BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

# Development of a Sensorimotor Algorithm Able to Deal with Unforeseen Pushes and Its Implementation Based on VHDL

Bachelor Thesis

Pablo Gabriel Lezcano Giménez

**Matrikelnummer**: 821978

**SS 2015**

**Betreuer BHT:** Prof. Dr. Hild
**Gutachter:** Prof. Kersten
**Beginn Datum:** 03. August 2015
**Ende Datum:** 03. November 2015
**Abgabe Datum:** 02. September 2015

## Acknowledgements

I would like to express my gratefulness to Prof. Dr. Hild for giving me the opportunity to do my Bachelor thesis in the robotics field.

My special thanks to Benjamin Panreck, my colleague and constant supervisor, for his patience guidance, encouragement and useful suggestions on my research work.

To my colleague Pablo de Miguel, who became a good friend during my year in Berlin and contributed with great effort with many common features to both our theses.

I would also wish to thank Marcus Janz and Christian Thiele for their great assistance in some difficulties I encountered during my work.

To Stefan Bethge, Jörg Meier, Peter Hirschfeld and the rest of the Neurorobotics Research Laboratory, for making me feel integrated from day one.

To my friends of the Universidad Politécnica de Madrid César, Fabi, Sancho, Fernando, Andrés, Sergio, Alberto and many others. I am particularly grateful to get to know each and every one of you and to spend countless hours both studying and having fun during the last 3 years. There's not enough money in the world to thank you, so I won't be paying any rounds.

This thesis is dedicated to my sister and my parents, who supported me throughout the whole degree.

## Abstract

There's no doubt about the increasing role played by robotics in today's society. We see it every day applied to the fields of medicine, industry, consumer electronics or even art. Some of these applications are fundamentally made with the purpose of interacting between humans and machines. This human machine interaction is one of the countless parts in the field of robotics that is in continuous development.

"Defying gravity - A Minimal Cognitive Sensorimotor Loop Which Makes Robots With Arbitrary Morphologies Stand Up" [1] by Prof. Dr. Manfred Hild establishes the point of departure of this thesis. It defines an electronic circuit and a VHDL algorithm that fights against varying external forces without the employment of a sensor (in its most strict definition). Sometimes the application would require to ignore some of these forces (pushes) while still considering others (gravitational force). "Development of a Sensorimotor Algorithm Able to Deal with Unforeseen Pushes and Its Implementation Based on VHDL" is a Bachelor thesis with the object of developing, experimenting and documenting a Cognitive Sensorimotor Loop (CSL) algorithm able to differentiate a pushing motion from the gravitational force and decide whether or not to work against it. This new behavior in the CSL has a whole variety of applications and advantages as energy saving or submission to assistive forces among others. This thesis converse mainly around two models, one based on parameterized thresholds and the other one based on a filter, both of them with their own limitations.

This thesis, elaborated during my ERASMUS year in Berlin, concludes my bachelor studies in Electronics Engineering at the Universidad Politécnica de Madrid and Elektrotechnik/ Mechatronik at the Beuth Hochschule für Technik Berlin. In it I wrap up, among other competences, the knowledge I acquired in VHDL hardware modeling. This thesis has some common ground with the bachelor thesis "VHDL-Based System Design of a Cognitive Sensorimotor Loop (CSL) for Haptic Human-Machine Interaction (HMI)" by Pablo de Miguel Nogales and the master thesis "Entwicklung eines Adaptativen Regelungsmechanismus für die Bewegungsoptimierung modularer Aktuatoren" by Benjamin Panreck. The three theses contain some inherited work regarding data visualization among other things, while some other modules were written "as a team" to extend features which are common in the three of them.

## Table of contents

## Table of figures

# 1. Personal motivation

Since I was very young,  science fiction always caught my eye. As a kid all started with spaceship toys and films, as usual. One of these films introduced me to short stories and novels by H.G. Wells, Aldous Huxley, Isaac Asimov, George Orwell, etc.

To me, robotics is making science fiction real or at least getting as close as possible, if that sounds too corny to some. When the time came to choose a field of research for my thesis, robotics seemed like the best choice. It has also a lot of professional opportunities in the fields of biomedical engineering, industry, consumer electronics, automation etc.

Once I was introduced to the Neurorobotics Research Laboratory (NRL) team, I noticed that It wasn't the average workplace. The team comprehends degrees in computer science, mechatronics,  industrial engineering, physics, mathematics, economics and psychology, each and every one of them playing a role to develop Myon, a humanoid autonomous robot. Working the last months in the NRL has challenged my knowledge in both electronics and VHDL hardware modeling. The second has been particularly challenging, since I am not especially fond of programming. Nevertheless, It felt really motivating to research on something that could be applied in the future to anything much more complex e.g. Myon and in the event of not having satisfying results, help the next researcher to learn from my mistakes.

"*Science doesn't purvey absolute truth. Science is a mechanism. It's a way of trying to improve your knowledge of nature. It's a system for testing your thoughts against the universe and seeing whether they match. And this works, not just for the ordinary aspects of science, but for all of life*"

- Isaac Asimov

## 2. Objectives

The objective of this thesis is to document the experiments and overall work done on push ignoring contractive sensorimotor algorithms, meaning sensorimotor algorithms that ignore large magnitude forces (compared to gravity) applied in a short time interval on a pendulum system. This main objective is divided in two sub-objectives:

- Developing a system based on parameterized thresholds.
- Developing a system based on a push bypassing filter.

## 3. Basics and mathematical foundations

Before going any further, there are some fundaments that need to be addressed. These fundaments, which are needed for the system based on a push bypassing filter, comprehend digital signal processing (DSP), representations in binary numbers and its mathematical explanations.

### 3.1. Infinite Impulse Response (IIR) Filters

As the name indicates, IIR filters have an impulse response that continues indefinitely, as opposed to finite impulse response (FIR) filters. The reason whether to choose IIR or FIR filters depends on the characteristics desired for the application. For example, FIR filters can have a generalized linear phase but they cannot be described by closed-form equations, while IIR filters can. [2]

Since the main desire for this system is to reduce complexity, the right choice is to use an IIR discrete-time filter:

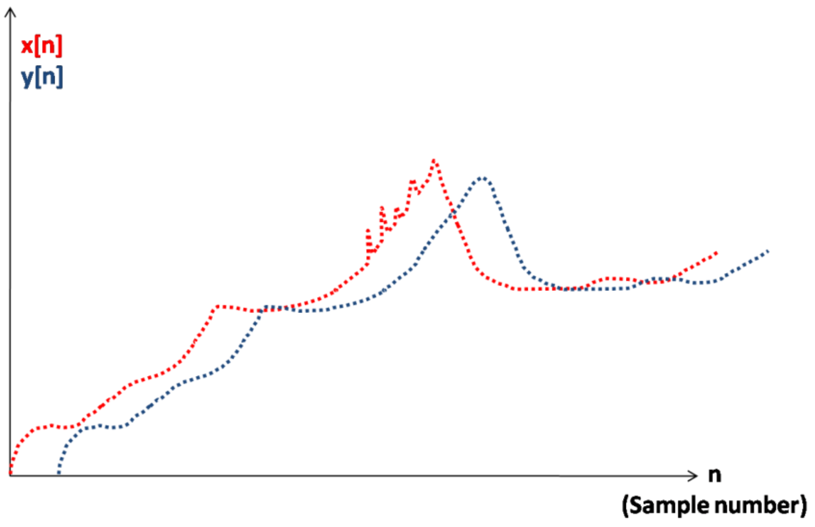


**Figure 1: IIR discrete-time low-pass filter example**

Where $x[n]$ and $y[n]$ are the input and output samples at the present moment respectively. A delayed sample (in a past moment) would be represented by $x[n-N]$ or $y[n-N]$ where $N$ is the sample delay.

The graph shows a filter example with low-pass response, which can be observed by the spikes at the $x[n]$ signal disappearing at the $y[n]$ signal. In other words, the fast variations of the signal within a time period (high frequencies) are cut, hence the low-pass response.

Another thing that can be seen in the graph is the delay introduced by the filter. The blue signal is displaced at the right of the red one. Depending on the system speed, this can become a major issue as will be seen further in the 11th chapter.

Now that the IIR filters have been briefly introduced, we can focus in the one required to achieve the push bypassing. The desired behavior for the IIR filter is as follows:

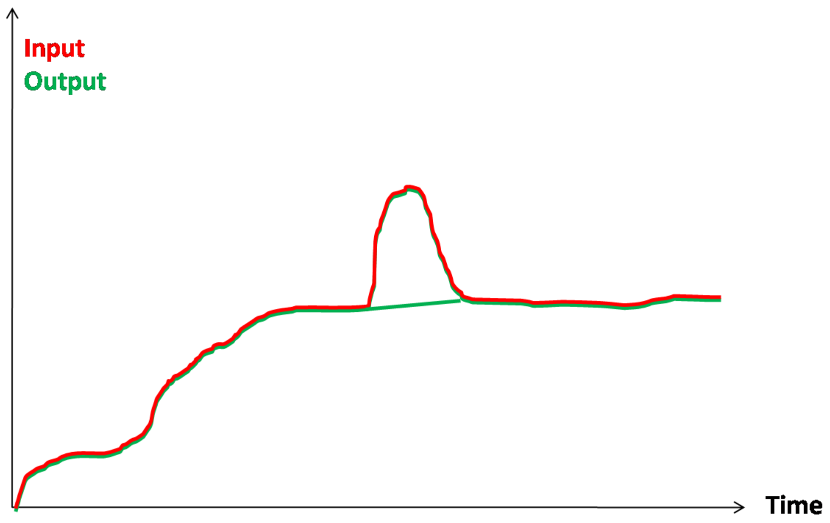


**Figure 2: Integral filtering**

This means that the system needs to cut the high frequencies until the abrupt increases/decreases are smoothened. In other words, a low pass filter is required. This can be done analogically with active or passive RC filters or digitally with the FPGA resources. Since DSP is already a huge subject, this document only will focus in a solution with a low-pass digital 1st order IIR filter[1]:

$$a_0 \cdot y[n] = b_0 \cdot x[n] + a_1 \cdot y[n-1]$$

$x[n]$: filter input

$y[n]$: filter output

$y[n-1]$: previous filter output (1 period delay)

$a_0, b_0, a_1$: filter coefficients

One major concern is to calculate the filter coefficients. Since the length of the pushes are not really specified, it is necessary that the cutoff frequency of the filter can be changed during the experiments (e.g. via keyboard faders) in order to tune it in real time. Hence a further simplification was applied to the filter. That way, there's not an actual coefficient calculation:

$$y[n] = a \cdot x[n] + (1 - a) \cdot y[n-1]$$

Where $a$ is a coefficient between $0$ and $1$.

There are a few things in terms of computational performance that are interesting to take note. Considering that a multiplication is a very expensive operation in terms of resources, a good alternative is using shift operators. Given coefficients between $0$ and $1$, the operation needed is right shifting. Right shifts are then reduced to mere $2^n$ divisions, being $n$ the number of shifts ($\frac{1}{2} = 0.5$, $\frac{1}{2^2} = 0.25$, $\frac{1}{2^3} = 0.125$ and so on).

The use of shift operators brings up another topic regarding data types. The "integral" variable is type signed, meaning that the 18 bit variable is a two's complement representation. It's important to make sure that shifting right produces a correct number. Let's do a simplified explanation based on 4 bit numbers[1]:

| | **Bit** | | | |
|---|---|---|---|---|
| **Number** | 3 | 2 | 1 | 0 |
| **-8** | 1 | 0 | 0 | 0 |
| **-4** | 1 | 1 | 0 | 0 |
| **2** | 1 | 1 | 1 | 0 |

| | **Bit** | | | |
|---|---|---|---|---|
| **Number** | 3 | 2 | 1 | 0 |
| **4** | 0 | 1 | 0 | 0 |
| **2** | 0 | 0 | 1 | 0 |
| **1** | 0 | 0 | 0 | 1 |

**Figure 3: right-shift operation of a 4 bits two's complement number**

A quick view of the tables shows that a correct shift in two's complement representation must introduce into the MSB position the previous MSB value. From another point of view, a two's complement number can be increased in range (without changing the value) just adding '1's at the left if the number is negative and '0's if the number is positive:

$$0100_{BIN} = 0 \ldots 0000\ 0100_{BIN}$$

$$1011_{\mathrm{BIN}} = 1 \ldots 1111\ 1011_{\mathrm{BIN}}$$

In VHDL, this problem is automatically solved by using the *"IEEE.NUMERIC_STD"* package, which supports shift operations for signed numbers.

## 3.2.Fixed comma data type

As seen in the sub-chapter above, the filter uses decimal coefficients between 0 and 1. This characteristic is mandatory or the filter won't work properly. Therefore, it generates a need of a number representation with decimals for VHDL. For this matter, there are two options available.

First option would be to download an extended package that supports fixed or floating comma data types. Learning syntax rules for custom packages takes time and even after

[1] 4 bit numbers can represent from 0 to 15 in normal binary and from -8 to 7 in two's complement.

that there can be some unsolvable synthesis problems with the developing software, since the package is not official. That being said, it's highly recommended to go for a second option.

The second option consists on making a few operations to customize the signed data type and use it like a fixed comma data type. Let's go back to 4 bit numbers for the explanation, e.g. number $7$ in decimal notation.

$$0111_{BIN} = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7_{DEC}$$

If we add  negative powers at the right of the zero power, where the comma would be located, we can obtain a kind of fixed comma representation.

$$0100\,(.)0111_{BIN} = 2^2 + 2^{-2} + 2^{-3} + 2^{-4} = 4 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} =$$

$$= 4 + \ 0.25 + 0.125 + 0.0625 \ = \ 4.4375_{DEC}$$

Now it's possible to have $n$ decimals after the comma, being the resolution of the part after the comma $\frac{1}{2^n}$ .

## 4. A quick introduction to Cognitive Sensorimotor Loops (CSLs)

The original CSL used in [1] by Prof. Dr. Hild is based on an electronic circuit with two main parts: a 1 bit delta-sigma ($\Delta\Sigma$) modulator and a H-bridge driver connected to a Complex Programmable Logic Device (CPLD).

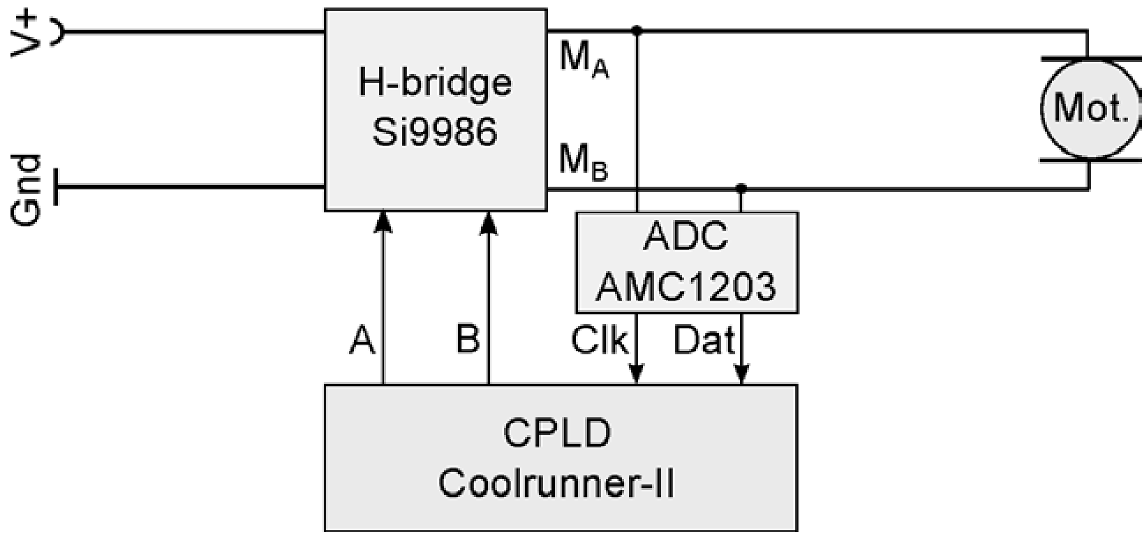


**Figure 4: System overview used by Prof. Dr. Hild in [1]. (figure imported from [1])**

The $\Delta\Sigma$ modulator (AMC1203) works as an analog-to-digital converter (ADC)[5]. It is used to measure the back electromotive force[2] (back EMF) in the terminals of a DC motor. The back EMF is a force that opposes to the electric current that induced it, as described in Faraday's law of induction and Lenz's law. In DC motors, this back EMF is proportional to the speed of rotation and therefore it can be used to infer the spinning speed. Applying a voltage in the terminals of a DC motor will produce the motor to start spinning and thus it will create a back EMF, which can be measured as a voltage with the opposite direction to the voltage applied. If the source that applies the voltage is disconnected and connect a multimeter to those same terminals, the multimeter will measure then a voltage proportional to the back EMF and therefore the rotation speed will be inferred. With this principle, the CSL operates first measuring the back EMF and then driving in relation to the magnitude of the force in such a way that the DC motor *serves both as an actuator and sensor* -[1] and it will fight against the forces applied such as gravity or external pushes.

