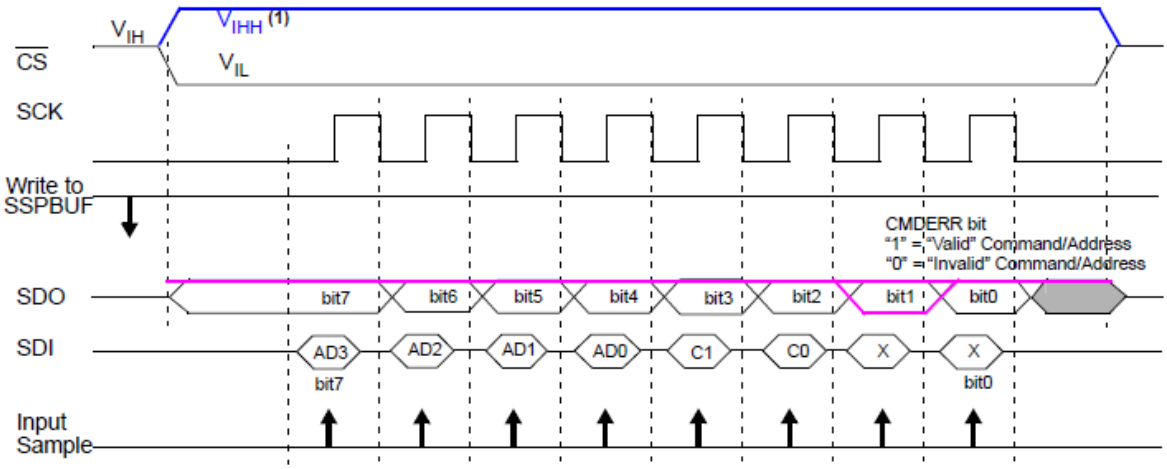


Figure 16. 8-bit SPI increment or decrement command, mode 0,0; from [20]

3.5. Central frequency control loop by phase matching

To implement the central frequency loop two blocks must be designed, the Phase Mismatch Detector and the Central Frequency Control.

3.5.1. Phase mismatch detector circuit

Two possible ways to detect the phase of an analogue signal with an FPGA board are considered.

The first technique is based on an ADC. First, with the on chip XADC it could be possible to sample the two analogue filter signals to compare. Then analysing the read values of each signal it could be stated when each signal cross the zero point. The zero point is the instant where the analogue signal have zero voltage, and also where the phase reference starts. Although there are two zero points within a period, it is considered only the first one, where signal goes from negative voltages to positive voltages. So, knowing which group of read values reaches first the zero point it could be finally stated that a signal is advanced to the other one or vice versa.

But this technique has two disadvantages, both related with the ADC. The first is that to truly detect a zero voltage value in an analogue signal it would require a really fast ADC. The second is that is so complicated to catch a sample at the right instant of crossing zero that may occur that the sample taken it is never zero. The only solution would be to implement a voltage range wide enough to consider it as zero point.

Considering the worst case on this project, as it is an audio band filter, the worst case is a 20 kHz analogue signal, then the signal will cross the zero point every 50 µs. Also considering the on board 1 Msps XADC, a sample would be read every 1 µs. So the possibility to identify the zero point is 50 over 1. To achieve a success with this technique a fast ADC would be needed. And it would need to be fast enough to take several samples during the zero point, to verify that the value read is truly the zero point.

So, knowing that the period is 50 µs, it can be deduced that a 1º lasts only 138 ns. Then if it is supposed that the zero point occurs during one phase degree (theoretically is zero) it would be necessary an ADC to read several samples within 138 ns.

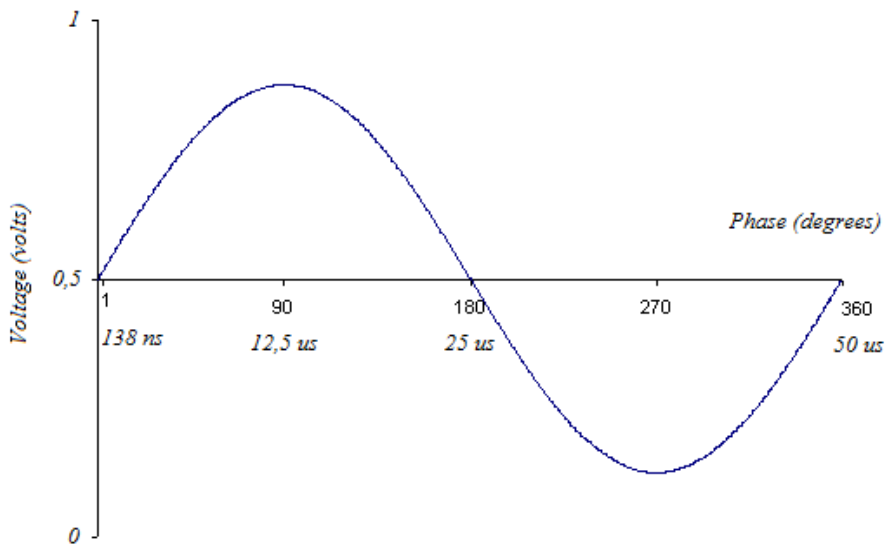


Figure 17. 20 kHz sinusoidal signal phase and time points within a period

Then with an ADC of 25 Msps only 3,45 samples could be done within 138 ns, 6,9 samples with a 50 Msps and 13,8 samples with a 100 Msps. Although there are commercial ADC with that characteristics it would increase the PCB complexity and cost.

The second technique considered is to use the FPGA logic directly to implement a phase detector block. In that case a circuit that generates two digital signals with the phase information must be first implemented. The circuit that generates a two-state signal from an analogic signal is a comparator. A comparator comparing an analogue input signal and a voltage reference, set to the maximum amplitude divided by two, would generate a digital signal with a rising edge when crossing the first zero point and a falling edge when crossing the second one, the waveform generated is shown in next picture, *Figure 18*.

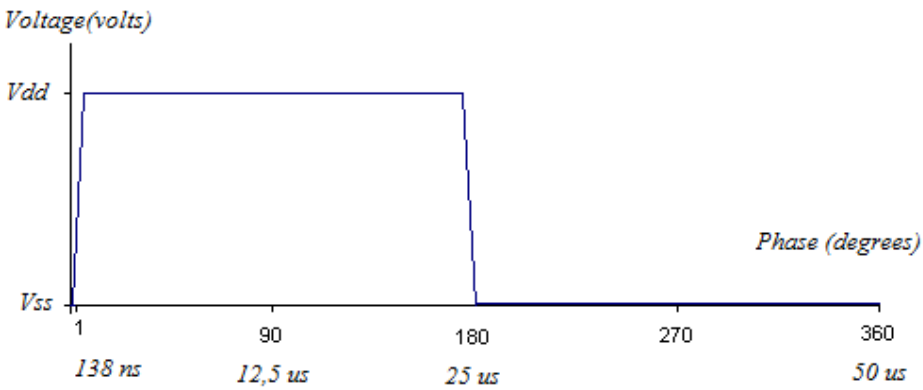


Figure 18. Comparator output

This technique only have two limitations. The first is the comparator settling time, which depends directly on the slew-rate. The second is the FPGA global clock frequency. So, the time needed to detect an analogue sinusoidal signal crossing the first zero point, is the comparator settling time plus the delay of the FPGA detection logic that must be implemented. But notice that such limitations only add a detection latency, because we could say that the sampling frequency of the comparator is infinite compared to the ADC, as the comparator is analogic. So, if both OPAMPs that detect input and output signals have the same latency, both zero points will be truly detected at the right time.

Because the second option is quite easily to implement, it has been discussed that can detect the zero point with more precision than the first technique and because the quad OPAMP chip used in the CTF have two spare OPAMP that can be used for this operation, it is decided to implement the second phase mismatch detection technique.

Then, the phase mismatch detection circuit is based on the chosen OPAMP (MCP6024) working as a comparator. This circuit allows to convert an analogue signal into a signal that switches between two discrete voltage levels. When the $V_{in}(t)$ input is higher than the V_{offset} signal the V_{out} value is V_{dd} voltage. But when the opposite occurs the V_{out} signal is V_{ss} voltage. Notice that to avoid the need of a voltage level-shifter stage, voltages Vdd and Vss to source the OPAMP must match the corresponding Vdd and Vss that sources the FPGA chip, which are 3,3 V and 0 V respectively.

Finally, to detect the phase mismatch between analogue filter input and analogue filter output two comparators are needed. One compares analogue filter input, while the other compares analogue filter output, both against the V_{offset} voltage. The two generated digital signals with phase information are called, BPF_Vin and BPF_Vout, which will be connected to the FPGA platform as digital inputs.

3.5.2. Phase mismatch detector logic

The phase mismatch detector logic is embedded in the FPGA. The VHDL code of this block is contained in a file called, “phase_detector_comp.vhd”, which is a sub-entity under the top-level entity called “DTAF_Controller.vhd”. Next figure, *Figure 19*, represents the block diagram of phase_detector_comp entity. Notice that “_c” ending is used to specify block signals, instead of global signals. This is useful to differentiate block internal signals from the input/output block signals, and also for an easy portmap connection when connecting all blocks to the top-level entity.

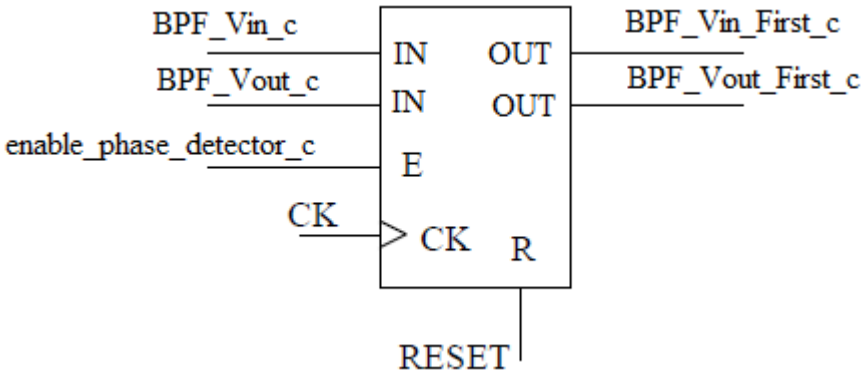


Figure 19. Phase detector block diagram

First the two external digital signals coming from the phase mismatch detector circuit, BPF_Vin_c and BPF_Vout_c, must be synchronized, into BPF_Vin_Sync and BPF_Vout_Sync. That is why both signals are twice registered in synchronous registers with enable and reset. The enable signal enables or disables the registering of the external inputs, and is controlled from the top level entity. The reset is synchronous. Next picture shows the correspondent block diagram*, Figure 20*.

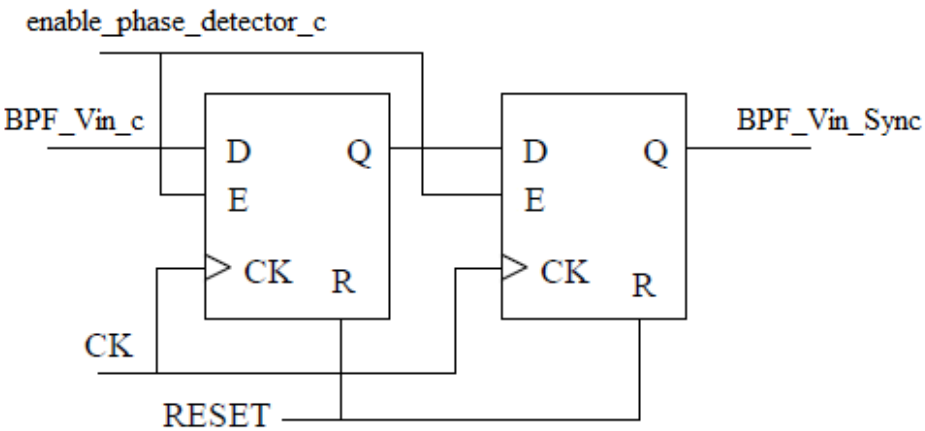


Figure 20. Synchronization of external BPF_Vin_c signal block diagram

Once the external signals, BPF_Vin_c and BPF_Vout_c, are synchronized is detected if there is a mismatch between them. To do that it is enough to implement a XOR between synchronized BPF_Vin_Sync and BPF_Vout_Sync. Then the BPF_Not_in_Phase signal generated will be at logic level one when inputs signals are not equal, and at logic level zero when both inputs are equal, *Figure 21*.

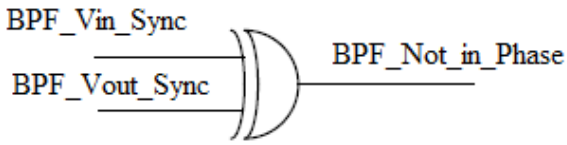


Figure 21. XOR gate to detect a phase mismatch

Secondly, although it is already known that here is a mismatch between external signals, it is necessary to know which signal is advanced or delayed to the other one. Because without that information it will not be possible to decide whether to increment or decrement the digital potentiometers value in order to correct the phase mismatch.

Now the rising edge of both signals must be obtained. To do that, a NOT and a AND gates are added to the synchronization logic. The negated output of the second register must pass through a AND gate together with the output of the first register. This logic gates are implemented with a combinational logic. Next picture, *Figure 22*, shows the logic implemented and the waveform diagram of its operation with BPF_Vin_c signal.

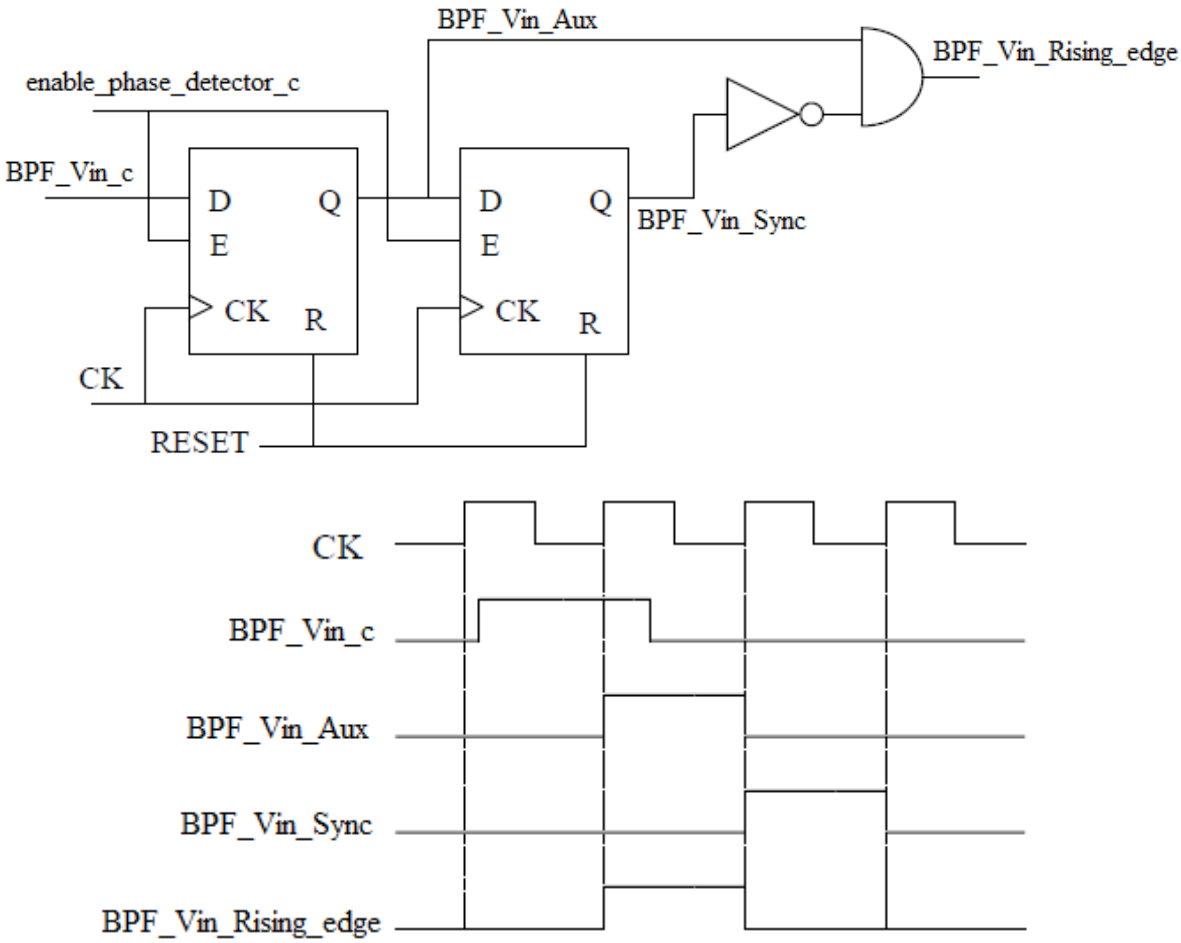


Figure 22. Block diagram and waveform of the rising edge detector

Finally, with BPF_Vin_Rising_edge and BPF_Vout_Rising_edge signals, the BPF_Not_in_Phase signal, and the synchronized BPF_Vin_sync and BPF_Vout_sync, two signals are obtained with the advanced or delayed information. The logic function applied is a three-input AND gate. So, if both external signals are not in phase, and a rising edge is detected in the BPF_Vin_c input while BPF_Vout_c input is at logic level zero, it stated

that BPF_Vin_c is advanced to BPF_Vout_c. The same approach is done to detect BPF_Vout_c delayed to BPF_Vin_c.

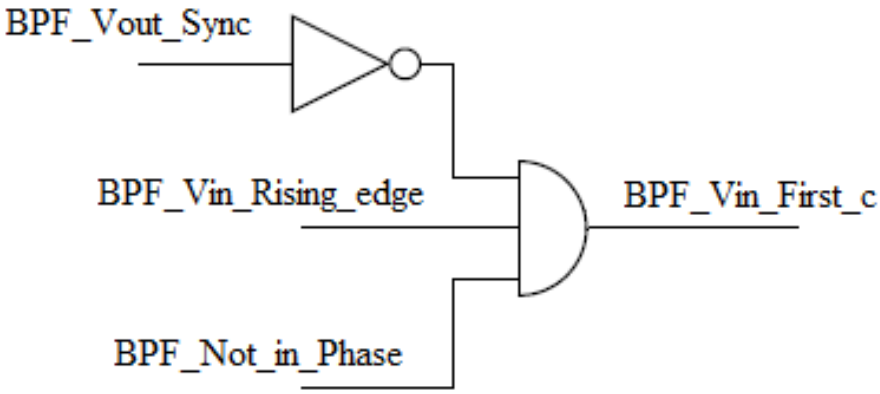


Figure 23. Phase mismatch detector gates

But when last step is simulated, the BPF_Vin_First_c and BPF_Vout_First_c signals do not behave as expected. That happened because the BPF_Not_in_Phase signal is generated from the synchronized external signals (it has been registered twice) while the rising edge detection it is only registered once. Then there is a clock period difference that cause the last three input AND gate work in a not expected way. To solve this a third register is added, and the rising edge is detected from the second and the third ones.

In the next picture, *Figure 24*, there is the final block diagram of the phase mismatch detection logic, and the waveforms simulated. Notice in the waveform, that first BPF_Vin_c goes high for several clock periods. Then the signal is registered three times. The rising edge is detected as the signal pass through the second register. Also, the signal BPF_Not_in_Phase goes high as the BPF_Vin_c is not equal to BPF_Vout_c. Finally the BPF_Vin_First goes high during a clock period to indicate that BPF_Vin_c signals is advanced to BPF_Vout_c.

See also that at 300 µs the BPF_Vout_c goes high also for 100 ns. But in this case the implementation is done only with two registers, as in Figure 22. That is why the signal BPF_Vout_first does not go high. Be aware that is only for explanation purposes, the final implementation incorporates the correct logic for both BPF_Vin_c and BPF_Vout_c.

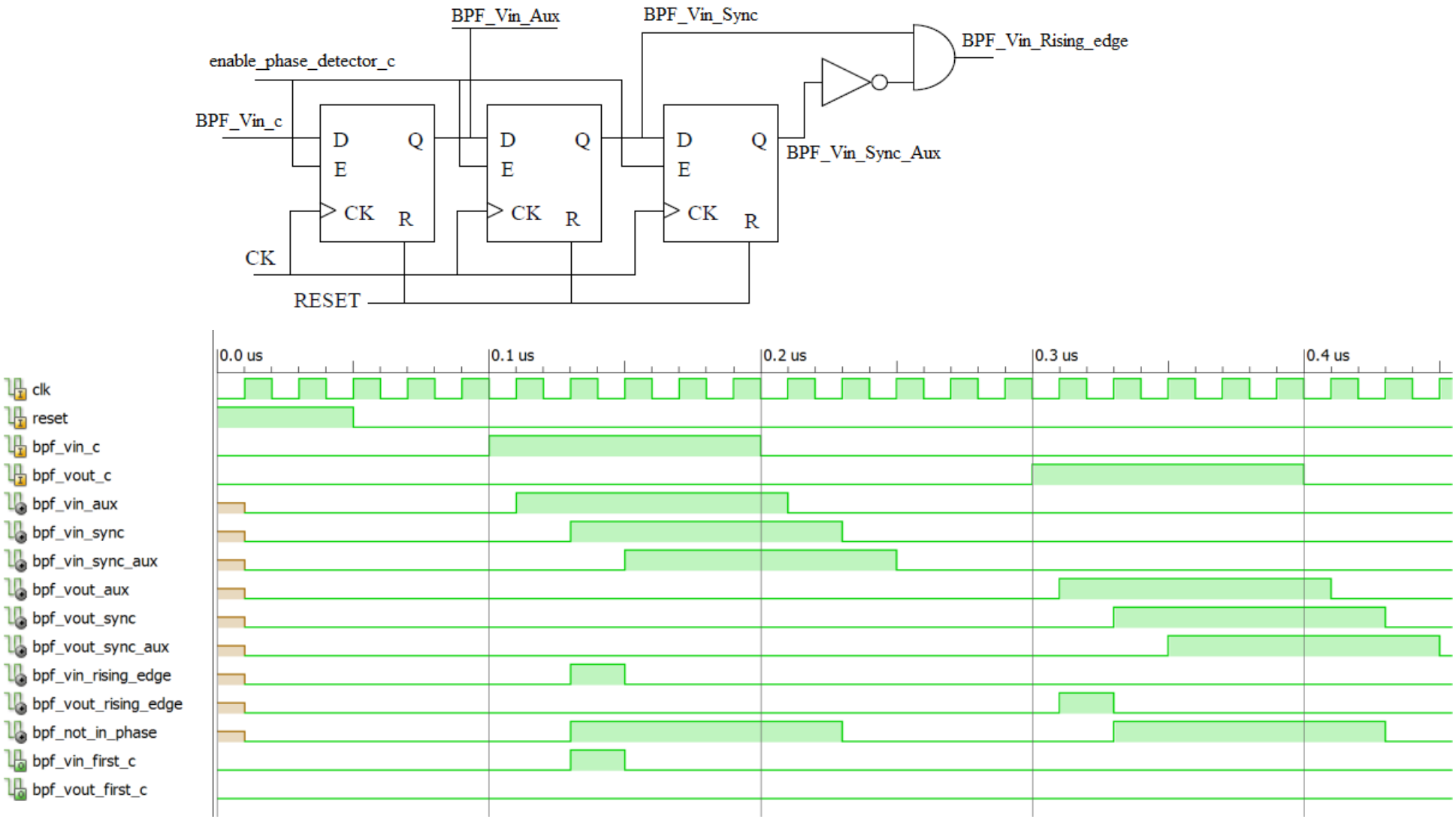


Figure 24. Vin advanced detection and waveform diagrams

3.5.3. Central frequency loop digital potentiometer controller

The central frequency loop digital potentiometer controller is embedded in the FPGA. The VHDL code of this block is contained in a file called, “phase_ctrl_comp.vhd”, which is a sub-entity under the top-level entity called “DTAF_Controller.vhd”. This block controls the digital potentiometers connected in the central frequency loop, corresponding to R_{in}, R_1 and R_3.

The block logic objective is to reduce the phase mismatching between the filter input and output analogue signals, by incrementing or decrementing the Digital Potentiometers resistance depending on the feedback provided by the Phase Detector Block.

The block inputs/outputs are shown in the next image, *Figure 25*. BPF_Vin_First_c and BPF_Vout_First_c signals come from the prior block Phase Detector Block. Freq_limit and enable_phase_ctrl_c are generated in the top-level entity and regulate the logic action of this block. While there are three outputs to control the Digital Potentiometers SPI Bus, and a POT_SDO_c input signal with Digital Potentiometer feedback information.

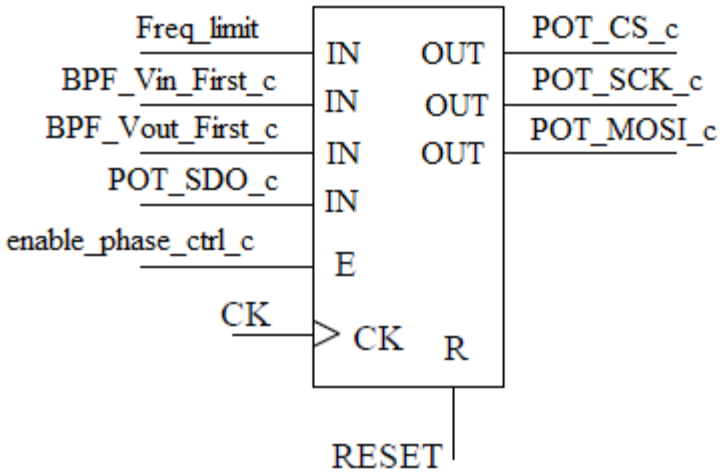


Figure 25. Phase control block diagram

The Phase Control block is mainly composed by a state machine that controls the digital potentiometers SPI Bus, operating as Master. The state machine is included inside the process “phase_ctrl_comp_state_machine”. Such state machine have evolved adapting to different needs through the project. First of all, the state machine was implemented taking into account 10 kΩ digital potentiometers, in single potentiometer operation (3.4.1.2). So, the state machine had two states, Idle and Phase Matching. But in order to improve the wiper range, the state machine was improved to concatenate two digital potentiometers wipers together, as in 3.4.1.3. Then a third state called Pot Address control was added. As finally 100 kΩ potentiometers were used, the third state was erased, returning to the state machine original design. In case of willing to operate with this potentiometers with a variable resistor range of 200 kΩ, the double potentiometer operation and hence the three states state machine would be needed again. Changing from double potentiometer to single potentiometer operation requires another change on the Phase matching block, the send commands are slightly different. Next image, *Figure 26*, shows both state machine diagrams:

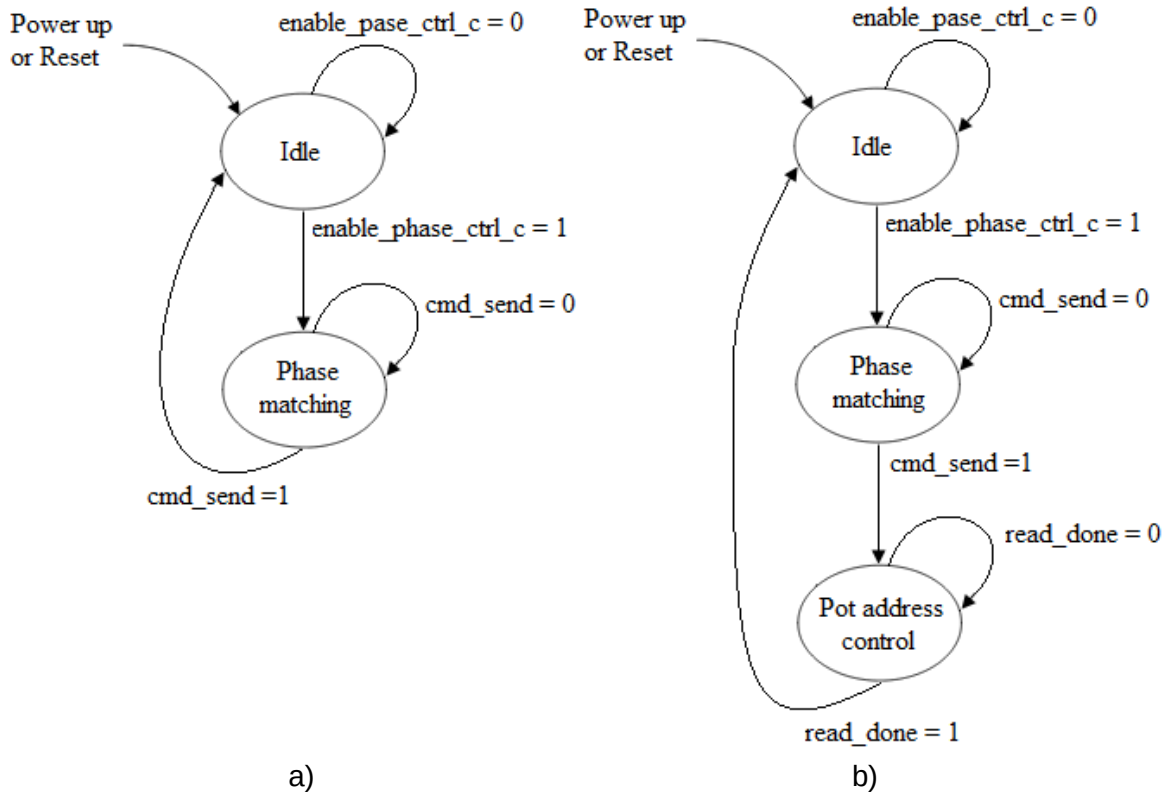


Figure 26. Phase control state machine diagram. a) For Single potentiometer. b) For Double potentiometer.

