PHYS 479: High-Speed Information Processing for Laser-locking System Applied to Optomechanical Cavities

Laurent René de Cotret
McGill University,
Supervised by Jack C. Sankey

August 23, 2013

Abstract

The use of optical cavities in Optomechanics requires that the laser and cavity are locked together. To counter the effect of environmental mechanical noise on the length of the cavity, one needs to apply a cancelling response to the cavity length (via piezoelectric elements) as quickly as possible. This report details the making of a high-speed digital signal processing system built using a field-programmable gate array (FPGA) capable of displaying any filter and the design of the filters that could be used to attenuate this mechanical noise.

1 Introduction

The Sankey Laboratory is concerned with the interaction between light and mechanical objects, or more specifically, radiation pressure and thin membranes. The typical setup consists of an optical cavity (two mirrors facing each other), with a mechanical object (membrane) in between.
A typical setup of the Sankey Laboratory uses a thin silicon nitride membrane as mechanical object, which is weakly tethered to the environment (see figure 1). This membrane is held in an optical cavity, in an ultra-high vacuum environment (pressure less than 10^{-7} Pa). The nominal cavity length is 1 cm, with a membrane 100 nm thick [1]. The laser used to stimulate the membrane has a wavelength of 1550 nm. At this wavelength, silicon nitride membranes of 100 nm thick have a transmittance of approximately 30% [2], which enables light to pass through the membrane and reflect on the end mirror; therefore, light is continually interacting with the membrane from both sides.

Maximum interaction between light and membrane occurs when the power circulating in the cavity is maximum. Power is maximum when the laser light is in resonance with the cavity length; it is said that the laser is locked to the cavity, and that the cavity is at resonance length. If the cavity length drifts away from the resonance length, circulating power drops and there are fewer photons-membrane interactions. Therefore, it is important that the cavity length stays as close as possible to resonance length.

Alternatively, a laser-locking system would enable the team to precisely tune the cavity length, which could be used to adjust the cavity resonant frequency with respect to the laser frequency. This intentional "detuning" could be used for laser cooling [3].

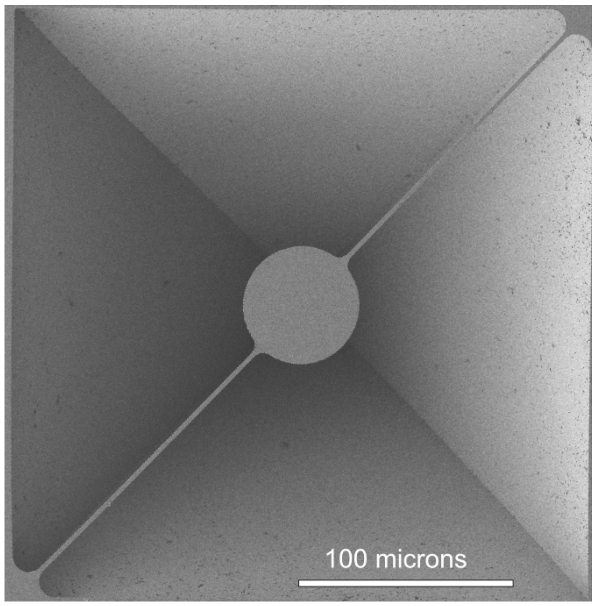


Figure 1: A weakly-tethered silicon nitride membrane to be used in an optical cavity, made by the Sankey Laboratory at McGill. Picture taken by Christoph Reinhardt.

Mechanical noise from the environment (vibrations from colleagues talking, traffic nearby, etc.) contributes to vary the cavity length randomly. By examining the light that exits the cavity, it is possible to determine the difference between the cavity length and the resonance length (called the detuning length). This measurement is not part of the project, and thus it is assumed that the system to be built receives information about the detuning length in the form of a voltage. As time goes by,

the detuning length varies due to mechanical noise; this sends a voltage signal to the system described in this report. To adjust the cavity length, piezoelectric elements are attached to one of the mirrors; applying a voltage to the piezoelectric elements moves the mirror.
The general idea is to process the detuning length signal in order to apply an appropriate response to piezoelectric elements, with the objective of keeping the detuning length small. This procedure is called feedback locking.

Section 2 describes the feedback locking theory and digital signal processing theory necessary to understand the system design. Section 3 describes the project's requirements and the system design. Section 4 details how filters are implemented. Finally, section 5 details the testing done on some digital filters.

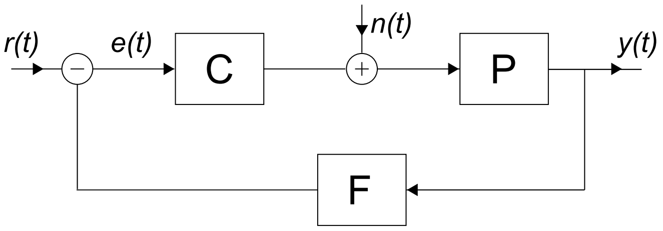


Figure 2: General feedback loop. The three components are the controller (C), the plant (P), and the sensor (F). The objective is to create C so that the output signal $y(t)$ be the same as the reference input signal $r(t)$. Noise is added into the system via $n(t)$. The signal $e(t) = r(t) - y(t)$ is the tracking error, which has to be minimized.

2 Theory

We start by explaining what is a transfer function, and then move on to describe a general feedback control system. Then we show how to "translate" a filter into discrete-time hardware.

Linear Time-Invariant Systems

Figure 2 shows a block diagram of a feedback loop. The arrows represent signals moving through the circuit, which is composed of three components: a controller (C), a plant (P), and a sensor (F).

For a physical circuit, it is reasonable to assume that the output of the components do not depend on time explicitly, but only on the input; moreover, it is also reasonable to assume that at low frequencies, the components behave in a linear way. With these assumptions, the feedback loop of figure 2 becomes a linear time-invariant (LTI) system. Analog LTI systems are governed by laws expressed as linear differential equations; it is therefore natural to analyze analog systems using the Laplace Transform.
A convention in this report is that the Laplace transform of the function $f(t)$ is $LT\{f(t)\} = F(s)$, where t is a real number, and s is complex.

We define the transfer function of a component to be the ratio of the output and the input in the Laplace s-domain; thus, for an input signal $x(t)$ and an output signal $y(t)$, the transfer function $T(s)$ satisfies:

$$T(s) = Y(s)/X(s)$$

An important property of LTI systems is that the transfer function is always a rational function in the Laplace s-domain [4, p. 32]. We note that a realizable LTI system cannot have any poles in the right half of the s-plane (which is a complex plane), as it would mean that a finite input can create an unbounded output (the right half s-plane represents exponentials growing in time) [5, p. 22].

2.1 Feedback Control Theory

All feedback control systems have three components: a controller, a sensor, and a plant (see figure 2). The plant is the object to be controlled; in our case, the plant is th ecavity length. The sensor measures the output from the plant; in our case, a photodiode examines the light coming out of the cavity. The controller uses the feedback from the sensor to control the plant [4, p. 31]. There are two inputs: noise affecting the plant, which is represented in figure 2 as $n(t)$, and a reference signal $r(t)$ that tells the controller what is the desired output $y(t)$. In general it is assumed that the transfer function of the sensor is unity, and so it could be omitted from the diagram of figure 2.

The controller is the only component over which we have total control. The objective in the design of a feedback loop is two fold. First, given a plant (and a sensor) with known frequency-domain transfer functions P and F, the controller has to be designed so that the whole system is stable; that is, for finite input and noise $r(t)$ and $n(t)$, the output $y(t)$ is also finite. Second, the controller has to be designed so that the plant output $y(t)$ follows the reference input $r(t)$ as close as possible [4, p. 63]. Note that $r(t)$ is often set to 0 or a constant value. In our case, $r(t)$ might be the desired cavity length.

Stability in the Feedback Loop

With the additional assumption that the transfer function of the sensor $F(s)$ equals unity, it is possible to use specify the transfer function of the controller such that the whole system is stable.