The H-bridge driver is entrusted to give the motor a particular power to rotate with a certain speed. This H-bridge topology allows to run the DC motor in both directions among two other features: brake and coast. Braking the motor happens when the driver outputs a low-level logic value to both terminals. In other words, the terminals are short-circuited and the motor will eventually stop. To coast the motor, the driver outputs a high impedance (Hi-Z) logic level, which is in practice an open circuit. The behavior of the motor will then be as nothing is connected to its terminals.

[2] Sometimes also addressed as counter-electromotive force (counter EMF).

The driver can be powered with a single unipolar positive voltage for both driving directions, which is a major advantage. The CSL algorithm drives the motor by using the Pulse Width Modulation (PWM) technique. This means that the power that receives the motor depends on the duration of the pulses of an square signal:

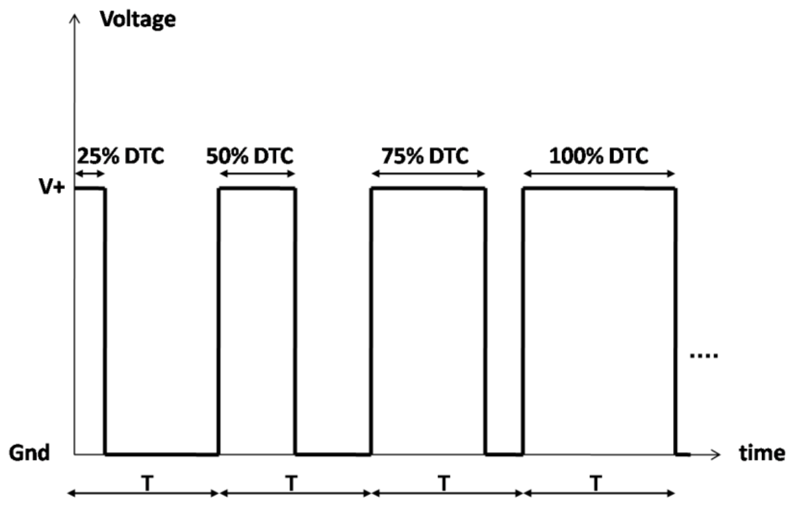


**Figure 5: Pulse Width Modulation example**

The above figure shows different duty cycles (DTCs) for the square signal with period $T$. The higher the magnitude in the force applied to the motor, the higher the DTC on the PWM signal applied by the H-bridge driver will be and therefore the faster the rotation speed of the motor will be.

The CPLD that run the original sensorimotor algorithm in [1] has been substituted in this system for a Field Programmable Gate Array (FPGA) , which has a vast amount of resources compared to CPLDs. The late CSLs are based on an electronic circuit connected to a development board via one of its Pmod inputs. This board, which has among other elements the previously mentioned FPGA, is configured to behave like a certain specific hardware by the use of the hardware description language VHDL[3]. In [1] the CPLD ran a Finite State Machine (FSM) with two states: sense and drive.

In the sense phase, the ΔΣ-Modulator measures the back EMF and transmits a binary signal to the CPLD or FPGA. This sense phase, which is constant, lasts 10ms. In order to have a correct back EMF measure, the H-bridge must be configured to coast position, otherwise the measure will be compromised when a voltage is applied to the terminals. The FPGA gathers the data until the sense phase is finished. Then, a transition in the FSM will happen, entering the drive phase. [1]

In the drive phase the algorithm will decide, depending on the back EMF gathered in the sense phase, whether or not to drive the motor and for how long. If the back EMF is positive, it means that the force that produced it was negative. The algorithm drives the motor "Forward" when the back EMF is positive and "Reverse" when it's negative. This

[3] in this case it's used VHDL. There are other hardware description languages e.g. Verilog, which was the language used by Prof. Dr. Hild in [1].

way the CSL drives the motor opposing to the external force that produced it when it was collected in the sense phase. The other issue is how much time in which the drive phase occurs (since it's not constant as the sense phase), in other words, the time in which the motor is being driven. Here is where the PWM takes place. Depending on the magnitude of the back EMF measured in the previous sense phase, the driving time will be lower or higher, hence and for all practical purposes, a PWM. [1][5][6]

These sense-drive transitions are in constant loop during the operation and producing the desired behavior.

# 5. System overview and components

This chapter contains a brief description of the elements that form the system and the connections between them.

## 5.1. Connection overview

The figure below shows how the inputs and outputs of each component are related, as well as the cables used.

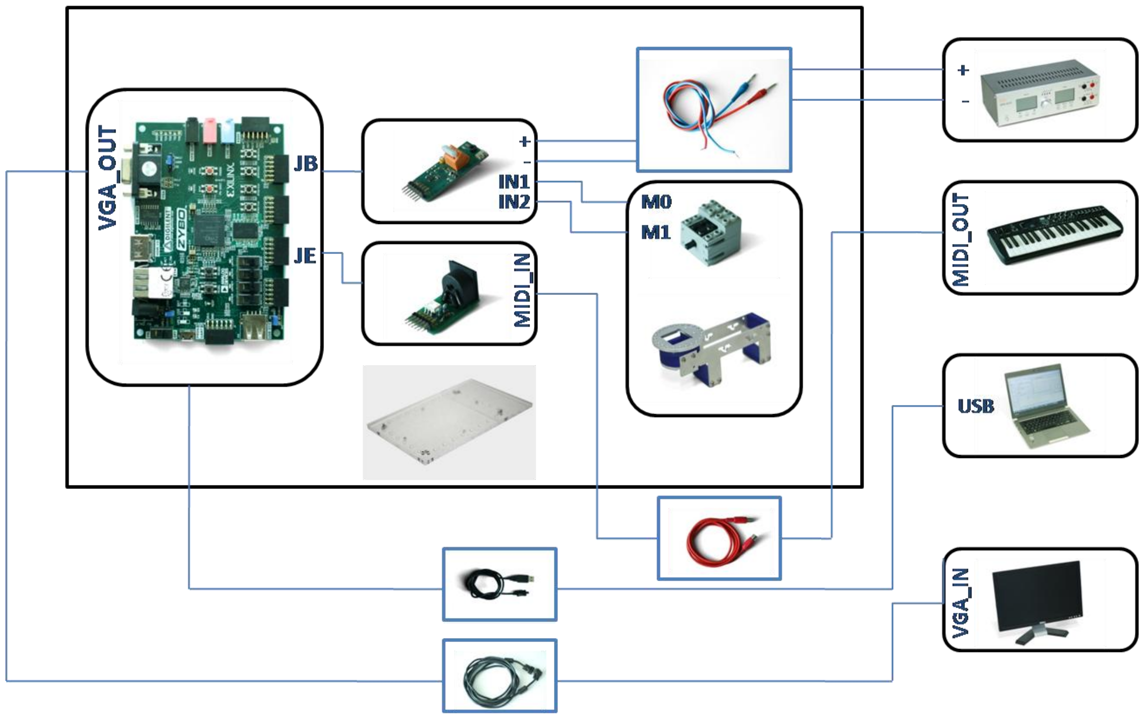


**Figure 6: Connection overview**

Where every element of the figure above is further described in the next sub-chapter

## 5.2.ZYBO Zynq-7000 Development Board

Development board by the manufacturer Xilinx, where the whole system converges around. It has a 4.400 logic slices FPGA (each slice with four 6-Input LUTs and 8 flip-flops), PLLs (to create clock sources), fast block RAM, 16-bits per pixel VGA port and Pmod connectors, among other elements. [3], [4]

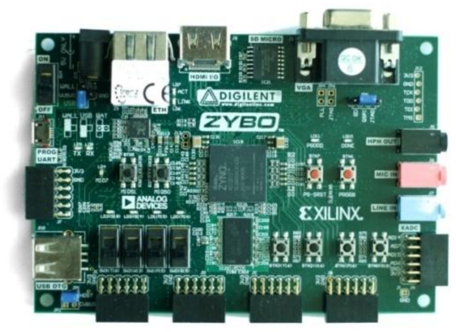


Figure 7: ZYBO board

## 5.3.Cognitive Sensorimotor Loop (CSL) Pmod

Circuit that combines both sensor and driver. It's based on a H-Bridge driver and a Delta-Sigma modulator. [1] [5] [6]

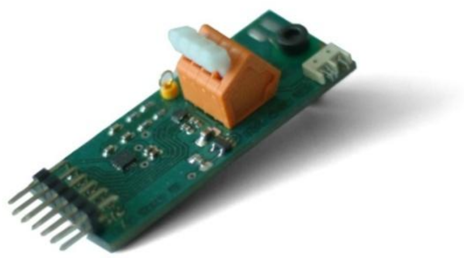


Figure 8: CSL Pmod

## 5.4.MIDI Pmod

Converts the DIN-5 connector into Pmod standard. It's used to receive MIDI messages sent from the MIDI keyboard.

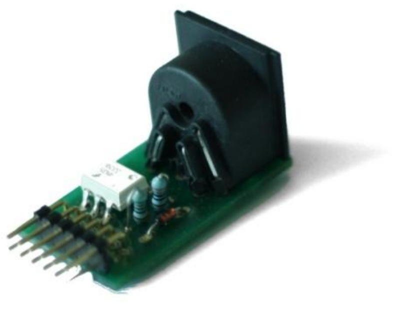


Figure 9: MIDI Pmod

## 5.5.miditech i2-Control 37 keyboard

Keyboard used to transmit MIDI messages via its analog faders.

Figure 10: MIDI keyboard

### 5.6. VGA Display

Used to visualize pertinent data. Commercial DELL display where the VGA modules will be running at 1024x768 resolution and 70Hz refresh.

Figure 11: VGA display

### 5.7. Lego Technic Motor 9 Volts

DC Motor run by the CSL Pmod.

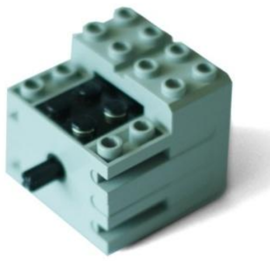


Figure 12: DC motor

### 5.8. Mounting Surface

estructural base for the ZYBO and the motor structure.

Figure 13: Mounting surface

### 5.9. ELV DPS-5315 DC-Power Supply

Provides DC power to the CSL Pmod and indirectly the motor.

Figure 14: Power supply

### 5.10. Motor structure

Fixes safely the motor against mechanical forces and torques.

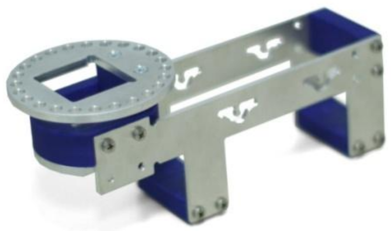


Figure 15: Motor structure

## 5.11. Wiring

Self-explanatory.

Figure 16: MIDI cable

Figure 17: VGA cable

Figure 19: USB cable

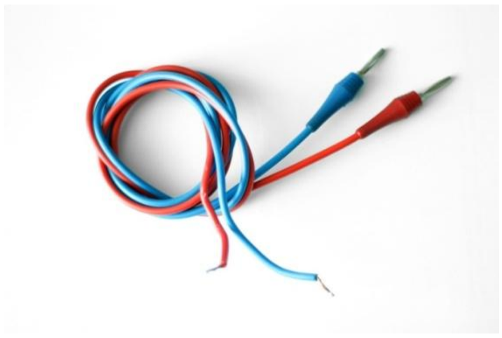


Figure 18: banana cable

## 5.12. PC

Used to run the Xilinx Vivado SW and program the ZYBO.

Figure 20: Laptop

## 6. Voltage integration. Drift in the measure

To measure the counter-electromotive force (counter EMF) of the motor, the CSL-PMod has an analogical-digital converter based on a Delta-Sigma Modulator (ΔΣ-Modulator).

This ΔΣ-Modulator generates a flow of bits depending on the differential voltage between terminals of the motor. Here's an example taken from the datasheet [4] to explain its operation:

| **Differential Voltage** | **Percentage of '1's** | **Percentage of '0's** |
|---|---|---|
| **256mV** | 80% | 20% |
| **0V** | 50% | 50% |
| **-256mV** | 20% | 80% |

**Figure 21: Differential Voltage-bit density of Delta-Sigma modulator**

This means that when the motor is at rest, the ΔΣ-Modulator should output 50% of '1's (high-level logic values) and '0's (low-level logic values). The problem is that, in practice, the percentage of '0's is bigger than 50% when the motor is at rest, and the captured value with VHDL is drifting from the one corresponding to 0V.

In VHDL coding, this simplifies in having a counter that is incremented when a '1' is received and decremented when a '0' is received.

```
elsif mdat='1'then
  if v_ena='1' and voltage<131071 then
    voltage<=voltage+1;
  end if;
else
  if v_ena='1' and voltage>-131072 then
    voltage<=voltage-1;
  end if;
end if;
```

As already stated, the counter will tend to zero when the differential voltage is 0V and it will tend to a positive value when the density of '1's is higher than the density of '0's.

To check the behavior of this drift in VHDL, a series of measures were conducted.:

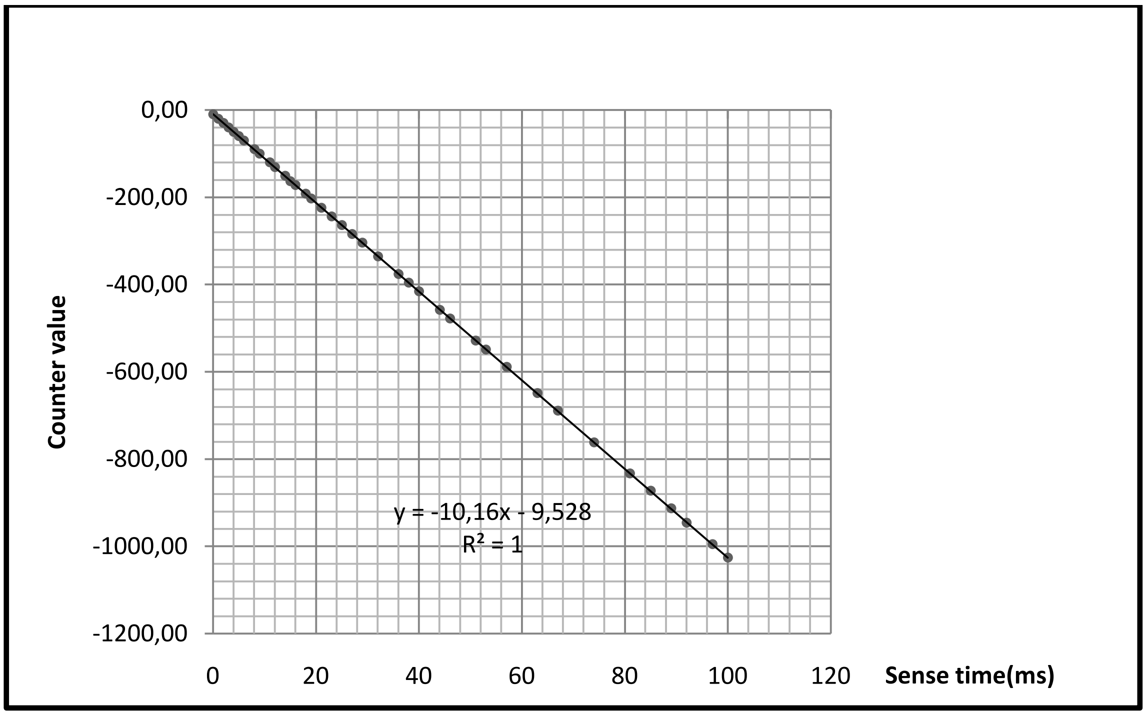


**Figure 22: Drift vs Sense-time**

The graph above shows that the drift has linear behavior depending on the sense time and therefore it can be compensated at the end of every sense phase, regardless of what time it has (remember the sense phase is a constant parameter in the CSL operation).