The Idle state it is used to hold the steady values of the SPI bus signals: POT_CS_c = '1', POT_SCK_c = '0', POT_MOSI_c = '0'. It also resets the values of POT_cmd_send and POT_read_done, to allow the state machine to launch future commands when an enable signal arrives. If there is a reset signal or there is no enable this values are kept.

When enable_phase_ctrl_c signal goes high for a clock period, the state changes into Phase Matching. The Phase Matching and Pot address control states have a sub-state machine that truly control the SPI bus waveform generation timing and values. So, when the main state-machine reaches either Phase matching or Pot address control, the actual running process is the corresponding sub-state-machine. Next picture, *Figure 27,* shows the mentioned sub-state-machine diagram.

As the SPI bus signals must follow a strict waveform timing pattern the signal Freq_Limit, generated in the top-level entity, controls the timing access to each state of the sub-state-machine. So, next state is accessed only when a new Freq_Limit pulse arrives. Then, the SPI clock is generated alternating POT_Load (SPI_Clock = '0') and POT_Send states (SPI_Clock = '1'). In this way the frequency of the SPI clock can be tuned to reach different clock frequencies. That is important since the maximum clock frequency that the Digital potentiometer can operate is "only" 10 MHz. The chosen frequency for the SPI clock is 6,25 MHz to avoid working close to the maximum frequency. Such frequency is originated in a counter at the top-level entity that generates the "Freq_limit" signal. Notice that the "Freq_limit" signal must be 12,5 MHz because the SPI clock is formed by switching from POT_Load and POT_Send states on each positive clock edge.

In the same way the signal POT_cmd_send is set to '1', in the POT_Stop state, when a complete command is already sent which causes the main state-machine to go to Idle state, so the the SPI bus goes to Idle state.

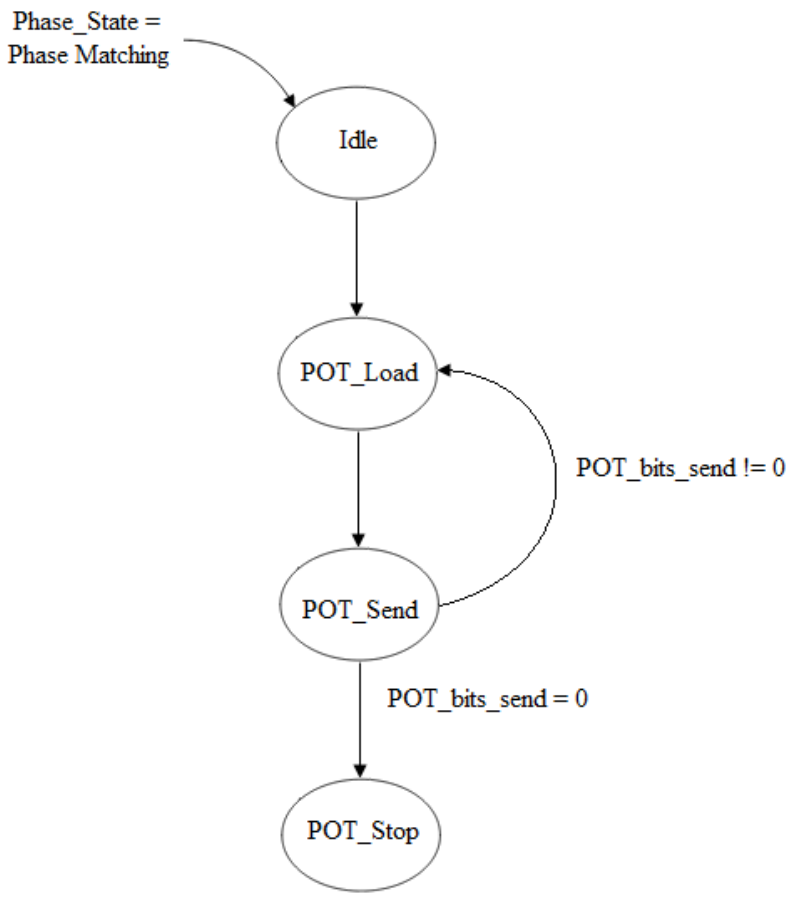


Figure 27. Phase matching or Pot address control sub-state machine diagram

What the Phase Matching sub-state-machine does is to send a 16 bit message through SPI to the corresponding Digital Potentiometers. The message consists of two commands bytes, one for each wiper on a Digital Potentiometer chip. So the first command byte is addressed to the wiper 0 (address 0h) while the second command byte is addressed to the wiper 1 (address 1h). The command bits can be either “01” for an increment or “10” for a decrement. Finally as the data bits are not important, zeros are sent. Although only two command bytes are sent, the command used is continuous increment. So there is no need to return the chip select signal to steady state until both bytes are already sent.

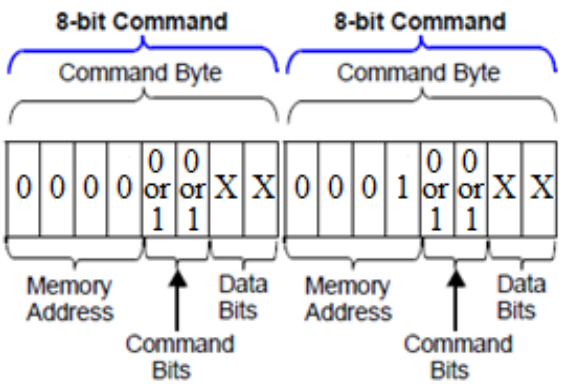


Figure 28. Command bytes sent structure in single potentiometer operation

The command bits vary depending on a parallel process called “inc_or_dec_sel”. Such process decides whether to send an increment or decrement command to the Digital potentiometer depending on the BPF_Vin_First_c and BPF_Vout_First_c input signals. If BPF_Vin_First_c is high the command bits are switch to decrement, but if BPF_Vout_First_c is high the commands bits are switched to incremented instead.

The signal POT_SDO_c is not used in single potentiometer operation, because there is no need of tracking the current position of the wiper. That is why because the digital potentiometer logic does not execute an increment command when the wiper has reached the top position. And does not execute neither a decrement command when wiper reaches the bottom position. Apart of tracking the wiper position, the POT_SDO_c signal could be used as error SPI bus checker, because this signal receives monitoring of the state command error bit. In this project it has not been implemented any command error verification within the SPI bus controller because the maximum clock frequency is not reached, and the prototype fabricated is not intended to operate in harsh conditions.

Finally, when double potentiometer operation was desired some modifications were done. First a different message was sent in the Phase Mismatching state. Secondly a third state was added, following the state machine diagram shown in *Figure 26b*, and a new process was implemented.

In double potentiometer operation (3.4.1.3), each variable resistor within the filter used a whole digital potentiometer chip, then two potentiometers are concatenated to be used by the same resistor. That is why both wipers cannot receive the same command each time. Instead only a byte command is send to only one wiper at each time. So, the Phase Mismatching sub-state-machine was modified to send only 8 bits instead of 16*, Figure 29*.

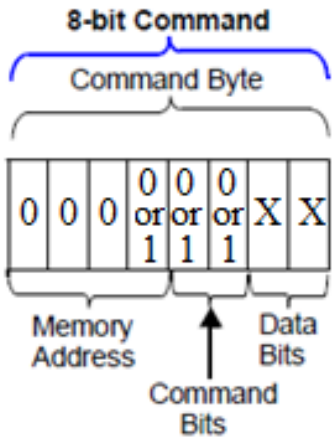


Figure 29. Command byte sent structure in double potentiometer operation

Then apart of deciding whether to increment or decrement, it is needed to decide which wiper have to be addressed at each time. That is why another parallel process called “wiper0_or_wiper1_sel” is implemented. This process detects when each wiper reaches the wiper’s respective bottoms or tops. But to do it, there is the need of implementing the read command. That is why the third state called “Pot address control” was added. This state operates similar as the Phase Matching in single potentiometer, because sixteen bits are also sent. But when the read command bits (“11”) are received by the digital potentiometer, it sends 10 data bits back to the Master (first bit received is spare). Then, although the remaining bits sent by the master are not important, the SPI clock must still be generated. So, on each clock period a bit is received through the POT_SDO_c signal. The data received is used in the “wiper0_or_wiper1_sel” process.

3.6. Quality factor control loop by amplitude matching

3.6.1. Amplitude mismatch detector circuit

The first step in order to implement the amplitude mismatch detector is to acquire the analogue filter input and output peak value.

To detect a peak value two possibilities are considered. The first one is based on a precision rectifier. With a simple precision rectifier circuit it could be possible to eliminate the half negative part of a sinusoidal signal, while the diode voltage will not affect to the half positive part. Then if a capacitor and a resistor are connected as the load of the precision rectifier circuit, the capacitor charges and discharges adapting to the peak voltage of the analogue signal. Finally an ADC could read the peak value generated.

As it is necessary to acquire the peak value of both input and output of the tunable filter, two precision peak detectors would be needed. Next image shows a precision peak detector circuit, *Figure 30*.

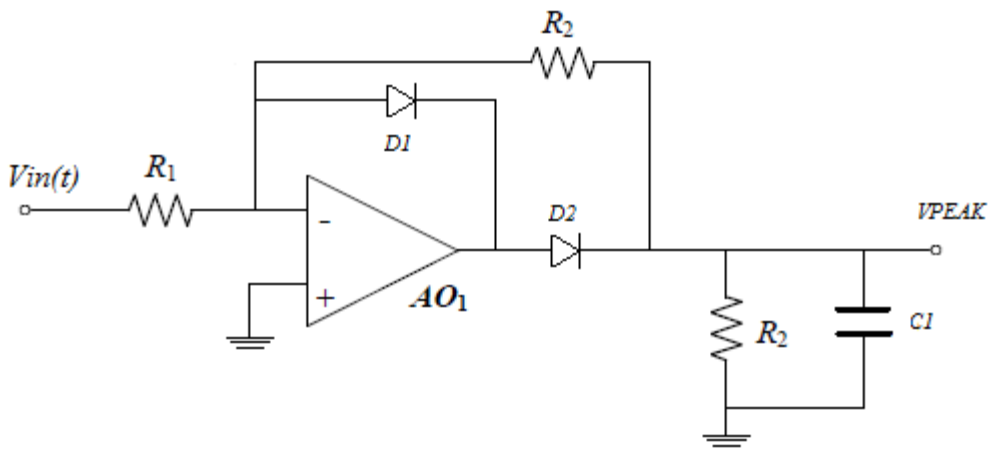


Figure 30. Precision peak detector circuit

Then two more OPAMPs, four diodes, six resistors, two capacitors and an ADC would be needed, incrementing the bill of materials.

The second possibility considers the use of only an ADC. To obtain the peak value of an analogue signal with an ADC, ideally it is only needed to sample the analogue signal at enough sampling frequency to validate the Nyquist-Shannon sampling theorem. The theorem states that to correctly reconstruct a sampled analogue filter, the sampling frequency must be at least twice the signal frequency sampled. But to reduce the error between the real maximum amplitude and the obtained value, the sampling frequency will be increased to be able to obtain several samples around the maximum amplitude point.

Considering the next image, *Figure 31*, the region where the maximum amplitude occurs is 16º degrees wide. Considering a worst case of 20 kHz signal, 16º are equivalent to 2,2 µs. Then with a 1 Msps ADC a maximum of two samples could be done during the maximum amplitude region. If we consider that the analogue signal maximum amplitude will be stable during several periods, it is even possible to do an averaging of different period maximum amplitude points to have a more accurate value.

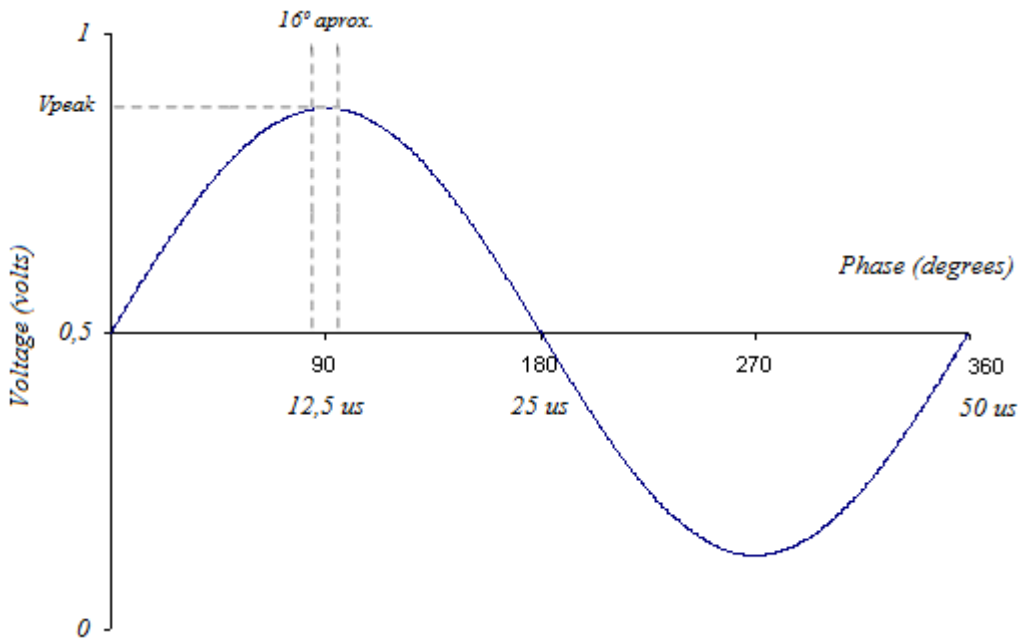


Figure 31. Maximum amplitude region of a 20 kHz sinusoidal

Although the ADC option in the phase detector is discarded, chapter 3.5.1, because a fast ADC is needed, in the amplitude detector 1 Msps ADC is fast enough. Because while before the required data was the time of crossing zero, now instead the exact time of the maximum amplitude is not required, but the voltage value. Also, the second option does not require extra circuitry that would increase costs and PCB complexity, because the ADC required fits the on board ADC characteristics. That is why the amplitude mismatch detector circuit implemented is the second option, using the on board ADC.

3.6.2. Amplitude mismatch detector logic

As explained before in chapter 3.3.4, the amplitude mismatch detector logic was first implemented in the Spartan-3A Platform. On that board, the analogue capture circuit is formed by a programmable gain amplifier and an external ADC, both driven by SPI communication bus. In that case a block to send commands via SPI bus was designed, fulfilling all requirements in “Analogue Capture Circuit” chapter from [15]. Then, configuring the programmable gain amplifier and reading both channels of the ADC was achieved. But as the code implemented was rather similar to the code used in Phase Mismatch detector block and it is completely not reusable for the current Nexys4 Platform it won’t be detailed on this document.

Then, the amplitude mismatch detector logic implemented on the Artix-7 Nexys4 Platform is spread into three parts: the XADC core and the XADC controller, due to the ADC used is the Xilinx 7 series FPGAs XADC Functionality (detailed in [17]) and the Peak Amplitude detector.

First, the XADC is an IP core from Xilinx that enables analogue mixed signal functionality, providing a complete control of the on chip dual 1 Msps ADC. Although the Artix-7 FPGA has one dedicated differential analogue input and sixteen auxiliary analogue inputs it is important to realize that the Nexys4 only wires four of this auxiliary analogue inputs to the edge board connector called JXADC, which is barely explained in chapter 10.1 within [19], but detailed in the “Nexys4_Master.ucf” file obtained from the platform vendor [14].

3.6.2.1. XADC core

The picture below, *Figure 32*, represents the on-chip XADC block diagram. The whole XADC system is controlled by the DRP (Dynamic Reconfiguration Port). This port can be accessed either from the JTAG bus or the FPGA logic. But while the communication to the JTAG can be direct stablished (with appropriate JTAG cable), the interconnection to the FPGA logic requires the XADC to be instantiated within the VHDL project.

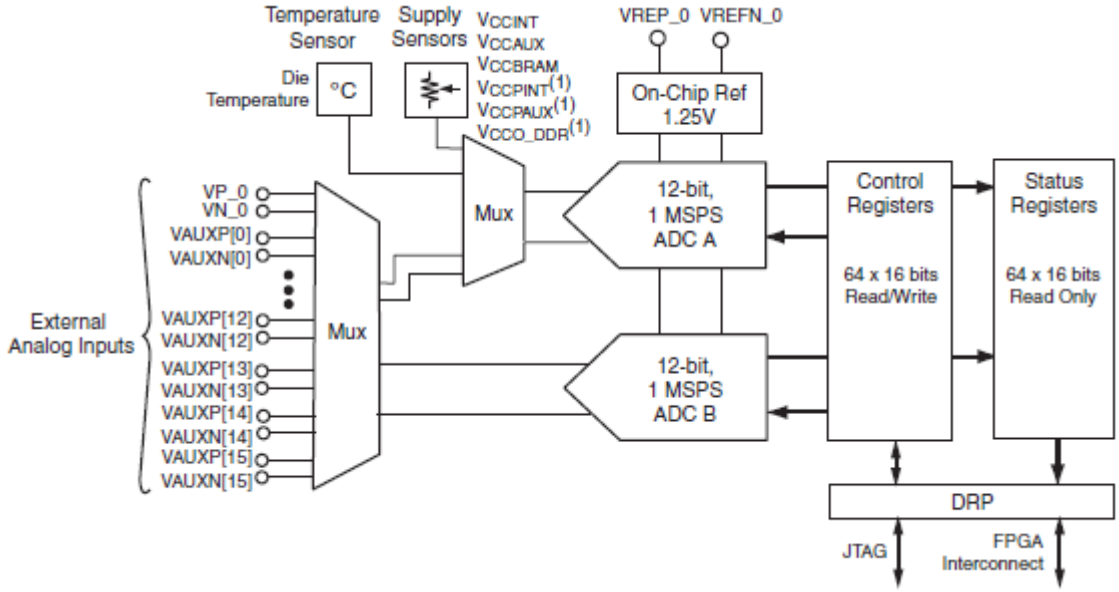


Figure 32. XADC Block diagram extracted from [17]

As there are many sampling modes and different ways to use the XADC the best way in order to implement the XADC core is to use the Xilinx Core Generator tool, which is located in the ISE Project Navigator at the Tool tab. To work with this tool is necessary to create a project and select the exact FPGA chip to work with. In this case is the family Artix7, device XC7A100T, package CSG324 and the -3 speed grade is selected. Then a whole list with all Xilinx IP cores appear, and the IP cores that are compatible with the selected device are highlighted. At the end of the list appears the XADC wizard. When the wizard is selected at the right appear some information about the XADC wizard software and a list of actions that are allowed from which the “Custom and Generate” is selected. The XADC wizard opened guides the design of the XADC core.

The XADC wizard includes a user guide [18] that helps understanding all the steps that must be followed to obtain the desired core.

Since the XADC has a Dual ADC the most suitable operation mode seems to be the “Independent ADC mode”, but as detailed in [17], in such mode one ADC must be used for monitoring internal voltages and temperatures, while the other ADC can be used to sample external analogue input channels. As for this project monitoring internal voltages and temperatures is relevant this XADC mode is not used. Instead, the XADC operation mode used is Simultaneous Sampling Mode. This mode assigns the external analogue input channels 0 to 7 to one ADC and channels 8 to 15 to the other ADC, so one channel of each group can be sampled at the same time. Although the channel choice should be CH2 and CH10 or CH3 and CH11 to accomplish the previous statement, the wizard does not allow this selection. The next table shows the final configuration selected:

Paramater	Option Selected
Operation Mode	Simultaneous Sampling
Timing Mode	Continous Mode
DRP Timing Options	Enable DCLK, DCLK 100 MHz, 1 Msps
Dynamic Reconfiguration Port	Enable DRP
Control Ports	Enable RESET_IN
Status Outputs	Enable CHANNEL_OUT, EOC_OUT, EOS_OUT, BUSY_OUT
ADC Setup	None Channel Averaging, None Calibration
Enable Alarms	NO Alarms enabled
Channel Sequencer Setup	VAuxP2/VAuxN2 (unipolar), VAuxP3/VAuxN3 (unipolar)

Table 2. XADC configuration

The continuous timing mode allows the XADC not to wait to a trigger signal to start the next conversion, so when a conversion ends the next one starts and so on. The DRP is enabled and set to work with 100 MHz clock then the corresponding basic inputs and outputs will appear on the ADC core. Other XADC signals are set in the Control Port and Status Outputs areas which are useful for the XADC control. In order to simplify the core and the system the averaging, calibaration and alarms options are not selected. Anyway the Alarm_out signal appears in the XADC core, but it will not be used. Finally the desired external analogue input channels selected are, VAuxP2/VAuxN2, VauxP3/VAuxN3.

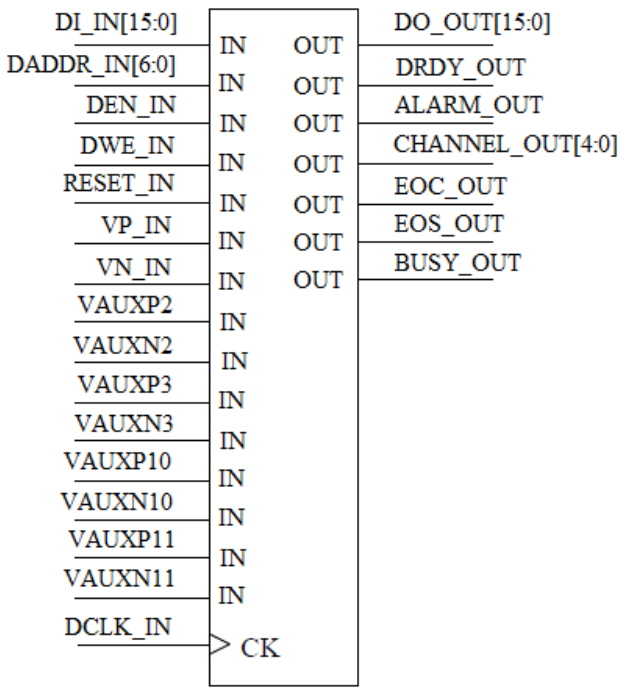


Figure 33. XADC core block diagram

Notice that such auxiliary analogue inputs are prepared for differential sampling, when implemented properly the voltage read at the negative channel can be subtracted from the read at the positive channel removing then the noise. This method is recommended for noisy digital environments, but it will not be applied within this project, that is why both VauxN2 and VauxN3 are tied to ground internally.

In order to proof that the XADC core is well generated the VHDL code is simulated. For a quick verify process the same Xilinx Core Generator Tool provides an ISE project with the XADC core, an example design block that calls the XADC core and a testbench as a top-level entity. Next image, *Figure 34*, is the result of the simulation. It is possible to see that each 1.040 ns a sample is taken to a different channel. So the effective sampling rate is 480,77 Ksps per channel instead of 1Msps per channel.

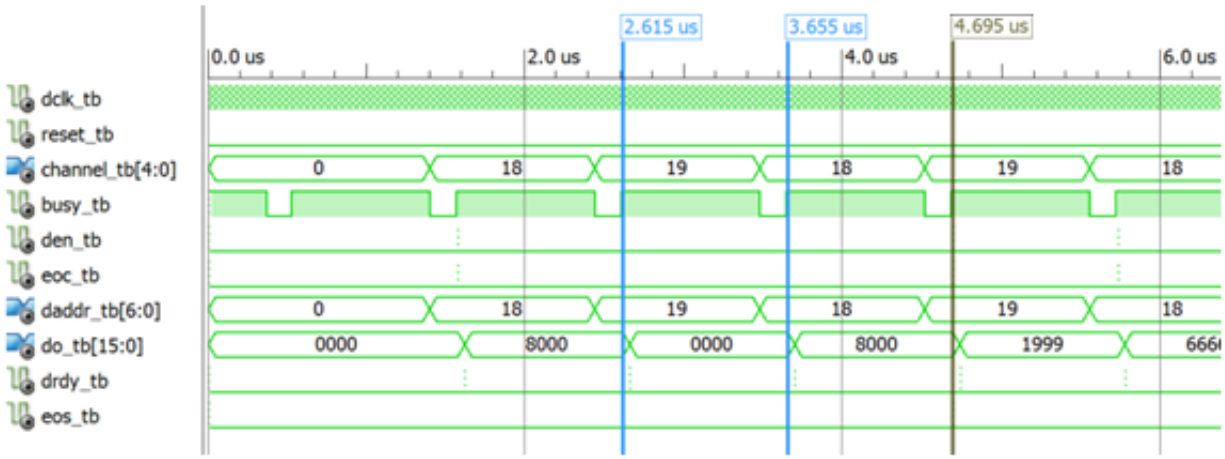


Figure 34. XADC testbench simulation

Then to use the XADC block within the Digitally Tunable Analogue Filter project the core contained in a file called “xadc_wiz_v2_1.vhd” is added to the Filter ISE project folders.

3.6.2.2. XADC Controller

The example design block and the logic within the XADC testbench project generated with the wizard must also be added, with nomenclature modifications, to the top-level entity in order to call the core and control the DRP to read the desired channels. This logic block implemented among the top-level entity file, “DTAF_Controller.vhd”, is called XADC Controller. This block could have been also placed in a different file as the phase detector block, but it was preferred to hold it in the top-level entity file in order to avoid possible extra issues.

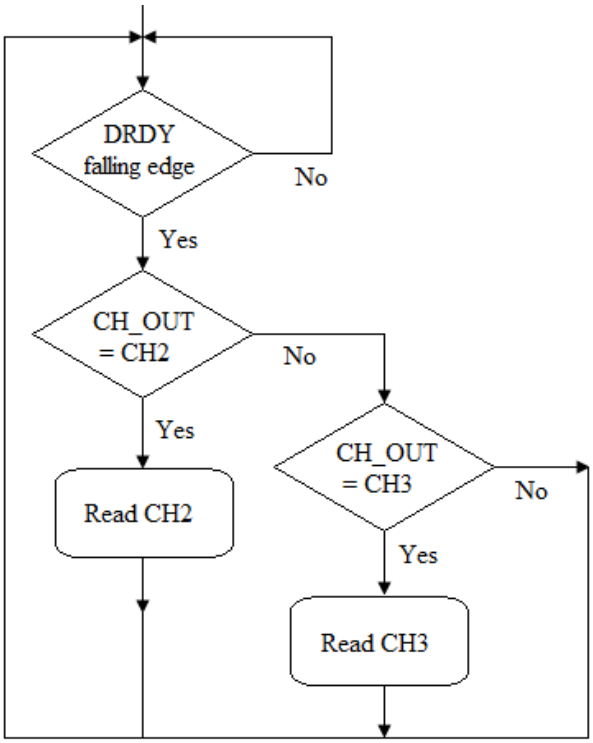


Figure 35. XADC controller ASM diagram

So the XADC controller implemented instantiates the XADC core from the top-level entity called DTAF_Controller.vhd”, configures some XADC inputs signals, and implements the logic to carry out the read of the external analogue input channels VauxP2 and VauxP3. In the image above, *Figure 35*, there is the ASM Diagram that explains how the XADC controller works.