Let A be the set of stable, real-valued rational functions. Let P(s) and C(s) be the transfer functions of the plant and controller respectively, and assume $P(s), C(s) \in A$. Since P(s) is a rational function (because linear and time-invariant), we can write $P(s) =$

$N(s)/M(s)$, where $N(s)$ and $M(s)$ are polynomials in s, and $N(s), M(s) \in A$. Using Euclid's algorithm, it is possible to find two polynomials $X(s)$ and $Y(s)$ such that

$$N \cdot X + M \cdot Y = 1 \tag{1}$$

Doyle et al. [4, p. 71] described that the set of all controller transfer functions $C(s)$ for which the feedback loop is stable (with the assumptions of an LTI system and $F(s) = 1$) is defined as:

$$C \in \left\{ \frac{Q}{1 - PQ} : Q \in A \right\} = \left\{ \frac{X + MQ}{Y - NQ} : Q \in A \right\} \tag{2}$$

that is, there exists a stable and real-valued transfer function $Q(s)$, which can be used to find the form of $C(s)$ according to equation 2. Therefore, designing a controller that makes the feedback loop stable results in finding the right $Q(s)$.

Asymptotic Tracking

The second goal of designing the controller transfer function $C(s)$ consists of choosing $C(s)$ according to equation 2 such that the output $y(t)$ follows $r(t)$. While we cannot expect $y(t)$ to be equal to $r(t)$ at all times due to noise $n(t)$, we require that $y(t)$ asymptotically tend to $r(t)$; we would like $y(t) \in [r(t) - \epsilon, r(t) + \epsilon]$, for some small real error ϵ, at all times.

Assuming the transfer function $P(s)$ is known, the procedure for finding $C(s)$ is described as follows [4, p. 74]. Define the transfer function from $r(t)$ to $e(t)$ in the $n(t) = 0$ case to be S, and the transfer function from $r(t)$ to $y(t)$ in the $n(t) = 0$ case to be T. From $R(s) - Y(s) = E(s)$ and $Y(s) = E(s) \cdot C(s) \cdot P(s)$:

$$S = \frac{1}{1 + P(s) \cdot C(s)}$$
$$T = \frac{P(s) \cdot C(s)}{1 + P(s) \cdot C(s)}$$

S is called the sensitivity function, while T is called the complementary sensitivity function (as $S + T = 1$). By denoting P by N/M and C by its parametrization (from equation 2), S and T reduce to the following in the case of a stable feedback loop:

$$S = M\,(Y - NQ) \tag{3}$$
$$T = N\,(X + MQ) \tag{4}$$

The sensitivity function S gets its name from another standard result in feedback control theory: since S is the transfer function from $r(t)$ to $e(t)$, setting an error bound of $|e(t)| < \epsilon$ amounts to the requirement:

$$\|S\|_{\infty} = \sup_{s} S < \epsilon \tag{5}$$

A procedure for designing $C(s)$ given a parametrization (from equation 2), the transfer functions S and T, and $r(t)$, can be stated as:

1. Use equations 3, 4 and 5 to find constraints on Q
2. Solve for $Q \in A$
3. Back-substitute to get the controller transfer function $C(s)$

The only missing part needed before using the techniques described in section 2, are the transfer functions from the cavity, piezoelectric elements and photodiode. Once these transfer functions are measured or approximated, it would be possible to find a controller transfer function (section 2.1), translate it in digital form (section 2.2) and wire the digital fitlter into the FPGA (section 4).

Until the measurements are made, we will concentrate on making an arbitrary filter with the FPGA. The rest of this work will only be concerned with making a digital filter apparatus, a crucial step before creating a feedback locking system.

2.2 Digital Implementation

After having specified the controller transfer function $C(s)$ from section 2.1, this section explains how to realize the controller using digital (or discrete-time) hardware. The convention used is that digital signals are represented with square brackets ($y[n]$), where n is an integer representing the number of time steps (if t is time, and T the sampling period, n is such that $t = n \cdot T$).

The z-plane

Analog filters are analyzed using the Laplace Transform and the s-plane, because such filters are described by linear differential equations. However, discrete-time filters are described using difference equations, and are analyzed using the z-transform. The z-transform of a digital signal $y[n]$ is written $ZT\{y[n]\} = Y(z)$, and the transform is defined as:

$$ZT\{y[n]\} = \sum_{n=-\infty}^{n=\infty} y[n] z^{-n} \tag{6}$$

for z a complex number. A typical difference equation for a digital filter with input $x[n]$ and output $y[n]$ has the following form:

$$\begin{aligned} y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + ... \\ + b_1 y[n-1] + b_2 y[n-2] + ... \end{aligned} \tag{7}$$

The z-transfer function is defined similarly to the transfer function of equation 1: if $Y(z)$ and $X(z)$ are the z-transforms of $y[n]$ and $x[n]$ respectively, then the z-transfer function $H(z)$ is defined as:

$$H(z) = Y(z)/X(z) \tag{8}$$

The main advantage of using the z-transform is the link between a system's difference equation and its z-transfer function [5, p. 246]; for a general difference equation (equation 7), the transfer function is given by:

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + ...}{1 - b_1 z^{-1} - b_2 z^{-2} - ...} \tag{9}$$

Therefore, given a z-transfer function $H(z)$, one can find the difference equation; the difference equation can then be implemented in an FPGA. Additionally, one can find the frequency response (both gain and phase) of a z-transfer function $H(z)$ by subsituting $H(e^{-i\omega})$; Then, the gain response is given by $|H(e^{-i\omega})|$, and the phase response by $\arg(H(e^{-i\omega}))$. [6, p. 611].

We also note that the frequency response of a digital filter is inherently symmetric. The gain response will always be symmetric around the half-sampling frequency, while the phase response will always be antisymmetric. This is due to the discrete-time nature of sampling: beyond the half-sampling frequency, a sinusoidal signal will look (after sampling) like a signal of lower frequency, with phase shift of 180° [5, p. 235].

The Bilinear Transform

The z-plane and s-plane are closely related because their respective transforms are discrete-time and continuous-time equivalents [6, p. 609]. Similarly to the condition that no realizable transfer function can have a pole in the right half of the s-plane, no realizable z-transfer function can have poles in the $|z| \geq 1$ region [6, p. 611].

It is thus reasonable to look for a way of translating a transfer function $H(s)$ into a z-transfer function $G(z)$ in the z-plane. Tustin's Method (or the Bilinear Transform) is an approximation that relates the z-plane and the s-plane in a way that preserves the properties of the inital function (either in going from the z-plane to the s-plane or vice-vera) [5, p. 450].

From a discretization of time $t = n \cdot T$ (with T the sampling period), we can express s as $s = T^{-1} \ln z$. By Taylor expansion, a first-order approximation of s as a function of z simplifies to:

$$s = \frac{2}{T}\left(\frac{z-1}{z+1}\right) \tag{10}$$

As Oppenheim et al. [5, p. 450] explains, the Bilinear Transform preserves important properties; most importantly, it preserves stability [7, p. 212].

Procedure for Designing a Feedback Loop

With the Bilinear Transform, it is now possible to realize a digital filter. First, given a transfer function, the Bilinear Transform can be used to translate this transfer function into the z-plane. Second, by relating equations 7 and 9, the system's difference equation can be computed and implemented in digital hardware.

3 Design Choices

3.1 Project requirements

A similar apparatus used by Prof. Sankey at Yale could apply 100 000 corrections per second (that is, it could react in 10 μm); one requirement is to best this correction rate by a factor of 10 or more. Another requirement is that the design should be adaptable: if the apparatus is changed (e.g. using a different material for the membranes), the feedback locking system should be easily adjustable by the experimenter. The adaptability requirement mandates the use of digital hardware, and this is why section 2.2 is relevant for the project. Below is an argument concerning which digital hardware to use.

3.2 System Design

Since typical mechanical noise has a frequency range from 0 to 100 kHz, it is necessary that signal processing takes place at a higher rate [5, p. 147]. Such a frequency of processing is not possible on an ordinary lab computer, since computers proceed linearly; while computers are fast in carrying simple operations, the sheer number of operations necessary just to do nothing (e.g. running the operating system) implies a low frequency of operation (relative to the project needs) for analyzing data. In fact, most hardware based on a central processing unit (CPU) (e.g. Arduino, Rasberry Pi) would not be suitable because of speed limitations.