It is also interesting to calculate the voltage difference that is producing the drift. If we pick for example a sense time of 68ms the value of the counter is -689. This means that there were 689 more '0's than '1's received. The total number of bits sent in 68ms with a 10Mhz clock is 680000. The problem is then reduced to a simple linear system with two equations:

$$\begin{cases} x + y = 680000 \\ x - y = 689 \end{cases}$$

$x$: quantity of '0's

$y$: quantity of '1's

The solution to $y$ is $y = 339655.5$, therefore the percentage of '1's is

$$339655.5/680000 \approx 49.9493\%$$

Because the behavior is linear, the differential voltage that causes the drift can be represented with a linear form equation:

$$y = mx + n$$

Being $m$ the slope of the curve and $n$ the point at which the line crosses with the y-axis.

Particularizing the equation to this case:

$$V_{drift} = \left(\frac{Vmax - Vmin}{P(1's)max - P(1's)min}\right) * P(1's) - V_o$$

The $n = V_o = -426.67mV$ is calculated by a known point and the slope $m$.

$P(1's)$ is the density of '1's between 0 and 1. $P(1's) = 0.499493$

$$V_{drift} = \left(\frac{256mV - (-256)mV}{0.8 - 0.2}\right) * 0.499493 - 426.67mV$$

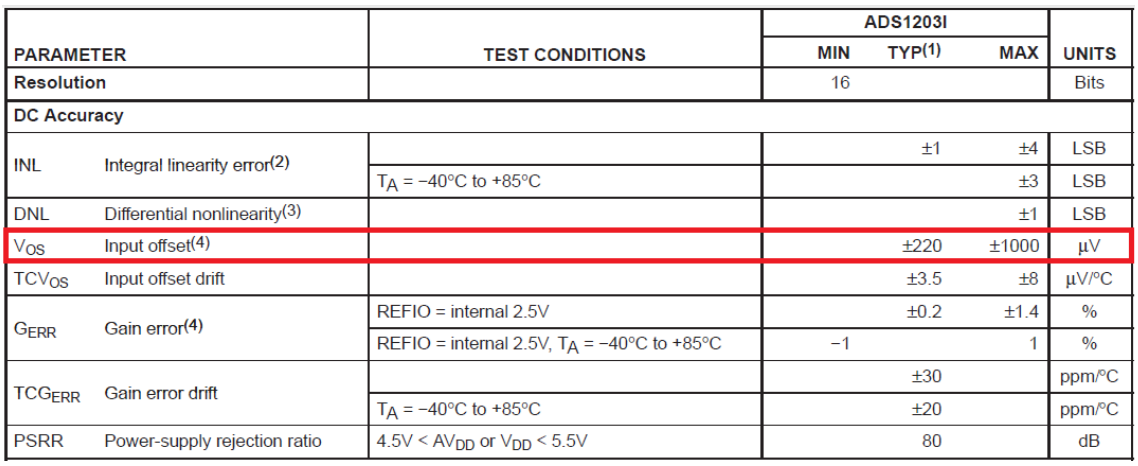
$$V_{drift} \approx -436\mu V$$

This small value is enough to produce a drift in the integrative control of the CSL-algorithm. If the counter value is not reset over time the differential voltage will tend to decrement the counter until its negative saturation.

To end this subject, another interesting matter is to figure out the origin of these

-426μV.

The answer to this matter can be found in the ΔΣ-Modulator datasheet [4]:



| PARAMETER | | TEST CONDITIONS | ADS1203I | | | UNITS |
|---|---|---|---|---|---|---|
| | | | MIN | TYP[1] | MAX | |
| **Resolution** | | | 16 | | | Bits |
| **DC Accuracy** | | | | | | |
| INL | Integral linearity error[2] | | | ±1 | ±4 | LSB |
| | | $T_A = -40°C$ to $+85°C$ | | | ±3 | LSB |
| DNL | Differential nonlinearity[3] | | | | ±1 | LSB |
| $V_{OS}$ | Input offset[4] | | | ±220 | ±1000 | µV |
| $TCV_{OS}$ | Input offset drift | | | ±3.5 | ±8 | µV/°C |
| $G_{ERR}$ | Gain error[4] | REFIO = internal 2.5V | | ±0.2 | ±1.4 | % |
| | | REFIO = internal 2.5V, $T_A = -40°C$ to $+85°C$ | −1 | | 1 | % |
| $TCG_{ERR}$ | Gain error drift | | | ±30 | | ppm/°C |
| | | $T_A = -40°C$ to $+85°C$ | | ±20 | | ppm/°C |
| PSRR | Power-supply rejection ratio | $4.5V < AV_{DD}$ or $V_{DD} < 5.5V$ | | 80 | | dB |

**Figure 23: Delta-Sigma datasheet extract**

The explanation is that the drift comes from the input offset in the ΔΣ-Modulator, hence the linearity of the drift (a constant offset produces a linear drift over time).

# 7. Drift compensations. Fixing the ΔΣ-Modulator offset

This chapter contains the designs tried in order to correct the ΔΣ Modulator offset

## 7.1.Dead Zone

The easiest way to solve the problem with the ΔΣ offset is to introduce a dead zone. The characteristic curve of a dead zone is as follows:

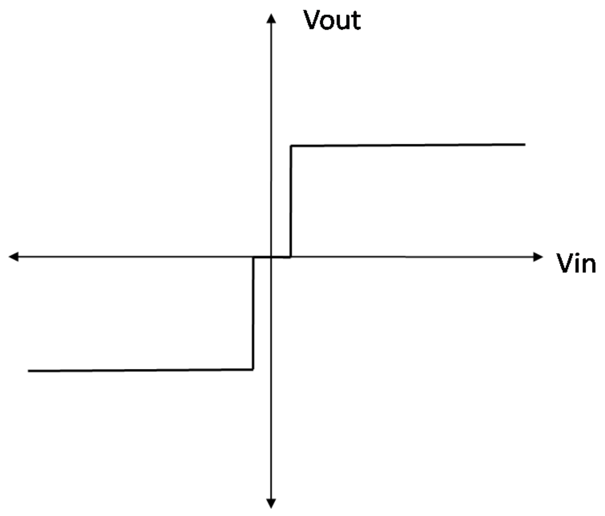


**Figure 24: Dead zone characteristic curve**

The graphic shows that no output will be produced for certain small values.

Applied to the VHDL sensorimotor loop, this consists of setting a threshold in the integrative control. The algorithm checks the value of the integral after finishing a sense phase. In this case, the offset is negative. If the integral value is below the threshold imposed by the dead zone, the control will jump to the drive phase with the timer set to '0' and there will be no driving motion. Introducing a dead zone on the system has an important disadvantage. It will cause the integral value to be close to the threshold value when no external forces are applied. In the case a force were applied, the algorithm would react faster to one direction more than the other. This is because in order to drive ( in this case counterclockwise) the integral value needs to rise from the threshold up to the energy threshold set up by Prof. Dr. Hild in [1], therefore  sensitiveness is lower on this direction. Additionally, it's a good choice to reset the integral variable. This way the motor will be as sensitive in both directions. Another matter is that the drift is going to happen only in one direction so the best option is to apply an asymmetric dead zone, in this case that means to displace the characteristic curve to the negative side, as seen on the figure:

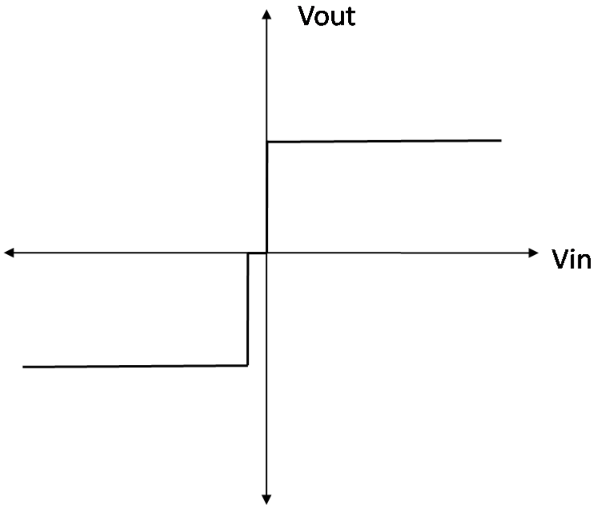


**Figure 25: Asymmetrical Dead Zone characteristic curve**

The reason for this, it's that there is no interest in having a dead zone for the positive values because the offset is always going to be, in this case, negative. This way, It's achieved at least the best possible response in one of the two directions of the motor.

## 7.2.Linear compensation

Being proved in the 6th chapter, that the drift is linear on a time basis, it can be corrected by capturing the offset for a certain sense time and then apply basic geometry (Thales theorem) to calculate the compensation needed to correct the drift for any sense time:

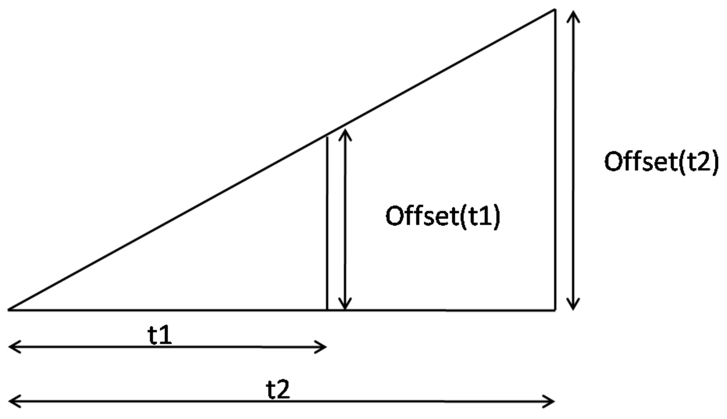


Figure 26: Compensation triangle

$$\left(\frac{Offset(t_1)}{t_1}\right) = \left(\frac{Offset(t_2)}{t_2}\right)$$

$$Offset(t_1) \cdot t_2 = Offset(t_2) \cdot t_1$$

In VHDL:

```
Compensation_27<=std_logic_vector((Offsetcapture*
(signed("0"&tsense)+1))/(signed("0"&captureTime)/9999));
Offsetcompensation <= Compensation_27(17 downto 0);
```

Where `Offsetcapture` and `captureTime` are $Offset(t_1)$ and $t_1$ respectively and `Offsetcompensation` and `tsense` are $Offset(t_2)$ and $t_2$.

`Compensation_27` is just an auxiliary variable to deal with the operation without overflowing.

The advantage of this method against the dead zone is that sensitiveness of the CSL is the same for both directions. The algorithm has the same response on the motor no matter the value of the voltage.

A last thing to point out is the error in the calculation of the compensation. In practice, this translates into not fixing completely the drift on the measure.

## 7.3.Measures module. Finite State Machine

There's a more stylish solution to fix finally and for all the drift caused by the offset of the ΔΣ-Modulator. The "measures.vhd" module can be modified to capture the offset for a certain sense time and then subtract the offset value every time a sense phase ends. That way, when the drive phase starts, the voltage will be already compensated.

As said before, the CSL algorithm [1] is based on an integrative control. The integration is never reset on a time basis. That means the drift caused by the ΔΣ Modulator offset can only be compensated once a sense phase is finished. A good way to overcome this problem would be building the integral through the addition of average voltages.

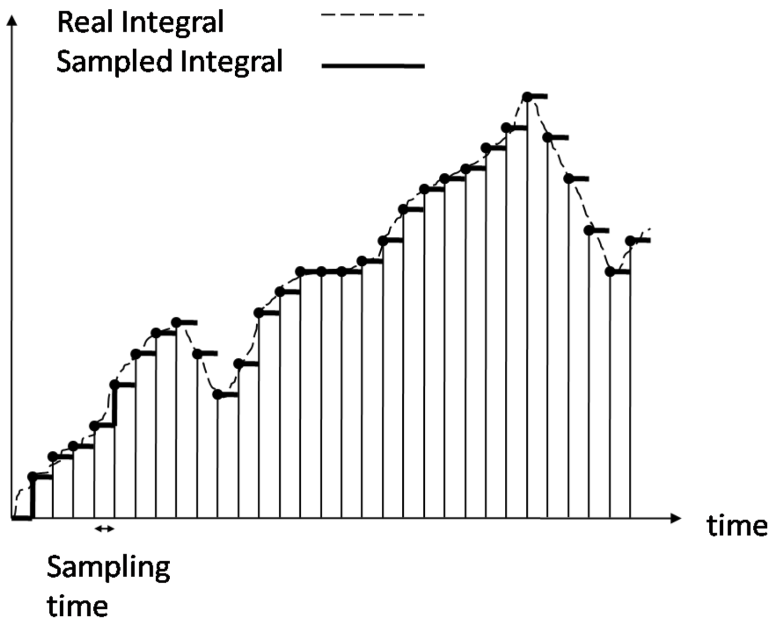


**Figure 27: Measure sampling**

The average voltage is a representative value of the back EMF measured during a time window (sampling time). The integral[4] is built by the contribution of all previous voltages.

The advantage of this method is that it's possible to compensate the offset before the integral is obtained and therefore the algorithm will work with an already adequate measure without drift.

Once this is understood, fixing the offset in VHDL becomes a trivial problem. A Finite State Machine (FSM) with two states can (1st) capture the offset. This is just a normal voltage capture when no external forces are applied to the motor. And (2nd) obtain the voltage value, depending on the sense time.

[4] Of course writing about "real integral" is not an accurate term, considering that the analogical differential voltage is being sampled with a 10MHz clock. From now on, the reader should understand this term as an abbreviation of the correct one.