3.6.2.3. Peak Amplitude Detector

Once both external analogue input channels are sampled, the read values are stored in a signal called “Vin_CH” for VauxP2 and “Vout_CH” for VauxP3. Then there are two similar process called “Peak_detector_Vin” and “Peak_detector_Vout” which extract the peak amplitude value of each channel.

This logic block implemented among the top-level entity file, “DTAF_Controller.vhd”, is called Peak Amplitude Detector.

At this point is important to know the count range of the ADC. It goes from 0 to 4095 where one count is 244 µV and full scale is 1 V, see next *Figure 36*.

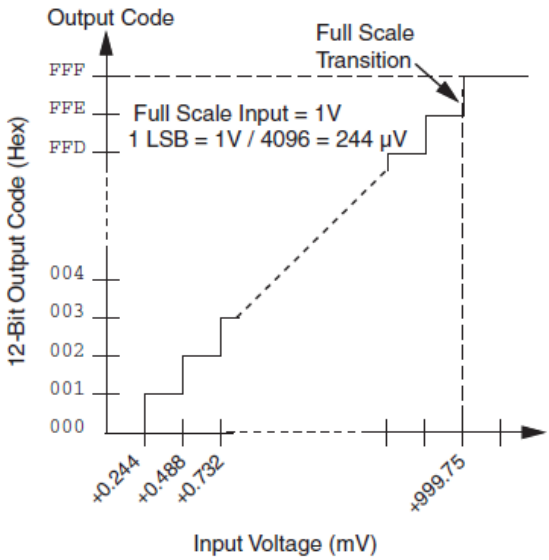


Figure 36. XADC unipolar transfer function from [17]

What each peak detector does, see below *Figure 37*, is to identify the major values samples which values are over a reference value. When the current sample is under the reference value, the major value sample obtained during the process is the new peak value. This operation allows to refresh the peak value detected on each period of the analogue input signal.

Ideally the reference value must be set to 800h which is the mid-scale of the XADC operating in unipolar mode and corresponds to the applied offset voltage to the filter OPAMP. So it is possible to detect peaks until 500 mV. But reducing the reference to 500h generated a better quality factor loop response. Actually reducing the reference means that the peak detector has more time and more samples to determine which the peak value is. Although the drawback is that is not possible to detect peaks under 200 mV, is not an issue because the Vin_CH to sample is fixed to 350 mVpeak and the Vout_CH is intended to follow Vin_CH.

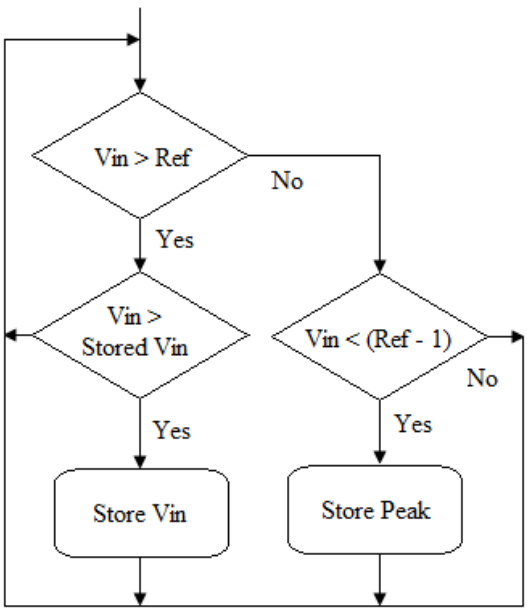


Figure 37. Vin Peak Detector ASM diagram

Finally both peak values detected, from Vin_CH and VOut_CH, are compared to obtain which is bigger than the other. The result allows to identify if the Vin_CH is over the Vout_CH, then AMP_Vin_Over signal goes high, or the opposite and AMP_VOut_Over signal goes high. This two signals are the output of the whole Amplitude mismatch detector logic explained in this section. Both signals are connected to the next block, the "Amp_ctrl_comp" block.

3.6.3. Quality factor loop digital potentiometer controller

The quality factor loop digital potentiometer controller is embedded in the FPGA. The VHDL code of this block is contained in a file called, "amp_ctrl_comp.vhd", which is a sub-entity under the top-level entity called "DTAF_Controller.vhd". This block controls the digital potentiometer connected in the quality factor loop, corresponding to R_2.

As this block have the same logic as the phase_ctrl_comp block (3.5.3) is not explained in detail, see the VHDL code similarities in the appendix. The main differences are the inputs and outputs of the block. In this case the SPI bus signals are connected to a different digital potentiometer which is solved by a nomenclature change (adding "amp" to all names). The inputs AMP_Vin_Over_c and AMP_Vout_Over_c come from the amplitude detector logic within the top-level entity. While Freq_limit and enable_amp_ctrl_c are generated in the top-level entity and regulate the logic action of this block.

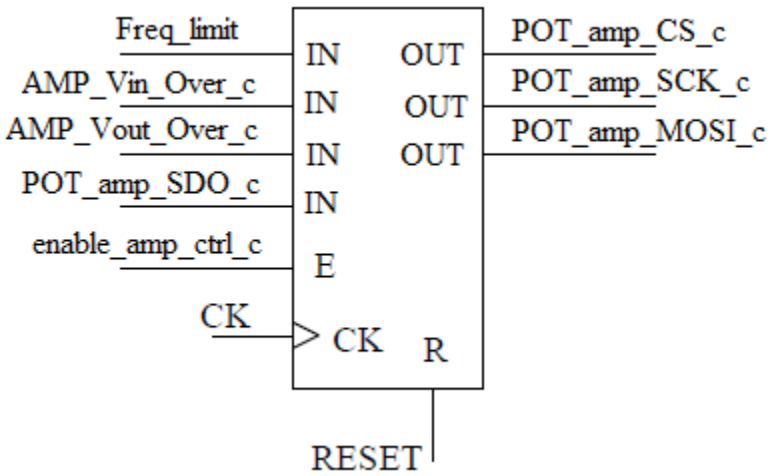


Figure 38. Amplitude control block diagram

One could think that a unique SPI controller block could have been implemented instead of having the phase_ctrl_comp and amp_ctrl_comp blocks. In such a way it may be possible to occupy less logic cells within the FPGA, despite that multiplexing input/output signals would be required as there are two different SPI busses, adding complexity to the design. But the most important reason is that a real-time control action is desired. Having two dedicated blocks ensure that in any clock period where the corresponding enable signal goes high the control action can start.

3.7. DTAF controller (Top-level entity)

As introduced before the top-level entity called DTAF controller is contained in the file "DTAF_Controller.vhd". It contains all the I/O that interconnect the FPGA with the external circuitry such as the digital potentiometers, the analogue input channels, phase feedback signals and the buttons. During the project development this block have been adapted to different needs, for example including LEDs to display ADC read values or slide switches to interact with the FPGA logic. The image below, *Figure 39*, shows al the I/O of the DTAF_Controller.

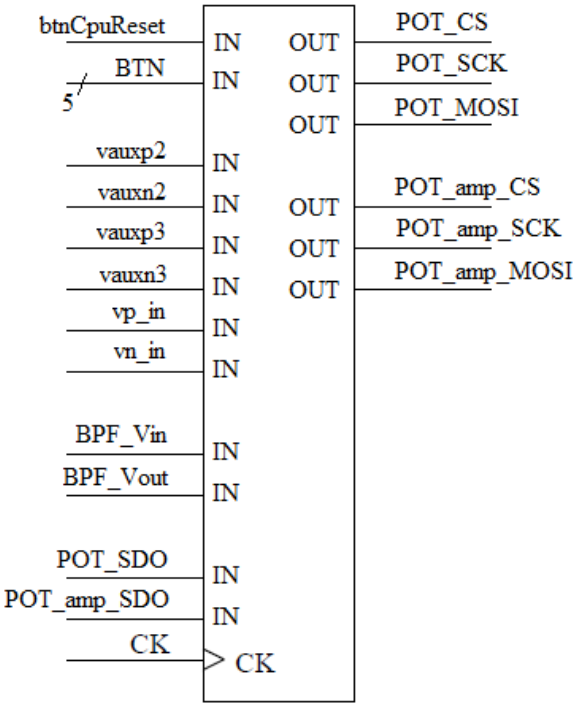


Figure 39. DTAF_Controller block diagram

As the DTAF_Controller is the top-level entity it contains the declaration and the mapping of all sub-entities involved in the project. But as the DTAF_Controller does not only join the different blocks there are also other process and concurrent logic within the DTAF_Controller vhd file. For example there is the logic that decides when to enable the different control or detection blocks and as specified before, in chapters 3.6.2.2 and 3.6.2.3, there are the XADC Controller and the Peak Amplitude Detector. Next image, *Figure 40,* shows the distribution and relations of the DTAF_Controller content.

Finally there is a process called "main_counter" that divides the global clock frequency of 100 MHz into a 12,5 MHz signal called "Freq_limit" which is an input of the amplitude control and phase control blocks. That signal is used to generate the SPI Clock signals for both SPI buses.

In the image below there is a diagram of how the "DTAF_Controller.vhd" code is distributed. The squares are the external .vhd files while the rest are process and concurrent logic within the top-level entity. Where the arrows represent the signal path between blocks.

Although the debouncer.vhd is the less important block as it debounces the external input button signals, it also debounces the reset button placed on the FPGA platform which then is connected to all the process through the project.

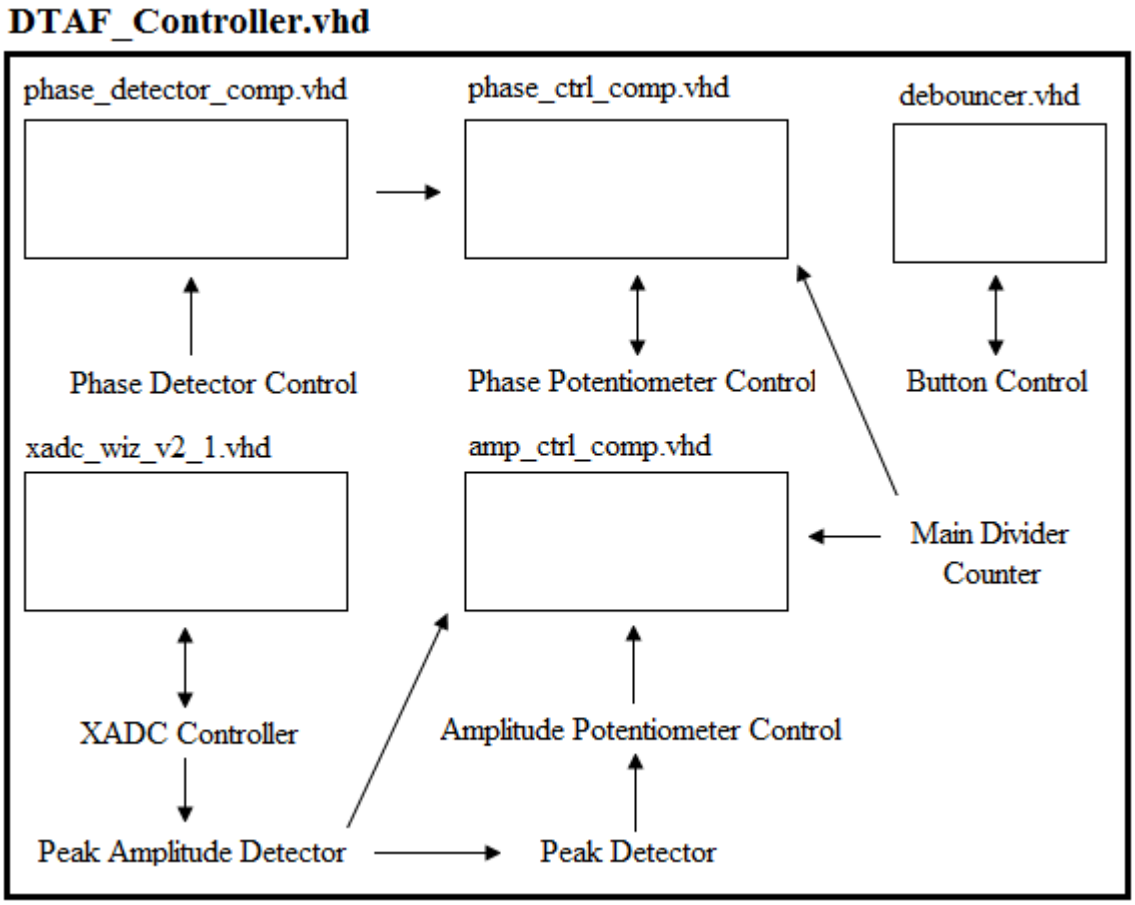


Figure 40. DTAF_Controller distribution

3.7.1. Central frequency loop control logic

Next diagram, *Figure 41,* represents the central frequency control loop implemented. As the "Advanced or Delayed" and "Wo loop Potentiometer Control" blocks were already detailed, the point of interest now is how the control logic, after receiving the detection signals, decides to trigger the control action.

The strategy followed consists on reducing the phase mismatch by incrementing or decrementing one step each potentiometer wiper involved in the loop. Then, as the clock frequency of the FPGA is three orders of magnitude higher than the worst case filter input (20 kHz) it is possible to consider a different control action per each new period of the input signal. So, the desired situation is that the mismatch is reduced so much that each new control action is the opposite of the last one. But even in that situation the control loop does not stop or changes, there is only one control triggering mode.

Then considering that the minimum time between two phase mismatch notifications is 50 µs of the worst case period, the control loop must actuate under that time. And it does, because the phase mismatch detection only require 3 clock periods, the RS flip-flop only takes 1 clock period, and finally the potentiometer control takes 3 µs to actuate being the slowest one.

So, as soon as both phase mismatch signals (BPF_Vin_First and BPF_Vout_First) are received by the DTAF_Controller an OR gate between them generates the enable_phase_ctrl signal that will trigger the "Wo loop Potentiometer Control" block. So, anytime that there is a mismatch between the input and output of the CTF circuit the control action is triggered. In a parallel way, there is a RS flip-flop that together with a multiplexor decides whether to send an Increment or Decrement command. Finally, when the enable_phase_ctrl triggers the "ω_o loop Potentiometer Control" block, the selected command corresponding to the actual clock period it is already selected because BPF_Vin_First and BPF_Vout_First are synchronous signals.

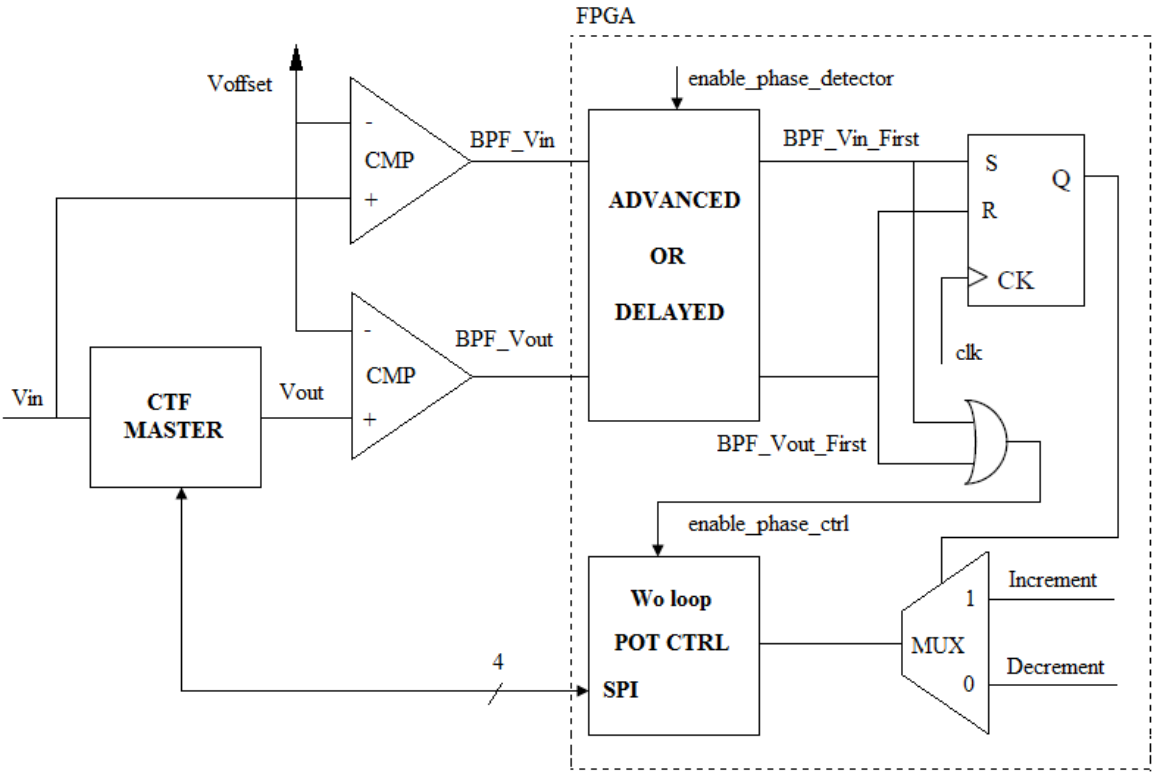


Figure 41. Central frequency control loop block diagram

3.7.2. Quality factor loop control logic

In a similar way works the quality factor loop control logic. Next diagram, *Figure 42*, represents the quality factor control loop implemented. As the "XADC", "Peak Detector" and "Q loop Potentiometer Control" blocks were already detailed, the point of interest now is how the control logic, after receiving the detection signals, decides to trigger the control action.

The RS flip-flop and the multiplexer are also present in this control logic, while the OR gate is changed by a process called "New_Peak". Such process generates a one clock period pulse output when the input peak detector detects a new amplitude peak. Then the control action is related to the filter input frequency.

Notice that the capability of tuning the quality factor is not implemented, as the control designed is intended to obtain always a quality factor of Q = 1, because the filter output is modified in order to match the filter input for all the frequencies.

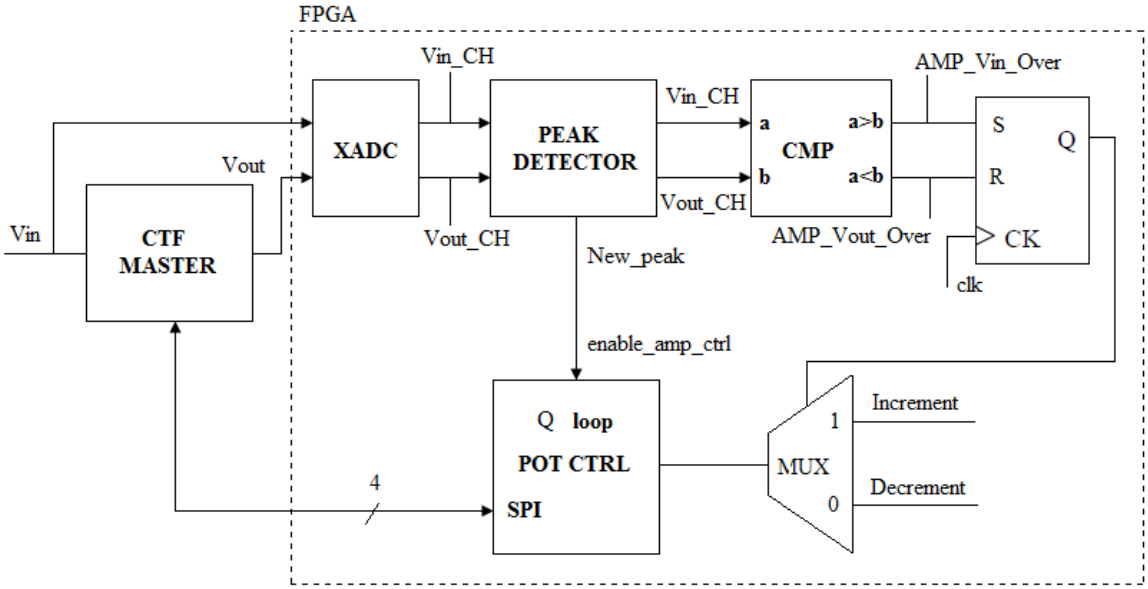


Figure 42. Quality factor control loop block diagram

3.8. PCB design and fabrication

In order to finally verify the whole DTAF in a proper way it is necessary to design and fabricate a PCB including all components within the filter. That means to fabricate both master and slave filters. On one hand the CTF topology with automatic control loops first tested on a breadboard will now be replaced for a PCB and it will become the master filter. And on the other hand a second PCB, but without loops, will be fabricated to perform the slave filter. Finally connecting master and slave PCBs as in *Figure 2,* correctly power up both boards and connecting reference voltage for the master filter and input signal for the slave filter will lead into the complete circuit of the DTAF.

The software used to design the PCB is Proteus 7 Professional, from Labcenter Electronics. This software is formed by two programs, one called Isis that has all typical commands and environment for electronic schematic design, including a basic but large enough libraries to work with. And another one called Ares, which has all typical commands, tools and environment to develop a PCB up to 14 inner layers.

Both programs have the utilities to modify library components or create new ones. Which is so useful when a part needed is missing in the library. So in the Isis program new schematic parts can be created. While in the Ares program new layouts can be developed.

The only disadvantage of this software is that, as is an English company software, some distance parameters can only be set by inches or thou units (thousandth of an inch) making the task harder for engineers that are not used to the imperial system.

3.8.1. Schematic

First PCB design step is to define the schematic of the circuit. The schematic must have all the connections that will form the electrical circuit. Also some consideration about how will be the whole system connected must be taken into account. That is why as the circuits for both master and slave filter are almost the same, it is decided to design a PCB compatible for both circuits. So two PCB with the same design are fabricated, one working as a slave filter, and the other one as master filter. That implies that some parts within the PCB will be common but some not.

First of all, the PCB have three banana terminals to easily connect BNC cables. J3 terminal is the filter's analogue input signal. J4 is the ground terminal to connect to oscilloscope and function generator, or other laboratory equipment. And finally J5 is the filter's analogue output signal.

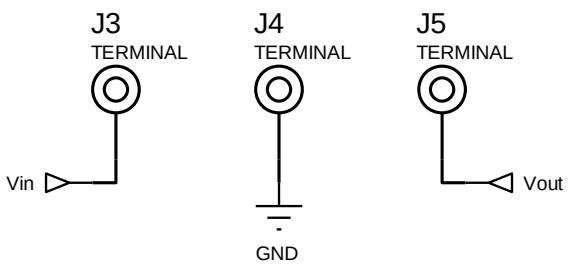


Figure 43. Banana terminals for BNC cable connection

The main circuit of the PCB to design is the CTF with digital potentiometers, shown in the next image, *Figure 44*. The digital potentiometer connection is made by ports. Ports allows designer to have a tidy schematic because large wire connections are avoided.

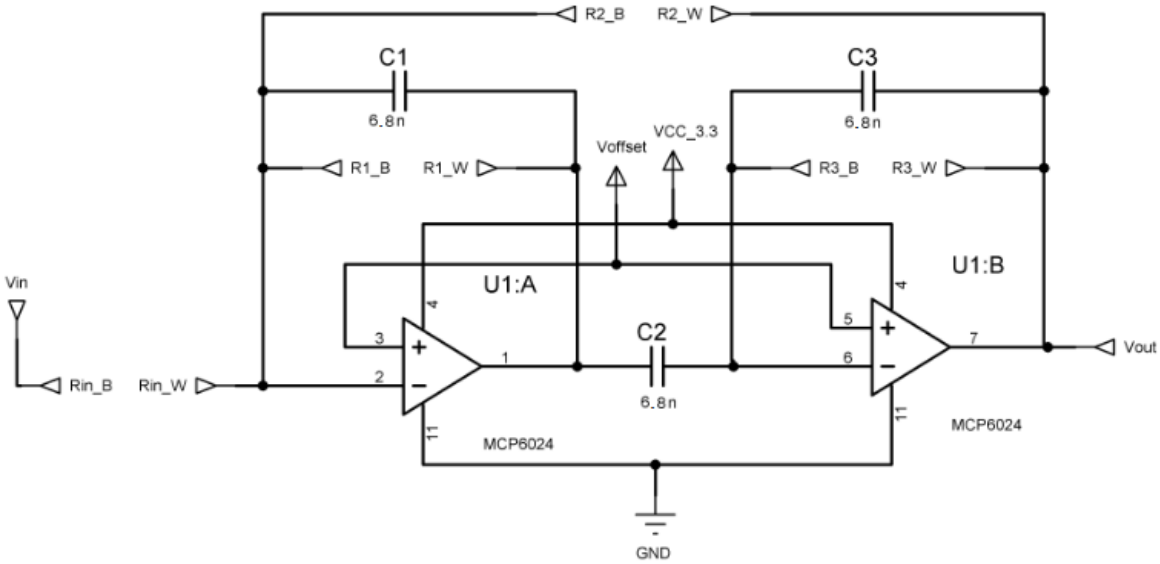


Figure 44. CTF with digital potentiometers schematic

The digital potentiometers schematic is presented below, *Figure 46*. Here are the connections of the three digital potentiometers chips that are common in the master and slave PCBs. There are placed 100 nF decoupling capacitors on each chip. Although only four wipers are needed for the circuit operation, as there are two different SPI buses three chip are needed. U2 and U3 are connected to the central frequency SPI bus control (POT_CS, POT_SCK, POT_MOSI and POT_SDO), while U4 is connected to the quality factor SPI bus control (POT_AMP_CS, POT_ AMP_SCK, POT_ AMP_MOSI and POT_ AMP_SDO).

Hence a wiper on each chip is not used for circuit operation. That wipers are instead used for test purpose, because it is not possible to measure others wipers resistance as they are connected into the circuit and this type of potentiometers must be powered to switch resistance values. To easily access the test wipers, four test terminals are placed within the PCB, *Figure 45*.