The best solution for speed would be to use an Application-Specific Integrated Circuit (ASIC). Such a circuit is custom-made as a printed circuit-board for a single application. The advantage of ASICs is that they can carry operations simultaneously if the design specifies it, a clear advantage over CPU-based hardware ; however, once an ASIC has been manufactured, it can only do one thing. Therefore, using an ASIC for the feedback locking system would make the system not adaptable.

There is a solution that is similar in performance to an ASIC, but can be rewired at will: a Field-Programmable Gate Array (FPGA). Specifically, an FPGA has a matrix of transistors, and the user controls the wiring. When a wiring is implemented and a task is running, the FPGA cannot be rewired; in this state, it behaves just like an ASIC.
With its matrix of transistors, an FPGA can implement any logic operation from Boolean Algebra [8]. Thus, it can, in principle, carry any operation that a computer could carry, given enough resources.

The matrix of an FPGA contains logic blocks (array of transistors that can be wired into any logical circuit), memory blocks and input/output (I/O) ports [8].

Moreover, FPGA are often manufactured on boards that have clocks and external memory.

Because the user can wire the matrix at will, an FPGA can have many independent circuits running concurrently; if such a circuit depends on a clock (which is very common), it is called a process. One example of processes running concurrently could be writing to memory and reading to memory; while a computer would require two clock cycles to perform this task (read, then write), an FPGA can have a writing process and a reading process working concurrently, increasing the speed at which reading/writing can be performed. Processes are described in greater detail in the following section.

4 Implementation

To implement a wiring to an FPGA, one must describe the wiring using a hardware description language (HDL). The language was chosen to be VHDL (Very high-speed integrated circuit Hardware Description Language), because of the quality and amount of information available. Alternatively, Verilog is another HDL that could have been used.

The month of May and most of June were devoted to learning VHDL using *Circuit Design and Simulation with VHDL* by V. A. Pedroni [9]. Then, the hardware was tested, and modifications had to be made. Finally, we wrote a digital filter VHDL code that could be uploaded to the FPGA.

4.1 Hardware Characterization and Modification

While learning VHDL, we researched for hardware capable of converting analog signals to digital signals, and vice versa. A Terasic DE0 FPGA development board was bought, which includes an Altera Cyclone III FPGA, USB interface, external Random Access Memory (RAM) of 512 MB, 50 MHz clock, and 80 multi-purpose I/O pins [10].

To convert between analog signals and digitals signals, a data conversion card (AD/DA card) with 14-bit resolution (built to infertace with the DE0 board using multipurpose pins) was purchased [11]. The AD/DA card is composed of two data converters (one analog-to-digital or A/D [12], and one digital-to-analog or D/A [13]) and additional circuitry needed to protect the converters and prefilter the input and output [14]. See figure 3 for a graphical representation of the devices.

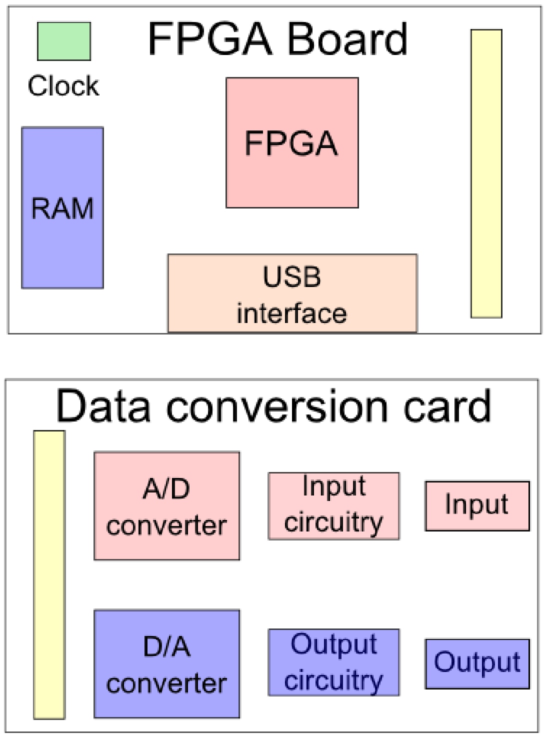


Figure 3: Graphic representation of the FPGA board (top) and data conversion card (bottom) where the major components are drawn. Note that the yellow bands (multipurpose I/O pins) represent the junction point between the two devices. The input and output circuitry are detailed in figure 4.

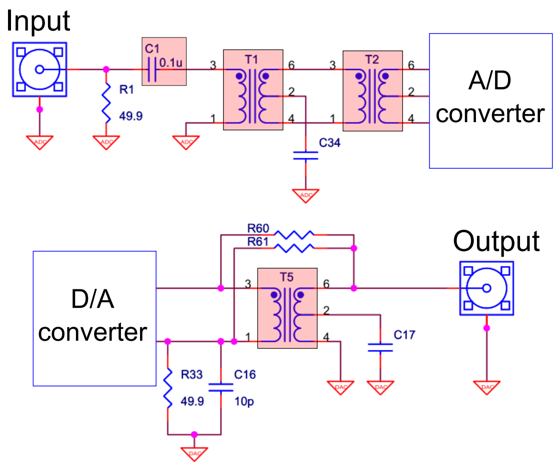


Figure 4: Circuit schematic of the input and output circuits wired into the data converter card. A/D and D/A represent analog-to-digital and digital-to-analog respectively. The components in red rectangles were removed to allow the A/D and D/A converters to measure low frequency signals. Modified from [14].

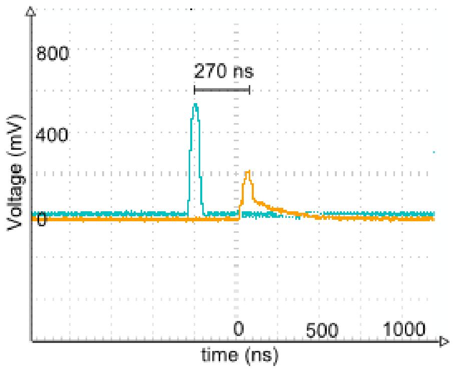
After receiving the hardware, we noticed that the AD/DA card can only convert signals with frequencies between 50 kHz and 20 MHz; this was confirmed by a Terasic engineer [15]. Because the data converters on their own are capable of handling all signals below 50 MHz [12] [13], we set to modify the card to allow the processing of low-frequency signal. Figure 4 shows the schematic of the input and output circuits. We expect that capacitors and transformers in series will inhibit the ability to measure low-frequency signals. Moreover, the transformers make it impossible to add a voltage offset to the input or output of the AD/DA card. Therefore, we removed the capacitor C1 and transformers T1, T2 and T5 in the red rectangles of figure 4.

4.1.1 Group Delay

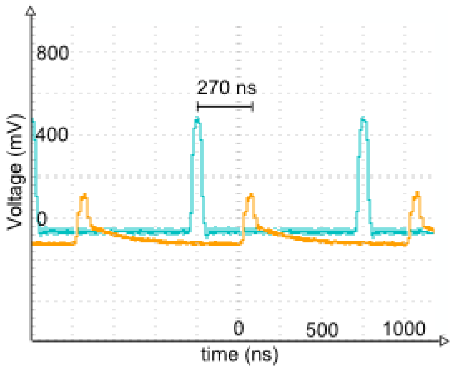
After the modifications, we are ready to test the hardware. The first important characteristic is the AD/DA card intrinsic delay (also called group delay); that is, the time it takes to simply sample and output. A simple code that wires the output of the A/D converter to the input of the D/A converter was uploaded to the FPGA board. Then, to measure the intrinsic delay, we fed pulses with very small duty cycles (less than 2%) to the input of the A/D converter (sampling at 50 MHz) and swep the input frequency between 1 kHz and 1 MHz. A delay equivalent to the sampling period (20 ns during this test) is expected because the data converters take one sampling period to read or write data. Any additional delay represent the group delay of the AD/DA card.

The results for frequencies 8.2 kHz and 1 MHz are presented in figure 5, which show that the group delay of this AD/DA card revolves around 250 ns for all frequencies. The frequency-independence of the group delay is consistent with [5, p. 243].