The next figure shows the FSM flow diagram:

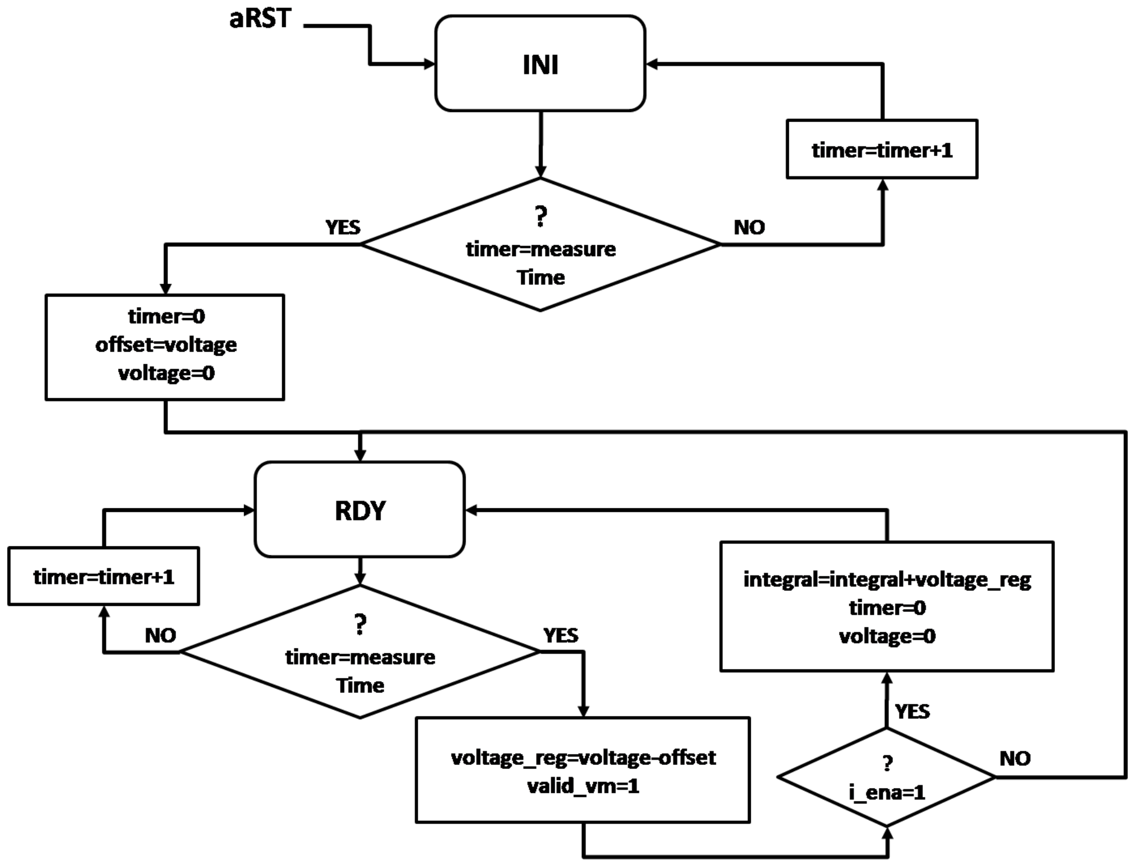


**Figure 28: Measure FSM flow diagram**

This flow diagram is a simplified version of the FSM implemented in VHDL. First, pressing the switch "SW0" of the ZYBO produces an asynchronized reset "aRST" which resets the registers and sets the FSM into "INI" state. Every rising edge of the clock, the condition "timer=measureTime" is checked, while the algorithm is gathering the bits from the ΔΣ Modulator and accumulating the voltage signal. If the condition is true, the timer and voltage registers are reset and the offset captured. Then the FSM jumps to "RDY" state. Once again, the condition is checked every rising edge of the clock. If the condition is satisfied, the output voltage "voltage_reg" gets compensated with the offset and the integral is build adding the current voltage to the accumulation of the previous ones.

In order to work properly, it's important to add some additional conditions not included in the diagram to prevent overflows in the values of "integral" and "voltage" before the '+' and '-' operations:

```
if ((voltage(17)&voltage)-(offset(17)&offset)<131071)
and ((voltage(17)&voltage)-(offset(17)&offset)>-131072) then
    voltage_reg<=voltage-offset;
end if;
```

and also:

```
if ((integral(17)&integral)+(voltage_reg(17)&voltage_reg)<131071)
and ((integral(17)&integral)+(voltage_reg(17)&voltage_reg)>-131072)
and (voltage_reg>4 or voltage_reg<-4) then
    integral<=integral+voltage_reg;
end if;
```

This guarantees that "voltage_reg" or "integral" are not overflowed. Because both are signed variables, a way to achieve it, is to concatenate the MSB to the whole value and increment the range as it would be done in a Two's Complement number. If the value is not between the range of a signed 18 bit variable [-131072, 131071] the operation doesn't take place (it does computationally, but the result is not saved into the variable register).

Additionally, a small dead zone in "voltage_reg" (between 4 and -4 units per sense phase) is added in order to improve the correction. This fixes the small fluctuations produced in the representative voltage and stops completely the measure drift.

# 8. Common modules description

This chapter covers a brief description of the modules that are common to this thesis and the other two theses mentioned in the abstract.

## 8.1. Clock sources

They constitute the clock input signals for all the different sequential elements in the system.

### 8.1.1. ClockTree.vhd (Inherited)

Generates and outputs different clocks from the 125MHz oscillator. These are required to be routed in dedicated tracks to prevent skew.

**Inputs:**

Clk_125MHz: source from external oscillator

**Outputs:**

Clk_75MHz: clock input for the VGA

Clk_12_288MHz: not used

Clk_3_072MHz: not used

Clk_500kHz: not used

Clk_250kHz: clock input for the MIDI interface

Clk_48kHz: not used

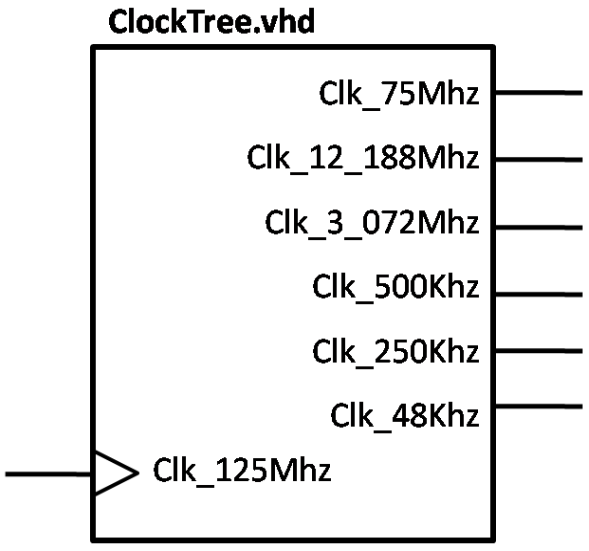


Figure 29: ClockTree module interface

## 8.2.Visual modules

In order to visualize pertinent data, some modules were added to the project. These VHDL modules define shapes and numbers for a 1024x768 VGA standard. Some of the modules were previously made to this project while the others were made to extend this tool.

### 8.2.1. VGA1024.vhd (BP[5])

Generates the pertinent synchronization signals to display data on a VGA with 1024x768/ 70Hz resolution points and 16bits pixel color.

**Inputs:**

Clk_75MHz: clock input for the VGA

VGA_Color[ 15..0]: inputs the color of the actual pixel.

VGA_BackColor[ 15..0]: inputs the color for the background.

**Outputs:**

VGA_Addr [21..0]: vectors the position in the screen for the pixel.

VGA_R [4..0]: depth for the red color

VGA_G [5..0]: depth for the green color

VGA_B [4..0]: depth for the blue color

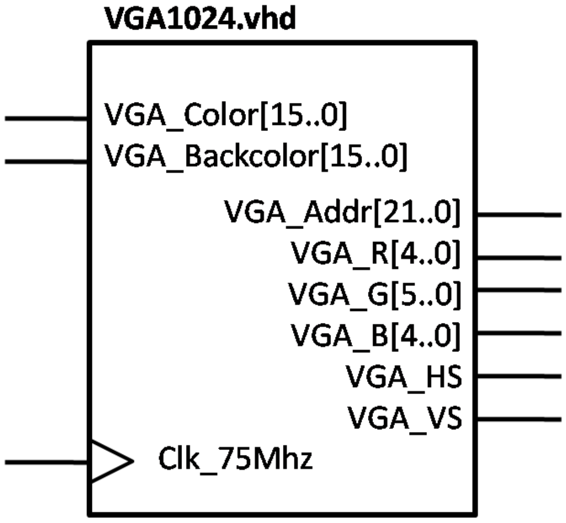


**Figure 30: VGA1024 module interface**

VGA_HS: synchronization signal that moves the pixel to one position right.

VGA_VS: synchronization signal that moves the pixel to one position down.

[5] implemented by Benjamin Panreck

### 8.2.2. GridPaper.vhd (Inherited)

Displays a grid for the VGA. It prints the current pixel of the grid with the background color when no other module outputs a pixel.

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

**Outputs:**

VGA_BackColor[ 15..0]: outputs the color for the background.

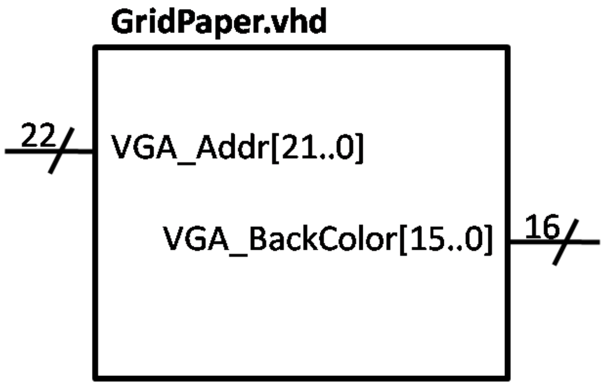


**Figure 31: GridPaper module interface**

### 8.2.3. ASCII_canvas.vhd (PdMN[6])

Groups together characters in ASCII. Displays text information on the VGA screen.

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

push_vga: writes "PUSH" on the screen when the logic level is '1'.

**Outputs:**

Color_out [15..0]: indicates the color of

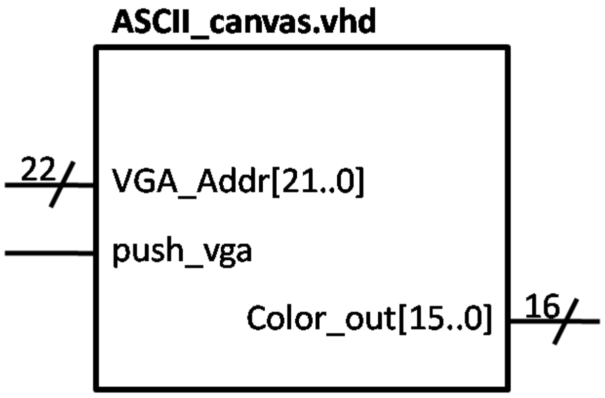


**Figure 32: ASCII_canvas module interface**

the actual pixel to print.

[6] implemented by Pablo de Miguel Nogales

### 8.2.4. ASCII_sign.vhd (PdMN)

Contains the data on how to print ASCII characters on the screen.

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

ASCII [6..0]: indicates the ASCII character to print on the screen

**Outputs:**

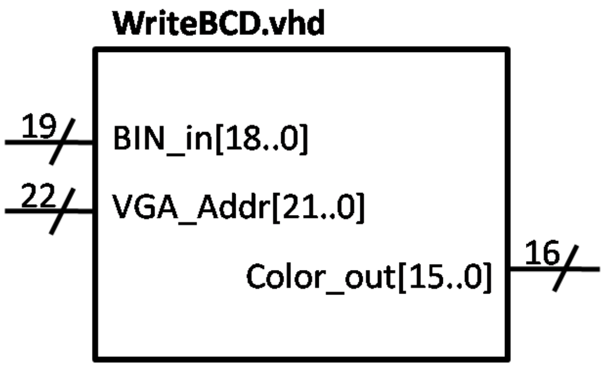


**Figure 33: WriteBCD module interface**

Pixel: sets a high-level logic value when part of an ASCII character is supposed to be printed on the actual position.

### 8.2.5. WriteBCD.vhd (PdMN)

Converts a binary number to BCD using the double-dabble algorithm.

**Inputs:**

BIN_in [18..0]: a 19 bit number in binary

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

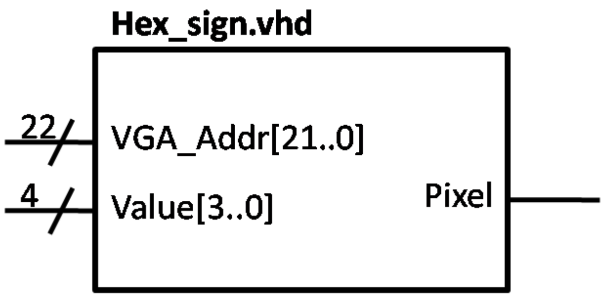


**Figure 34: Hex_sign module interface**

**Outputs:**

Color_out [15..0]: outputs the color of the BCD number.

### 8.2.6. Hex-Sign.vhd (Inherited)

Contains the information on how to print a hexadecimal number on the VGA

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

Value [3..0]: indicates the hexadecimal number (0 to F)

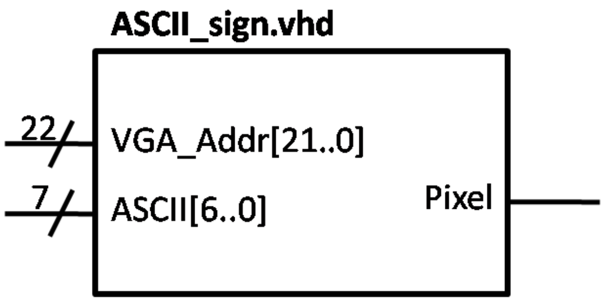


**Figure 35: ASCII_sign module interface**

**Outputs:**

Pixel: Pixel: sets a high-level logic value when part of a hexadecimal number is supposed to be printed on the actual position.

### 8.2.7. WriteSigned.vhd (PdMN)

Converts an 18bit number to a signed decimal value with 6 digits for displaying it on the screen.

**Inputs:**

BIN_in [17..0]: binary number interpreted in Two's complement.

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

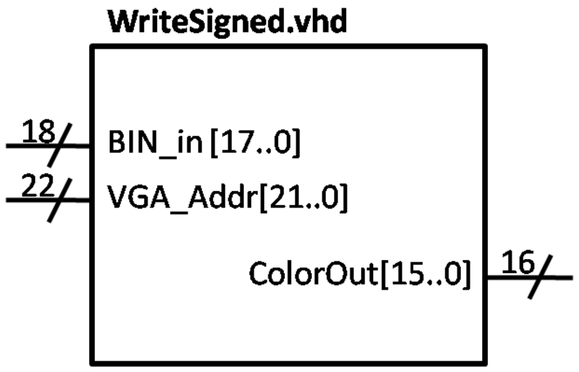


Figure 36: WriteSigned module interface

**Outputs:**

Color_out [15..0]: outputs the color of the 6 digit decimal number.

### 8.2.8. ShowScope.vhd (Inherited)

Draws a time graph on the screen based on a 7 bit value.

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

Value [6..0]: 7 bit binary value to show on a graph

**Outputs:**

VGA_Color [15..0]: outputs the color of the graph.

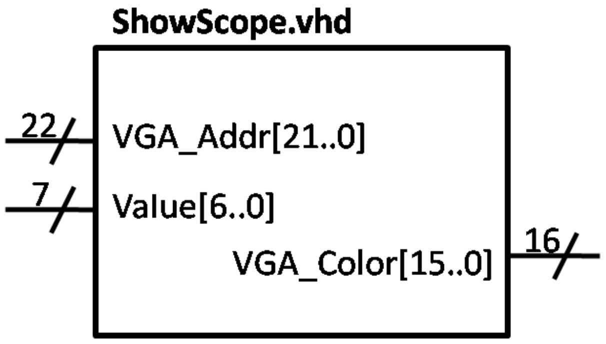


Figure 37: ShowScope module interface

### 8.2.9. ShowVBar.vhd (Inherited)

Draws a vertical bar on the screen with its size depending on a 7-bit value.

**Inputs:**

VGA_Addr[21..0]: vectors the position in the screen for a pixel.

Value [6..0]: 7 bit binary value that sets the size of the bar.