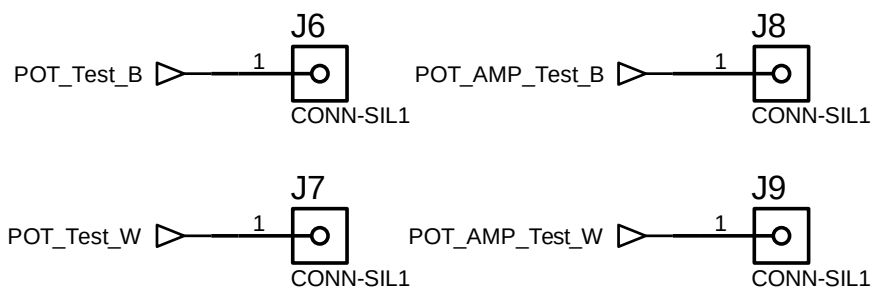


Figure 45. Test terminals for easy access on test wipers

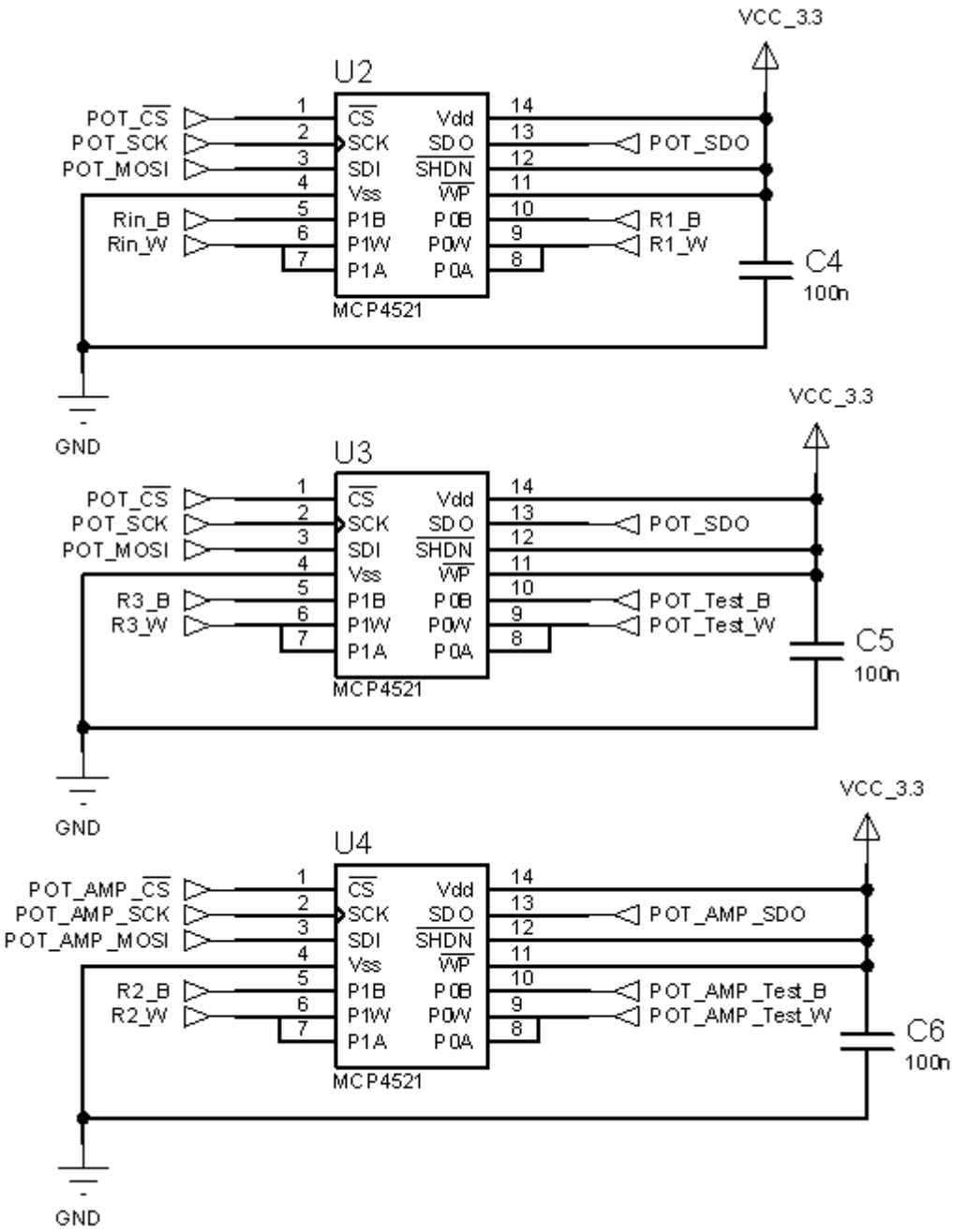


Figure 46. Digital potentiometers schematic

To easily connect both digital potentiometers SPI buses from FPGA board to both filter PCBs two SPI buses connectors are designed which are J1 and J1-1 connectors, *Figure 47*. In master PCB the J1 connector will connect directly to the JA PMOD connector in the FPGA board. But in the slave PCB the connector J1 will connect to the J1-1 connector in the master PCB. This way all signals from the FPGA board will go into both PCBs avoiding any kind of physical cable ramification. In the slave PCB, the J1-1 connector will be not mounted.

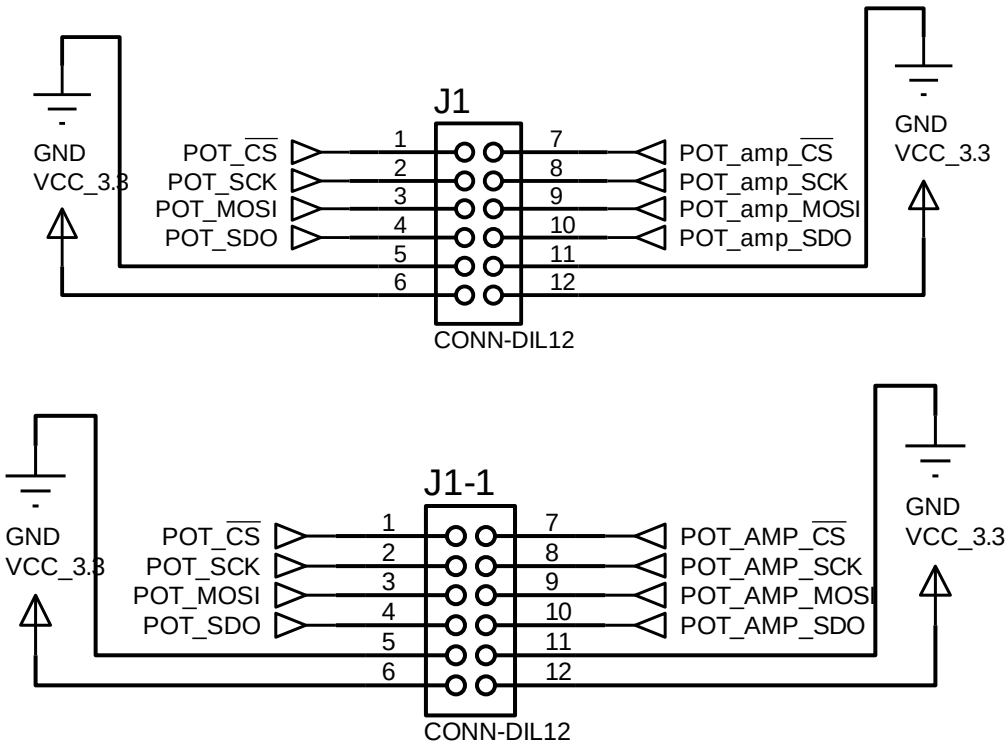


Figure 47. J1 and J1-1 header connectors

Another difference between master and slave PCBs is the feedback loop. While in the master PCB the J2 connector connects to the FPGA board. In the slave PCB the feedback J2 connector will not be mounted. J2 connector is composed of BPF_Vin and BPF_Vout signals that go to JB connector in FPGA board. Although both are digitals inputs they don't go to JA connector because is full. Each Jx connector in the FPGA board has 12 pins, but only 8 are I/O pins; other pins are power pins. J2 connector is also composed of Vin and Vout, which are both analogue signals, and GND and VCC_3.3 that all go to JXADC connector.

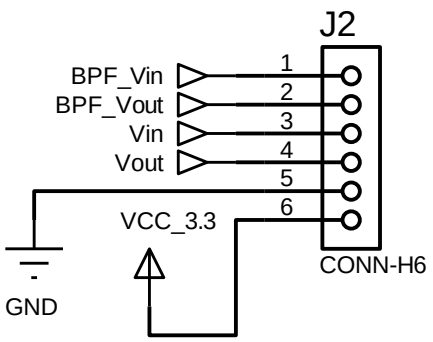


Figure 48. J2 header connector

While the feedback for quality factor only needs two analogue signals to be connected to the FPGA platform. The designed feedback for central frequency loops require some external circuitry. Next image shows the two OPAMP used in comparator configuration. The OPAMPs are called U1:C and U1:D because the used chip is a quad OPAMP. Then U1:A and U1:B OPAMPs are used in the CTF topology, *Figure 44*, but U1:C and U1:D in the central frequency feedback generation, *Figure 49.* The two signals result of each comparison are BPF_Vin and BPF_Vout detailed in *Figure 48.*

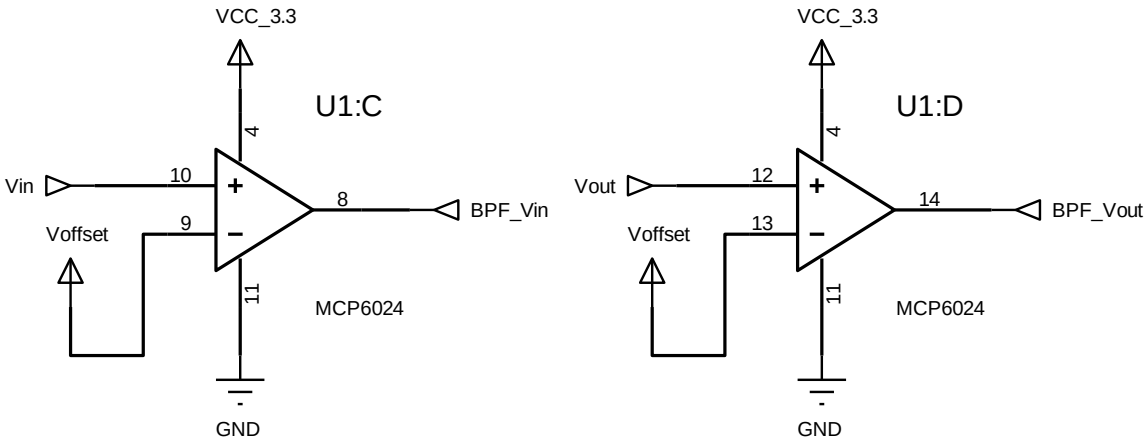


Figure 49. Input and Output voltage comparators

Finally, in order to polarise correctly the four OPAMPs, an offset voltage is generated within the PCB. As this voltage source supplies only the four OPAMP, and the typical supply current of the MCP6024 is 1 mA, a two resistors voltage divider is used. Then the voltage divider is formed by two ¼ W resistors. Notice that the Master Filter board and the Slave Filter board have different offset voltages.

The Master filter board offset voltage needs to be adapted to the on board ADC input voltage range which goes from 0 V to 1V. Then the middle point of 0,5 V is selected. While the Slave Filter board offset voltage is set to 1,65 V in order to take benefit of the whole voltage range of 3,3 V.

The next equations determines which are the resistors values required to obtain a 0,5 V offset voltage. To obtain the 1,65 V two 1 kΩ resistors are used.

$$V_{offset} = V_{dd} \cdot \frac{R_2}{R_1 + R_2} = 3{,}3\ V \cdot \frac{1\ k\Omega}{5{,}6\ k\Omega + 1\ k\Omega} = 0{,}5\ V$$

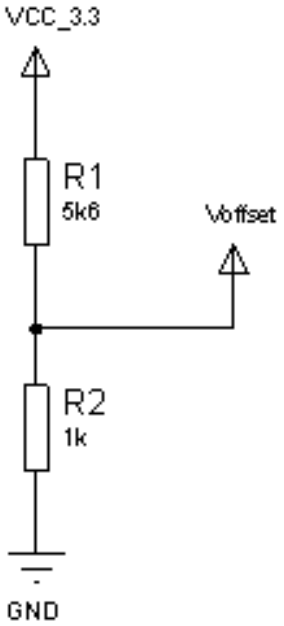


Figure 50. V_{offset} voltage generation

3.8.2. PCB layout design

The first parameter to set is the size of the PCB. As there are no are area restrictions, the size chosen follows the Eurocard format. Then the PCB dimensions are 100 mm x 160 mm. This format must be followed when mounting PCBs within a rack is desired, because the size matches 3U, where U is the rack unit with a value of 1 U = 44,45 mm. So when mounted into a rack the board would be 100 mm height and 160 mm width. The difference between 3U (133,35 mm) and 100 mm is used for the mechanical supports. Although this circuit is not intended to be mounted in a rack always is a good strategy to follow standards sizes.

The two main steps of a PCB layout design are the placement and routing, and first step is the component placement.

3.8.2.1. Placement

Although many software come with an automate placement tool, usually a manual placement performed by the designer is better. It is important to first plan how the external signals will access the PCB. In this case, the external connections are done in the connectors J1, J1-1, J2 and the banana connectors J3, J4 and J5. Then is preferred to place this connectors at the PCB's border, which will allow an easy plug of the external cables.

Secondly, it is desired to maintain the circuit order flow. That means to place all input signals at the left side then all the components in the middle, and the outputs at the right side. This organisation will help the PCB understanding when post fabrication reviews are done. Notice that sometimes a circuit may require to have the inputs as much closer as possible from certain circuits to avoid interferences from, for example power switches. In that cases there is a compromise between PCB handling and signal integrity that HW engineer must deal with.

As the placement is the first step, during the placement engineers must imagine how the routing is going to be done. A wrong placement can cause extra-long routes or even impossible routing of several connections. Also it is important to decide the route thickness to place all components separated enough to fit all routes connections. In this case the route thickness is limited by the fabrication process, all routes must have a 1 mm thickness, with 1 mm clearance.

One key aspect are the number of layers used. When SMD (Surface Mounted Components) are used a higher number of layers may be possible. With the disadvantage that many more vias should be included to connect through layers. As in this case all components are through hole, less vias may be used but the number of layers will be reduced to only two. Notice that higher number of layers means higher fabrication complexity that leads into more expensive PCBs.

Sometimes there are components where the layout is critical for their operation. Usually vendors offer a detailed explanation of placement and routing that should be strictly followed, to avoid malfunction or non-expected situations.

Finally, it may be necessary to do several loops between placement and routing. It may happen that a first placement design does not fit all required routings and a re-placement must be considered. In this project, during the PCB design 5 different placements steps were necessary.

In the next image the final placement is shown, *Figure 51*. All board-to-board jumpers (J1, J1-1 and J2) and the banana connectors (J3, J4 and J5) are placed close to the border. But the test jumpers are placed more inside the PCB, as they use is only occasional, only for test purpose. Jumpers J1 and J1-1 accomplish the condition of having inputs at the left side. Also J2, because is at the right side as it is an output connector. Although J3 is an input signal connector and J4 is the ground connection that can be considered either input or output, in this case was preferred to situate all banana connectors in the same region because that will be really useful when testing the whole system at the laboratory, as all BNC cable coming from oscilloscopes or waveform generation will go together.

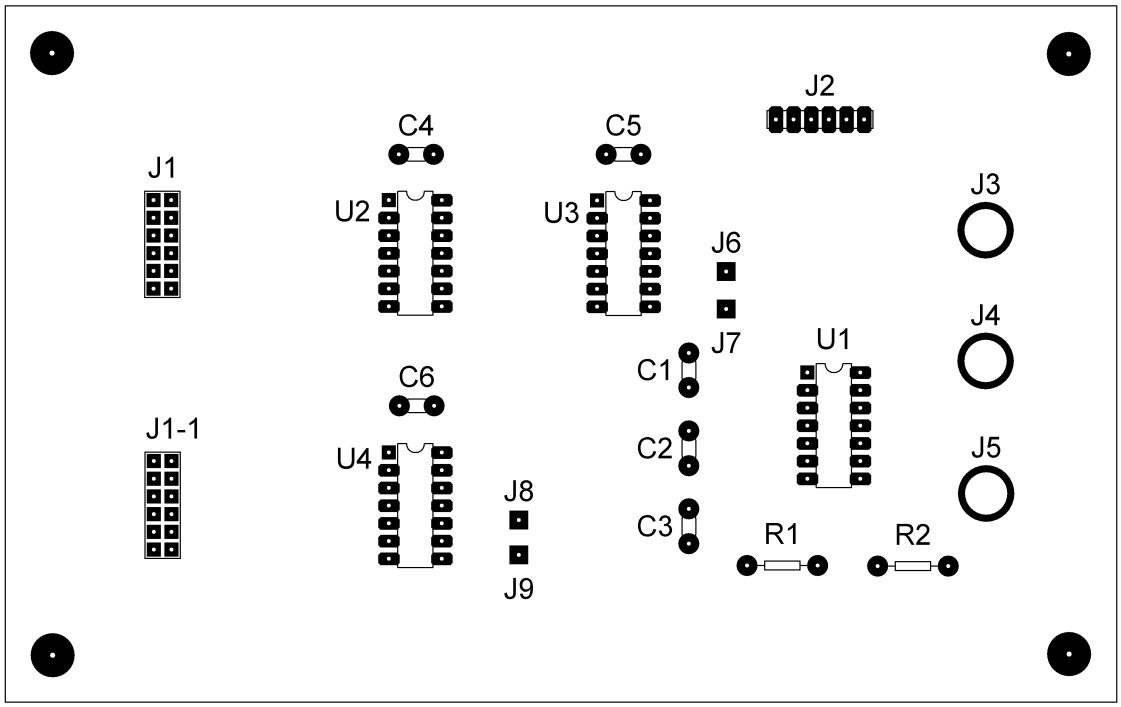


Figure 51. PCB Placement

In each PCB's corner, a drill hole is placed. This technique is used when the board is going to be used on a table, because four plastic or metal supports can be screwed with a nut elevating the board some centimetres.

3.8.2.2. Routing

When designer considers that the placement step is completed, the routing process begins. If simple circuit design is performed software given auto-route tools provide a quick way of routing all circuit signals. But in complex designs where number of pins is more than two-hundred, and separation between components is more than a few centimetres auto-route tool may fail and experienced HW designer would have to solve the routing maze.

Whether auto-route or manual routing is used is preferred to set all design rules before starting routing. Then, the first consideration in the routing consists on defining all design routing rules. In the Ares software there is a menu called, "Edit Strategies", where it is possible to define a whole routing strategy, *Figure 52*. Even it is possible to set a different strategy for signal routes than for power routes.

In the PCB designed, the same strategy is applied to signal and power routes.

- The Trace style is set to T40 (1 mm).
- The via style is V100-40, custom made one, with a 2,54 mm of copper diameter and a drill mark of 1 mm.
- All vias used are through hole ones, because other types (top blind, bottom blind and buried) would increase the PCB costs.
- Top copper and bottom copper are allowed.
- Pad-pad clearance is 0,254 mm.
- Pad-trace clearance is 1 mm.
- Trace-trace clearance is 1 mm.

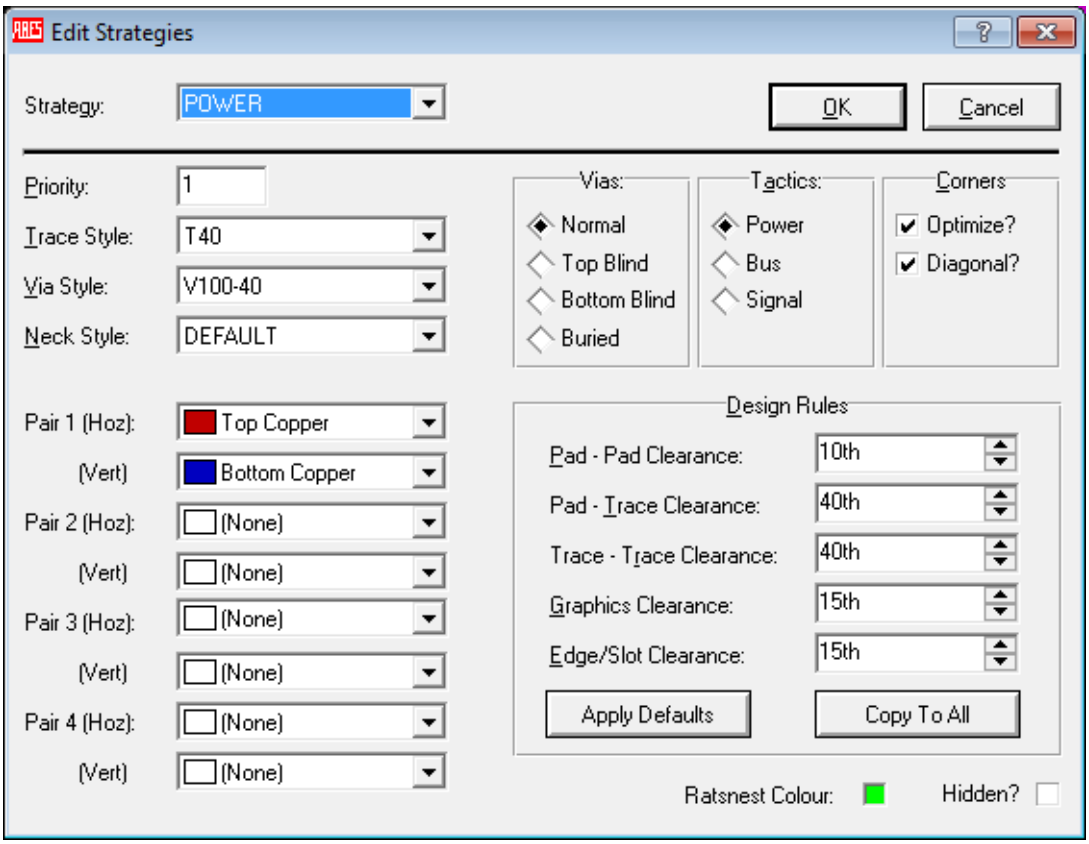


Figure 52. Routing Strategies Menu

One not evident but critical point are the layout pads. In SMD components, vendors normally offer documentation of their recommended land pattern, as it may change from one component to another. But as through hole PDIP packages have been used for decades, and mainly the only variation is the pin count there is no land pattern reference from vendors. That is why it is used the layout from the ARES libraries. But as the holes are drilled manually is preferred to have a wide copper area. Then the pad style in the 14 pins PDIP layouts for the digital potentiometers and the OPAMP is changed from a simple circular pad to a custom DIL pad, of 3 mm x 1,78 mm, *Figure 53*. Notice that the corresponding pin number one pad is left squared (2,03 mm edge) as a reference when soldering the components on the PCB. In the J2 connector the same pad style is used although it is bigger, 3,81 mm x 2,03 mm. Also because of manual drilling, all circular pads of capacitors and resistors within the PCB are set to 3,05 mm of diameter.

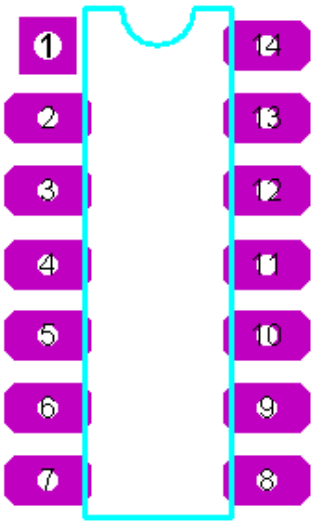


Figure 53. 14 pin PDIP layout

Finally all drill holes except banana connectors and corner board holders have a drill mark of 1 mm. Instead, the banana connectors have a M6 (6 mm) drill hole, and the corner board's holders have a M3 (3 mm) one. The test terminals have a square pad of 2,54 mm edge, while connector J1 and J1-1 have squared pad of 2,03 mm edge.

Next image shows the completed PCB routing, *Figure 54*. Colour red represents top copper, while blue represents bottom copper. Instead violet colour represents areas where top and bottom copper are used. In grey colour are shown the drill holes diameters. Turquoise colour represents the top silk layer. Top silk layer is really useful in complex PCBs as gives key information to relate PCB components with schematic circuit symbols. Even end-user annotations can be added to give explanations about some components or circuit utilization.

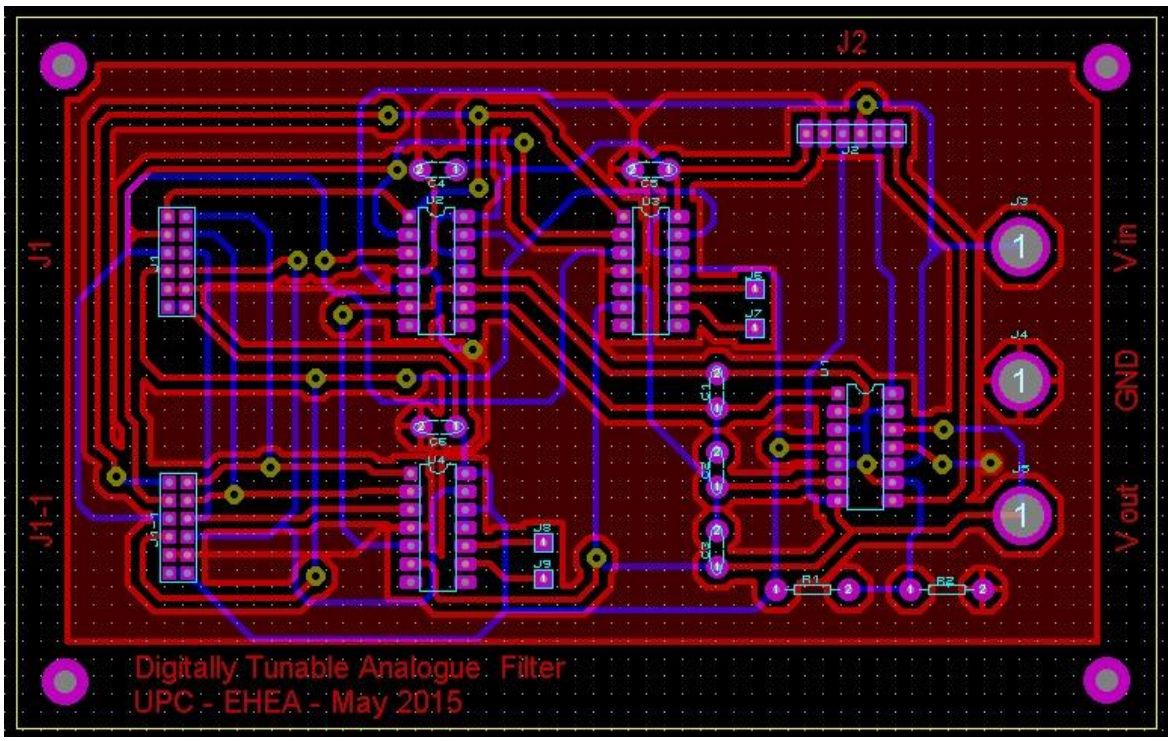


Figure 54. Completed PCB Layout

In the fabricated PCB it is not possible to have a silk layer. That is why that annotations are implemented in the same top copper layer. Next images shows separately the top copper, *Figure 55*, and bottom, *Figure 56*, layers. Notice that a whole ground plane has been implemented in the top copper layer. A ground plane serves as a return path for current of all components connected on the board.