This group delay imply that AD/DA card can react in 270 ns at best (at a sampling frequency of 50 MHz), which means a correction frequency of 3.7 MHz. This is 37 times faster than what was used by Prof. Sankey at Yale. However, even using a faster external clock, the AD/DA card group delay will always be 250 ns at least, which gives a correction rate of 4 MHz at best.



(a) Input frequency of 8.2 kHz and duty cycle of 0.48 %.



(b) Input frequency of 1 MHz and duty cycle of 2.00 %

Figure 5: Group delay characterization of the data conversion card at a sampling frequency of 50 MHz. The input signal (blue) is a pulse of 500 mV amplitude. The output of the data conversion card is shown in yellow. The time offset between the two peaks shows the group delay to be around 250 ns for low-frequencies and high-frequencies alike (taking into account the 20 ns sampling period).

4.1.2 Intrinsic Transfer Function

The second important characteristic of the AD/DA card is its intrinsic transfer function. To measure this transfer function, we wire the input of the AD/DA card directly to its output, so that we are wiring the transfer function $Y(s)/X(s) = 1$. To measure the transfer function, we use a Zurich Instruments Lock-in Amplifier located in the Sankey Laboratory, and sweep the frequency between 1 kHz and 1 MHz using an input with amplitude of 0.2V.

The results are plotted in figure 6. While the linear phase response is expected from the constant group delay, the amplitude response acts as a low-pass filter with a very small cutoff frequency (less than 10 % of the sampling frequency). We note that changing the sampling frequency does not alter the overall shape shown in figure 6; for example, for a sampling frequency of 1000 Hz, the artifact shown in figure 6 still occurs around $0.5f_s$. This artifact is documented in the DA converter's datasheet [13]. However, due to interference, the gain and phase responses of the artifact is not constant in time. Finally, the last important characteristic of the dig-

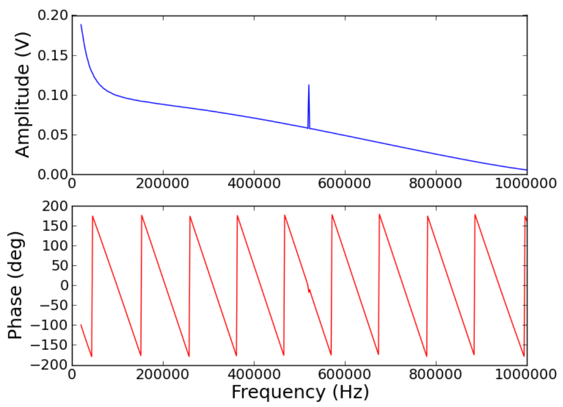


Figure 6: Measurement of the AD/DA card's intrinsic transfer function, taken at a sampling frequency f_s = 1 MHz. The input has an amplitude of 0.2V. We note that an artifact always appears near $0.5 f_s$, but that its amplitude is not constant in time.

ital filter apparatus is the dynamic range. From the converter datasheets, we know the A/D converter can read voltages between 0 and 1V, and assigns a 14 bit number to that voltage (between 0 and 16383) [12]. Moreover, the D/A converter also takes a 14 bit number, and outputs a corresponding voltage between 0 and 1V [13]. This requires that all inputs have a voltage offset so that the input never has negative voltages. Negative voltages are converted just like a voltage of 0V by the A/D converter [12].

Hardware Description Language

The following section describes what we have learned about HDL (and more specifically VHDL) during the first part of the project.

While the use of HDL is similar to computer programming, the philosophy behind HDL "programs" is quite unique. The user describes a function, that is, how the ouputs are related to the inputs. Once this is done, the HDL program is then compiled, which amounts to a translation to a truth table of the same form as 1, and the creation of the simplest logical circuit that implements the truth table. Finally, the compiled program is then run through a fitter, which determines how to wire the transistors in the FPGA. Note that before the fitting operation, the code is entirely portable; it is possible to move between different FPGAs with an HDL program, and the fitter will make sure that the program is wired appropriately.

For example, consider the logical half-adder, which adds two bits A and B, and returns the sum (S), and the carry bit (C). A half-adder can be represented in

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 1: Truth table for a logical half-adder, with inputs A and B, and ouputs sum (S) and carry (C). The sum and carry are computed according to equation 11

logic form as the following operations:

$$\begin{aligned} S &= A \text{ XOR } B = A \cdot \overline{B} + \overline{A} \cdot B \\ C &= A \text{ AND } B \end{aligned} \tag{11}$$

where $\overline{A}$ = NOT A and $A \cdot B$ = A OR B. Alternatively, the half-adder can be described by a truth table, as shown in tab. 1.

By describing equation 11 in HDL, the compiler will try to simplify the logical circuit as much as possible before passing on to the fitting process. The advantage is that the compiler can look for the simplest logical circuit that has the same truth table as table 1, rather than reproducing exactly what the user has written. This is much different than programming languages such as Python, which does exactly what the user write. It also implies that even an HDL beginner with no experience can produce fully-optimized circuits.

4.2 Realizing a Digital Filter

As stated previously, one main advantage of FPGAs over traditional computer is the ability to do computations in parallel. In VHDL, an independant computation is called a process. The idea is to divide our VHDL program into the smallest independant chunks, and translate them into processes; then, interference is minimized, errors are easier to track, and computation is much faster. The final VHDL code is available in Appendix B.

Memory

While there are many ways (in theory) to realize a digital filter using an FPGA, all realizations have a common component: memory. Since we want to do computations using current and previous inputs ($x[n-i]$ for $i \geq 0$) and previous outputs ($y[n-j]$ for $j \geq 1$), it is necessary that the inputs and outputs be stored somewhere. While our first (unsuccessful) attempts involved using external RAM, we will only describe the realization using registers.

A register is a logical circuit capable of holding a binary value for one clock cycle. It is possible to tell the register when to be completely transparent (thus acting as a wire), or when to hold its value until the next clock cycle. There are tens of thousands of registers in a typical FPGA (15 408 in our FPGA [10], each holding one bit of information. By using 14 registers together, it is possible to hold an input $x[n]$ or output $y[n]$ (since any input from the A/D converter or any output from the D/A converter are 14 bit numbers). By using 28 registers, it is possible to hold two values (e.g. $x[n]$ and $x[n-1]$), and so on. The advantage of using registers is that all the information contained in the register array can be read at once. This is much faster than the RAM on our board, which can only manipulate two 16-bit numbers each 55 ns (3 clock cycles at 50 MHz) [10]. Therefore, using registers, it is possible to compute a difference equation of the form of equation 7 in a single clock cycle, whereas a RAM-based computation would have taken three clock cycles per term in the difference equation. Thus, the use of registers enables the use of a faster sampling frequency, and lower group delay due to an almost instant response.

The idea of register memory is the following: assume we want to keep the last six 14-bit inputs; we require 84 registers. We arrange those registers into a vector; the first 14 registers are one coordinate, then the registers 15 to 28 are another coordinate, and so on. At the start of a sampling cycle, we store the current input $x[n]$ in the first coordinate of the register array. At the same time, we move the first coordinate to the second, the second coordinate to the third, and so on. Therefore, during the n^{th} sampling step, the register array has the form:

$$(x[n], x[n-1], x[n-2], ..., x[n-5])$$

At the next step (step n+1), the register array is composed as follows:

$$(x[n+1], x[n], x[n-1], ..., x[n-4])$$

The main advantages of these register arrays above are simplicity and speed. It is more simple to use because there is no need to track where each input is in the memory, as with RAM: the input of the current cycle is always in the first coordinate, the input from last cycle in the second, and so on. And the register arrays are faster because all coordinates can be read/written to at the same time.

By using two register arrays in parallel (in two VHDL processes), we can store the A/D converter inputs $(x[n], x[n-1], ...)$ in one array, and the filter outputs $(y[n-1], y[n-2], ...)$ in another. Then at each clock cycle, we can retrieve the coordinates we need and prepare to compute the current output $y[n]$.