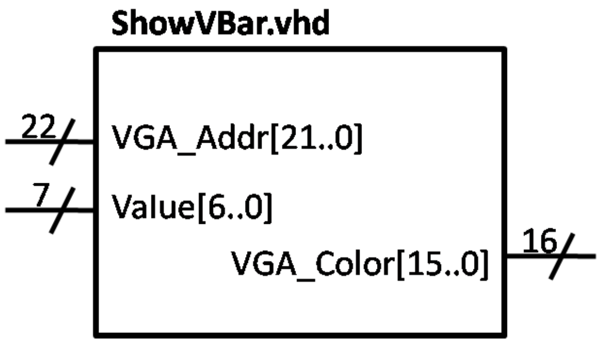


Figure 38: ShowVBar module interface

**Outputs:**

VGA_Color [15..0]: outputs the color of the bar.

## 8.3.MIDI data acquisition

These modules define all the necessary tools to obtain configuration parameters from the user via MIDI, in this case the keyboard faders.

### 8.3.1. GetMIDI.vhd (Inherited)

Defines the MIDI standard to receive messages, in this case, the keyboard.

**Inputs:**

ClkMIDI: MIDI clock input. Must be 500KHz

MidiIn: input for the MIDI data received.

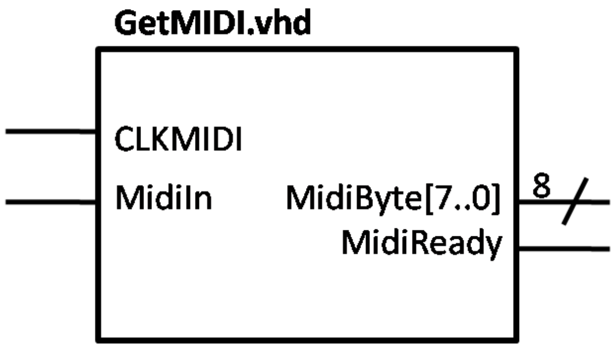


Figure 39: GetMIDI module interface

**Outputs:**

MidiByte: received message converted to 8-bit binary.

MidiReady: sets a high-level logic value when the reception is finished.

### 8.3.2. faders.vhd(PLG[7])

Implements the keyboard's faders

**Inputs:**

midibyte [7..0]: received MIDI message converted to 8-bit binary.

midiready: indicates that reception is finished

**Outputs:**

fader1-9: outputs values between 0 and 127 from the faders.

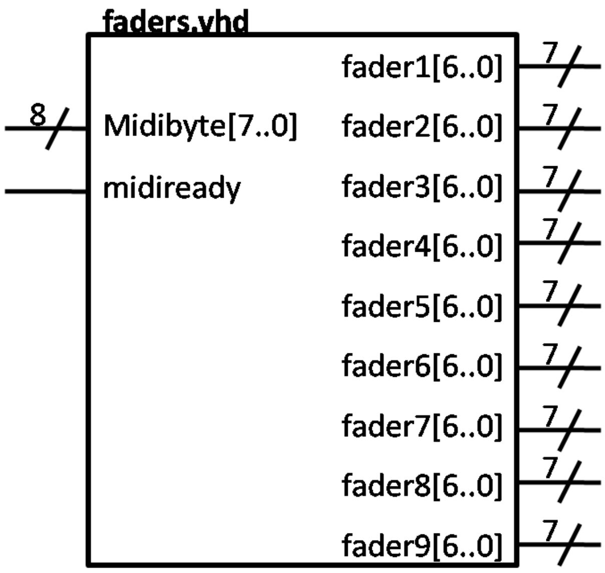


**Figure 40: faders module interface**

[7] Pablo Lezcano Giménez. A full description of this module is explained the next chapter.

## 9. Faders module. Keyboard's battery of faders.

In order to introduce any kind of parameters in real time, it's very useful to have the whole fader battery of the keyboard implemented in a module. Each fader can be identified with an specific 8 bit hexadecimal address. Although the module is configured by default for a *miditech i2-Control keyboard*, it can work for any other MIDI keyboard if the addresses are known.

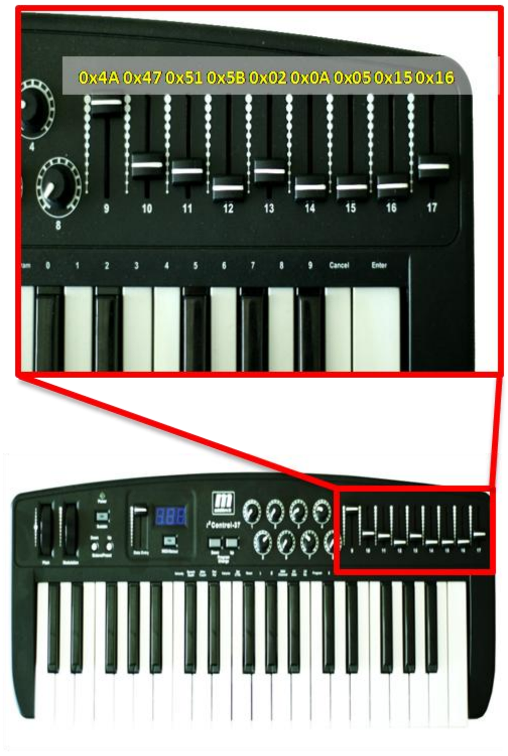


**Figure 41: faders addresses of the MIDI keyboard**

In VHDL, the address is set by default by the use of "generic" syntax:

```
Generic( -- default adresses for the faders in a miditech i2 control-
37 keyboard
            fa1: std_logic_vector( 7 downto 0):=x"4A";
            fa2: std_logic_vector( 7 downto 0):=x"47";

                  ...........etc.............
```

That way, if the keyboard changes, there's no need in internally modifying the file. The new addresses can be set at the module definition using "generic map" syntax:

```
faderControl:     entity work.faders
generic map  ( add1, add2, add3, add4, add5, add6, add7, add8, add9)
port map     (MidiByte, MidiReady, fader9, fader10, fader11, fader12,
fader13, fader14, fader15, fader16, fader17);
```

Where "`addX`" is the 8-bit hexadecimal address of each fader.

The FSM is as follows:

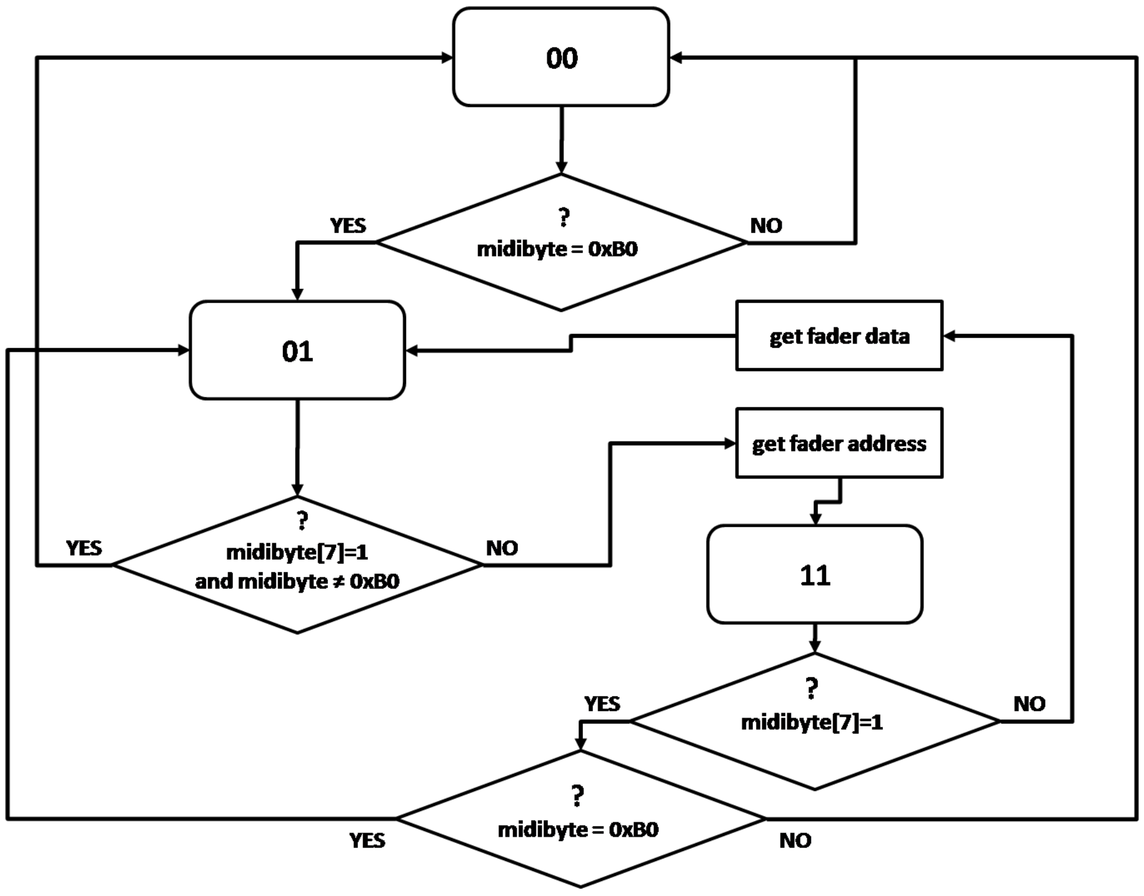


**Figure 42: fader battery FSM**

First, the FSM checks if there's a change event in the faders ("midibyte = 0xB0"). If the condition is true, it means that the message is in fact a "Control Change" message [7] and the FSM can jump to the next state "01". Then the algorithm waits for the next byte, in other words, until the byte has changed. The MIDI standard compels to have a '0' in its MSB for the first data byte [7]. If both of them are correct, the FSM can spot which fader was moved by the address and jump to state "11". Finally is checked once again the MSB of the byte [7] and if everything is correct the FSM acquires the value between 0 and 127 from the fader (the MSB is always '0'[7]) and jumps back to "01".

In VHDL, the FSM reads as follows:

```
wait until rising_edge (MidiReady);
      case state is
      when "00"=> -- waiting for controller message (hex value "B0")
            if midibyte = x"B0" then
                  state <="01";
            end if;
      when "01"=>
            if midibyte(7) then
                  if midibyte /= x"B0" then
                        state<="00";
                  end if;
            else
                  ctrlnr<=midibyte(6 downto 0);
                  state<="11";
            end if;
      when "11"=>                   -- waiting for controller value
            if midibyte(7) then         -- command byte
                  if midibyte = 8x"B0" then
                        state <= "01";
                  else
                        state <= "00";
                  end if;
            else
                  case ctrlnr is    --choosing fader
                  when fa1=>
                        fader1<=midibyte(6 downto 0);
                  when fa2=>
                        fader2<=midibyte(6 downto 0);
                  when fa3=>
                        fader3<=midibyte(6 downto 0);
                  when fa4=>
                        fader4<=midibyte(6 downto 0);
                  when fa5=>
                        fader5<=midibyte(6 downto 0);
                  when fa6=>
                        fader6<=midibyte(6 downto 0);
                  when fa7=>
                        fader7<=midibyte(6 downto 0);
                  when fa8=>
                        fader8<=midibyte(6 downto 0);
                  when others=>
                        fader9<=midibyte(6 downto 0);
                  end case;
            state <= "01";
            end if;
      when others=>
            state<="00";
      end case;
    end process;
```

Let's analyze the process step by step:

```
wait until rising_edge (MidiReady);
      case state is
      when "00"=> -- waiting for controller message (hex value "B0")
            if midibyte = x"B0" then
                  state <="01";
            end if;
```

The process is invoked only when a rising edge of the signal "MidiReady" occurs. The first state "00" waits for the first byte to indicate a "Control Change" event ("midibyte = 0xB0") [7]. If that happens, the FSM will jump to the state "01".

```
when "01"=>
            if midibyte(7) then
                  if midibyte /= x"B0" then
                        state<="00";
                  end if;
            else
                  ctrlnr<=midibyte(6 downto 0);
                  state<="11";
            end if;
```

The FSM gets ready for the second byte. The standard compels the MSB of the byte "midibyte(7)" to be '0' [7]. If the condition is not satisfied, the FSM will jump to the initial state "00" or stay at "01" if the byte is considered again as an status byte with a control event. If none of this happened, the byte correctly indicates the address of the corresponding fader and the FSM can jump to the last state.

```
when "11"=>                   -- waiting for controller value
            if midibyte(7) then         -- command byte
                  if midibyte = 8x"B0" then
                        state <= "01";
                  else
                        state <= "00";
                  end if;
            else
                  case ctrlnr is    --choosing fader
                  when fa1=>
                        fader1<=midibyte(6 downto 0);
                  when fa2=>
                        fader2<=midibyte(6 downto 0);
                  .................etc................
                  when others=>
                        fader9<=midibyte(6 downto 0);
                  end case;
            state <= "01";
            end if;
      when others=>
            state<="00";
      end case;
    end process;
```

The "11" state also checks the MSB of "midibyte" as required by the standard [7]. Then the first data byte (ctrlnr) is compared with the addresses of the faders which are either generically stored or specified in the module definition. When the fader address matches the one captured in "ctrlnr", the second data byte gets registered in its respective variable "faderX".

There's also one last thing to point out at the end of the code:

```
when others=>
            state<="00";
```

The reason for this is simple. In the absence of an asynchronous reset, the FSM could start in a unknown state. This forces it to jump to "00" and wait for the bytes to arrive.

# 10. Systems based on parameterized thresholds

## 10.1. System overview

As the name indicates, these models are all based on setting up a threshold that, when it's surpassed, will stop the normal operation of the sensorimotor loop. This is achieved using a module that receives the voltage together with the threshold parameter and decides if a push happened. If a push occurs, an output signal will be generated. The "ignore" signal is checked in the sense phase of the CSL algorithm. If the signal has high-level logic value, the driving phase will be skipped.

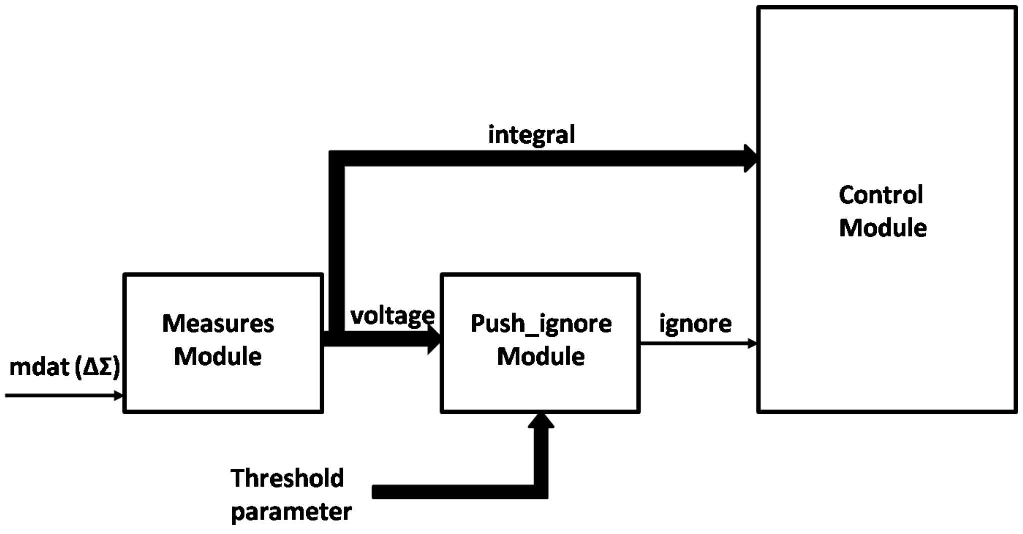


**Figure 43: Parameterized thresholds simplified diagram**

Where "mdat" and "ignore" are binary signals, "voltage" and "Threshold parameter" are 18 bit bus type signals.

Working with an external module to detect the pushes allows to implement the full behavior without needing to change the control module where the sensorimotor algorithm is executed and therefore no major changes are produced in the module.

