Development of a Digital Offset Laser Lock

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Physics from
The College of William and Mary

by

Ian W. Hage

Accepted for Honors
(Honors or no-Honors)

Seth Aubin, Physics (advisor)

Henry Krakauer, Physics

Matthew Haug, Philosophy

Williamsburg, VA
April 26, 2016

Development of a Digital Offset Laser Lock

Ian Hage

Abstract

An offset laser lock is an optoelectronic system capable of stabilizing a laser's optical frequency to a variable offset from another stable laser's frequency. The lock system described in this report relies on a digital frequency comparison of the beat note of two lasers to determine and fix the frequency difference, or offset, between them. This thesis describes the construction of a configurable proportional-integral (PI) control circuit to regulate a laser's frequency based on the error signal generated by a digital comparison of the beat note frequency, as well as the integration of formerly unconnected circuit elements within the laser lock. Testing confirms the proper operation of both the PI circuit and the laser lock system.

1 Introduction

The primary goal of a laser locking system is to fix or stabilize a laser's frequency. Diode lasers become unstable due to temperature drifts as well as mirror vibrations in the laser cavity [1]. Furthermore, the laser diode is extremely sensitive to fluctuations in its injection current. Atomic spectral lines provide a robust frequency reference that is both stable and reproducible. However, spectral lines are few and far apart [2]. Spectral locking is a complicated and expensive procedure, so it is convenient to arrange a system by which one laser can borrow stability from another. An electronic system capable of locking a laser at a variable frequency offset allows for an economically viable lock that can achieve a range of frequencies. The lock system in this thesis is designed to be capable of an offset range of ± 7 GHz. This range encompasses the possible hyperfine splitting of most alkali metals, elements commonly used in atomic experiments[3]. Unlocked lasers such as the titanium-sapphire laser used in the William and Mary Ultra-cold AMO lab exhibit excursions of several MHz in the frequency domain, but the locked laser should be stable to better than 1 MHz when given a stable source.

A paradigmatic application for the offset lock is optical pumping and probing in rubidium (Fig. 1). For ^{87}Rb, the D2 transition between $5^2S_{1/2}$ and $5^2P_{3/2}$ levels is at 780 nm or 384 THz. $5^2S_{1/2}$ is the ground level and has hyperfine levels that have a separation of 6.83 GHz, whereas the four hyperfine levels of the excited state span only 495.8 MHz. Therefore, an offset lock with a range of 7 GHz or

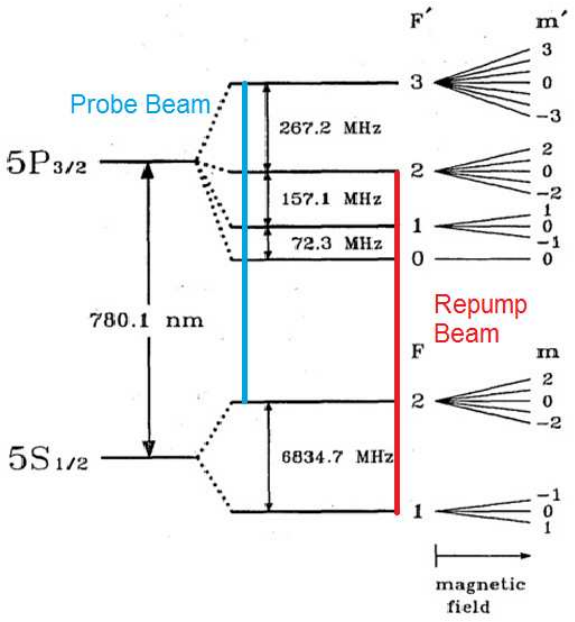


Figure 1: Hyperfine splitting of Rubidium 87's D2 Line. Energy levels will shift due to the presence of a magnetic field. The 6.8 GHz separation between the F=1 and F=2 levels justifies the 7 GHz range of the lock system. Repumper laser at a frequency offset from an already locked probe beam is one potential application of the lock system.

higher should be able to target any D2 transition in ^{87}Rb. A variable offset is particularly useful in the case of an applied magnetic field, in which the hyperfine levels, and thus the optical transitions, are shifted due to the Zeeman effect, typically on the order of a few 1-140 MHz for fields of up to 100 Gauss. In such a case the variable offset allows the laser frequency to track the new hyperfine levels by simply changing the offset proportionally to the magnetic field.

The lock can also be used without an offset to increase laser power by simply allowing the addition of another laser at the same frequency. This technique could be used to increase the power of the Ultra-Cold AMO lab's magneto-optical trap, allowing for more trapped atoms for experiments.