Arithmetic

To fully implement a difference equation of the form of equation 7, we also need to implement a way of quickly multiplying a factor and an input (or output). The limited amount of transistors in the FPGA means we cannot implement floating point arithmetic (which is used in CPU-based hardware). While floating point arithmetic is the most precise arithmetic in approximating real-numbers arithmetic, it is also the slowest. Therefore, we chose to use fixed point arithmetic, which we now explain.

The fastest arithmetic that can be done on a FPGA is integer arithmetic. Multiplying two integers (in the form of binary numbers) is almost instant, taking only one clock cycle. The idea behind fixed point arithmetic is to use integer arithmetic for improved speed over floating point arithmetic, at the cost of precision. The name "fixed point" refers to the position of the point in the representation. A typical fixed point number is of the form $a.b$, where both a and b are binary numbers. For example, given $a = 01$ and $b = 101$, the number $a.b$ is decomposed as follows:

$$\begin{aligned} a.b &= \left(0 \cdot 2^1 + 1 \cdot 2^0\right) + \left(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}\right) \\ a.b &= (1_{10}) + (0.5_{10} + 0.125_{10}) = 1.625_{10} \end{aligned} \tag{12}$$

where the first parenthesis is the binary representation of a, the second parenthesis is the binary representation of $.b$, and the subscript $_{10}$ represents numbers in base 10. In a fixed point arithmetic, we must fix the length (in bits) of b, called the precision of the calculation. For example, set the precision to p. Then, given a multiplication $a.b \times c.d$, we can use integer arithmetic as follows:

$$\begin{aligned} a.b \times c.d &= (a + .b) \times (c + .d) \\ &= \frac{(a + .b) \cdot 2^p \times (c + .d) \cdot 2^p}{2^{p+1}} \\ &= \frac{(a \cdot 2^p + b) \times (c \cdot 2^p + d)}{2^{p+1}} \end{aligned} \tag{13}$$

The logic of equation 13 is similar to computing $0.3 \times 0.5 = (0.3 \cdot 10 \times 0.5 \cdot 10)/100 = (3 \times 5)/100 = 0.15$. We note that in the last line of equation 13, the numerator is an integer multiplication, and the denominator is also an integer. This means that the fixed-point arithmetic developped this summer only require a combination of basic logic gates, and thus the computation speed is only limited by the propagation speed of the electric field in the FPGA [8].

To approach the accuracy of floating point arithmetic, we chose to implement fixed point arithmetic with a precision of 32; the factors of equation 7 (typically real numbers) are translated using a computer into fixed point numbers with 32 numbers after the point. Note that it is very simple to change this precision to any desired number, and that it does not affect computation speed.

5 Testing

This section describes the testing done using the arbitrary filter program from section 4. For each test filter, a measurement of the frequency response was made using the lock-in amplifier. Each filter was set to a sampling frequency of 1 MHZ; this is done by scaling the 50 MHz sampling clock. As we will see in section 5.1, this scaling is not precise.

To account for the AD/DA card's intrinsic frequency response, we proceeded to normalize the measurements.

This normalization procedure is made of two processes. First, the test filter's amplitude response is divided by the AD/DA card's amplitude response; this gives the test filter's normalized gain response. Then, the AD/DA card's phase response is subtracted from the test filter's phase response, which yields the test filter's normalized phase response. The experimental procedure for measuring the transfer functions is the same as presented in section 4.1.2.

For brevity, the raw measurements from the lock-in amplifier can be seen in the following subsection for the integrator filter; however, for the other filters, the raw measurements are presented in the appendix A, while we only present the normalized frequency responses.

We note that on each frequency response, there is an artifact near one-half of the sampling frequency. This artifact is a feature of the D/A converter [13], as stated in the previous section.

5.1 Integrator Filter

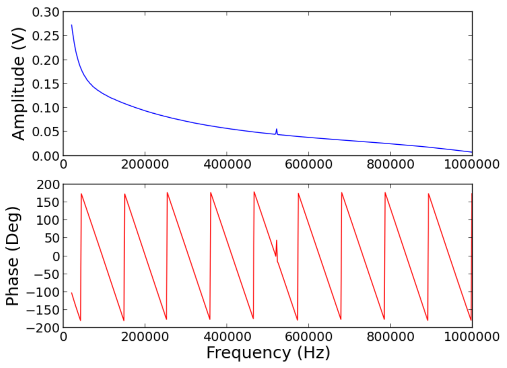
The first filter we test is the digital "integrator filter" [16]. An integrator filter has the following z-transfer function:

$$I(z) = \frac{z}{z-a} = \frac{1}{1-\frac{a}{z}} \tag{14}$$

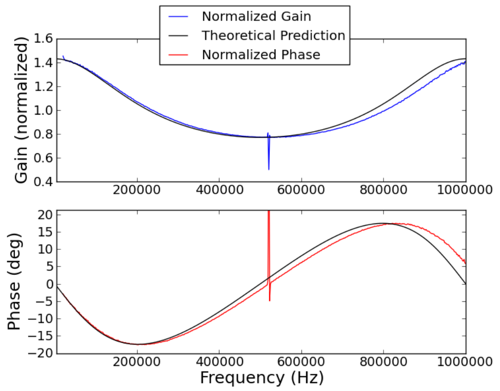
From equations 7 and 9, the right side of equation 14 indicates that the integrator filter has a difference equation of the form $y[n] = x[n] - a \cdot y[n-1]$. Note that the integrator filter is not always stable; since we require that all poles of the z-transfer function are located inside the $|z| < 1$ zone, an integrator filter is stable only if $|a| < 1$. For this particular test, we choose $a = 0.3$.

The raw measurement of frequency response from the realization of the z-transfer function of equation 14 with $a = 0.3$ is presented in figure 7, along with the normalized measurement. The normalized measurement is compared to the theoretical frequency response (in black).

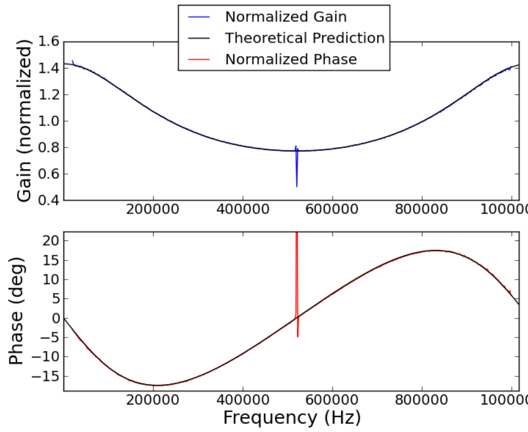
We can see that the normalized response on figure 7(b) has the same overall shape as the theoretical curves.



(a) Raw frequency response



(b) Normalized frequency response compared to theoretical prediction



(c) Normalized frequency response with theoretical predictions scaled for a sampling frequency of 1.04 MHz

Figure 7: Response measurements of an integrator filter with z-transfer function of the form of equation 14 with $a = 0.3$. Fig. (a) shows the raw frequency response for the digital filter. Fig. (b) presents the frequency response normalized to the AD/DA card's intrinsic transfer function presented in figure 6, overlayed to the theoretical predictions at a sampling frequency of 1 MHz. Fig. (c) shows an instance of the theoretical curves adjusted for a 1.04 MHz sampling rate. We can conclude that the digital filter apparatus has an effective sampling rate close to 1.04 MHz.

A possible explanation for the horizontal difference in scaling between the theoretical predictions and the normalized curves might be due to the sampling clock not being exactly 1 MHz; the sampling clock is a scaling of the 50 MHz clock on the FPGA board, and the 50 MHz clock has a frequency uncertainty of $\pm 5\%$ [10].

We can conduct a visual test to try and approximate the effective sampling frequency. By plotting the theoretical curves over a frequency of 1.04 MHz (as opposed to 1 MHz in figure 7b), the normalized frequency response of the integrator filter fits the theoretical predictions better. Figure 7c shows the result of the visual test.

This visual test shows that while we write the FPGA to sample at 1 MHz, it is sampling near 1.04 MHz in reality, We will adjust the theoretical curves to a sampling frequency of 1.04 MHz for the rest of this work, which we call the "effective sampling frequency".