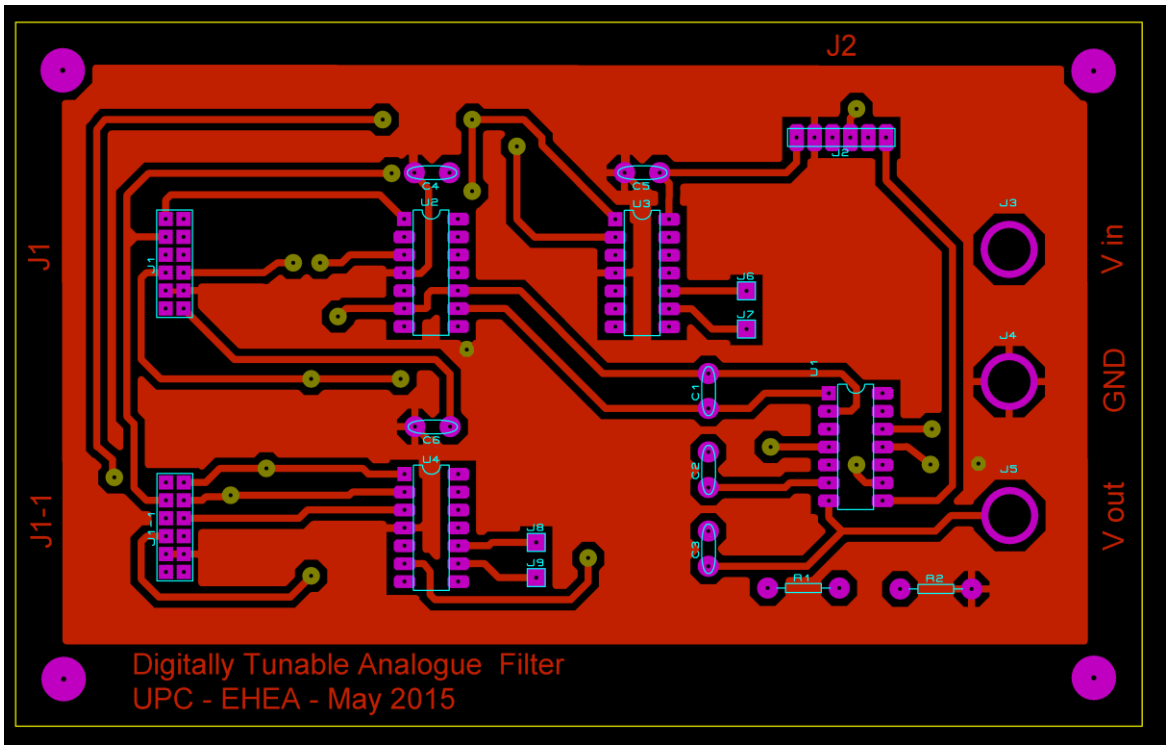


Figure 55. Top layout view

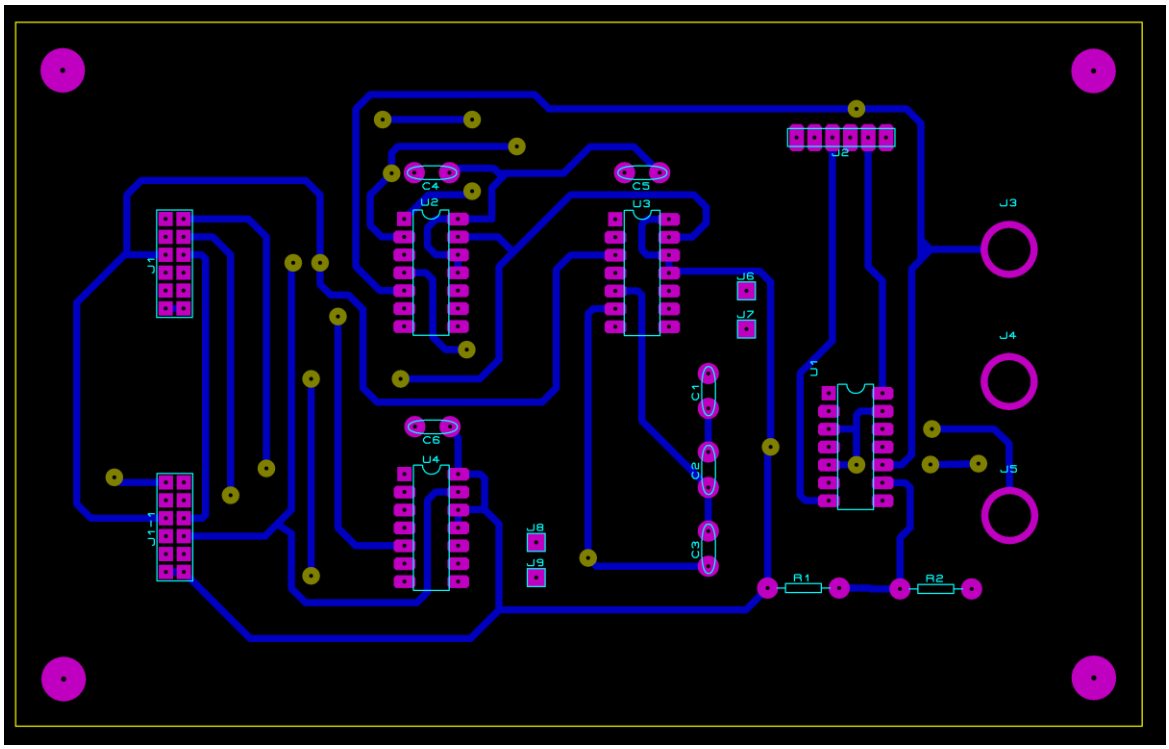


Figure 56. Bottom layout view

4. Results

4.1. Simulation

To have a first contact with the circuit operation, a bode plot simulation of the DTAF is done. Actually a parametric simulation is done with the extreme values that the potentiometers can achieve and several points in between.

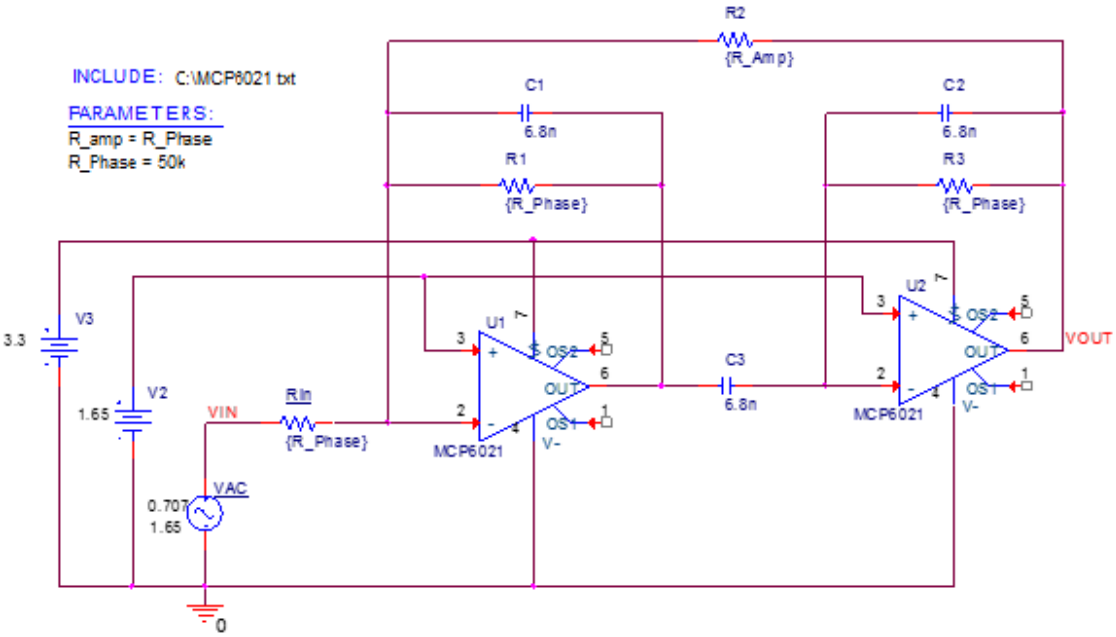


Figure 57. DTAF OrCAD Schematic

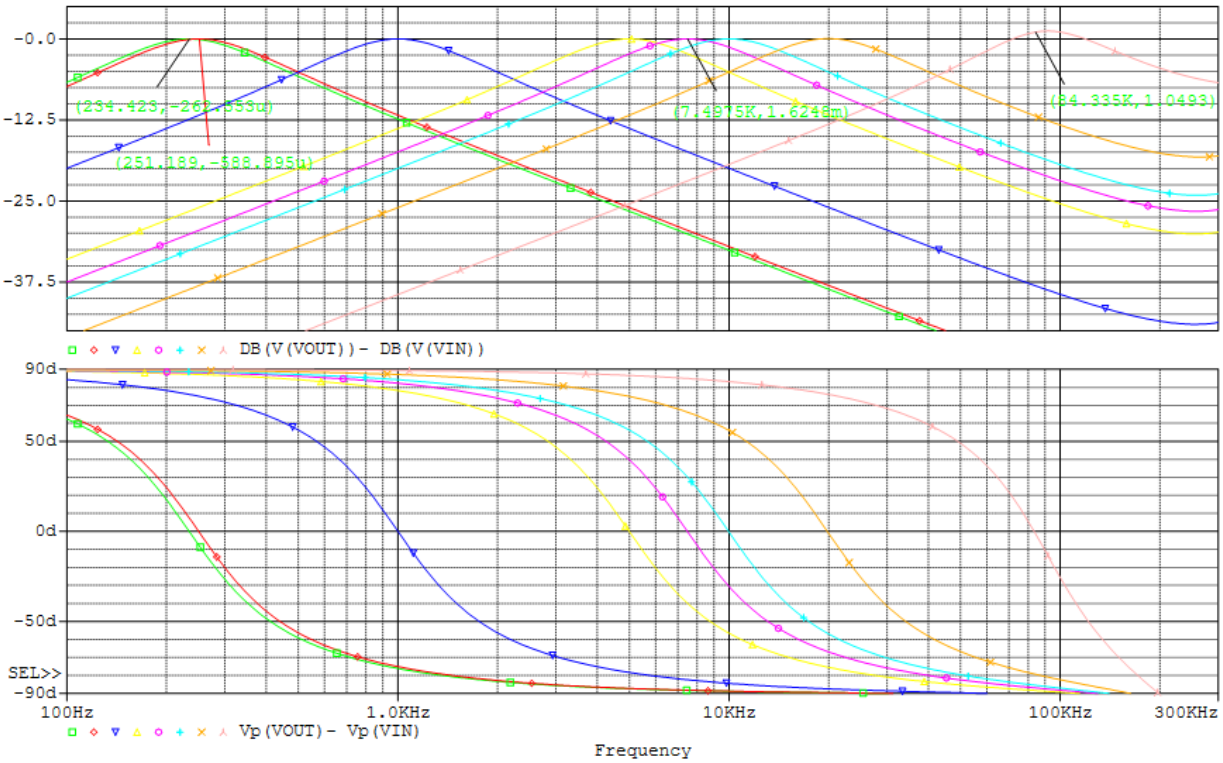
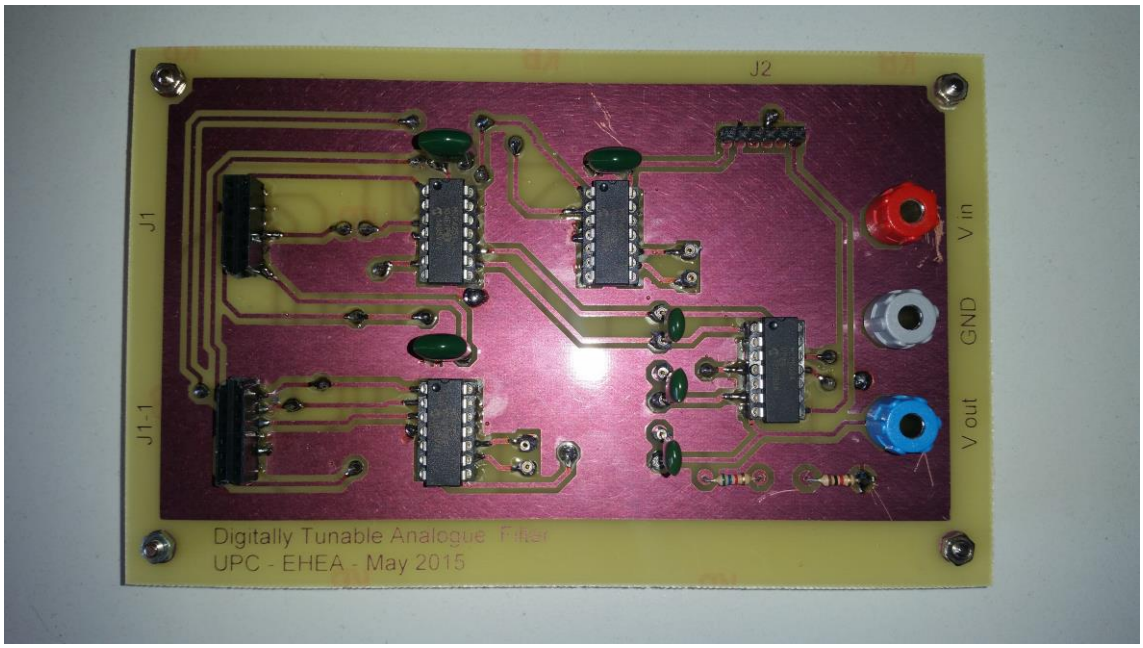


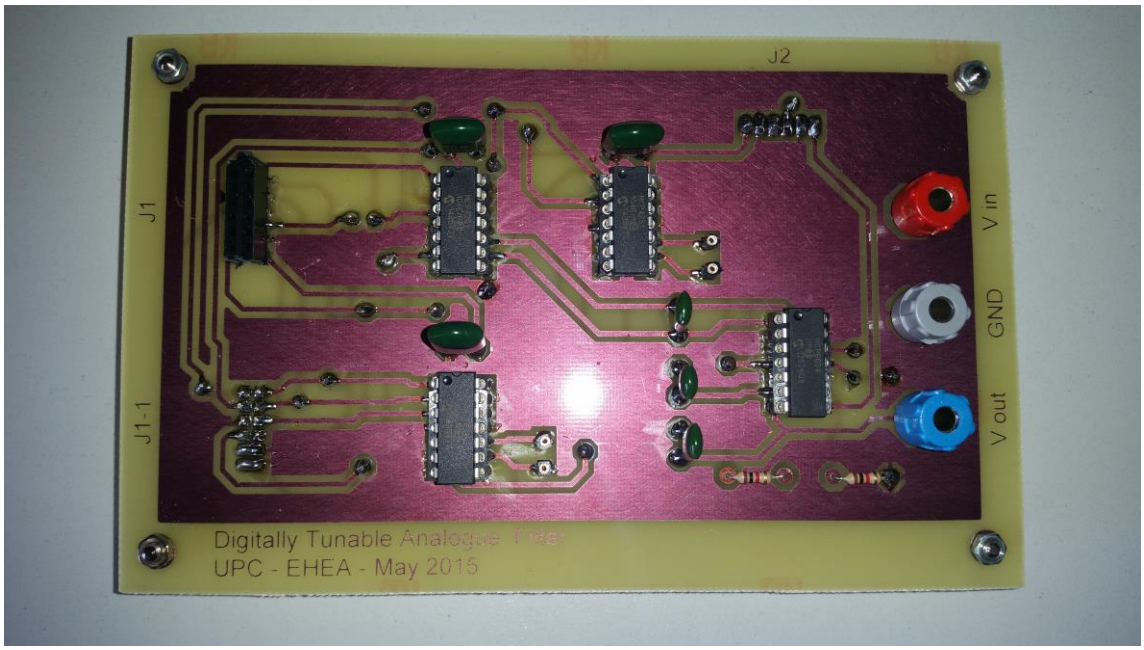
Figure 58. DTAF Bode plot simulation

4.2. Set up

Next are shown the fabricated PCBs after mounting all components. Although both board are apparently equal the connectors J1-1 and J2 are not mounted in the Slave Filter. Also, even that the connector is not necessary it was necessary to solder junctions between layers. Also the resistors are different to obtain the different offset voltages.



a) Master Filter



b) Slave Filter

Figure 59. Fabricated and Mounted PCBs

The picture below shows the connections of the complete DTAF system. At left there is the Nexys4 board, in the middle the Master Filter board and at the right the Slave Filter board.

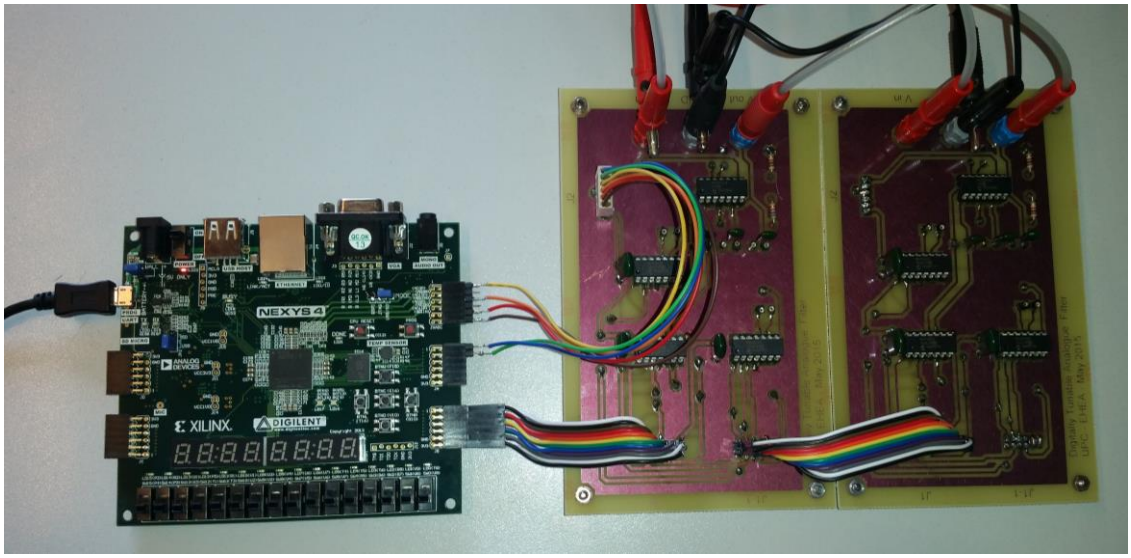


Figure 60. DTAF system

In order to obtain measurements of the DTAF input and output, the set-up shown in the next picture was used. There are two function waveform generators (RIGOL, DG5101) one to source the Master reference signal, and the other one to source the Slave input signal. Both generated signals are also connected to the oscilloscope (RIGOL, DS1064B) together with the outputs of each filter board to obtain phase and voltage measurements. Finally another oscilloscope (AD INSTRUMENTS, DS2202A) is used to obtain FFT waveforms. As the whole system is powered directly from the FPGA Platform no laboratory power supply is used. Oscilloscope connections are done with banana to BNC coaxial cable, except to obtain the SPI waveform where passive probes (AD INSTRUMENTS, RP1100) were used.

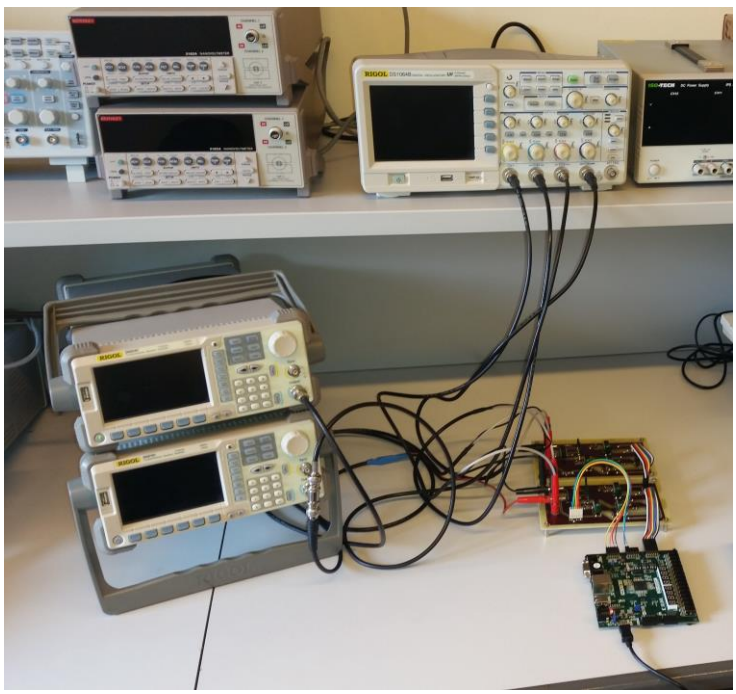
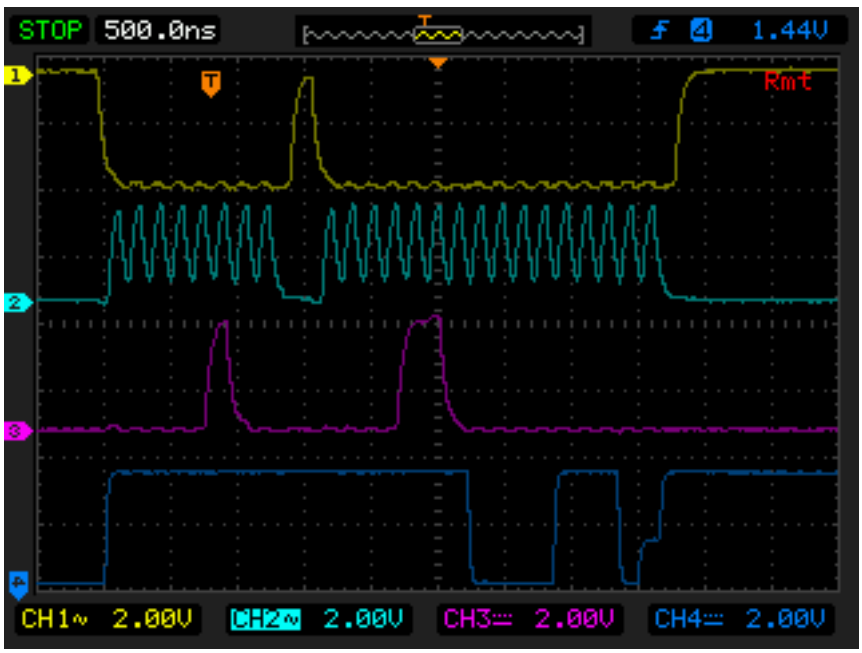


Figure 61. Measurement Set-Up

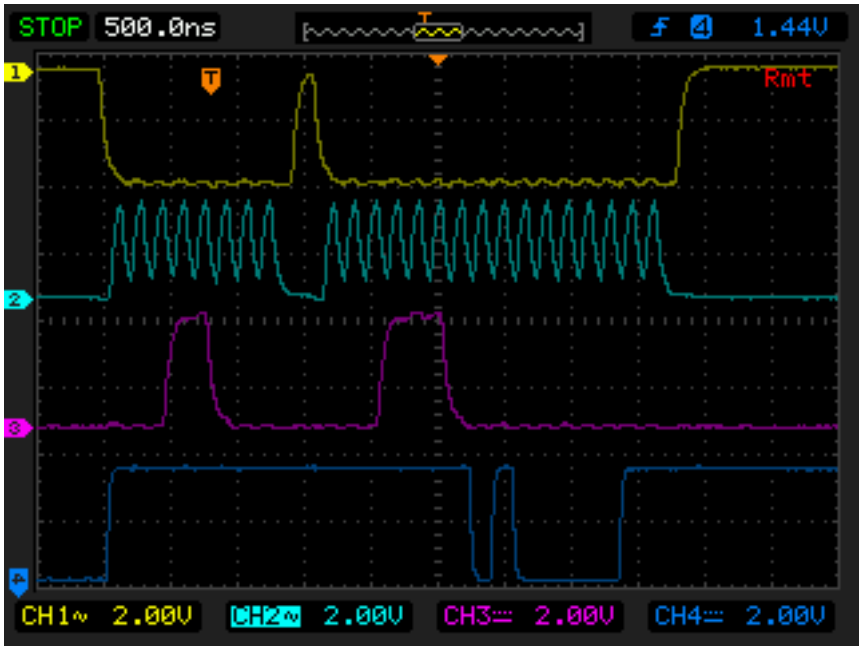
4.3. SPI Waveforms

As detailed in chapter 3.4.2, the digital potentiometers must receive specific SPI commands in order to modify their resistor value according to the control actions. As in early project steps a double potentiometer configuration was used, next *Figure 62* shows all the possible commands implemented within the control action in double potentiometer operation. The different channels are: CH1 is $\overline{CS}$, CH2 is SCK, CH3 is MOSI and CH4 is SDO.

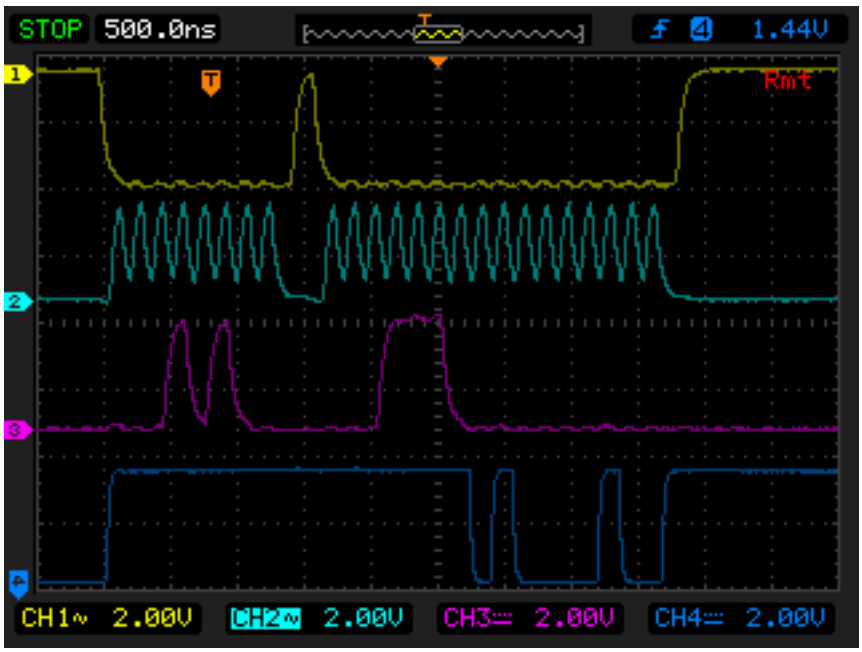
a) Decrement and Read on Wiper 0



b) Increment and Read on Wiper 0



c) Decrement and Read on Wiper 1



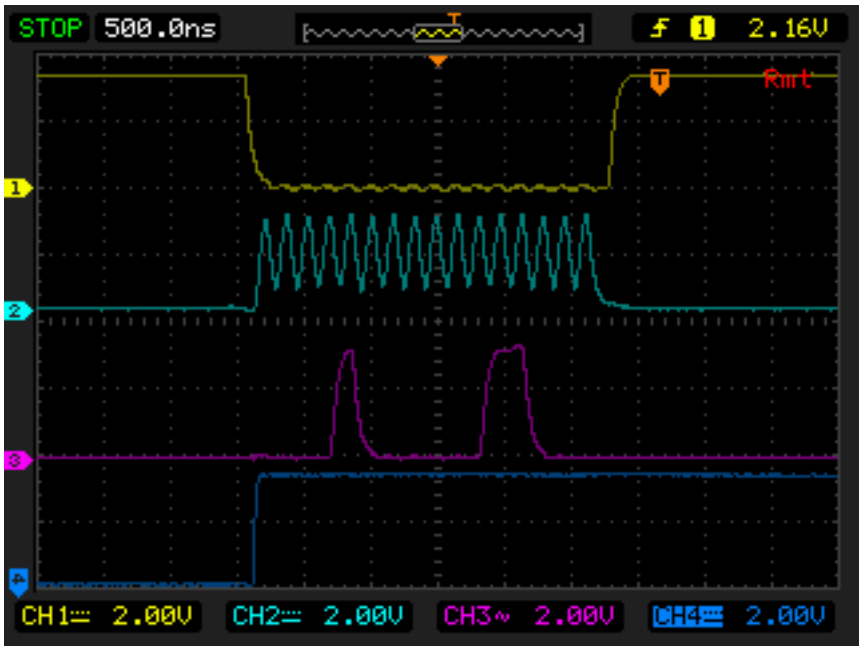
d) Increment and Read on Wiper 1

Figure 62. Double Potentiometer operation SPI Commands

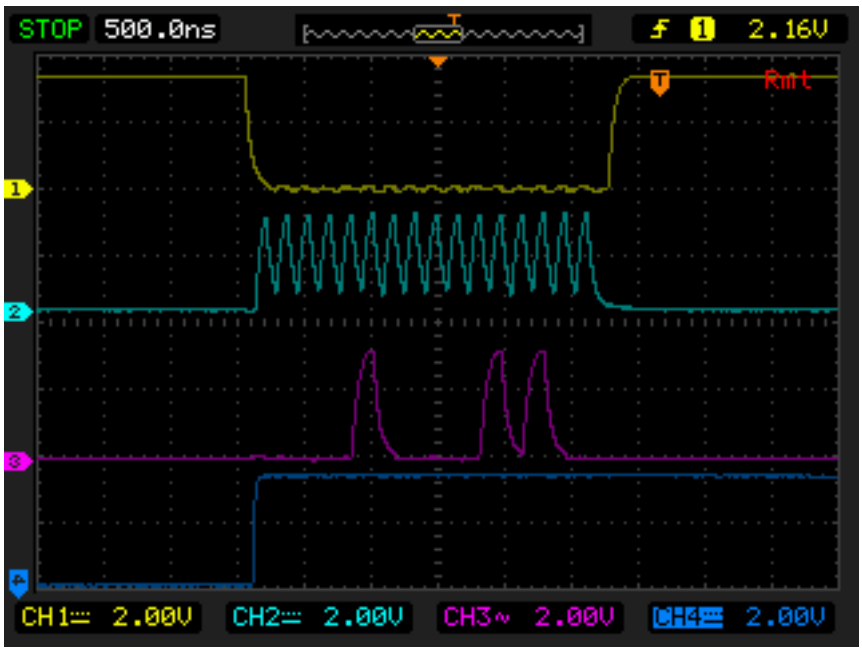
Notice that in double potentiometer operation the commands are not addressed to both potentiometers at the same time. So the commands actuate on one potentiometer or the other depending on the process explained in chapter 3.4.1.3.