## 10.2. Static threshold model

The first and simplest idea to come in mind on detecting pushes is to establish a voltage threshold. This consists on stopping the sensorimotor algorithm when a certain voltage is reached. It's important to understand that, when the word "voltage" is referred in this document, it means a representative quantity obtained in a time window based on the '1's and '0's received by the ΔΣ Modulator.

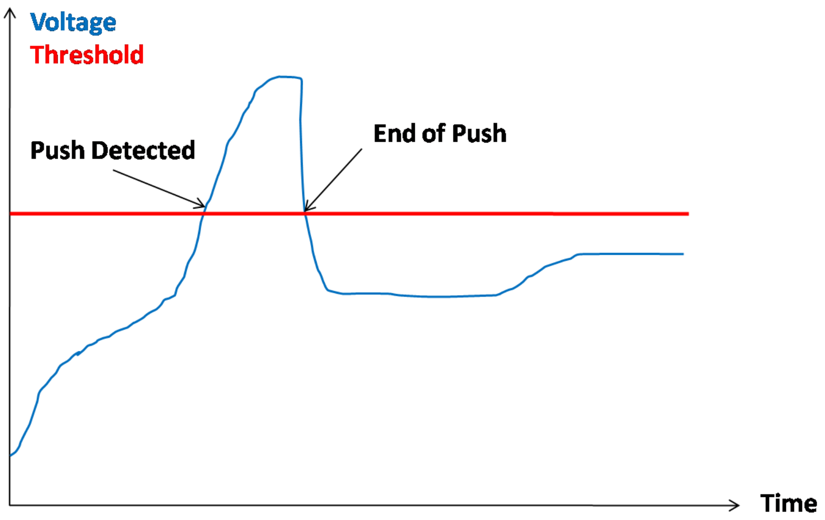


**Figure 44: Push detection with static threshold**

This way, the sensorimotor algorithm won't be operational between the "Push Detected" event and the "End of Push" event.

This solution is also very easy to implement in VHDL.

```
ignore<='1' when abs(voltage) > abs(threshold) else
        '0';
```

Which defines a combinational circuit, in this case a comparator, that changes the ignore output to '1' when the condition is satisfied. Of course the logic needs some time to propagate but this is minimal compared to the speed of the system.

Of course, a module this simple has many flaws. The first one to notice would be the detection of slow in time but high magnitude forces as pushes:

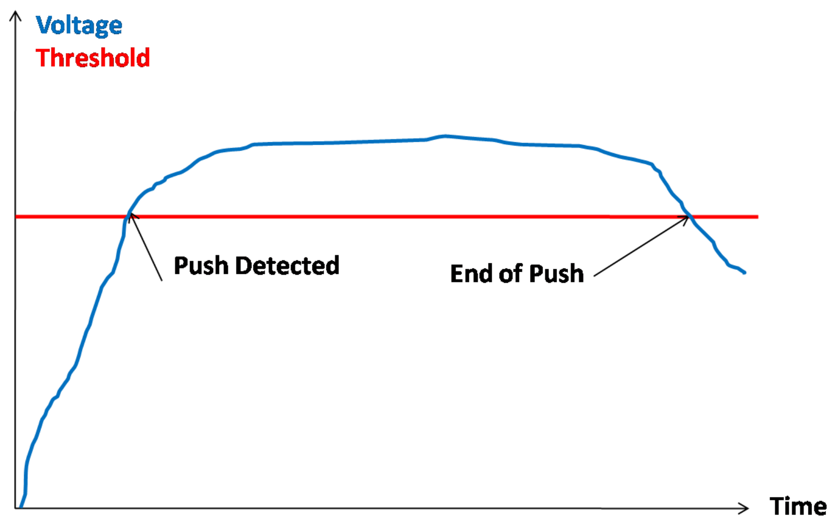


Figure 45: False push detection with static threshold

That could happen for example if the threshold is lower than the maximal voltage obtained in the pendulum motion, which occurs at $\pm 90°$ from its equilibrium positions.

The second flaw on this model would be the inability to detect pushes below the threshold. This means that pushes below a certain voltage won't be detected and translates into systems with low sensitiveness.

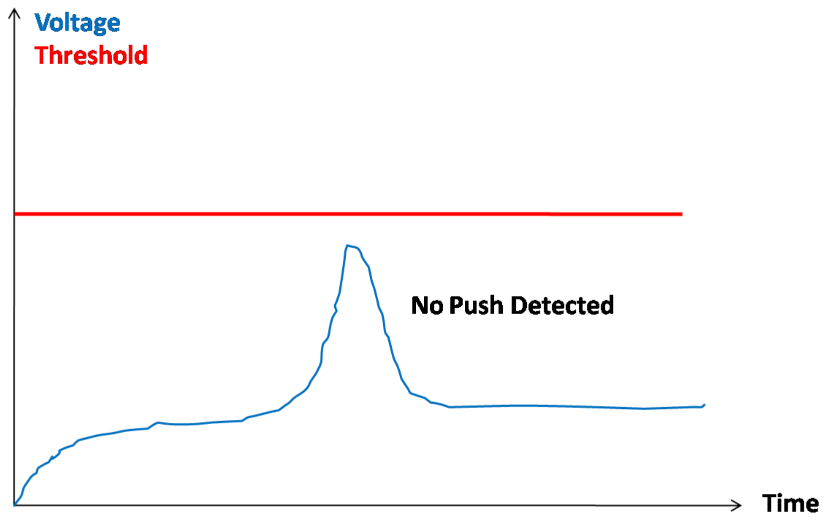


Figure 46: Missed push with static threshold

These two flaws reveal the main problem about using this model. As said before, the maximal voltage occurs at $\pm 90°$ from the equilibrium positions of the pendulum. In order to maximize the sensitiveness of the detection, the threshold should be set up with a value slightly higher than the maximal voltage so the control loop operates with no interruption when no pushes are produced. The problem is that if the system is heavier (e.g. Myon's joints), the maximal voltage required would be bigger and therefore a bigger threshold would also be needed. Pushes would need to be really strong in order to be detected and being too abrupt could come into making damage to the parts.

## 10.3. Incremental threshold model

In the last chapter were explained several flaws in the detection of pushes using a static threshold. One way to improve the module would be to detect the variation of the voltage within time. If this value is higher than the threshold, it will consider a push has occurred.

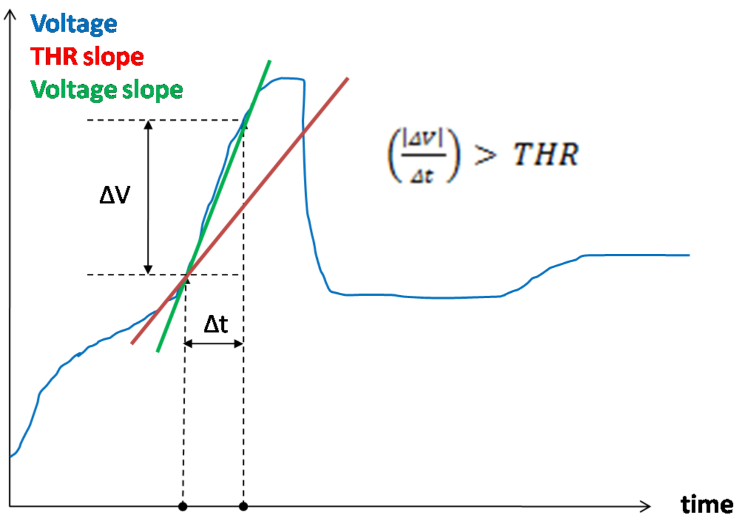


**Figure 47: push detection with incremental threshold**

In the upper figure can be seen the basics of the model. Several of the problems in the previous model are fixed. Now a push would have to be shorter in time than $\Delta t$ in order to go unnoticed. In practice this is nearly impossible, because the physical forces applied by a human are much slower than the processing speed.

It's important to take the absolute value of $\Delta V$ so the detection can work for both directions of the motor.

## 10.4. VHDL implementation. push_ignore.vhd

**Inputs:**

CLK: 10Mhz clock signal coming from the ΔΣ Modulator.

aRST: asynchronous global reset (high-level logic value).

ena: enable signal

voltage[17..0]: representative voltage of the back EMF within a time window.

threshold [17..0]: parameter formed by concatenating two faders of the MIDI keyboard

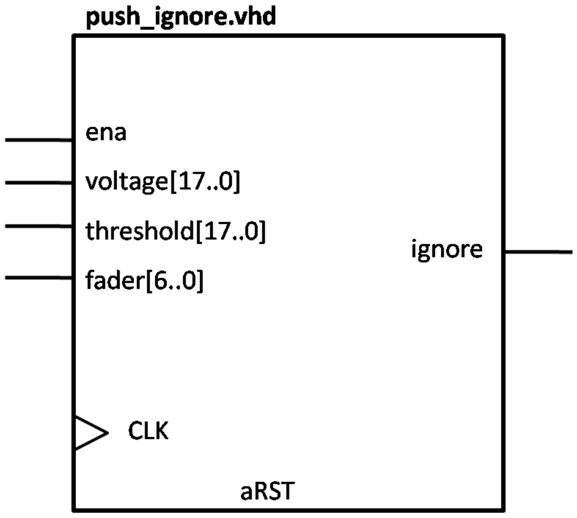


**Figure 48: push_ignore module**

fader[6..0]: parameter coming from the MIDI keyboard used to calculate the Δt.

**Outputs:**

ignore: binary signal that indicates if a push was detected.

The implementation on VHDL of the module is as follows:

```
process (arst, clk) begin
    if arst='1' then
      voltage_reg1<=(others=>'0');
      voltage_reg2<=(others=>'0');
    elsif clk'event and clk='1' then
      if ena='1' and timer=measureTime then
        timer<=(others=>'0');
        voltage_reg2<=voltage_reg1;
        voltage_reg1<=voltage;
        aux<=abs(voltage_reg1-voltage_reg2);
        if aux>abs(threshold) then
          ignore<='1';
        else
          ignore<='0';
        end if;
      else
        timer<=timer+1;
      end if;
    end if;
  end process;
```

Let's analyze the process step by step.

```
process (arst, clk) begin
```

```
    if arst='1' then
      voltage_reg1<=(others=>'0');
      voltage_reg2<=(others=>'0');
    elsif clk'event and clk='1' then
```

The process models a sequential circuit sensitive only to "arst" and "clk" signals. This means that the process is executed only when these two produce an event (change of logic value, rising edge, etc.). "arst" is the asynchronous reset global to all the modules. It's assigned to the "SW0" of the ZYBO. When the switch is pushed (high-level logic value in the ZYBO's switches [2]), the two voltage registers are reset. If there's not a reset, when a rising edge occurs, the main body of the process executes:

```
      if ena='1' and timer=measureTime then
        timer<=(others=>'0');
        voltage_reg2<=voltage_reg1;
        voltage_reg1<=voltage;
        aux<=abs(voltage_reg1-voltage_reg2);
        if aux>abs(threshold) then
          ignore<='1';
        else
          ignore<='0';
        end if;
      else
        timer<=timer+1;
      end if;
    end if;
```

The process has a enable signal "ena" that can be also assigned to a switch in order to activate/deactivate the module. When this signal is active and the timer has arrived to the target value "measureTime", the old capture of the voltage registers in "voltage_reg2" and the new one in "voltage_reg1". Then, if the absolute of the new value minus the old one is higher than the threshold, the ignore signal will stay activated during "measureTime" until it checks the conditions again.

## 10.5. VHDL implementation. Control.vhd

The control module is based on Prof. Dr. Hild's algorithm in [1]. It differs from its predecessor, among other things, in an additional waiting state, scaling on the drive time by MIDI parameters and the data gathering from the ΔΣ Modulator moved to a different module.

**Inputs:**

CLK: 10Mhz clock signal coming from the ΔΣ Modulator.

aRST: asynchronous global reset (high-level logic value).

valid_vm: indicates the end of a voltage measure

sw1: switch 1 of the ZYBO.

ignore: signal that indicates if a push was detected.

senseparam [6..0]: parameter that establishes the sense time.

driveparam [6..0]: parameter that scales the drive time calculated in the sense phase.

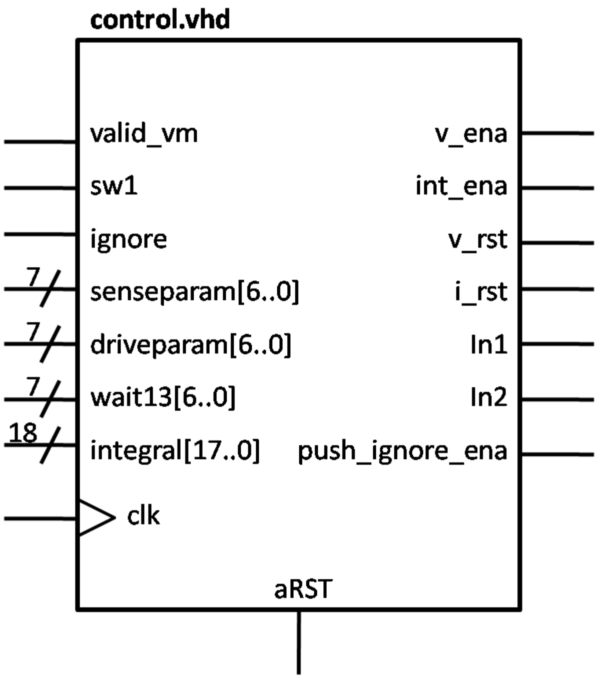


**Figure 49: control module (thresholds model)**

wait13[6..0]: parameter that establishes the waiting time for oscillations.

integral [17..0]: integration of the back EMF.

**Outputs:**

v_ena, int_ena: voltage and integral measure enable signals.

v_rst, i_rst: voltage and integral measure synchronous resets.

in1, in2: settings for the H-bridge driver. This pins configure the driving direction, coast and brake.

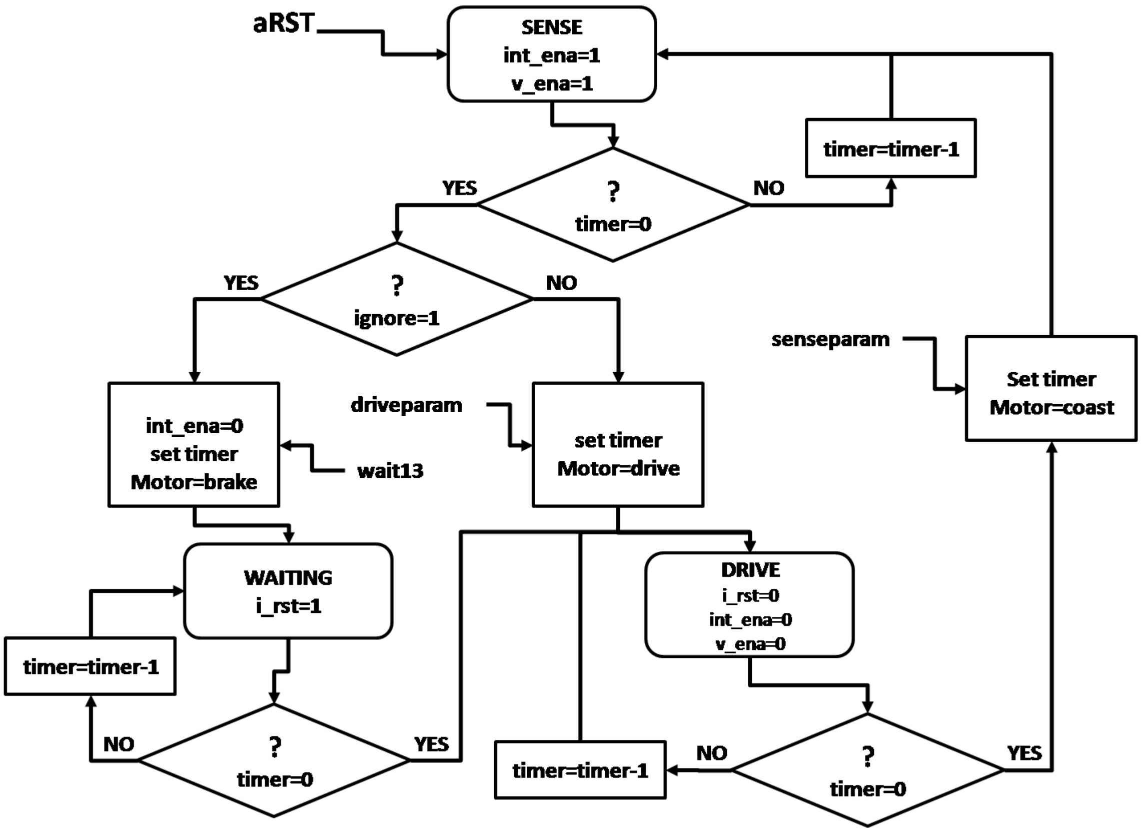


Figure 50: Control FSM flow diagram

The figure above shows the flow diagram of the control module. To start, the switch of the ZYBO "SW0" produces the usual asynchronous reset when is pressed. The FSM enters then into a sense phase and sets the timer. That enables the "measures.vhd" module to gather data from the ΔΣ Modulator. When the timer runs out, the algorithm checks if the ignore signal was activated.