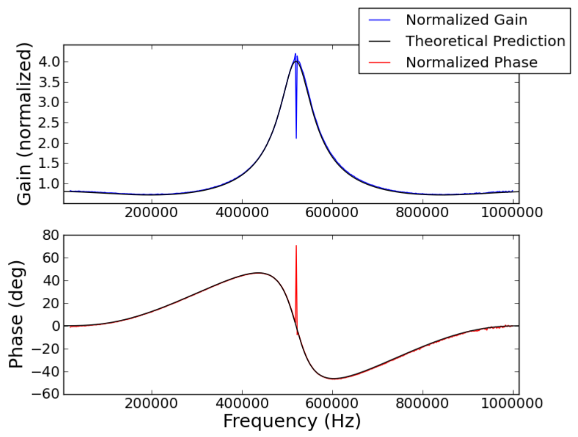


Figure 8: Normalized frequency response measurement of a resonator with z-transfer function of the form of equation 15 with $(b_1, b_2) = (-0.5, 0.25)$, and an effective sampling frequency of 1.04 MHz.

5.2 Resonance Filter

The second filter to be tested is a resonance filter. The simplest digital resonance filter has the following z-transfer function:

$$R(z) = \frac{1}{1 + b_1 z^{-1} - b_2 z^{-2}} \tag{15}$$

where $0 < b_2 < 1$ determines the width of the resonance peak (b_2^2 close to one implies narrow peak) [16]. After looking at the frequency response for various values of (b_1, b_2), we choose $(b_1, b_2) = (-0.5, 0.25)$. After uploading the difference equation $y[n] = x[n] - 0.5y[n-1] + 0.25y[n-2]$ into the digital filter apparatus, the frequency response was measured using a lock-in amplifier. For brevity, the raw measurement is presented in figure I. The normalized measurements are presented in figure 8, along with the theoretical curves adjusted for an effective sampling frequency of 1.04 MHz.

Again, the normalized measurements closely follow the theoretical predictions of equation 15.

5.3 Double Peak Filter

The last filter to be tested was found by playing with the coefficients in a difference equation. This filter has a z-transfer function of the form:

$$R(z) = \frac{-1 + z^{-1} - z^{-2} + z^{-3}}{1 - \frac{1}{5}\left(z^{-1} - z^{-2} + z^{-3} - z^{-4}\right)} \tag{16}$$

This filter is interesting because its phase response has a lot of sharp features, as can be seen in figure 9.

The normalized frequency response measurement is presented in figure 10, overlayed with the theoretical prediction presented in figure 9. The raw measurements are presented in figure II. Again, ignoring the artifact, the results are in sync with the theoretical predictions. This filter is interesting because it completely cancels some frequencies. This test (along with the two previous subsections) demonstrates that the digital filter setup is predictable.

6 Conclusion

The present work detailed the necessary mathematics required for locking a laser to an optical cavity. While laboratory setup is not final, we decided to implement an arbitrary digital filter setup until we are ready. We optimized the performance of the digital filter setup for speed. We developped a procedure in order to implement filters, predict their outcome, and test them. Everything linked to this project is also portable: it does not depend on any particular FPGA, any particular filter, memory available, or clock speed. It can also be quickly customized for more (or less) accuracy, and various data conversion resolutions.

We have met the goals that were set in section 3.1. The design was made adaptable: it can be ported in less than 60 seconds to another FPGA, and modifying its parameters is easy. The setup currently has a group delay low enough to permit a correction rate of 3.7 MHz, an improvement over Prof. Sankey's last setup (with its correction rate of 0.1 MHz).

Our results also demonstrate that FPGAs are well suited to high-speed digital signal processing. The creation of register memory for almost-instant read and write operations, and the fast multiplication routine developped in this work make the digital filter setup only

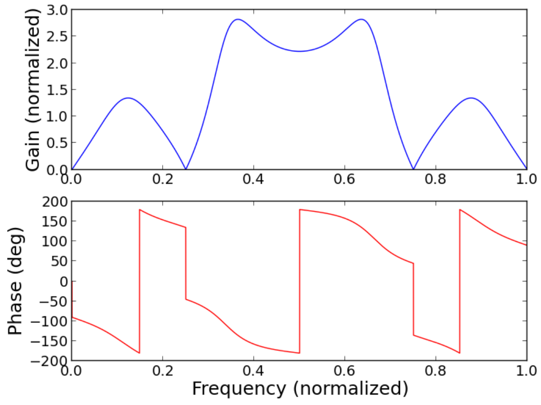


Figure 9: Predicted frequency response of the digital filter with z-transfer function of equation 16. The frequency axis is normalized to the sampling frequency. This filter is interesting to test because of the sharp features in its phase response.

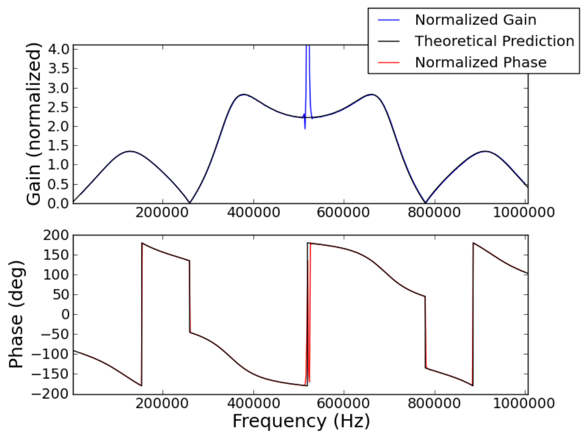


Figure 10: Normalized frequency response measurement of a digital filter with z-transfer function of the form of equation 16, with an effective sampling frequency of 1.04 MHz. The frequency response prediction of figure 9 is plotted over the data. This plot shows that the digital filter apparatus can closely replicate sharp phase shifts.

limited by its on-board clock.

We also showed that the digital filter's performance was predictable. The artifact might represent a problem if we continue using our the same AD/DA card. It is unlikely that we can modify the AD/DA card to get rid of the artifact; our best bet is to design our filters so that we are always working in the $\leq 0.5 f_s$ regime. Alternatively, by increasing the sampling frequency, we can "move" this artifact towards higher values of frequency.

Future Development

While we are satisfied with how the project is going, it is far from complete. The first concern is about the hardware: the AD/DA card has a group delay that limits the performance of the digital filter apparatus. In the future, we might want to build our own AD/DA card; this would enable the group to use higher-resolution A/D and D/A converters with lower group delay, and increased input voltage supported.

There is also an elephant in the room: we need to find a way to "cancel" the AD/DA card's intrinsic transfer function. While it is be possible to correct the AD/DA card's gain response for a wide range of frequencies (by multiplying any digital filter we want to implement by a high pass filter), any data conversion will result in a linear phase (as seen on figure 6. Therefore, it might prove difficult to remove completely the influence of the digital filter setup, both phase and gain response.

Another development that we will try is to simulate the transfer functions of the experimental setup (cavity, laser, photodiode) and test the digital filtering techniques presented in section 2. This would enable us to prepare for when the setup is ready.

Also, we might want to add keyboard and screen support to the digital filter apparatus, in order for the apparatus to be computer-independant: it would then be possible to tweak the digital filter without re-compiling a VHDL program and re-uploading to the FPGA.

Aknowledgments

I would like to thank Jack Sankey for trusting me with this project, and for helping me find many bugs. I also thank the members of the Sankey Laboratory team, for fruitful discussions and bug-hunting: Simon Bernard, Alexandre Bourassa, Chris McNally, Christoph Reinhardt and Maximilian Ruf.

All data analysis was done using SPYDER, a scientific-oriented Python development environment, as part of the free Python distribution Python(x,y). The plots were also made using SPYDER, and the Python library matplotlib.pyplot.

All Python and VHDL code described in this work are available upon request at `laurent.renedecotret@mail.mcgill.ca`.

References

[1] Jack C. Sankey. Sankey Laboratory, private communication, July 3rd 2013.

[2] Christoph Reinhardt. Sankey Laboratory, private communication, June 27th 2013.

[3] F. Marquardt, J. P. Chen, A. A. Clerk, and S. M. Girvin. Quantum theory of cavity-assisted sideband cooling of mechanical motion. *Physical Review Letters*, August 2007.