In addition, having the SDO signal at high while increment or decrement commands are performed means that the address or combination sent to the Potentiometer are valid. Knowing this behaviour a process to detect communication errors could have been implemented, but was not considered a priority for the project development.

On the other hand, there is the Single Potentiometer operation SPI commands which are the used in the final DTAF version as 100 kΩ potentiometers were used. In this case the read command is not performed, so the only commands used are continuous increment or decrement. As are continuous operations there is no need to turn back the $\overline{CS}$ to high between commands. Hence it reduces the time to send both SPI commands. Now both wipers registers receive the same command at each time, then the potentiometers are either incremented or decremented depending on the control action.



c) Decrement wipers 0 and 1



d) Increment wipers 0 and 1

Figure 63. Single Potentiometer operation SPI Commands

4.4. Functionality Analysis

In order to verify the DTAF functionality different set of bode curves are plotted, with experimental results, see *Figure 64* below. The values have been obtained in the laboratory, manually, with the RIGOL DS1064B oscilloscope. The Master reference voltage is set to an amplitude of 700 mVpp with an offset of 500 mVdc, while the Slave input signal is set to 1,414 Vpp with an offset of 1,65 Vdc. In order to obtain the output amplitude and the phase sample points, the Master reference frequency is used first to tune the band-pass filter center frequency. Then to obtain the graph below it has been tuned respectively to 250 Hz, 1 kHz, 5 kHz, 7,5 kHz and 10 kHz. Finally the Slave input signal frequency is swept from 40 Hz to 50 kHz.

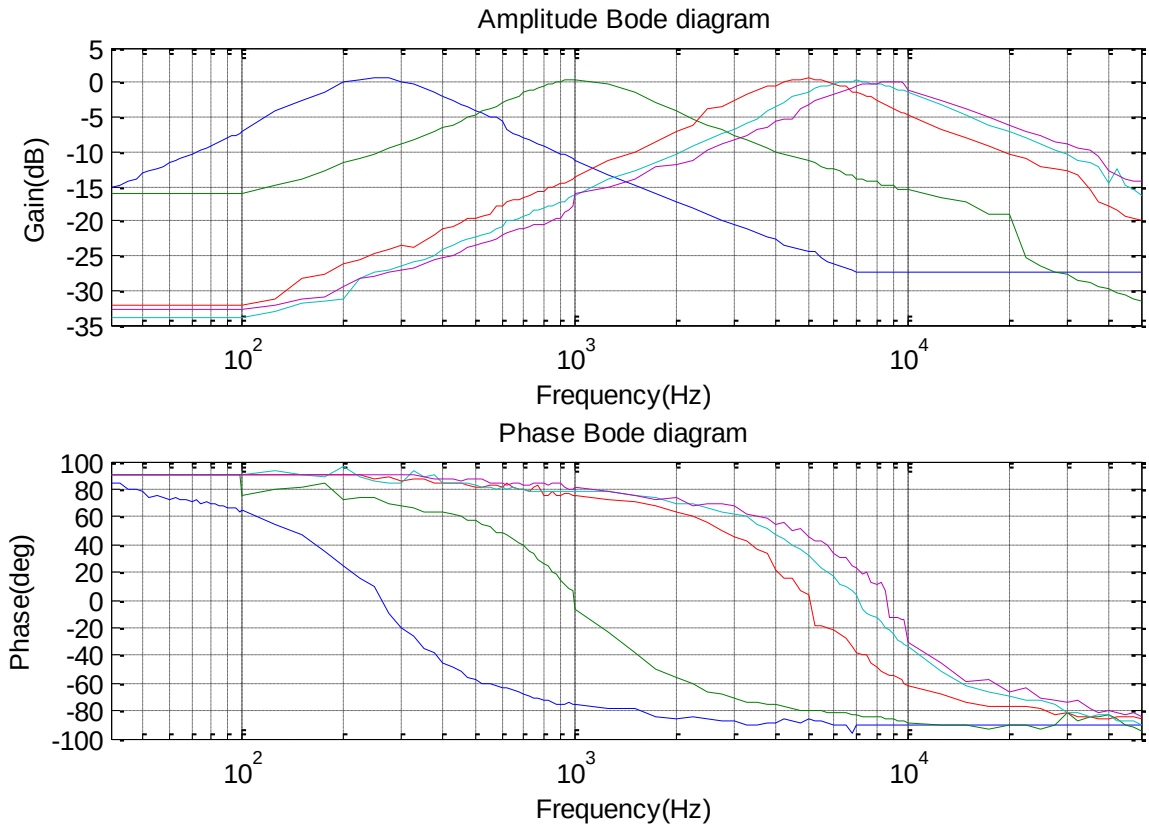


Figure 64. DTAF Bode plot experimental results

The result shows how the filter is able to adapt the center frequency for more than a decade. With a minimum frequency of 250 Hz up to a 10 kHz. Notice that the 10 kHz curve, actually is not correctly centred to 10 kHz. That is why at frequencies over 9-10 kHz the filter potentiometers almost reach their minimum resistance, and the control has less steps to adapt.

In addition the next *Figure 65*, shows experimental results of the variation of the digital potentiometers resistance over the frequency while both control loops are operating within the DTAF. Although the curves are almost equal, notice that this are instant values, because actually the resistance is continuously changing due to the loops control action.

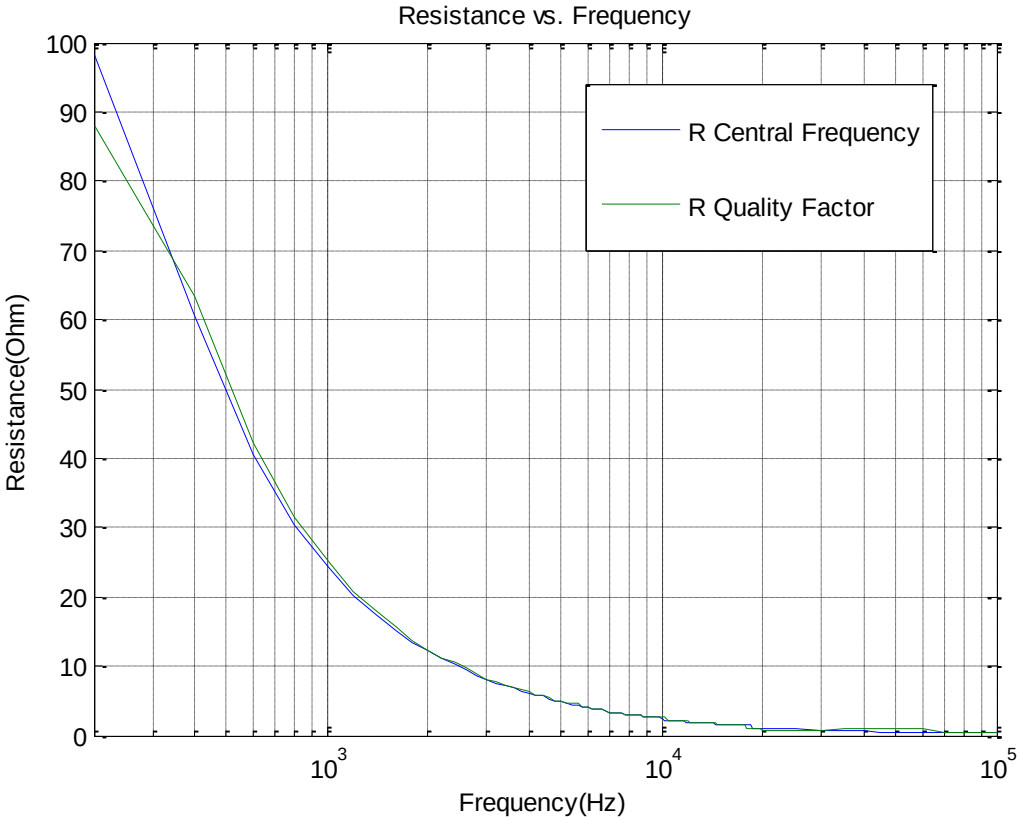


Figure 65. Resistance vs. Frequency from experimental results

4.5. Spectral Analysis

In order to quantify how much distortion adds the DTAF system to the Slave input signal the DTAF output THD is measured. To obtain the THD measurement several oscilloscope (DS2202A) FFT screenshots are done. The captures done are for different Slave input signal voltage amplitudes and for the same five different frequencies used in the bode plot. Next, *Figure 66*, there are two screenshot examples of the captured FFTs.

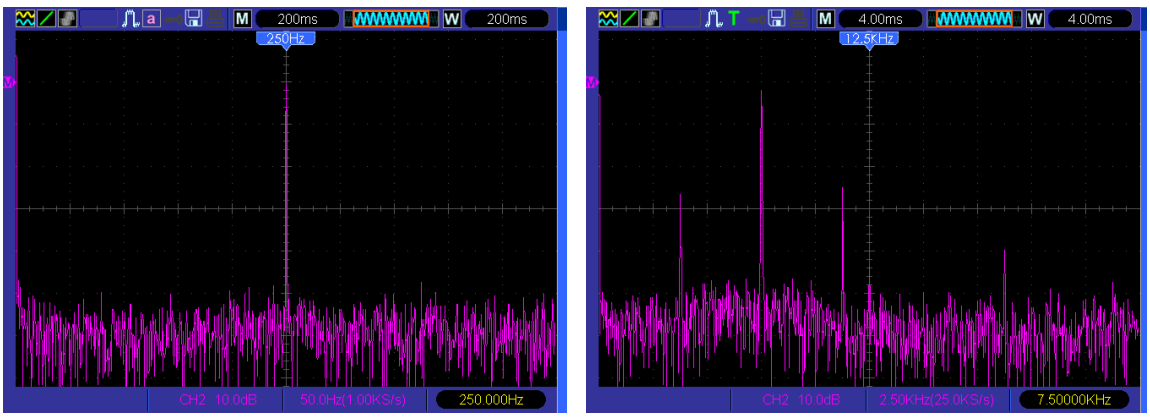


Figure 66. DTAF output FFT screenshots for V_{in} (slave) 1,414 Vpp

The formula used to calculate the THD is shown below. It consists on dividing the Vrms amplitude of several harmonics to the fundamental Vrms amplitude.

$$THD = \frac{\sqrt{V_2^2+V_3^2+\cdots+V_n^2}}{V_1} \quad [3]$$

The next *Figure 67* resumes the different THD values obtained. This measurement is done also with the CTF circuit by changing the resistors values manually to adapt the central frequency. The graph shows that the THD is always under 2% below 7 kHz, but over that the THD rises considerably especially for the 2,5 Vpp case.

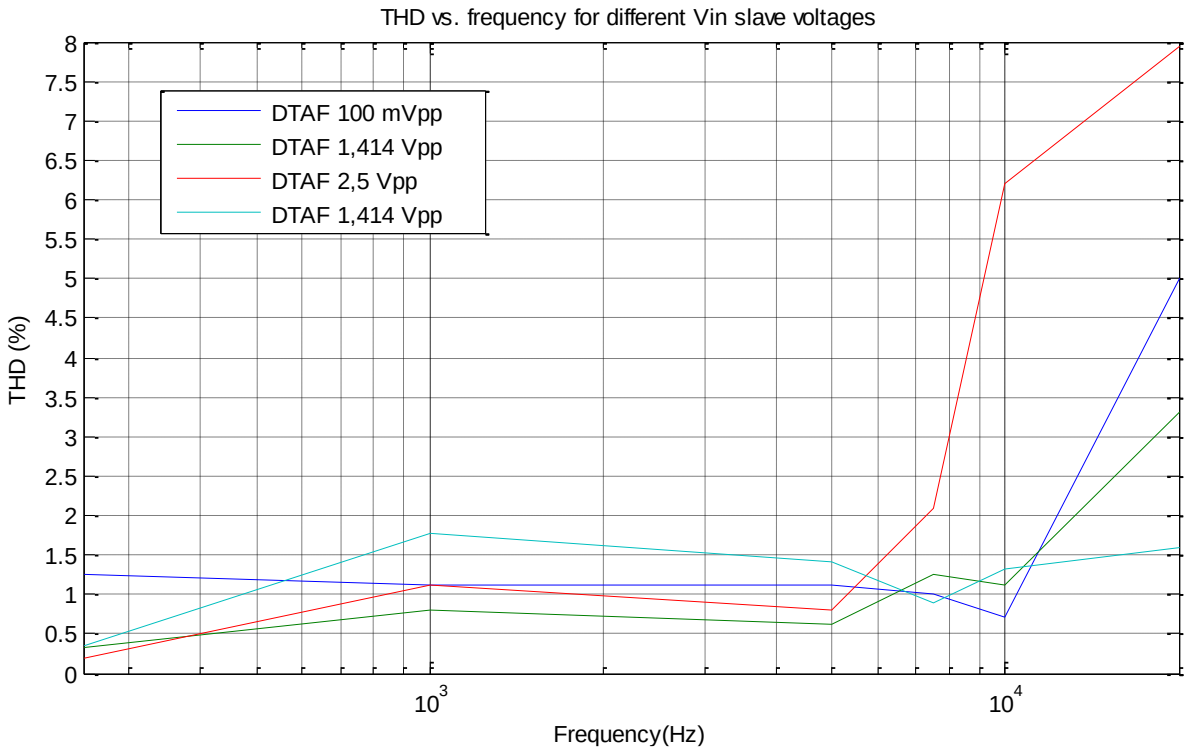


Figure 67. THD vs frequency for different slave voltages experimental results

5. Cost Estimation

As the fabricated system is a prototype and there is no intention to launch a product commercialization plan, just a cost estimation of its fabrication is presented.

Component	Quantity	Reference	Value	Farnell Code	Unit Price €	Total €
Resistor	3	R1 Slave, R2 Slave, R2 Master	1k	1652663	0,167	0,501
Resistor	1	R1 Master	5k6	1565324	0,032	0,032
Capacitor	6	C1-C3	6,8 nF	9752943	0,174	1,044
Capacitor	6	C4-C6	100 nF	2395773	0,030	0,180
MCP6024	2	U1	Quad	1627199	1,940	3,880
MCP4521	6	U2-U4	100k	1840692	0,964	5,784
Connector	3	J1 Master, J1-1 Master, J1 Slave	2x6 Through	1803667	1,150	3,45
Connector	1	J2 Master	1x6 Thorugh	1022255	0,138	0,138
Connector	2	J3	Banana	1698951	0,581	1,162
Connector	2	J4	Banana	1698952	0,661	1,322
Connector	2	J5	Banana	1698950	0,627	1,254
Socket	8		14-DIP	1077311	0,798	6,384
Jumper Cable	3 packs of 10		M-M	2452749	2,04	6,12
Photoresisitve Board	2			(RS) 159-6108	5,3	10,6
			TOTAL (MASTER and SLAVE Boards)			**41,851€**
NEXYS4 ARTIX-7 FPGA Board	1			Digilent		288,51 or 161,38 (academic)
TOTAL DTAF		330,361€	**TOTAL DTAF (academic)**			203,231€

6. Conclusions and future development

A DTAF system with central frequency and quality factor FPGA based control loops has been implemented using discrete components. In the results chapter have been demonstrated with real measurements that the filter operation works as expected, and according to the simulation.

On one hand the central frequency tuning capability have been completely implemented since it is possible to change the center frequency of the band-pass filter by means of modifying the frequency of the external input signal at the master filter. Even the goal of having a tunability frequency range wider than a decade have been achieved. Notice that such frequency range have been set to start at hundreds of Hertz frequencies to focus on the more audible part of the audio band.

But on the other hand the quality factor tuning capability is not implemented, since the control loop implemented cannot be configured as it is only intended to always keep a quality factor of Q = 1. That is why during the project development major of the time was dedicated to the integration of the whole system. Anyway, the DTAF can operate for a range amplitude input voltages of 100 mVpp up to 2,5 Vpp while keeping the central frequency tunability and a constant quality factor of Q = 1.

Another point is that the filter implemented is intended to be a second order filter while either the simulation and the real measurements exhibit a typical curve for a first-order filter since the slope in the bode is only 20 dB/decade instead of 40 dB/decade.

Although the DTAF system is completed and the functionality has been proved, the only input signal applied for such verification has been a pure sinusoidal signal. Then, the next step would be to pass a complete test of the whole DTAF system against other types of waveforms and of course audio signals.

In order to obtain the bode plot data in a quick way would be interesting to implement an automatic testbench. Such testbench could remotely configure one waveform generator to set the master reference signal at the center frequency under test, and then configure the other waveform generator to vary the slave input signal. For each slave input frequency set the testbench should read the oscilloscope information of the output signal of the DTAF. So, the data of a complete bode diagram could be automatically be saved into a file for a later processing and plotting.

Also would be interesting to measure the real THD of the system with a THD analyser instrument in order to directly evaluate the distortion of the output signal.

In addition, a study of the control loops applied and the corresponding improvement may reduce the THD of DTAF.

Finally in order to increment the frequency operation range it may be considered to introduce a sort of variable capacitor, instead of fixed capacitors. In such a way the actual range operation allowed by the digital potentiometers may be displaced to higher or lower frequencies. Such variable capacitors may be implemented by means of varicap diode, MEMS capacitors or switched FET.

Bibliography

[1] Martinez-Garcia, H.; Cosp-Vilella, J.; Manzanares-Brotons, M., "Observation of chaotic behavior in automatic tuning loops for continuous-time filters," Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on , vol., no., pp.742,745, 4-7 Aug. 2013

[2] Richard C. Cabot. Tunable bandpass filter system and filtering method. https://www.google.com/patents/US5136267 (accessed June 5, 2015)

[3] Odame, K.M.; Hasler, P., "An Adaptive Quality-Factor Bandpass Filter," Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on , vol., no., pp.3295,3298, 27-30 May 2007

[4] Un-Ku Moon; Bang-Sup Song, "Design of a low-distortion 22-kHz fifth-order Bessel filter," Solid-State Circuits, IEEE Journal of , vol.28, no.12, pp.1254,1264, Dec 1993

[5] Osa, J.I.; Carlosena, A.; Lopez-Martin, A.J., "MOSFET-C filter with on-chip tuning and wide programming range," Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on , vol.48, no.10, pp.944,951, Oct 2001

[6] Martinez, H.; Vidal, E.; Alarcon, E.; Poveda, A., "Design and implementation of an MRC-C TQE filter with on-chip automatic tuning," Circuits and Systems, 2001. MWSCAS 2001. Proceedings of the 44th IEEE 2001 Midwest Symposium on , vol.1, no., pp.196,199 vol.1, 2001

[7] Rahul Prakash, Eugenio Mejia. Digitally Tunable MDAC-Based State Variable Filter TI Designs – Precision: Verified Design. TIDU543-October 2014.

http://www.ti.com/lit/ug/tidu543/tidu543.pdf (accessed June 15, 2015)

[8] Low-power Widely Tunable Gm-C Filter with an Adaptive DC-blocking, Triode-biased MOSFET Transconductor.

[9] Soliman, Eman A.; Mahmoud, Soliman A., "Tunable second order bi-quad filter using current conveyor based FPAA," Microelectronics (ICM), 2011 International Conference on , vol., no., pp.1,4, 19-22 Dec. 2011

[10] Khan, M.Z.; Ansari, M.S.; Siddiqui, M.J., "Digitally programmable second-order current mode continuous-time filters," Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International , vol., no., pp.1,4, 27-30 April 2013

[11] Becker, J.; Henrici, F.; Trendelenburg, S.; Ortmanns, M.; Manoli, Y., "A Field-Programmable Analog Array of 55 Digitally Tunable OTAs in a Hexagonal Lattice," Solid-State Circuits, IEEE Journal of , vol.43, no.12, pp.2759,2768, Dec. 2008

[12] MCP6024 Rail-to-Rail Input/Output, 10 MHz Op Amp Datasheet

http://ww1.microchip.com/downloads/en/DeviceDoc/21685d.pdf (accessed Feb 25, 2015)

[13] Spartan-3A/3AN FPGA Starter Kit Board Design Examples.

http://www.xilinx.com/products/boards/s3astarter/reference_designs.htm (accessed May 10, 2015)

[14] Nexys™4 Artix-7 FPGA Board Support Documents

http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4 (accessed June 2, 2015)

[15] Spartan-3A/3AN FPGA Starter Kit Board User Guide. UG334 (v1.1) June 19, 2008

http://www.xilinx.com/support/documentation/boards_and_kits/ug334.pdf (accessed May 10, 2015)

[16] FX2 Breadboard Support Documents.

http://www.digilentinc.com/Products/Detail.cfm?Prod=FX2BB (accessed May 10, 2015)

[17] 7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide. UG480 (v1.7) May 19, 2015.

http://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf (accessed June 3, 2015)

[18] LogiCORE IP XADC Wizard v2.2 User Guide. UG772 July 25, 2012.

http://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v2_2/ug772_xadc_wiz.pdf (accessed June 3, 2015)

[19] Nexys4™ FPGA Board Reference Manual. Nexys4 rev. B; Revised November 19, 2013.

http://www.digilentinc.com/Data/Products/NEXYS4/Nexys4_RM_VB2_Final_5.pdf (accessed June 2, 2015)

[20] MCP4251 7/8-Bit Single/Dual SPI Digital POT with Volatile Memory. DS22060B.

http://ww1.microchip.com/downloads/en/DeviceDoc/22060b.pdf (accessed Feb 25, 2015)

Appendices

VHDL Code Implemented:

- DTAF_Controller.vhd
- phase_detector_comp.vhd
- phase_ctrl_comp.vhd
- amp_ctrl_comp.vhd
- xadc_wiz_v2_1.vhd
- debouncer.vhd
- Nexys4_Master.ucf

```
--------------------------------------------------------------------------------
-- University: UPC - Universitat Politècnica de Catalunya
-- Engineer: Joan Francesc Serra Bou
--
-- Create Date:    15:28:36 28/04/2015
-- Design Name:    Digitally Tunabale Analogue Filter controller
-- Module Name:    Top level entity
-- Project Name:   Digitally Tunabale Analogue Filter
-- Target Devices: XC7A100T-1CSG324C - Nexys4 - Artix-7
-- Tool versions:  ISE Design Suite 14.2
-- Description: DTAF Top-level entity also including Amplitude detector logic
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity DTAF_controller is
    Port ( btnCpuReset  : in  STD_LOGIC;
           BTN          : in  STD_LOGIC_VECTOR (4 downto 0);
           CLK          : in  STD_LOGIC;
           vauxp2       : in STD_LOGIC; -- Master CTF Output
           vauxn2       : in STD_LOGIC;
           vauxp3       : in STD_LOGIC; -- Master CTF Input
           vauxn3       : in STD_LOGIC;
           vp_in        : in STD_LOGIC;
           vn_in        : in STD_LOGIC;

           -- Filter signals for phase detection
           BPF_Vin      : in std_logic;
           BPF_Vout     : in std_logic;

           -- POT signals for phase matching
           POT_CS        :   out std_logic;  -- POT chipselect
           POT_SCK       :   out std_logic;  -- POT SCK
           POT_MOSI      :   out std_logic;  -- POT SDI
           POT_SDO       :   in  std_logic;  -- POT SDO  // POT SHDN is wired on PCB
```

```
            -- POT signals for amplitude matching
            POT_amp_CS    :  out std_logic;  -- POT amp chipselect
            POT_amp_SCK   :  out std_logic;  -- POT amp SCK
            POT_amp_MOSI  :  out std_logic;  -- POT amp SDI
            POT_amp_SDO   :  in  std_logic   -- POT SDO  // POT SHDN is wired on PCB
            );
end DTAF_controller;

architecture Behavioral of DTAF_controller is

---- Debounce component ----
component debouncer
Generic(
        DEBNC_CLOCKS : integer;
        PORT_WIDTH : integer);
Port(
        SIGNAL_I : in std_logic_vector(5 downto 0);
        CLK_I : in std_logic;
        SIGNAL_O : out std_logic_vector(5 downto 0)
        );
end component;

---- XADC Core component ----
component xadc_wiz_v2_1
    port (
          DADDR_IN            : in  STD_LOGIC_VECTOR (6 downto 0);
         -- Address bus for the DRP (dynamic reconfiguration port)
          DCLK_IN             : in  STD_LOGIC;                        -- DRP Clock input
          DEN_IN              : in  STD_LOGIC;                        -- DRP Enable Signal
          DI_IN               : in  STD_LOGIC_VECTOR (15 downto 0);   -- DRP Input data bus
          DWE_IN              : in  STD_LOGIC;                        -- DRP Write Enable
          RESET_IN            : in  STD_LOGIC;             -- Reset signal for the System Monitor control logic
          VAUXP2              : in  STD_LOGIC;                        -- Auxiliary Channel 2
          VAUXN2              : in  STD_LOGIC;
          VAUXP3              : in  STD_LOGIC;                        -- Auxiliary Channel 3
          VAUXN3              : in  STD_LOGIC;
          VAUXP10             : in  STD_LOGIC;                        -- Auxiliary Channel 10
          VAUXN10             : in  STD_LOGIC;
```

```
        VAUXP11              : in  STD_LOGIC;                          -- Auxiliary Channel 11
        VAUXN11              : in  STD_LOGIC;
        BUSY_OUT             : out  STD_LOGIC;                         -- ADC Busy signal
        CHANNEL_OUT          : out  STD_LOGIC_VECTOR (4 downto 0);     -- Channel Selection Outputs
        DO_OUT               : out  STD_LOGIC_VECTOR (15 downto 0);    -- DRP Output data bus
        DRDY_OUT             : out  STD_LOGIC;                         -- DRP Data ready signal
        EOC_OUT              : out  STD_LOGIC;                         -- End of Conversion Signal
        EOS_OUT              : out  STD_LOGIC;                         -- End of Sequence Signal
        ALARM_OUT            : out STD_LOGIC;                          -- OR'ed output of all the Alarms
        VP_IN                : in  STD_LOGIC;                          -- Dedicated Analog Input Pair
        VN_IN                : in  STD_LOGIC
);
end component;

---- Phase ctrl component ----
component phase_ctrl_comp
port (
       -- All component signals are ended with "_c" unless global ones
        clk                     : in   std_logic;
        reset                   : in   std_logic;
        enable_phase_ctrl_c     : in   std_logic;
        POT_CS_c                : out  std_logic;  -- POT chipselect
        POT_SCK_c               : out  std_logic;  -- POT SCK
        POT_MOSI_c              : out  std_logic;  -- POT SDI
        POT_SDO_c               : in   std_logic;  -- POT SDO  // POT SHDN is wired on PCB
        Freq_limit              : in   std_logic;
        BPF_Vin_First_c         : in   std_logic;
        BPF_Vout_First_c        : in   std_logic
);
end component;

---- Phase detector component ----
component phase_detector_comp
port (
        -- All component signals are ended with "_c" unless clk and reset
        clk                          : in   std_logic;
        reset                        : in   std_logic;
        enable_phase_detector_c      : in   std_logic;
```

```
        BPF_Vin_c                   : in   std_logic;
        BPF_Vout_c                  : in   std_logic;
        BPF_Vin_First_c             : out  std_logic;
        BPF_Vout_First_c            : out  std_logic
);
end component;

---- Amp ctrl component ----
component amp_ctrl_comp
port (
        -- All component signals are ended with "_c" unless global ones
        clk                         : in   std_logic;
        reset                       : in   std_logic;
        enable_amp_ctrl_c           : in   std_logic;
        POT_amp_CS_c                : out  std_logic;   -- POT amp chipselect
        POT_amp_SCK_c               : out  std_logic;   -- POT amp SCK
        POT_amp_MOSI_c              : out  std_logic;   -- POT amp SDI
        POT_amp_SDO_c               : in   std_logic;   -- POT amp SDO
        Freq_limit                  : in   std_logic;
        AMP_Vin_Over_c              : in   std_logic;
        AMP_Vout_Over_c             : in   std_logic
);
end component;

 ---- XADC core signals ----
  signal DADDR_IN                          : std_logic_vector(6 downto 0);
  signal CHANNEL_READ                      : std_logic_vector(4 downto 0);
  signal DEN_IN                            : std_logic;
  signal DWE_IN                            : std_logic;
  signal DI_IN                             : std_logic_vector(15 downto 0);
  signal DO_OUT                            : std_logic_vector(15 downto 0);
  signal XADC_CH2, Vout_CH                 : std_logic_vector(11 downto 0);
  signal XADC_CH3, Vin_CH                  : std_logic_vector(11 downto 0);
  signal DRDY_OUT                          : std_logic;
  signal DRDY_OUT_falling_edge, D1, D2     : std_logic;
  signal RESET_IN                          : std_logic;
  signal FLOAT_VCCAUX_ALARM                : std_logic;
  signal FLOAT_VCCINT_ALARM                : std_logic;
```

```
  signal FLOAT_USER_TEMP_ALARM    : std_logic;
  signal ALARM_OUT                : std_logic;
  signal BUSY_OUT                 : std_logic;
  signal EOC_OUT                  : std_logic;
  signal EOS_OUT                  : std_logic;
  signal MEASURE_DONE             : std_logic;

---- DTAF signals ----
signal BPF_Vin_First, BPF_Vout_First : std_logic;
signal AMP_Vin_Over, AMP_Vout_Over, neg_sample_Vin, neg_sample_Vout : std_logic;

signal Vin_CH_peak,  Vin_CH_stored:  std_logic_vector(11 downto 0) := (others => '0');
signal Vout_CH_peak, Vout_CH_stored: std_logic_vector(11 downto 0) := (others => '0');

-- Frequency limitation
signal Freq_limit : std_logic;

-- Peak Detector signals
signal E1, E2, Peak_detected : std_logic;

-- Reset signal
signal reset : std_logic;

-- cpu reset concatenation four push buttons
signal reset_and_BTN : std_logic_vector(5 downto 0);

-- Main divider counter signal
signal counter_3: std_logic_vector(2 downto 0);

-- Component enable signals
signal enable_phase_ctrl, enable_phase_detector, enable_amp_ctrl, enable_amp_detector : std_logic;
--------------
---- Debouncer signals ----
-- Used to determine when a button press has occured
signal btnReg : std_logic_vector (3 downto 0) := "0000";
signal btnDetect : std_logic;

-- Debounced btn signals used to prevent single button presses
```

```
-- from being interpreted as multiple button presses.
signal btnDeBnc : std_logic_vector(5 downto 0);

signal clk_cntr_reg : std_logic_vector (4 downto 0) := (others=>'0');

begin

-----------------------------------------------------------
------              Button Control                 -------
-----------------------------------------------------------
--Buttons are debounced and their rising edges are detected
--to trigger UART messages

-- Cpu reset button is concatenated with the five push buttons
reset_and_BTN <= btnCpuReset & BTN;

-- Assignment of debounced reset
reset <= not btnDeBnc(5);

--Debounces btn signals
Inst_btn_debounce: debouncer
    generic map(
        DEBNC_CLOCKS => (2**16),
        PORT_WIDTH => 6)
    port map(
        SIGNAL_I => reset_and_BTN,
        CLK_I => CLK,
        SIGNAL_O => btnDeBnc
    );

--Registers the debounced button signals, for edge detection.
btn_reg_process : process (CLK)
begin
    if (rising_edge(CLK)) then
        btnReg <= btnDeBnc(3 downto 0);
    end if;
end process;
```

```
--btnDetect goes high for a single clock cycle when a btn press is
--detected. This triggers a UART message to begin being sent.
btnDetect <= '1' when ((btnReg(0)='0' and btnDeBnc(0)='1') or
                                (btnReg(1)='0' and btnDeBnc(1)='1') or
                                (btnReg(2)='0' and btnDeBnc(2)='1') or
                                (btnReg(3)='0' and btnDeBnc(3)='1')  ) else
                  '0';

-----------------------------------------------------------
------              XADC Controller                -------
-----------------------------------------------------------
 xadc_wiz_inst : xadc_wiz_v2_1
port map (
      DADDR_IN(6 downto 0)     => DADDR_IN(6 downto 0),
      DCLK_IN                  => CLK,--DCLK_IN,
      DEN_IN                   => DEN_IN,
      DI_IN(15 downto 0)       => DI_IN(15 downto 0),
      DWE_IN                   => DWE_IN,
      RESET_IN                 => RESET_IN,
      VAUXP2                     => VAUXP2,
      VAUXN2                     => '0',
      VAUXP3                     => VAUXP3,
      VAUXN3                     => '0',
      VAUXP10                    => '0',
      VAUXN10                    => '0',
      VAUXP11                    => '0',
      VAUXN11                    => '0',
      BUSY_OUT                 => BUSY_OUT,
      CHANNEL_OUT(4 downto 0)  => CHANNEL_READ(4 downto 0),
      DO_OUT(15 downto 0)      => DO_OUT(15 downto 0),
      DRDY_OUT                 => DRDY_OUT,
      EOC_OUT                  => EOC_OUT,
      EOS_OUT                  => EOS_OUT,
      ALARM_OUT                => ALARM_OUT, -- OR'ed output of all the Alarms
      VP_IN                    =>  '0',  -- NOT used
      VN_IN                    =>  '0'   -- NOT used
      );
```

```
DADDR_IN <= "00" & CHANNEL_READ;
DI_IN <= "0000000000000000";     -- Spare data, only reading from XADC
DWE_IN <= '0';                   -- Null, only reading from XADC
DEN_IN <= EOC_OUT;               -- For continuous acquisition
RESET_IN <= '0';

  DRDY_edge_detection: process(clk)
  begin
      if (clk'event and clk='1') then
          if (reset = '1') then
             D1 <= '0';
             D2 <= '0';
          else
             D1 <= DRDY_OUT;
             D2 <= D1;
          end if;
      end if;
  end process;

  DRDY_OUT_falling_edge   <= D2 and (not D1);

 XADC_Read: process(clk)
  begin
      if (clk'event and clk='1') then
          if (DRDY_OUT_falling_edge = '1' ) then
              if ( CHANNEL_READ = "10010" ) then
                  XADC_CH2 <= DO_OUT(15 downto 4);
              else -- ( CHANNEL_READ = "10011" )
                  XADC_CH3 <= DO_OUT(15 downto 4);
              end if;
          end if;
      end if;
  end process;
```

```
-------------------------------------------------------
------          Peak Amplitude Detector          -------
-------------------------------------------------------
Vin_CH  <= XADC_CH3;
Vout_CH <= XADC_CH2;

  -- Peak detector Vin_CH
  Peak_detector_Vin: process(clk)
  begin
    if (clk'event and clk='1') then
        if (reset = '1' ) then
            Vin_CH_peak <= (others => '0');
            Vin_CH_stored <= (others => '0');
            neg_sample_Vin <= '0';
            -- Samples when sinus signal is over 312.320 mVdc (x"500")
            elsif ( Vin_CH > x"500") then -- ideally x"800" for 499.712 mVdc
                neg_sample_Vin <= '0';
                -- Each new sample bigger than the last one is stored.
                if ( Vin_CH > Vin_CH_stored ) then
                    Vin_CH_stored <= Vin_CH;
                end if;
            -- Samples when sinus signal is under 312.076 mVdc (x"4FF")
            -- Then the max amplitude sample stored is the new positive peak value
            elsif ( Vin_CH <= x"4FF" and neg_sample_Vin = '0') then -- ideally x"7FF" for 499.468 mVdc
                neg_sample_Vin <= '1';
                Vin_CH_peak <= Vin_CH_stored;
                Vin_CH_stored <= (others => '0');
            else
                -- nothing
            end if;
        end if;
   end process;
```

```
 -- Peak detector Vout_CH
 Peak_detector_Vout: process(clk)
 begin
   if (clk'event and clk='1') then
       if (reset = '1' ) then
           Vout_CH_peak <= (others => '0');
           Vout_CH_stored <= (others => '0');
           neg_sample_Vout <= '0';
           -- Samples when sinus signal is over 312.320 mVdc (x"500")
           elsif ( Vout_CH > x"500") then -- ideally x"800" for 499.712 mVdc
               neg_sample_Vout <= '0';
               -- Each new sample bigger than the last one is stored.
               if ( Vout_CH > Vout_CH_stored ) then
                   Vout_CH_stored <= Vout_CH;
               end if;
           -- Samples when sinus signal is under 312.076 mVdc (x"4FF")
          -- Then the max amplitude sample stored is the new positive peak value
           elsif ( Vout_CH <= x"4FF" and neg_sample_Vout = '0') then -- ideally x"7FF" for 499.468 mVdc
               neg_sample_Vout <= '1';
               Vout_CH_peak <= Vout_CH_stored;
               Vout_CH_stored <= (others => '0');
           else
               -- nothing
           end if;
       end if;
  end process;

AMP_Vin_Over  <= '1' when Vout_CH_peak < Vin_CH_peak  else '0';
AMP_Vout_Over <= '1' when Vin_CH_peak  < Vout_CH_peak else '0';
```

```
-------------------------------------------------------
------        AMPLITUDE Potentiometer Control      -------
-------------------------------------------------------
-- Instantiation of amplitude controller
amp_ctrl : amp_ctrl_comp
port map (
            clk => clk,
            reset => reset,
            enable_amp_ctrl_c => enable_amp_ctrl,
            POT_amp_CS_c => POT_amp_CS,
            POT_amp_SCK_c => POT_amp_SCK,
            POT_amp_MOSI_c => POT_amp_MOSI,
            POT_amp_SDO_c => POT_amp_SDO,
            Freq_limit => Freq_limit,
            AMP_Vin_Over_c => AMP_Vin_Over,
            AMP_Vout_Over_c => AMP_Vout_Over
);
-- Amplitude control enable adapts to the input frequency thanks to the peak detection of input signal
enable_amp_ctrl <= '1' when (Peak_detected = '1') else '0';

    New_peak: process(clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                E1 <= '0';
                E2 <= '0';
            else
                E1 <= neg_sample_Vin;
                E2 <= E1;
            end if;
        end if;
    end process;
```

```
    Peak_detected   <= E2 and (not E1);



-------------------------------------------------------
------            PHASE Potentiometer Control       -------
-------------------------------------------------------
-- Instantiation of phase controller
phase_ctrl : phase_ctrl_comp
port map (
            clk => clk,
            reset => reset,
            enable_phase_ctrl_c => enable_phase_ctrl,
            POT_CS_c => POT_CS,
            POT_SCK_c => POT_SCK,
            POT_MOSI_c => POT_MOSI,
            POT_SDO_c => POT_SDO,
            Freq_limit => Freq_limit,
            BPF_Vin_First_c => BPF_Vin_First,
            BPF_Vout_First_c => BPF_Vout_First
);
-- Phase Control enable is triggered always that there is a phase mismatch
enable_phase_ctrl <= '1' when (BPF_Vin_First = '1' or BPF_Vout_First = '1') else '0';

-------------------------------------------------------
------              PHASE Detector Control         -------
-------------------------------------------------------
-- Instantiation of phase detector
phase_detector : phase_detector_comp
port map (
            clk => clk,
            reset => reset,
            enable_phase_detector_c => enable_phase_detector,
            BPF_Vin_c => BPF_Vin,
            BPF_Vout_c => BPF_Vout,
            BPF_Vin_First_c => BPF_Vin_First,
            BPF_Vout_First_c => BPF_Vout_First
```

```
);

-- Phase Detector enable is always at high. It is implemented to keep the same structure as other blocks
enable_phase_detector <= '1';

-------------------------------------------------------
------              Main Divider Counter          -------
-------------------------------------------------------
-- Main frequency divider counter, 100 MHZ divided by 8 to have 12,5 MHz signal
-- Which leads into an SPI_CLK at 6,25 MHz because is generated in two clock periods.
main_counter: process(clk)
begin
    if (clk'event and clk='1') then
        if (reset = '1' ) then
            counter_3 <= (others => '0');
            Freq_limit <= '0';
        else
            if (counter_3 = "111") then
                Freq_limit <= '1';
                counter_3 <= (others => '0');
            else
                Freq_limit <= '0';
                counter_3 <= counter_3 + '1';
            end if;
        end if;
    end if;
end process;

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- University: UPC - Universitat Politècnica de Catalunya
-- Engineer: Joan Francesc Serra Bou
--
-- Create Date:    15/03/2015
-- Design Name:    Digitally Tunabale Analogue Filter controller
-- Module Name:    phase_detector_comp - Behavioral
-- Project Name:   Digitally Tunabale Analogue Filter
-- Target Devices: XC7A100T-1CSG324C - Nexys4 - Artix-7
-- Tool versions:  ISE Design Suite 14.2
-- Description:    This component detects BPF_Vin_c and BPF_Vout_c phase mismatch.
--                 BPF_Vin_First_c and BPF_Vout_First_c are outputs with the mismatching information.
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity phase_detector_comp is
port (
-- All components signals are ended with "_c" unless clk and reset
        clk                         :   in   std_logic;
        reset                       :   in   std_logic;
        enable_phase_detector_c     :   in   std_logic;
        BPF_Vin_c                   :   in   std_logic;
        BPF_Vout_c                  :   in   std_logic;
        BPF_Vin_First_c             :   out  std_logic;
        BPF_Vout_First_c            :   out  std_logic
);
end phase_detector_comp;

architecture Behavioral of phase_detector_comp is



-- Phase detection signals
signal BPF_Vin_Aux, BPF_Vin_Sync, BPF_Vin_Sync_Aux, BPF_Vout_Aux, BPF_Vout_Sync, BPF_Vout_Sync_Aux  : std_logic;
signal BPF_Vin_Rising_edge, BPF_Vout_Rising_edge, BPF_Not_in_Phase  : std_logic;
```

```
begin

---- ---- PHASE MISMATCH DETECTOR ---- ----
-- Synchronization of external signals BPF_Vin and BPF_Vout
phase_det_process: process(clk)
begin
   if (clk'event and clk = '1') then
        if (reset = '1') then
            BPF_Vin_Aux <= '0';
            BPF_Vin_Sync <= '0';
            BPF_Vout_Aux <= '0';
            BPF_Vout_Sync <= '0';
            BPF_Vin_Sync_Aux <= '0';
            BPF_Vout_Sync_Aux <= '0';
        elsif (enable_phase_detector_c = '1') then
            BPF_Vin_Aux <= BPF_Vin_c;   -- -- External digital Vin input
            BPF_Vin_Sync <= BPF_Vin_Aux;
            BPF_Vin_Sync_Aux <= BPF_Vin_Sync;
            BPF_Vout_Aux <= BPF_Vout_c;  -- -- External digital Vout output
            BPF_Vout_Sync <= BPF_Vout_Aux;
            BPF_Vout_Sync_Aux <= BPF_Vout_Sync;
        else -- (enable_phase_detector_c = '0')
            BPF_Vin_Aux <= '0';
            BPF_Vin_Sync <= '0';
            BPF_Vout_Aux <= '0';
            BPF_Vout_Sync <= '0';
            BPF_Vin_Sync_Aux <= '0';
            BPF_Vout_Sync_Aux <= '0';
      end if;
   end if;
end process;



-- Rising edge detection of synced external signals
BPF_Vin_Rising_edge <=  BPF_Vin_Sync  and (not BPF_Vin_Sync_Aux);
BPF_Vout_Rising_edge <= BPF_Vout_Sync and (not BPF_Vout_Sync_Aux);
```

```
-- Phase mismatching detection between BPF_Vin and BPF_Vout
BPF_Not_in_Phase <= '1' when (BPF_Vin_Sync /= BPF_Vout_Sync) else '0';

-- Output signal assignment
BPF_Vin_First_c <= '1' when (BPF_Not_in_Phase and BPF_Vin_Rising_edge and (not BPF_Vout_Sync_Aux)) = '1' else '0';

BPF_Vout_First_c <= '1' when (BPF_Not_in_Phase and BPF_Vout_Rising_edge and (not BPF_Vin_Sync_Aux)) = '1'else '0';

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- University: UPC - Universitat Politècnica de Catalunya
-- Engineer: Joan Francesc Serra Bou
--
-- Create Date:    20/03/2015
-- Design Name:    Digitally Tunabale Analogue Filter controller
-- Module Name:    phase_ctrl_comp - Behavioral
-- Project Name:   Digitally Tunabale Analogue Filter
-- Target Devices: XC7A100T-1CSG324C - Nexys4 - Artix-7
-- Tool versions:  ISE Design Suite 14.2
-- Description: This component controls digital potentiometers that are capable of phase matching.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity phase_ctrl_comp is
port (
        clk                         :   in   std_logic;
        reset                       :   in   std_logic;
        enable_phase_ctrl_c         :   in   std_logic;   -- All components signals are ended with "_c" unless clk
and reset
        POT_CS_c                    :   out  std_logic;   -- POT chipselect
        POT_SCK_c                   :   out  std_logic;   -- POT SCK
        POT_MOSI_c                  :   out  std_logic;   -- POT SDI
        POT_SDO_c                   :   in   std_logic;   -- POT SDO // POT SHDN is wired on PCB
        Freq_limit                  :   in   std_logic;
        BPF_Vin_First_c             :   in   std_logic;
        BPF_Vout_First_c            :   in   std_logic
);
end phase_ctrl_comp;

architecture Behavioral of phase_ctrl_comp is
```

```
---- Constant declarations
-- POT address
constant POT_addr_P0: std_logic_vector(3 downto 0) := ( '0',  '0',  '0',  '0' );
constant POT_addr_P1: std_logic_vector(3 downto 0) := ( '0',  '0',  '0',  '1' );
-- POT commands
constant POT_cmd_wr:  std_logic_vector(1 downto 0)  := ( '0',  '0' );
constant POT_cmd_inc: std_logic_vector(1 downto 0)  := ( '0',  '1' );
constant POT_cmd_dec: std_logic_vector(1 downto 0)  := ( '1',  '0' );
constant POT_cmd_rd:  std_logic_vector(1 downto 0)  := ( '1',  '1' );
-- POT spare bit
constant POT_bit_9:  std_logic := '0';

---- Type declarations
type Phase_State_type is (Idle, Phase_matching, Pot_addr_ctlr);
type POT_State_type is (POT_Idle, POT_Load, POT_Send, POT_Stop);

-- SIGNALS
signal Phase_State: Phase_State_type := Idle;

---- POT Signals
signal POT_State: POT_State_type := POT_Idle;
signal POT_data: std_logic_vector(8 downto 0) := b"000000000";
signal POT_cmd: std_logic_vector(1 downto 0) := b"00";
signal POT_addr: std_logic_vector(3 downto 0) := b"0000";
signal POT_addr_last: std_logic_vector(3 downto 0) := b"0000";
signal POT_message: std_logic_vector(15 downto 0) := (others => '0');
signal POT_bits_send: integer range 0 to 16 := 16;
signal POT_cmd_send: std_logic := '0';
signal POT_SCK_AUX: std_logic := '0';

---- Double potentiometer operation signals
--signal POT_message_short: std_logic_vector(7 downto 0) := (others => '0');
--signal POT_read_done: std_logic := '0';
--signal POT_Read: std_logic_vector(15 downto 0) := (others => '0');
--signal POT_Read_Data_P0: std_logic_vector(8 downto 0) := (others => '0');
--signal POT_Read_Data_P1: std_logic_vector(8 downto 0) := (others => '0');
```

```
begin

    phase_ctrl_comp_state_machine: process(clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                POT_CS_c <= '1';
                POT_SCK_c <= '0';
                POT_MOSI_c <= '0';
                Phase_State <= Idle;
            else
                case Phase_State is
                    when Idle =>
                        POT_CS_c <= '1';
                        POT_SCK_c <= '0';
                        POT_MOSI_c <= '0';
                        POT_cmd_send <= '0';
                        --POT_read_done <= '0';  -- For Double_potentiometer operation
                        if (enable_phase_ctrl_c = '1') then
                            Phase_State <= Phase_matching;
                        else
                            Phase_State <= Idle;
                        end if;

                    when Phase_matching =>
                       -- POT inc or dec depending on phase mismatch between Vin and Vout
                        if ( (POT_cmd_send = '0') and (Freq_limit = '1') ) then
                            -- POT State Machine
                            case POT_State is
                                when POT_Idle =>
                                    -- initialize POT SPI
                                    POT_CS_c <= '1';
                                    POT_MOSI_c <= '0';
                                    POT_SCK_c <= '0';
                                    POT_message <= POT_addr_P0 & POT_cmd & POT_bit_9 & POT_bit_9 &  -- Fisrt cmd
                                                   POT_addr_P1 & POT_cmd & POT_bit_9 & POT_bit_9;   -- Second cmd
                                    -- POT_message_short <= POT_addr & POT_cmd & POT_bit_9 & POT_bit_9;
                                    -- For Double_potentiometer operation
```

```
                POT_bits_send <= 16;
                --POT_bits_send <= 8; -- For Double_potentiometer operation
                POT_State <= POT_Load;
            when POT_Load =>
                POT_CS_c <= '0';
                POT_SCK_c <= '0';
                POT_SCK_AUX <='0';
                POT_MOSI_c <= POT_message(15);
                --POT_MOSI_c <= POT_message_short(7); -- For Double_potentiometer operation
                POT_State <= POT_Send;
            when POT_Send =>
                if POT_SCK_AUX = '0' then
                   POT_message <= POT_message(14 downto 0) & "0";
                    --POT_message_short <= POT_message_short(6 downto 0) & "0";
                    -- For Double_potentiometer operation
                    POT_bits_send <= POT_bits_send - 1;
                end if;
                if (POT_bits_send = 0) then
                    POT_State <= POT_Stop;
                else
                    POT_SCK_c <= '1';  -- In Stop state SPI_SCK is not needed
                    POT_SCK_AUX <= '1';
                    POT_State <= POT_Load;
                end if;
            when POT_Stop =>
                POT_CS_c <= '1';
                POT_MOSI_c <= '0';
                POT_SCK_c <= '0';
                POT_cmd_send <= '1';
                POT_State <= POT_Idle;
            when others =>
                POT_State <= POT_Idle;
        end case;
    end if;
    -- Direct Transition to next state when inc or dec is done
    if (POT_cmd_send = '1') then
        Phase_State <= Idle;
        --Phase_State <= Pot_addr_ctlr; -- For Double_potentiometer operation
```

```
                        end if;

-- Uncomment for Double_potentiometer operation
--                  when Pot_addr_ctlr =>
--                      if ( (POT_read_done = '0') and (Freq_limit = '1') ) then
--                          -- POT State Machine
--                          case POT_State is
--                              when POT_Idle =>
--                                  -- READ POT SPI through SDO
--                                  POT_CS_c <= '1';
--                                  POT_MOSI_c <= '0';
--                                  POT_SCK_c <= '0';
--                                  POT_message <= POT_addr & POT_cmd_rd & POT_bit_9 & POT_data;
--                                  POT_bits_send <= 16;
--                                  POT_State <= POT_Load;
--                              when POT_Load =>
--                                  POT_CS_c <= '0';
--                                  POT_SCK_c <= '0';
--                                  POT_SCK_AUX <='0';
--                                  POT_MOSI_c <= POT_message(15);
--                                  POT_State <= POT_Send;
--                              when POT_Send =>
--                                  if POT_SCK_AUX = '0' then
--                                      POT_message <= POT_message(14 downto 0) & "0";
--                                      POT_Read <= POT_Read(14 downto 0) & POT_SDO_c;
--                                      POT_bits_send <= POT_bits_send - 1;
--                                  end if;
--                                  if (POT_bits_send = 0) then
--                                      POT_State <= POT_Stop;
--                                  else
--                                      POT_SCK_c <= '1';  -- In Stop state SPI_SCK is not needed
--                                      POT_SCK_AUX <= '1';
--                                      POT_State <= POT_Load;
--                                  end if;
--                              when POT_Stop =>
--                                  POT_CS_c <= '1';
--                                  POT_MOSI_c <= '0';
--                                  POT_SCK_c <= '0';
```

```
--                                  POT_read_done <= '1';
--                                  if (POT_addr = POT_addr_P0) then
--                                      POT_Read_Data_P0 <= POT_Read(9 downto 1);
--                                  else
--                                      POT_Read_Data_P1 <= POT_Read(9 downto 1);
--                                      POT_addr_last <= POT_addr_P1;
--                                  end if;
--                                  POT_State <= POT_Idle;
--                              when others =>
--                                  POT_State <= POT_Idle;
--                          end case;
--                      end if;
--
--                      -- Direct Transition to next state when inc or dec is done
--                      if (POT_read_done = '1') then
--                          Phase_State <= Idle;
--                      end if;

                    when others =>
                        Phase_State <= Idle;
                end case;
            end if;
        end if;
    end process;

--  Inc or dec selection. Increment when Vin is advanced to Vout. Decrement when Vin is delayed to Vout.
phase_inc_or_dec_sel: process(clk)
begin
   if (clk'event and clk = '1') then
      if (reset = '1') then
         POT_cmd <= POT_cmd_inc;
      else
            if (BPF_Vin_First_c = '1') then
                POT_cmd <= POT_cmd_dec;
            end if;
            if (BPF_Vout_First_c = '1') then
                POT_cmd <= POT_cmd_inc;
            end if;
```

```
      end if;
   end if;
end process;

-- Uncomment for Double_potentiometer operation
----  Potentiometer address selection.
--phase_wiper0_or_wiper1_sel: process(clk)
--begin
--   if (clk'event and clk = '1') then
--      if (reset = '1') then
--         POT_addr <= POT_addr_P0;
--      else
--          -- When POT_P0 reaches the full scale, addres changes to point to POT_P1
--          if (POT_Read_Data_P0 = b"100000000" and POT_addr_last = POT_addr_P0) then
--              POT_addr <= POT_addr_P1;
--          end if;
--          -- When POT_P1 reaches the zero scale, addres changes to point to POT_P0
--          if (POT_Read_Data_P1 = b"000000000" and POT_addr_last = POT_addr_P1) then
--              POT_addr <= POT_addr_P0;
--          end if;
--      end if;
--   end if;
--end process;

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- University: UPC - Universitat Politècnica de Catalunya
-- Engineer: Joan Francesc Serra Bou
--
-- Create Date:    05/04/2015
-- Design Name:    Digitally Tunabale Analogue Filter controller
-- Module Name:    amp_ctrl_comp - Behavioral
-- Project Name:   Digitally Tunabale Analogue Filter
-- Target Devices: XC7A100T-1CSG324C - Nexys4 - Artix-7
-- Tool versions:  ISE Design Suite 14.2
-- Description: This component controls digital potentiometer that is capable of amplitude matching.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity amp_ctrl_comp is
port (
        clk                         :   in   std_logic;
        reset                       :   in   std_logic;
        enable_amp_ctrl_c           :   in   std_logic;   -- All components signals are ended with "_c" unless
global ones
        POT_amp_CS_c                :   out  std_logic;   -- POT amp chipselect
        POT_amp_SCK_c               :   out  std_logic;   -- POT amp SCK
        POT_amp_MOSI_c              :   out  std_logic;   -- POT amp SDI
        POT_amp_SDO_c               :   in   std_logic;   -- POT amp SDO
        Freq_limit                  :   in   std_logic;
        AMP_Vin_Over_c              :   in   std_logic;
        AMP_Vout_Over_c             :    in   std_logic
);
end amp_ctrl_comp;

architecture Behavioral of amp_ctrl_comp is
```

```
---- Constant declarations
-- POT address
constant POT_amp_addr_P0: std_logic_vector(3 downto 0) := ( '0',  '0',  '0',  '0' );
constant POT_amp_addr_P1: std_logic_vector(3 downto 0) := ( '0',  '0',  '0',  '1' );
-- POT commands
constant POT_amp_cmd_wr:  std_logic_vector(1 downto 0)  := ( '0',  '0' );
constant POT_amp_cmd_inc: std_logic_vector(1 downto 0)  := ( '0',  '1' );
constant POT_amp_cmd_dec: std_logic_vector(1 downto 0)  := ( '1',  '0' );
constant POT_amp_cmd_rd:  std_logic_vector(1 downto 0)  := ( '1',  '1' );
-- POT spare bit
constant POT_amp_bit_9:  std_logic := '0';

---- Type declarations
type Amp_State_type is (Idle, Amp_matching); --, Pot_amp_addr_ctlr);
type POT_amp_State_type is (POT_amp_Idle, POT_amp_Load, POT_amp_Send, POT_amp_Stop);

-- SIGNALS
signal Amp_State: Amp_State_type := Idle;

---- POT Signals
signal POT_amp_State: POT_amp_State_type := POT_amp_Idle;
signal POT_amp_data: std_logic_vector(8 downto 0) := b"000000000";
signal POT_amp_cmd: std_logic_vector(1 downto 0) := b"00";
signal POT_amp_addr: std_logic_vector(3 downto 0) := b"0000";
signal POT_amp_addr_last: std_logic_vector(3 downto 0) := b"0000";
signal POT_amp_message: std_logic_vector(15 downto 0) := (others => '0');
signal POT_amp_bits_send: integer range 0 to 16 := 16;
signal POT_amp_cmd_send: std_logic := '0';
signal POT_amp_SCK_AUX: std_logic := '0';

---- Double potentiometer operation signals
--signal POT_amp_message_short: std_logic_vector(7 downto 0) := (others => '0');
--signal POT_amp_read_done: std_logic := '0';
--signal POT_amp_Read: std_logic_vector(15 downto 0) := (others => '0');
--signal POT_amp_Read_Data_P0: std_logic_vector(8 downto 0) := (others => '0');
--signal POT_amp_Read_Data_P1: std_logic_vector(8 downto 0) := (others => '0');
```

```
begin

    phase_ctrl_comp_state_machine: process(clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                POT_amp_CS_c <= '1';
                POT_amp_SCK_c <= '0';
                POT_amp_MOSI_c <= '0';
                Amp_State <= Idle;
            else
                case Amp_State is
                    when Idle =>
                        POT_amp_CS_c <= '1';
                        POT_amp_SCK_c <= '0';
                        POT_amp_MOSI_c <= '0';
                        POT_amp_cmd_send <= '0';
                        --POT_amp_read_done <= '0'; -- For Double_potentiometer operation
                        if (enable_amp_ctrl_c = '1') then
                            Amp_State <= Amp_matching;
                        else
                            Amp_State <= Idle;
                        end if;

                    when Amp_matching =>
                       -- POT inc or dec depending on phase mismatch between Vin and Vout
                        if ( (POT_amp_cmd_send = '0') and (Freq_limit = '1') ) then
                            -- POT State Machine
                            case POT_amp_State is
                                when POT_amp_Idle =>
                                    -- initialize POT SPI
                                    POT_amp_CS_c <= '1';
                                    POT_amp_MOSI_c <= '0';
                                    POT_amp_SCK_c <= '0';
                                     POT_amp_message <= POT_amp_addr_P0 & POT_amp_cmd & POT_amp_bit_9 &
                                   POT_amp_bit_9 &  -- First cmd
                                   POT_amp_addr_P1 & POT_amp_cmd & POT_amp_bit_9 &   POT_amp_bit_9;-- Second cmd
```

```
                    -- POT_amp_message_short <= POT_addr & POT_cmd & POT_bit_9 & POT_bit_9;
                     -- For Double_potentiometer operation
                    POT_amp_bits_send <= 16;
                    --POT_amp_bits_send <= 8; -- For Double_potentiometer operation
                    POT_amp_State <= POT_amp_Load;
                when POT_amp_Load =>
                    POT_amp_CS_c <= '0';
                    POT_amp_SCK_c <= '0';
                    POT_amp_SCK_AUX <='0';
                    POT_amp_MOSI_c <= POT_amp_message(15);
                    --POT_amp_MOSI_c <= POT_amp_message_short(7); -- For Double_potentiometer
operation
                    POT_amp_State <= POT_amp_Send;
                when POT_amp_Send =>
                    if POT_amp_SCK_AUX = '0' then
                       POT_amp_message <= POT_amp_message(14 downto 0) & "0";
                        --POT_amp_message_short <= POT_amp_message_short(6 downto 0) & "0";
                        -- For Double_potentiometer operation
                        POT_amp_bits_send <= POT_amp_bits_send - 1;
                    end if;
                    if (POT_amp_bits_send = 0) then
                        POT_amp_State <= POT_amp_Stop;
                    else
                        POT_amp_SCK_c <= '1';  -- In Stop state SPI_SCK is not needed
                        POT_amp_SCK_AUX <= '1';
                        POT_amp_State <= POT_amp_Load;
                    end if;
                when POT_amp_Stop =>
                    POT_amp_CS_c <= '1';
                    POT_amp_MOSI_c <= '0';
                    POT_amp_SCK_c <= '0';
                    POT_amp_cmd_send <= '1';
                    POT_amp_State <= POT_amp_Idle;
                when others =>
                    POT_amp_State <= POT_amp_Idle;
            end case;
        end if;
        -- Direct Transition to next state when inc or dec is done
```

```
                        if (POT_amp_cmd_send = '1') then
                            Amp_State <= Idle;
                            --Amp_State <= Pot_amp_addr_ctlr; -- For Double_potentiometer operation
                        end if;

-- Uncomment for Double_potentiometer operation
--                  when Pot_amp_addr_ctlr =>
--                      if ( (POT_amp_read_done = '0') and (Freq_limit = '1') ) then
--                          -- POT State Machine
--                          case POT_amp_State is
--                              when POT_amp_Idle =>
--                                  -- READ POT SPI through SDO
--                                  POT_amp_CS_c <= '1';
--                                  POT_amp_MOSI_c <= '0';
--                                  POT_amp_SCK_c <= '0';
--                                  POT_amp_message <= POT_amp_addr & POT_amp_cmd_rd & POT_amp_bit_9 &
POT_amp_data;
--                                  POT_amp_bits_send <= 16;
--                                  POT_amp_State <= POT_amp_Load;
--                              when POT_amp_Load =>
--                                  POT_amp_CS_c <= '0';
--                                  POT_amp_SCK_c <= '0';
--                                  POT_amp_SCK_AUX <='0';
--                                  POT_amp_MOSI_c <= POT_amp_message(15);
--                                  POT_amp_State <= POT_amp_Send;
--                              when POT_amp_Send =>
--                                  if POT_amp_SCK_AUX = '0' then
--                                      POT_amp_message <= POT_amp_message(14 downto 0) & "0";
--                                      POT_amp_Read <= POT_amp_Read(14 downto 0) & POT_amp_SDO_c;
--                                      POT_amp_bits_send <= POT_amp_bits_send - 1;
--                                  end if;
--                                  if (POT_amp_bits_send = 0) then
--                                      POT_amp_State <= POT_amp_Stop;
--                                  else
--                                      POT_amp_SCK_c <= '1';  -- In Stop state SPI_SCK is not needed
--                                      POT_amp_SCK_AUX <= '1';
--                                      POT_amp_State <= POT_amp_Load;
--                                  end if;
```

```
--                                when POT_amp_Stop =>
--                                    POT_amp_CS_c <= '1';
--                                    POT_amp_MOSI_c <= '0';
--                                    POT_amp_SCK_c <= '0';
--                                    POT_amp_read_done <= '1';
--                                    if (POT_amp_addr = POT_amp_addr_P0) then
--                                        POT_amp_Read_Data_P0 <= POT_amp_Read(9 downto 1);
--                                        POT_amp_addr_last <= POT_amp_addr_P0;
--                                    else
--                                        POT_amp_Read_Data_P1 <= POT_amp_Read(9 downto 1);
--                                        POT_amp_addr_last <= POT_amp_addr_P1;
--                                    end if;
--                                    POT_amp_State <= POT_amp_Idle;
--                                when others =>
--                                    POT_amp_State <= POT_amp_Idle;
--                            end case;
--                        end if;
--
--                        -- Direct Transition to next state when inc or dec is done
--                        if (POT_amp_read_done = '1') then
--                            Amp_State <= Idle;
--                        end if;

                    when others =>
                        Amp_State <= Idle;
                end case;
            end if;
        end if;
    end process;

--  Inc or dec selection. Increment when Vin is under Vout. Decrement when Vin is over Vout.
amp_inc_or_dec_sel: process(clk)
begin
   if (clk'event and clk = '1') then
      if (reset = '1') then
         POT_amp_cmd <= POT_amp_cmd_inc;
      else
            if (AMP_Vin_Over_c = '1') then
```

```
                POT_amp_cmd <= POT_amp_cmd_dec;
            end if;
            if (AMP_Vout_Over_c = '1') then
                POT_amp_cmd <= POT_amp_cmd_inc;
            end if;
      end if;
   end if;
end process;

-- Uncomment for Double_potentiometer operation
----  Potentiometer address selection.
--phase_wiper0_or_wiper1_sel: process(clk)
--begin
--   if (clk'event and clk = '1') then
--      if (reset = '1') then
--         POT_amp_addr <= POT_amp_addr_P0;
--      else
--          -- When POT_P0 reaches the full scale, addres changes to point to POT_P1
--          if (POT_amp_Read_Data_P0 = b"100000000" and POT_amp_addr_last = POT_amp_addr_P0) then
--              POT_amp_addr <= POT_amp_addr_P1;
--          end if;
--          -- When POT_P1 reaches the zero scale, addres changes to point to POT_P0
--          if (POT_amp_Read_Data_P1 = b"000000000" and POT_amp_addr_last = POT_amp_addr_P1) then
--              POT_amp_addr <= POT_amp_addr_P0;
--          end if;
--      end if;
--   end if;
--end process;

end Behavioral;
```

```
-- file: xadc_wiz_v2_1.vhd
-- (c) Copyright 2009 - 2011 Xilinx, Inc. All rights reserved.
--
-- This file contains confidential and proprietary information
-- of Xilinx, Inc. and is protected under U.S. and
-- international copyright and other intellectual property
-- laws.
--
-- DISCLAIMER
-- This disclaimer is not a license and does not grant any
-- rights to the materials distributed herewith. Except as
-- otherwise provided in a valid license issued to you by
-- Xilinx, and to the maximum extent permitted by applicable
-- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
-- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
-- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
-- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
-- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
-- (2) Xilinx shall not be liable (whether in contract or tort,
-- including negligence, or under any other theory of
-- liability) for any loss or damage of any kind or nature
-- related to, arising under or in connection with these
-- materials, including for any direct, or any indirect,
-- special, incidental, or consequential loss or damage
-- (including loss of data, profits, goodwill, or any type of
-- loss or damage suffered as a result of any action brought
-- by a third party) even if such damage or loss was
-- reasonably foreseeable or Xilinx had been advised of the
-- possibility of the same.
--
-- CRITICAL APPLICATIONS
-- Xilinx products are not designed or intended to be fail-
-- safe, or for use in any application requiring fail-safe
-- performance, such as life-support or safety devices or
-- systems, Class III medical devices, nuclear facilities,
-- applications related to the deployment of airbags, or any
-- other applications that could lead to death, personal
```

```
-- injury, or severe property or environmental damage
-- (individually and collectively, "Critical
-- Applications"). Customer assumes the sole risk and
-- liability of any use of Xilinx products in Critical
-- Applications, subject only to applicable laws and
-- regulations governing limitations on product liability.
--
-- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
-- PART OF THIS FILE AT ALL TIMES.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
Library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

entity xadc_wiz_v2_1 is
    port (
          DADDR_IN            : in  STD_LOGIC_VECTOR (6 downto 0);
           -- Address bus for the DRP (dynamic reconfiguration port)
          DCLK_IN             : in  STD_LOGIC;  -- DRP Clock input
          DEN_IN              : in  STD_LOGIC;  -- DRP Enable Signal
          DI_IN               : in  STD_LOGIC_VECTOR (15 downto 0);  -- DRP Input data bus
          DWE_IN              : in  STD_LOGIC;  -- DRP Write Enable
          RESET_IN            : in  STD_LOGIC;
           -- Reset signal for the System Monitor control logic
          VAUXP2              : in  STD_LOGIC;                        -- Auxiliary Channel 2
          VAUXN2              : in  STD_LOGIC;
          VAUXP3              : in  STD_LOGIC;                        -- Auxiliary Channel 3
          VAUXN3              : in  STD_LOGIC;
          VAUXP10             : in  STD_LOGIC;                        -- Auxiliary Channel 10
          VAUXN10             : in  STD_LOGIC;
          VAUXP11             : in  STD_LOGIC;                        -- Auxiliary Channel 11
          VAUXN11             : in  STD_LOGIC;
          BUSY_OUT            : out  STD_LOGIC;                       -- ADC Busy signal
          CHANNEL_OUT         : out  STD_LOGIC_VECTOR (4 downto 0);   -- Channel Selection Outputs
          DO_OUT              : out  STD_LOGIC_VECTOR (15 downto 0);  -- DRP Output data bus
          DRDY_OUT            : out  STD_LOGIC;   -- DRP Data ready signal
```

```
          EOC_OUT              : out  STD_LOGIC;   -- End of Conversion Signal
          EOS_OUT              : out  STD_LOGIC;   -- End of Sequence Signal
          ALARM_OUT            : out  STD_LOGIC;   -- OR'ed output of all the Alarms
          VP_IN                : in   STD_LOGIC;   -- Dedicated Analog Input Pair
          VN_IN                : in   STD_LOGIC
);
end xadc_wiz_v2_1;

architecture xilinx of xadc_wiz_v2_1 is

  attribute CORE_GENERATION_INFO : string;
  attribute CORE_GENERATION_INFO of xilinx : architecture is
"xadc_wiz_v2_1,xadc_wiz_v2_1,{component_name=xadc_wiz_v2_1,dclk_frequency=100,enable_busy=true,enable_convst=false
,enable_convstclk=false,enable_dclk=true,enable_drp=true,enable_eoc=true,enable_eos=true,enable_vbram_alaram=false
,enable_vccddro_alaram=false,enable_vccpaux_alaram=false,enable_Vccint_Alaram=false,enable_Vccaux_alaram=false,ena
ble_vccpint_alaram=false,ot_alaram=false,user_temp_alaram=false,timing_mode=continuous,channel_averaging=None,sequ
encer_mode=on,startup_channel_selection=simultaneous_sampling}";

  signal FLOAT_VCCAUX_ALARM : std_logic;
  signal FLOAT_VCCINT_ALARM : std_logic;
  signal FLOAT_USER_TEMP_ALARM : std_logic;
  signal FLOAT_VBRAM_ALARM : std_logic;
  signal FLOAT_MUXADDR : std_logic_vector (4 downto 0);
  signal aux_channel_p : std_logic_vector (15 downto 0);
  signal aux_channel_n : std_logic_vector (15 downto 0);
  signal alm_int : std_logic_vector (7 downto 0);
begin

        aux_channel_p(0) <= '0';
        aux_channel_n(0) <= '0';

        aux_channel_p(1) <= '0';
        aux_channel_n(1) <= '0';

        aux_channel_p(2) <= VAUXP2;
        aux_channel_n(2) <= VAUXN2;

        aux_channel_p(3) <= VAUXP3;
```

```
 aux_channel_n(3) <= VAUXN3;

 aux_channel_p(4) <= '0';
 aux_channel_n(4) <= '0';

 aux_channel_p(5) <= '0';
 aux_channel_n(5) <= '0';

 aux_channel_p(6) <= '0';
 aux_channel_n(6) <= '0';

 aux_channel_p(7) <= '0';
 aux_channel_n(7) <= '0';

 aux_channel_p(8) <= '0';
 aux_channel_n(8) <= '0';

 aux_channel_p(9) <= '0';
 aux_channel_n(9) <= '0';

 aux_channel_p(10) <= VAUXP10; -- NOT USED
 aux_channel_n(10) <= VAUXN10; -- NOT USED

 aux_channel_p(11) <= VAUXP11; -- NOT USED
 aux_channel_n(11) <= VAUXN11; -- NOT USED

 aux_channel_p(12) <= '0';
 aux_channel_n(12) <= '0';

 aux_channel_p(13) <= '0';
 aux_channel_n(13) <= '0';

 aux_channel_p(14) <= '0';
 aux_channel_n(14) <= '0';

 aux_channel_p(15) <= '0';
 aux_channel_n(15) <= '0';
ALARM_OUT <= alm_int(7);
```

```
 XADC_INST : XADC
     generic map(
        INIT_40 => X"8000", -- config reg 0
        INIT_41 => X"4f0f", -- config reg 1
        INIT_42 => X"0400", -- config reg 2
        INIT_48 => X"0000", -- Sequencer channel selection
        INIT_49 => X"000c", -- Sequencer channel selection
        INIT_4A => X"0000", -- Sequencer Average selection
        INIT_4B => X"0000", -- Sequencer Average selection
        INIT_4C => X"0000", -- Sequencer Bipolar selection
        INIT_4D => X"0000", -- Sequencer Bipolar selection
        INIT_4E => X"0000", -- Sequencer Acq time selection
        INIT_4F => X"0000", -- Sequencer Acq time selection
        INIT_50 => X"b5ed", -- Temp alarm trigger
        INIT_51 => X"57e4", -- Vccint upper alarm limit
        INIT_52 => X"a147", -- Vccaux upper alarm limit
        INIT_53 => X"ca33",  -- Temp alarm OT upper
        INIT_54 => X"a93a", -- Temp alarm reset
        INIT_55 => X"52c6", -- Vccint lower alarm limit
        INIT_56 => X"9555", -- Vccaux lower alarm limit
        INIT_57 => X"ae4e",  -- Temp alarm OT reset
        INIT_58 => X"5999",  -- Vbram upper alarm limit
        INIT_5C => X"5111",  -- Vbram lower alarm limit
        SIM_DEVICE => "7SERIES",
        SIM_MONITOR_FILE => "XADC.txt"
        )

port map (
        CONVST              => '0',
        CONVSTCLK           => '0',
        DADDR(6 downto 0)   => DADDR_IN(6 downto 0),
        DCLK                => DCLK_IN,
        DEN                 => DEN_IN,
        DI(15 downto 0)     => DI_IN(15 downto 0),
        DWE                 => DWE_IN,
        RESET               => RESET_IN,
        VAUXN(15 downto 0)  => aux_channel_n(15 downto 0),
```

```
        VAUXP(15 downto 0)  => aux_channel_p(15 downto 0),
        ALM                 => alm_int,
        BUSY                => BUSY_OUT,
        CHANNEL(4 downto 0) => CHANNEL_OUT(4 downto 0),
        DO(15 downto 0)     => DO_OUT(15 downto 0),
        DRDY                => DRDY_OUT,
        EOC                 => EOC_OUT,
        EOS                 => EOS_OUT,
        JTAGBUSY            => open,
        JTAGLOCKED          => open,
        JTAGMODIFIED        => open,
        OT                  => open,

        MUXADDR             => FLOAT_MUXADDR,
        VN                  => VN_IN,
        VP                  => VP_IN
         );
end xilinx;
```

```
--------------------------------------------------------------------------------
--  debouncer.vhd -- Signal Debouncer
--------------------------------------------------------------------------------
-- Author:  Sam Bobrowicz
--          Copyright 2011 Digilent, Inc.
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
-- This component is used to debounce signals. It is designed to
-- independently debounce a variable number of signals, the number of which
-- are set using the PORT_WIDTH generic. Debouncing is done by only
-- registering a change in a button state if it remains constant for
-- the number of clocks determined by the DEBNC_CLOCKS generic.
--
-- Generic Descriptions:
--
--   PORT_WIDTH - The number of signals to debounce. determines the width
--                of the SIGNAL_I and SIGNAL_O std_logic_vectors
--   DEBNC_CLOCKS - The number of clocks (CLK_I) to wait before registering
--                  a change.
--
-- Port Descriptions:
--
--   SIGNAL_I - The input signals. A vector of width equal to PORT_WIDTH
--   CLK_I  - Input clock
--   SIGNAL_O - The debounced signals. A vector of width equal to PORT_WIDTH
--
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
-- Revision History:
--  08/08/2011(SamB): Created using Xilinx Tools 13.2
--  08/29/2013(SamB): Improved reuseability by using generics
--  05/05/2015(JFSB): Modified, debounce added to btnCpuReset signal
--------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;
USE IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;

entity debouncer is
    Generic ( DEBNC_CLOCKS : INTEGER range 2 to (INTEGER'high) := 2**16;
              PORT_WIDTH : INTEGER range 1 to (INTEGER'high) := 6);
    Port ( SIGNAL_I : in  STD_LOGIC_VECTOR ((PORT_WIDTH - 1) downto 0);
           CLK_I : in  STD_LOGIC;
           SIGNAL_O : out  STD_LOGIC_VECTOR ((PORT_WIDTH - 1) downto 0));
end debouncer;

architecture Behavioral of debouncer is

constant CNTR_WIDTH : integer := natural(ceil(LOG2(real(DEBNC_CLOCKS))));
constant CNTR_MAX : std_logic_vector((CNTR_WIDTH - 1) downto 0) := std_logic_vector(to_unsigned((DEBNC_CLOCKS -
1), CNTR_WIDTH));
type VECTOR_ARRAY_TYPE is array (integer range <>) of std_logic_vector((CNTR_WIDTH - 1) downto 0);

signal sig_cntrs_ary : VECTOR_ARRAY_TYPE (0 to (PORT_WIDTH - 1)) := (others=>(others=>'0'));

signal sig_out_reg : std_logic_vector((PORT_WIDTH - 1) downto 0) := (others => '0');

begin

debounce_process : process (CLK_I)
begin
   if (rising_edge(CLK_I)) then
   for index in 0 to (PORT_WIDTH - 1) loop
      if (sig_cntrs_ary(index) = CNTR_MAX) then
         sig_out_reg(index) <= not(sig_out_reg(index));
      end if;
   end loop;
   end if;
end process;
```

```
counter_process : process (CLK_I)
begin
    if (rising_edge(CLK_I)) then
    for index in 0 to (PORT_WIDTH - 1) loop

        if ((sig_out_reg(index) = '1') xor (SIGNAL_I(index) = '1')) then
            if (sig_cntrs_ary(index) = CNTR_MAX) then
                sig_cntrs_ary(index) <= (others => '0');
            else
                sig_cntrs_ary(index) <= sig_cntrs_ary(index) + 1;
            end if;
        else
            sig_cntrs_ary(index) <= (others => '0');
        end if;

    end loop;
    end if;
end process;

SIGNAL_O <= sig_out_reg;

end Behavioral;
```

```
##############################################################################
## Company:
## Engineer: Joan Francesc Serra Bou
##
## Create Date:    28/04/2015
## Project Name:   Digitally Tunabale Analogue Filter
## Target Devices: XC7A100T-1CSG324C - Nexys4 - Artix-7
## Tool versions:  ISE Design Suite 14.2
## Description:    .ucf file where all I/O connections are defined
##
##############################################################################

## Clock signal
NET "CLK"   LOC = "E3"  | IOSTANDARD = "LVCMOS33";
#Bank = 35, Pin name = IO_L12P_T1_MRCC_35, Sch name = CLK100MHZ
NET "CLK" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100 MHz HIGH 50%;

## Buttons
NET "btnCpuReset" LOC = "C12"   | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L3P_T0_DQS_AD1P_15, Sch name = CPU_RESET
NET "BTN<4>"      LOC = "E16"   | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L11N_T1_SRCC_15, Sch name = BTNC
NET "BTN<2>"      LOC = "F15"   | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L14P_T2_SRCC_15, Sch name = BTNU
NET "BTN<3>" LOC = "T16"    | IOSTANDARD = "LVCMOS33";
#Bank = CONFIG, Pin name = IO_L15N_T2_DQS_DOUT_CSO_B_14,    Sch name = BTNL
NET "BTN<0>"      LOC = "R10"   | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_25_14, Sch name = BTNR
NET "BTN<1>"      LOC = "V10"   | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_L21P_T3_DQS_14, Sch name = BTND

## Pmod Header JA
NET "BPF_Vin"     LOC = "B13"   | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L1N_T0_AD0N_15, Sch name = JA1
NET "BPF_Vout"    LOC = "F14"   | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L5N_T0_AD9N_15, Sch name = JA2
```

```
## Pmod Header JB
NET "POT_CS"        LOC = "G14" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L15N_T2_DQS_ADV_B_15, Sch name = JB1
NET "POT_SCK"       LOC = "P15" | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_L13P_T2_MRCC_14, Sch name = JB2
NET "POT_MOSI"      LOC = "V11" | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_L21N_T3_DQS_A06_D22_14, Sch name = JB3
NET "POT_SDO"       LOC = "V15" | IOSTANDARD = "LVCMOS33";
#Bank = CONFIG, Pin name = IO_L16P_T2_CSI_B_14, Sch name = JB4
NET "POT_amp_CS"    LOC = "K16" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_25_15, Sch name = JB7
NET "POT_amp_SCK"   LOC = "R16" | IOSTANDARD = "LVCMOS33";
#Bank = CONFIG, Pin name = IO_L15P_T2_DQS_RWR_B_14, Sch name = JB8
NET "POT_amp_MOSI"  LOC = "T9"  | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_L24P_T3_A01_D17_14, Sch name = JB9
NET "POT_amp_SDO"   LOC = "U11" | IOSTANDARD = "LVCMOS33";
#Bank = 14, Pin name = IO_L19N_T3_A09_D25_VREF_14, Sch name = JB10

## Pmod Header JXADC
NET "vauxp2"    LOC = "B16" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L7P_T1_AD2P_15, Sch name = XADC3_P -> XA3_P
NET "vauxn2"    LOC = "B17" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L7N_T1_AD2N_15, Sch name = XADC3_N -> XA3_N
NET "vauxp3"    LOC = "A13" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L9P_T1_DQS_AD3P_15, Sch name = XADC1_P -> XA1_P
NET "vauxn3"    LOC = "A14" | IOSTANDARD = "LVCMOS33";
#Bank = 15, Pin name = IO_L9N_T1_DQS_AD3N_15, Sch name = XADC1_N -> XA1_N
```