If it was activated (YES), the integral stops being built and the motor is set up to brake before jumping to wait state, which doesn't do anything except running out the timer set and then jump to "DRIVE".

If it wasn't activated (NO), the motor will start the drive phase, not allowing the "measures.vhd" module to capture data from the ΔΣ Modulator.

Once either the drive or the waiting states are finished, the motor is set to coast so it doesn't resist any force and jump again to a sense phase and therefore starting a new loop.

In VHDL, the control module is as follows below:

```
process(arst, clk) begin
      if arst='1' then
            state<=sense;
            timer<=25d"99999"; --10ms default
      elsif clk'event and clk='1' then
            case state is
            when sense=>
                  int_ena<='1';
                  v_ena<='1';
                  if timer=25d"0" then
                        if ignore='1' then
                              timer<=(18d"9999" * unsigned(wait13));
                              in1<='1';
                              in2<='1';
                              state<=waiting;
                              int_ena<='0';
                        elsif   (   and(integral(17   downto   13))   or
nor(integral(17 downto 13)) )then --threshold
                              timer  <=(others=>'0');--  drive-time  is
zero, when every bit of voltage(17 downto 13) is 1 or 0
                              state<=drive;
                        else
                              in1<=not integral(17);
                              in2<=integral(17);
                              timer <= "0" & unsigned( abs(integral(17
downto 1)) ) * unsigned(driveparam) / 25d"64";
                               state<=drive;
                        end if;
                  else
                        timer<=timer-1;
                  end if;
            when waiting=>
                  i_rst<='1';
                  if timer=25d"0" then
                        state<=drive;
                        i_rst<='0';
                  else
                        timer<=timer-1;
                  end if;
            when drive=>
                  int_ena<='0';
                  i_rst<='0';
                  v_ena<='0';
                  if timer=25d"0" then
                        timer  <=(18d"9999"  *  unsigned(senseparam))  +
25d"9999"; -- prepare Measure-time, fix value: 18d"9999" -> 1 ms, 1
ms minimum time
                        in1<='0';
                        in2<='0';
                        state<=sense;
                  else
                        timer<=timer-1;
                  end if;
            end case;
      end if;
end process;
```

Here's a step by step detailed explanation, splitting the above code:

```
process(arst, clk) begin
      if arst='1' then
            state<=sense;
            timer<=25d"99999"; --10ms default
      elsif clk'event and clk='1' then
```

The process is sensitive to the asynchronous global reset "arst" and the 10Mhz clock signal. After a global asynchronous reset, the FSM will start in a sense state of 10ms. Then, every rising edge of the clock the process will be executed.

```
case state is
            when sense=>
                  int_ena<='1';
                  v_ena<='1';
```

In the sense state, both the voltage and the integration are enabled in the "measures.vhd" module.

```
                  if timer=25d"0" then
                        if ignore='1' then
                              timer<=(18d"9999" * unsigned(wait13));
                              in1<='1';
                              in2<='1';
                              state<=waiting;
                              int_ena<='0';
```

If a push was detected (ignore='1') ending a sense phase, the timer will be set based on the fader 13 parameter ("wait13"). Then the H-bridge pins[5] will be set to brake the motor ( in1<='1', in2<='1') and change to "waiting" state. At this point is also good to disable the integral so it doesn't measure undesired oscillations.

```
                        elsif   (   and(integral(17   downto   13))   or
nor(integral(17 downto 13)) )then --threshold
                              timer  <=(others=>'0');--  drive-time  is
zero, when every bit of voltage(17 downto 13) is 1 or 0
                              state<=drive;
```

If no push was detected, the algorithm checks if the integral is below a threshold. If that's the case, the state machine will jump to the drive state with no time for driving motion. This threshold, set up by Prof. Dr. Hild in [1], checks if the integral value is high enough to move the motor in the drive phase. This makes sure that the timer time is enough to be worth to drive the motor, otherwise it will just produce power loss without moving motion.

```
                        else
                              in1<=not integral(17);
                              in2<=integral(17);
                              timer <= "0" & unsigned( abs(integral(17
downto 1)) ) * unsigned(driveparam) / 25d"64";
                               state<=drive;
                        end if;
                  else
                        timer<=timer-1;
                  end if;
```

If the integral is above the threshold, the pins of the H-bridge [5] (in1, in2) are configured to drive in the opposite direction of the back EMF previously measured in the sense phase. This is achieved checking the MSB of "integral", in other words, checks the sign of the integral.

The time to drive is set up by the "integral" value plus some scaling that enhances the driving strength. This can be configured by the fader 10 ("driveparam").

```
            when waiting=>
                  i_rst<='1';
                  if timer=25d"0" then
                        state<=drive;
                        i_rst<='0';
                  else
                        timer<=timer-1;
                  end if;
```

The "waiting" state is just a state that waits for the oscillations to happen. It keeps the integration reset so when it exits from this state, it doesn't have accumulated a high magnitude value. This way, the motor won't drive abruptly after a push was detected.

```
            when drive=>
                  int_ena<='0';
                  i_rst<='0';
                  v_ena<='0';
                  if timer=25d"0" then
                        timer <=(18d"9999" * unsigned(senseparam)) +
25d"9999"; -- prepare Measure-time, fix value: 18d"9999" -> 1 ms, 1
ms minimum time
                        in1<='0';
                        in2<='0';
                        state<=sense;
                  else
                        timer<=timer-1;
                  end if;
            end case;
      end if;
end process;
```

In the drive state, the voltage and integration measures are disabled. The reason of it is because, with sensorimotor loops, it's only possible to measure the back EMF when the motor is not being driven. After the drive time is finished and the motor configuration pins will be set to coast (in1<='0', in2<='0'). This prepares the motor to measure the back EMF again and jumps to the sense phase.

## 10.6. Resources and power consumption.

This section contains the resources utilized and power consumed by the system based on parameterized thresholds.

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| **Slice LUTs** | 1796 | 17600 | **10,20** |
| **Slice Registers** | 837 | 35200 | **2,38** |
| **Memory** | 12 | 60 | **20,00** |
| **DSP** | 3 | 80 | **3,75** |
| **IO** | 39 | 100 | **39,00** |
| **Clocking** | 4 | 32 | **12,50** |

**Figure 51: Thresholds system resources consumption overview**

The table above contains the different elements of the FPGA that were used. The percentages show that the system uses a small amount of resources. Lot of these resources correspond to the visual modules from chapter 8.

| Name | F7 Muxes | F8 Muxes | Slice[8] | LUT as Logic | LUT as Memo ry | LUT Flip Flop Pairs | Block RAM Tile | DSPs |
|---|---|---|---|---|---|---|---|---|
| **top** | 2.02 % | 1.65 % | 14.09 % | 8.61 % | 4.66 % | 11.46 % | 19.16 % | 3.75 % |
| **ASCII_canvas (ASCII_canvas)** | 0.25 % | 0.00 % | 0.75 % | 0.41 % | 0.00 % | 0.41 % | 8.33 % | 0.00 % |
| **ClockTree (ClockTree)** | 0.00 % | 0.00 % | 0.04 % | 0.02 % | 0.00 % | 0.02 % | 0.00 % | 0.00 % |
| **csl (csl_lezcan)** | 0.00 % | 0.06 % | 4.97 % | 3.34 % | 0.00 % | 3.64 % | 0.00 % | 3.75 % |
| **control (control)** | 0.00 % | 0.00 % | 0.63 % | 0.36 % | 0.00 % | 0.36 % | 0.00 % | 2.50 % |
| **measures (measures)** | 0.00 % | 0.06 % | 3.72 % | 2.73 % | 0.00 % | 2.81 % | 0.00 % | 1.25 % |
| **push (push_ignore)** | 0.00 % | 0.00 % | 0.68 % | 0.23 % | 0.00 % | 0.48 % | 0.00 % | 0.00 % |
| **faderControl (faders)** | 0.00 % | 0.00 % | 1.27 % | 0.58 % | 0.00 % | 0.81 % | 0.00 % | 0.00 % |
| **GetMidi (GetMIDI)** | 0.00 % | 0.00 % | 0.27 % | 0.09 % | 0.00 % | 0.16 % | 0.00 % | 0.00 % |
| **integral_sign ed (WriteSigned)** | 0.00 % | 0.00 % | 1.09 % | 0.54 % | 0.00 % | 0.67 % | 2.50 % | 0.00 % |
| **ShowScope1 (ShowScope)** | 0.31 % | 0.31 % | 0.65 % | 0.05 % | 0.93 % | 0.50 % | 0.00 % | 0.00 % |
| **ShowVBar9 (ShowVBar)** | 0.00 % | 0.00 % | 0.04 % | 0.03 % | 0.00 % | 0.03 % | 0.00 % | 0.00 % |

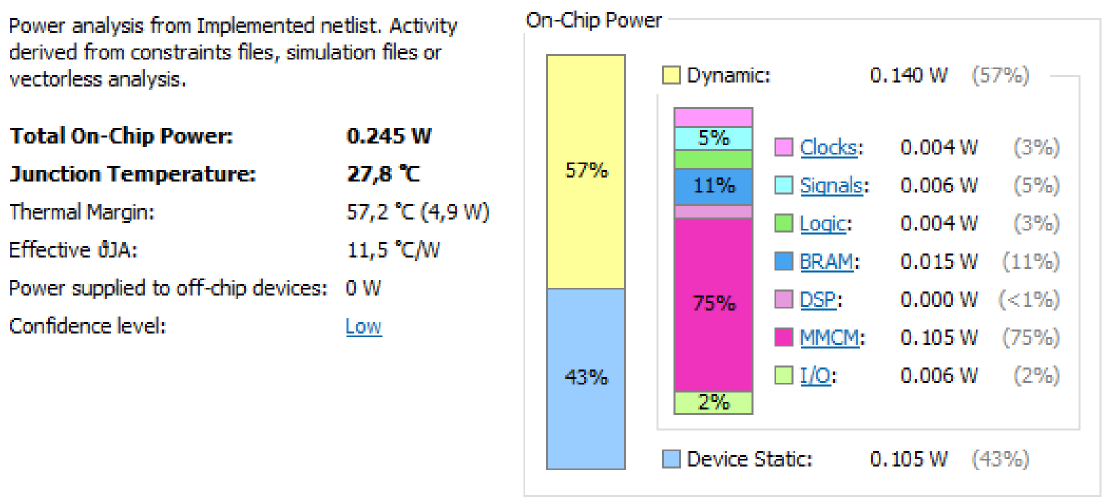
[8] Each Slice has four 6-input LUTs and 8 FFs

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **VGA1024 (VGA1024)** | 0.09 % | 0.00 % | 2.25 % | 1.19 % | 0.00 % | 1.30 % | 0.00 % | 0.00 % |

**Figure 52: Thresholds system resources (extended)**

It's important to remember that all the visual modules wouldn't be needed once the system has its application. In that case, the system would utilize a minimum amount of resources.

The next issue is the power consumption of the ZYBO:



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **0.245 W** |
| **Junction Temperature:** | **27,8 ℃** |
| Thermal Margin: | 57,2 ℃ (4,9 W) |
| Effective ϑJA: | 11,5 ℃/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

On-Chip Power

| | | |
|---|---|---|
| Dynamic: | 0.140 W | (57%) |
| Clocks: | 0.004 W | (3%) |
| Signals: | 0.006 W | (5%) |
| Logic: | 0.004 W | (3%) |
| BRAM: | 0.015 W | (11%) |
| DSP: | 0.000 W | (<1%) |
| MMCM: | 0.105 W | (75%) |
| I/O: | 0.006 W | (2%) |
| Device Static: | 0.105 W | (43%) |

**Figure 53: Thresholds system power consumption**

The chart show that  57% of the power is to keep the board on, while the rest of the power consumption (43%) is mainly spent in the Mixed Mode Clock Manager (MMCM) which role is to  generate the different clock signals.

It's important to remember that the CSL Pmod and the motor are powered separately with the DC power supply so they are not included in the above chart.

## 10.7. Outlook

The system shows generally a good behavior. Experimentally, it detects pushes with a wide variety of ranges: short in time, subtle, continuous and prolonged in time.

It keeps the simplicity of the original sensorimotor loop in [1] by Prof. Dr. Hild, since the push detection is achieved out of the control loop. Therefore, the control module just needs some minor adjusting and the introduction of the waiting state, which has no complexity at all.

The speed of the algorithm, with magnitudes of tens of milliseconds, is enough to have corrections close to the two equilibrium positions ($90°$ and $270°$) and stay there stable as long as there aren't any new pushes. One thing that could be improved for next versions could be the fluidity of the system for some situations. There are isolated cases, with the threshold really close to the maximum torque voltage, where the system stumbles until it finds a position with lower torque.

In conclusion, the advantages of this system against the one with push bypassing filter makes it a better solution unless some matters are corrected in the second one.

# 11. System based on a push bypassing filter

## 11.1. System overview

This model consists on filtering the representative back EMF and establish it as the new input of the sensorimotor loop in order to ignore pushes.

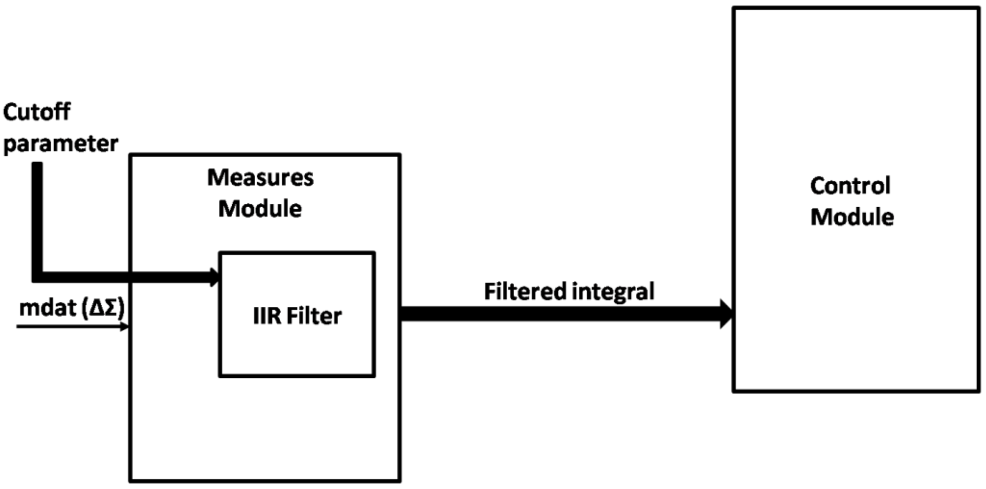


**Figure 54: Push bypassing filter simplified diagram**

In the above simplified diagram can be seen, in what things differs from the thresholds model. The "ignore.vhd" module has been removed since there's no push detection in strict terms. The measures module has a section of the code in which the filter is implemented. The control module stays similar but without detection conditions.