[4] J. Doyle, B. Francis, and A. Tannenbaum. *Feedback Control Theory*. Macmillan Publishing, London, 1990.

[5] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, New Jersey, second edition, 1999.

[6] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, 1997.

[7] C. J. Astrom. *Computer Controlled Systems, Theory and Design*. Prentice-Hall, New Jersey, second edition, 1990.

[8] V. A. Pedroni. *Digital Electronics and Design with VHDL*. Morgan Kaufmann Publishers, Massachusetts, 2008.

[9] V. A. Pedroni. *Circuit Design and Simulation with VHDL*. MIT Press, Massachusetts, 2010.

[10] Terasic Technologies. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=56&No=364&PartNo=4`, 2011. Manufacturer User Guide.

[11] Terasic Technologies. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=278&PartNo=3`, 2013. Manufacturer Datasheet, as part of the "ADA CD-ROM" download.

[12] Analog Devices. `http://www.analog.com/static/imported-files/data_sheets/AD9248.pdf`, 2010. Manufacturer Datasheet.

[13] Analog Devices. `http://www.analog.com/static/imported-files/data_sheets/AD9763_9765_9767.pdf`, 2011. Manufacturer Datasheet.

[14] Terasic Technologies. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=278&PartNo=3`, 2008. Manufacturer Schematic, as part of the "ADA CD-ROM" download.

[15] Terasic Engineer Amy Zhou. Terasic Support, private communication, May 22nd 2013.

[16] Gary P. Scavone. `http://www.music.mcgill.ca/~gary/307/week2/filters.html`, 2013. McGill MUMT 307 (class notes).

Appendix A: Figures

Resonance Filter

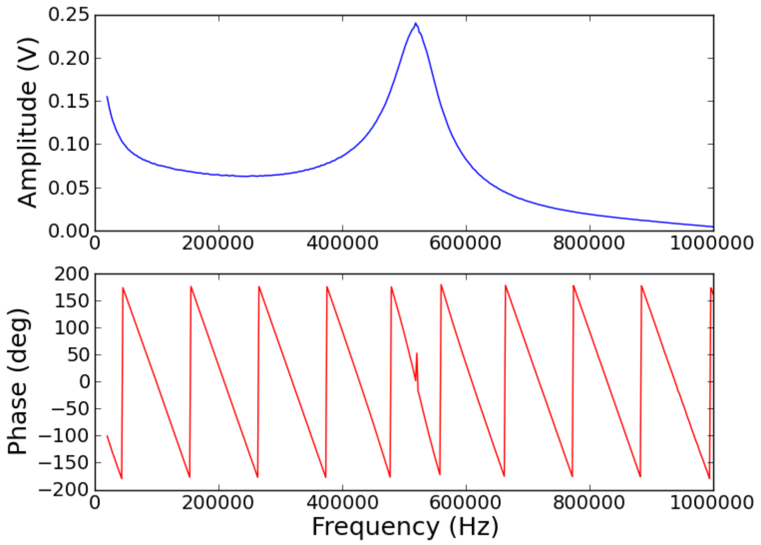


Figure I: Frequency response measurement of a resonator with z-transfer function of the form of equation 15, with $(b_1, b_2) = (-0.5, 0.25)$. The input has amplitude 0.2V.

Double Peak Filter

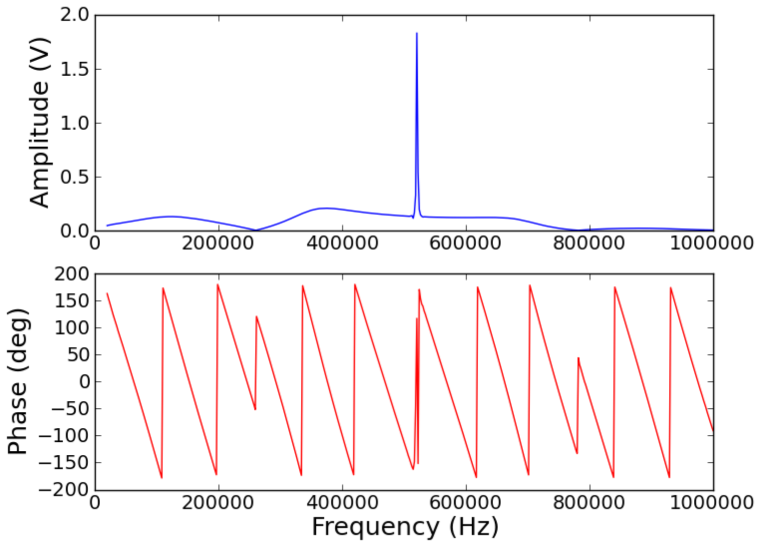


Figure II: Frequency response measurement of a resonator with z-transfer function of the form of equation 16. The input has amplitude 0.2V.