Glossary

A list of all acronyms and what they stand for.

FPGA - Field Programmable Gate Array

SMPS - Switched Mode Power Supply

CTF - Continuous Time Filter

TQE - Transimpedance Q Enhancement

EUETIB - *Escola Universitària d'Enginyeria Tècnica Industrial de Barcelona*

UPC - *Universitat Politècnica de Catalunya*

I2C - Inter-Integrated Circuit

SPI - Serial Peripheral Interface

ADC - Analogue to Digital Converter

PCB - Printed Circuit Board

LCD - Liquid Crystal Display

VGA - Video Graphics Array

PHY - Physical layer of the Open Systems Interconnection model

LED - Light Emitting Diode

Msps - Mega-Samples per Second

SMA - SubMiniature version A

PMOD - Peripheral Module

VHSIC - Very High Speed Integrated Circuit

VHDL - VHSIC Hardware Description Language

DUT - Device under Test

PROM - Programmable Read-Only Memory

JTAG - Join Test Action Group

USB - Universal Serial Bus

UART - Universal Asynchronous Reciever/Transmitter

MEMS - Microelectromechanical Systems

POR - Power-on Reset

PDIP - Plastic Dual in-line Package

SCK - Serial Clock

SDI - Serial Data Input

SDO - Serial Data Output

WP - Write Protection

SHDN - Shutdown

CS - Chip Select

MSB - Most Significant Bit

OPAMP - Operational Amplifier

BNC - Bayonet Neil-Concelman

SMD - Surface Mounted Device

DIL - Dual in-line

ASM - Algorithmic State Machine