## 11.2. VHDL Implementation. measures.vhd

Once all the DSP basics and mathematical foundations in the 3rd chapter are understood, the VHDL filter code can start to be analyzed. The "measures.vhd" is pretty much the same to the one in the system based on thresholds, so there's no need to explain the whole module. Instead, this chapter will just concentrate in the filter implementation. Although this section of the code is very compressed, it contains a serious amount of information within. The code employs this signals:

```
      signal integral, filtered_i: signed (17 downto 0);
      signal aux: signed (31 downto 0);
      signal cutoff: Natural;
```

The variable "integral" is, as explained before, a representation of the back EMF integrated in time. It's an 18 bit signed number with no decimals and It basically composes the input of the filter.

The variable "filtered_i" has the same data characteristics as "integral" and it's the output of the filter. "aux" is a 32 bit signed number and it's employed as an auxiliary variable where the fixed comma operations take place, that's the reason for the extra bits.

The variable "cutoff" defines the number of shifts executed by the filter. This is introduced via faders by the user. So this "cutoff" it's another representative value and not the actual cutoff frequency of the filter. Since the "shift_right" operation needs the second operator to be an integer, the incoming value from the fader is casted:

```
       cutoff<=to_integer(unsigned(fader2));
```

Let's get into the body of the filter:

```
       aux<=aux+shift_right((integral&4d"0"-aux), cutoff);
             if aux(31 downto 4)>131071 then
                   filtered_i<=18d"131071";
             elsif aux(31 downto 4)<-131072 then
                   filtered_i<='1'&17d"0";
             else
                   filtered_i<=aux(21 downto 4);
             end if;
```

Now, step by step:

```
       aux<=aux+shift_right((integral&4d"0"-aux), cutoff);
```

The first thing to notice is " integral&4d"0" ". This concatenates four zeroes at the right of the value to be used as decimals, so that the shift operation can be done with fixed comma data and store decimals.

After that the value is subtracted to "aux". This is just a form of increasing the performance in the filter. The easiest way to understand it is with an example. Let's say the value of "cutoff" is 2. That corresponds to 2 shifts to the right, which equates to multiplying by 0.25 ($\frac{1}{2^2} = \frac{1}{4}$). The mathematical equivalent would be:

$$\begin{aligned} aux + 0.25 \cdot (integral - aux) &= aux + 0.25 \cdot integral - 0.25 \cdot aux \\ &= 0.25 \cdot integral + 0.75 \cdot aux \end{aligned}$$

If we remember the 1 order IIR filter formula shown in the 3rd chapter:

$$y[n] = a \cdot x[n] + (1 - a) \cdot y[n-1]$$

And compare it:

$$a \cdot x[n] + (1 - a) \cdot y[n-1] = 0.25 \cdot integral + 0.75 \cdot aux$$

$$x[n] = integral, \;\; y[n-1] = aux$$

$$a = 0.25,\ (1 - a) = 0.75$$

It's now clear that both are basically the same equation.

Next part is as follows:

```
            if aux(31 downto 4)>131071 then
                  filtered_i<=18d"131071";
```

These sentences check if the accumulation in the filter overflows the 18 bit output variable. If this is the case, the output is set to its maximal value.

```
            elsif aux(31 downto 4)<-131072 then
                  filtered_i<='1'&17d"0";
```

Same here for negative values. If the value overflows from below, the output is set to its minimal possible value. This means setting a '1' in the MSB followed by '0's in a two's complement representation.

```
            else
                  filtered_i<=aux(21 downto 4);
            end if;
```

If none of the prior cases happened, the value is in range, and can be output in the "filtered_i" variable once the decimals (the four LSBs) are removed.

### 11.3. Resources and power consumption

This section contains the resources utilized and power consumed by the system based on a push bypassing filter.

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| **Slice LUTs** | 1589 | 17600 | **9,03** |
| **Slice Registers** | 707 | 35200 | **2,01** |
| **Memory** | 12 | 60 | **20,00** |
| **DSP** | 3 | 80 | **3,75** |
| **IO** | 39 | 100 | **39,00** |
| **Clocking** | 4 | 32 | **12,50** |

**Figure 55: Filter system resources overview**

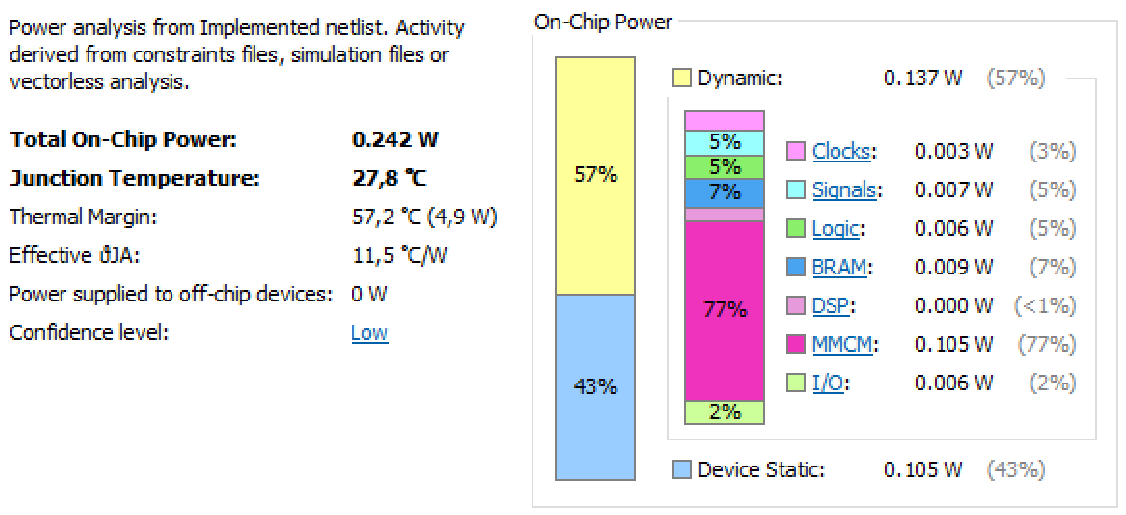
The table above displays a similar amount of resources consumed to the system based on parameterized thresholds. Comparing with the Figure 51: Thresholds system resources consumption overview, it can be seen that this model uses an insignificant less quantity. Thus won't be decisive when the time came to choose between both of them.

| Name | F7 Muxes | F8 Muxes | Slice | LUT as Logic | LUT as Memo ry | LUT Flip Flop Pairs | Block RAM Tile | DSPs |
|---|---|---|---|---|---|---|---|---|
| top | 1.27 % | 1.04 % | 12.22 % | 8.07 % | 2.80 % | 9.89 % | 19.16 % | 3.75 % |
| ASCII_canvas (ASCII_canvas) | 0.20 % | 0.00 % | 0.81 % | 0.36 % | 0.00 % | 0.38 % | 8.33 % | 0.00 % |
| ClockTree (ClockTree) | 0.00 % | 0.00 % | 0.04 % | 0.02 % | 0.00 % | 0.02 % | 0.00 % | 0.00 % |
| csl (csl_lezcan) | 0.00 % | 0.00 % | 4.04 % | 3.25 % | 0.00 % | 3.36 % | 0.00 % | 3.75 % |
| control (control) | 0.00 % | 0.00 % | 0.77 % | 0.56 % | 0.00 % | 0.56 % | 0.00 % | 3.75 % |
| measures (measures) | 0.00 % | 0.00 % | 3.65 % | 2.68 % | 0.00 % | 2.90 % | 0.00 % | 0.00 % |
| faderControl (faders) | 0.00 % | 0.00 % | 1.00 % | 0.31 % | 0.00 % | 0.59 % | 0.00 % | 0.00 % |
| filteredint_si gned (WriteSigned) | 0.09 % | 0.09 % | 1.04 % | 0.62 % | 0.00 % | 0.69 % | 3.33 % | 0.00 % |
| GetMidi (GetMIDI) | 0.00 % | 0.00 % | 0.25 % | 0.09 % | 0.00 % | 0.17 % | 0.00 % | 0.00 % |
| ShowScope1 (ShowScope) | 0.31 % | 0.31 % | 0.59 % | 0.03 % | 0.93 % | 0.48 % | 0.00 % | 0.00 % |
| ShowVBar9 (ShowVBar) | 0.00 % | 0.00 % | 0.04 % | 0.03 % | 0.00 % | 0.03 % | 0.00 % | 0.00 % |
| VGA1024 (VGA1024) | 0.01 % | 0.00 % | 2.18 % | 1.21 % | 0.00 % | 1.32 % | 0.00 % | 0.00 % |

**Figure 56: Filter system resources (extended)**

The extended tables verify that the resource utilization is very similar, given they both use the common and visual modules, which represent a significant percentage of the used resources.

Same as in the chapter before, another matter to check would be the power consumption:


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **0.242 W** |
| **Junction Temperature:** | **27,8 °C** |
| Thermal Margin: | 57,2 °C (4,9 W) |
| Effective ϑJA: | 11,5 °C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

On-Chip Power

| | | |
|---|---|---|
| Dynamic: | 0.137 W | (57%) |
| Clocks: | 0.003 W | (3%) |
| Signals: | 0.007 W | (5%) |
| Logic: | 0.006 W | (5%) |
| BRAM: | 0.009 W | (7%) |
| DSP: | 0.000 W | (<1%) |
| MMCM: | 0.105 W | (77%) |
| I/O: | 0.006 W | (2%) |
| Device Static: | 0.105 W | (43%) |

**Figure 57: Filter system power consumption**

The chart displays almost exact results to the thresholds model. Most of the power consumption is static, while the dynamic one is mainly produced by the MMCM. This results make sense given the resource utilization is pretty similar in both systems.

## 11.4. Outlook

The complexity of this system falls back onto understanding some basics in DSP and finding and alternative to represent data with decimals. Apart from that, it can be implemented in a few lines using this simple IIR filter.

Experimentally, it works well for non-equilibrium positions when the cutoff parameter is properly adjusted. When the pendulum approximates to the 90° equilibrium position, the algorithm is not fast enough to do small corrections and maintain the pendulum stable. This is surely because of the delay introduced by the filter. Since the filter is built with shift registers, if the cutoff parameter is set to e.g. six units, that means a sample arrives to the output of the filter six periods later. That means that using a 10 milliseconds sense phase, the control loop will process that same sample approx. 60 milliseconds late. This is definitely too slow for this motor, however for systems with more mass e.g. Myon could be enough to reach equilibrium positions.

To sum up, one major improvement for this particular motor and mass would be to increase the speed of the system. One thing that could be done is to use an external oscillator for the ΔΣ Modulator up to a maximum of 16MHz clock speed[5]. If the speed is still not enough it would be probably needed a faster ADC.

# ANNEXE 1: Video description. System based on parameterized incremental thresholds

### *Screen legend*

- OFFSET: compensation value for the ΔΣ modulator offset, applied at the end of every sense phase.

- VOLTAGE: representative value of the back EMF of the motor measured by the $\Delta\Sigma$ modulator during a time window. This value is already compensated with the $\Delta\Sigma$ offset.
- INTEGRAL: integration of the representative value of the back EMF.
- THRSHLD: value of the incremental threshold. If "VOLTAGE" grows more than this value in a time window, a push is detected.
- PUSH: appears on the screen when a push is detected.

### *Switches definition*

| Switches | ON |
|---|---|
| **SW0** | activate driver |
| **SW1** | Enable push ignoring |
| **SW2** | Graph scaling |

**Figure 58: switches set up on parameterized thresholds model**

### *BHT_BA_Thresholdmodel_1.mp4*

The video shows a correct behavior of the system based on parameterized incremental thresholds. The experiment consists on applying a series of different pushes in both directions. This pushes are also different in magnitude and applied on different angular positions.

| Parameter | Fader number | Value |
|---|---|---|
| **sense** | 9 | 10 (ms) |
| **drive** | 10 | 30 (-) |
| **threshold** | 11-12 | 1408 (-) |
| **wait** | 13 | 5 (ms) |

**Figure 59: BHT_BA_Thresholdmodel_1.mp4 set up**

### *BHT_BA_Thresholdmodel_2.mp4*

This second video uses also the same system based on parameterized incremental thresholds. It shows the importance of having a waiting state in the sensorimotor algorithm to prevent oscillations. The difference between this model and the one above is that there's no waiting time configured, so the system will oscillate after detecting a push.

| Parameter | Fader number | Value |
|---|---|---|
| **sense** | 9 | 10 (ms) |
| **drive** | 10 | 30 (-) |
| **threshold** | 11-12 | 1408 (-) |
| **wait** | 13 | 0 (ms) |

**Figure 60: BHT_BA_Thresholdmodel_2.mp4 set up**

## ANNEXE 2: Video description. System based on a push bypassing filter

### *Screen legend*

- OFFSET: compensation value for the ΔΣ modulator offset, applied at the end of every sense phase.
- VOLTAGE: representative value of the back EMF of the motor measured by the ΔΣ modulator during a time window. This value is already compensated with the ΔΣ offset.
- INTEGRAL: integration of the representative value of the back EMF.
- FILTEREDINT: "INTEGRAL" filtered by a low-pass IIR filter.

### *Switches definition*

| Switches | ON |
|---|---|
| **SW0** | activate driver |
| **SW1** | - |
| **SW2** | Graph scaling |
| **SW3** | - |

**Figure 61: switches set up on push bypassing filter model**

### *BHT_BA_IIRfiltermodel.mp4*

The video shows how the system based on a push bypassing filter reacts. It's configured in order to have certain equilibrium between speed and filtering correctly the pushes. It can be appreciated that the system is too slow to detect the top equilibrium position before reacting again.

| Parameter | Fader number | Value |
|---|---|---|
| **sense** | 9 | 10 (ms) |
| **drive** | 10 | 29 (-) |
| **cutoff** | 11 | 6 (-) |
| **wait** | 13 | 5 (ms) |

**Figure 62: BHT_BA_IIRfiltermodel.mp4 parameter set up**

## Bibliography

[1] Hild, Manfred: *Defying Gravity - A Minimal Cognitive Sensorimotor Loop Which Makes Robots With Arbitrary Morphologies Stand Up*. 11th International Conference on Accomplishments in Electrical and Mechanical Engineering and Information Technology (DEMI) 2013.

[2] Oppenheim, Alan V. : *Discrete-time signal processing* /Alan V. Oppenheim, Roland W. Schafer, with John R. Buck. -2nd ed.

[3] ZYBO™ FPGA Board Reference Manual.

Download site: http://digilentinc.com/Data/Products/ZYBO/ZYBO_RM_B_V6.pdf

[4] ZYBO™ schematics, VB3

Download site: http://digilentinc.com/Data/Products/ZYBO/ZYBO_sch_VB.3.pdf

[5] TEXAS INSTRUMENTS - ADS1203 datasheet. Motor Control, Current Measurement 1 bit, 10MHz, 2nd order, Delta-Sigma Modulator.

[6] TEXAS INSTRUMENTS - DRV8837 datasheet. LOW-VOLTAGE H-BRIDGE IC.

[7] MIDI MANUFACTURERS ASSOCIATION: Table 1 - Summary of MIDI messages.

Site: http://www.midi.org/techspecs/midimessages.php Date of query: June 2015.

## Statement of Autorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, September ...., 2015 ……………………………………………..