Appendix B: Digital Filtering Code

This appendix contains the VHDL code on which the work of section 5 was done. If you have any questions, do not hesitate to contact the author. Note: everything preceded by a double dash "–" is a VHDL comment.

```
-- Author: Laurent Rene de Cotret
--           for Sankeylab
-- Date: July 31st 2013
-- McGill University
--
-- ******* THIS PROGRAM IS MADE TO WORK WITH THE MODIFIED CHANNELS OF THE AD/DA CARD *******
-------------------------------------------------------------------------------------------------------
LIBRARY IEEE;
       USE IEEE.STD_LOGIC_1164.ALL;
       USE IEEE.NUMERIC_STD.ALL;

LIBRARY IEEE_proposed; -- For fixed point calculations
       USE IEEE_proposed.FIXED_PKG.ALL;
       USE IEEE_proposed.FIXED_FLOAT_TYPES.ALL;
-------------------------------------------------------------------------------------------------------
ENTITY Abstract_FSM IS
       GENERIC (
                     ----- Conversion Parameters -----

                     resolution: INTEGER := 14; -- Resolution of the AD converter is 14 bits for the
                         Terasic AD/DA card
                     precision: INTEGER := 32; -- Number of fractional bits to be used

                     ----- Filter Parameters -----
                     -- y[n] = a0 x[n] + a1 x[n-1] + a2 x[n-2] + ... + b1 y[n-1] + b2 y[n-2] + ...
                     -- Change the following coefficients to create a digital filter.
                     a0: REAL := 0.0;
                     a1: REAL := 0.0;
                     a2: REAL := 0.0;
                     a3: REAL := 0.0;
                     a4: REAL := 0.0;
                     a5: REAL := 0.0;

                     b1: REAL := 0.0;
                     b2: REAL := 0.0;
                     b3: REAL := 0.0;
                     b4: REAL := 0.0;
                     b5: REAL := 0.0;
                     b6: REAL := 0.0
                     );
       -- We note that an "OUT" port means the value is going into the AD/DA Card, while an "IN" port is
           a value coming from the AD/DA card into the DE0 board
       PORT (

       ----- DATA CONVERSION PORTS -----

                     AD_output_A: IN UNSIGNED(resolution-1 DOWNTO 0);  -- Output of the analog-digital
                         converter from channel A
                     AD_output_B: IN UNSIGNED(resolution-1 DOWNTO 0);  -- Output of the analog-digital
                         converter from channel B
                     AD_OTRA: IN STD_LOGIC;              -- Analog-digigal out-of-range indicator for
                         channel A
                     AD_OTRB: IN STD_LOGIC;              -- Analog-digigal out-of-range indicator for
                         channel B
                     AD_enable_A: OUT STD_LOGIC;         -- If driven low then channel A is enabled
                     AD_enable_B: OUT STD_LOGIC;         -- If driven low then channel B is enabled
                     AD_power_on: OUT STD_LOGIC;         -- Power-Down Function for Channel A & B
```

```
                AD_PLL_A: OUT STD_LOGIC;             -- Phase-Lock-Loop (PLL) clock input for
                    channel A
                AD_PLL_B: OUT STD_LOGIC;             -- Phase-Lock-Loop (PLL) clock input for
                    channel B

                DA_input_A: OUT UNSIGNED(resolution-1 DOWNTO 0);
                DA_input_B: OUT UNSIGNED(resolution-1 DOWNTO 0);
                DA_PLL_A: OUT STD_LOGIC;
                DA_PLL_B: OUT STD_LOGIC;
                DA_WRTA: OUT STD_LOGIC;
                DA_WRTB: OUT STD_LOGIC;
                DA_MODE: OUT STD_LOGIC;              -- if 0, mode = interleaved. If 1, mode = dual
                    port

        ----- DE0 PORTS -----

                clk: IN STD_LOGIC;
                reset: IN STD_LOGIC;
                button: IN STD_LOGIC;
                led: OUT STD_LOGIC_VECTOR(9 DOWNTO 0) -- To save power, will be driven logic "low"
                );
        END ENTITY;
------------------------------------------------------------------------------------
ARCHITECTURE main OF Abstract_FSM IS

--\\\\\\\\\ MULTIPLICATION FUNCTION //////////--
-- This function takes an output from the ADC and multiplies it by a real number, and outputs
-- the result in sfixed form.
-- As explained in Pedroni 2008, since the function has 2 inputs (factor and AD_output), the function
    call will look like:\
-- Example <= mult(factor,AD_output);
-- Note that the input voltage level is added to an offset of 8192 ( or 2**resolution / 2);
-- In this case, a negative factor will not destroy all living things with the force of a thousand suns
-- This offset is then subtracted before output. A "MOD" operator is used to "roll" over the range of 0
    to 2**resolution - 1

        FUNCTION mult (CONSTANT factor: REAL; SIGNAL AD_output: UNSIGNED ) --inputs
                RETURN SFIXED IS
                        VARIABLE output: SFIXED(2*resolution-1 DOWNTO -precision);
                        VARIABLE input_level: SFIXED(resolution-1 DOWNTO 0);
        BEGIN
                input_level := to_sfixed(to_integer(AD_output) - 8192,resolution-1,0); --Conversion from
                    offset binary to signed fixed point (from [0 to 16383] to [-8192.0 to 8191.0] for
                    resolution = 14);

                output := resize(input_level * to_sfixed(factor,resolution-1,-precision),output'HIGH,
                    output'LOW); -- Multiplication

                RETURN output;
        END mult;
--/////////////////////////////////////////////--

----- Memory Signals -----

        SIGNAL memory_inputs: UNSIGNED((6*resolution) - 1 DOWNTO 0); -- To memorize the last x inputs,
            change the coefficient before "resolution"
        SIGNAL memory_outputs: UNSIGNED((6*resolution) - 1 DOWNTO 0); -- To memorize the last x outputs,
            change the coefficient before "resolution"

----- Miscellaneous Signals -----

        SIGNAL AD_enable: STD_LOGIC; -- Turns the AD converter on (if AD_enable <= '0') or off (AD_enable
            <= '1')
        SIGNAL output: UNSIGNED (resolution - 1 DOWNTO 0);
        SIGNAL samp_clk: STD_LOGIC; -- The sampling clock of the filter.

BEGIN

--\\\\\\\\\\ SAMPLING CLOCK //////////--
-- This program creates a sampling clock, generally much slower than the 50 MHz
-- Currently, the clock changes every 25 cycles of clk, which menans samp_clk = 1 MHz

        sampling_clock: PROCESS (clk)
                VARIABLE prescaler: INTEGER RANGE 0 TO 24;
        BEGIN
                IF (reset = '1') THEN
                        prescaler := 0;
                        samp_clk <= '0';
                 ELSIF (clk'EVENT AND clk='1') THEN
                        prescaler := (prescaler + 1) MOD prescaler'HIGH;
                        IF (prescaler = 0) THEN
                                samp_clk <= NOT samp_clk;
                        END IF;
                END IF;
        END PROCESS sampling_clock;

--\\\\\\\\\\ MEMORY //////////--
--These process modify memory_inputs (and similarly for memory_ouputs) in the following way: every clock
    cycle, we affix the last input for the AD converter
-- at the beginning of memory_inputs, while deleting the last digits of memory_inputs. Thus, at any
    time, the first "resolution"
-- number of digits of memory_inputs are x[n], the next "resolution" number of digits are x[n-1], etc.

        memory_inputs_management: PROCESS (samp_clk) -- Inputs
        BEGIN
                IF (samp_clk'EVENT AND samp_clk='0') THEN
                        IF (reset = '1') THEN
                                memory_inputs <= (OTHERS => '0');
                        ELSE
                                memory_inputs <= AD_output_A & memory_inputs(6*resolution - 1 DOWNTO
                                    resolution);
                        END IF;
                END IF;
        END PROCESS memory_inputs_management;

        memory_outputs_management: PROCESS (samp_clk) -- Ouputs
        BEGIN
                IF (samp_clk'EVENT AND samp_clk = '0') THEN
                        IF (reset = '1') THEN
                                memory_outputs <= (OTHERS => '0');
                        ELSE
                                memory_outputs <= output & memory_outputs(6*resolution - 1 DOWNTO
                                    resolution); -- memory_outputs <= y[n] & y[n-1] & y[n-2]
                        END IF;
                END IF;
        END PROCESS memory_outputs_management;

--\\\\\\\\\\ OUTPUT CONTROL //////////--
-- This is where the magic happens (i.e. calculations)
        filter: PROCESS (samp_clk)
                VARIABLE temp_output: SFIXED(16*resolution - 1 DOWNTO 0);
        BEGIN
                IF (samp_clk'EVENT AND samp_clk='1') THEN
                -- The following calculation is written in many lines to improve readability, but it will
                be understood by the compiler to be a single calculation
            -- because "output" is a variable, and not a signal.
                   temp_output := resize(mult(a0,memory_inputs(6*resolution - 1 DOWNTO 5*resolution))
                       -- Feedforward
             + mult(a1,memory_inputs(5*resolution - 1 DOWNTO 4*resolution))
             + mult(a2,memory_inputs(4*resolution - 1 DOWNTO 3*resolution))
             + mult(a3,memory_inputs(3*resolution - 1 DOWNTO 2*resolution))
             + mult(a4,memory_inputs(2*resolution - 1 DOWNTO resolution))
             + mult(a5,memory_inputs(resolution - 1 DOWNTO 0))
             + mult(b1,memory_outputs(6*resolution - 1 DOWNTO 5*resolution)) --Feedback
             + mult(b2,memory_outputs(5*resolution - 1 DOWNTO 4*resolution))
             + mult(b3,memory_outputs(4*resolution - 1 DOWNTO 3*resolution))
             + mult(b4,memory_outputs(3*resolution - 1 DOWNTO 2*resolution))
             + mult(b5,memory_outputs(2*resolution - 1 DOWNTO resolution))
             + mult(b6,memory_outputs(resolution - 1 DOWNTO 0))
             , temp_output'HIGH,temp_output'LOW);
            END IF;

            output <= to_unsigned(to_integer(temp_output) + 12*8192,resolution); -- Output has to be
                at most "resolution" number of digits wide. Add one "+8192" per addition in temp_output

       END PROCESS filter;

--\\\\\\\\\\ OUTPUT SELECTOR //////////--
       sel: PROCESS (samp_clk)
       BEGIN
            IF (button= '1') THEN
                   DA_input_A <= output;
            ELSE
                   DA_input_A <= to_unsigned(to_integer(mult(1.0,AD_output_A))+8192,resolution);
            END IF;

       END PROCESS sel;

--\\\\\\\\\\ TEMPERATURE CONTROL \\\\\\\\\\--

       temp_control: PROCESS (reset)
       BEGIN
            IF (reset = '1') THEN
                   AD_power_on <= '0'; -- Logic low = disabled
            ELSE
                   AD_power_on <= '1'; -- enabled
            END IF;
       END PROCESS temp_control;

--\\\\\\\\\\ POWER SETTINGS \\\\\\\\\\--

       AD_enable_A <= '0'; -- AD converter CH A enabled
       AD_enable_B <= '0'; -- AD converter CH B enabled
       AD_PLL_A <= samp_clk;
       AD_PLL_B <= samp_clk;

       DA_MODE <= '1';
       DA_PLL_A <= NOT samp_clk;
       DA_PLL_B <= NOT samp_clk;
       DA_WRTA <= samp_clk;
       DA_WRTB <= samp_clk;

END ARCHITECTURE;
```