2 Lock Design Overview

The locking system begins with the Receiver Optical Sub Assembly (ROSA). This optical detector's role is to capture light from both the stable and unstable lasers and output the difference in frequency. This is accomplished in the photodetection medium through the principles governing electromagnetic waves. The ROSA detects the intensity of incoming light. For two laser fields, the intensity is given

by the following:

$$I \propto < E_{total}^2 >_\tau = < [E_1 cos(\omega_1 t) + E_2 cos(\omega_2 t + \varphi)]^2 >_\tau \quad (1)$$

Where $E_1$and E_2 are the electric field amplitudes of waves with frequencies ω_1, ω_2 and phase difference φ. is the time scale for the averaging of the photodetector. Typical laser frequencies are hundereds of terahertz as in the Rubidium example. For compairison the averaging time of the detector is on the order of a few gigahertz. Expanding equation 1 gives:

$$I \propto < E_1^2 cos^2(\omega_1 t) + E_2^2 cos^2(\omega_2 t + \varphi) + 2E_1 E_2 cos(\omega_1 t) cos(\omega_2 t + \varphi) >_\tau \quad (2)$$

The cross term can be expanded with the following trigonometric identity into a terms which have a frequencies that are the sum and difference of the original frequencies, respectively known as heterodynes.

$$cos(u)cos(v) = \frac{1}{2}[cos(u - v) + cos(u + v)] \quad (3)$$

$$I \propto < E_1^2 cos^2(\omega_1 t) >_\tau + < E_2^2 cos^2(\omega_2 t + \varphi) >_\tau + < E_1 E_2 cos((\omega_1 + \omega_2)t + \varphi) >_\tau + < E_1 E_2 cos((\omega_1 - \omega_2)t - \varphi) >_\tau \quad (4)$$

The optical laser frequencies incident on the ROSA are too fast to register electronically. Because of their high speed the squared cosine terms and the summed frequency term are averaged by the photodetection medium. The average of $cos^2(x)$ is $\frac{1}{2}$ and the average of $cos(x)$ is 0. The rightmost term which has a lower frequency will not be averaged so long as its period is greater than the integration time of the electronics.

$$I \propto \frac{1}{2}E_1^2 + \frac{1}{2}E_2^2 + E_1 E_2 cos((\omega_1 - \omega_2)t - \varphi) \quad (5)$$

The only remaining non-constant term has a greatly reduced frequency which is identical to the frequency difference between the the beams. This is exactly the quantity we want to measure and control to lock the laser. The ROSA simply reports the lower heterodyne, or "beat note", as an electronic signal with the same frequency. The lock should be capable of a 7 GHz offset, so high frequency electronics are required to handle such signals.

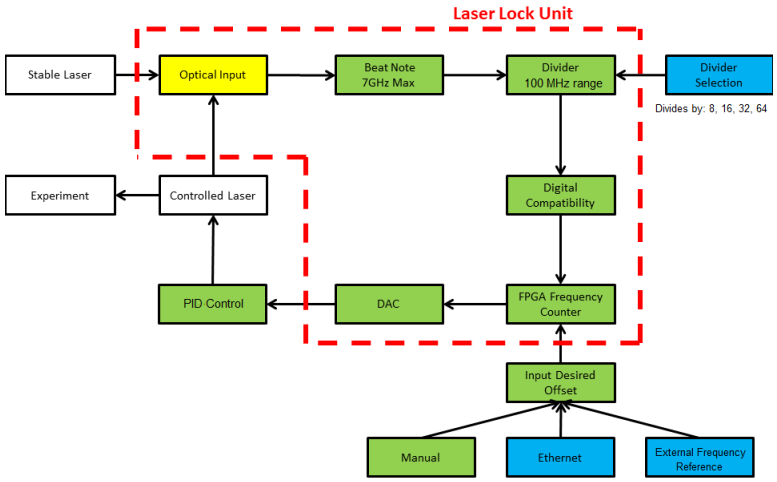


Figure 2: Lock system design. Items within the red dashed line are included in the laser locking unit. Arrows show the flow of user inputs and the control signal. Laser light from both stable and unstable lasers enters the optical receiver (yellow). The generated electronic signals are divided and amplified to suit the frequency counter. The frequency is then measured and compared to the desired frequency offset. The frequency difference is converted by the DAC to a proportional DC voltage and fed to an external PI controller to alter the laser frequency. Green: complete Yellow: incomplete Blue: planned

Figure 2 outlines the complete optical frequency lock loop, whereby the the frequency offset is recorded and fed back as a control signal. After the beat note is converted into an electronic signal by the ROSA, it is fed to a Hittite HMC750LP4 limiting amplifier where it becomes a digital square wave centered on 0V. The square wave is sent to a frequency divider to further reduce the signal frequency. The Analog Devices ADF4007 divider is capable of a division ratio of 8, 16, 32, or 64 over a wide band, reducing the incoming pulse train to a 1-100 MHz scale frequency. Once a DC bias is applied to make the signal compatible with the FPGA, this pulse train is sent to the system's frequency counter. The Field-Programmable Gate Array (FPGA) is a software configurable digital circuit programmed to count the pulses over a 10 ms interval. This frequency count is compared with a desired frequency offset input by the user. The difference of these frequencies is proportional to the amount of error in the laser frequency modulo the divider setting. The FPGA calculates this error and sends the information as serial bits to a digital-to-analog converter (DAC). The DAC outputs an analog signal between -10 and 10 volts that is proportional to this error. A stable lock requires a stable error signal, but digital-to-analog conversion introduces significant high speed digital noise. A low pass inductor-capacitor (LC) filter in a 'Chebyshev' configuration was constructed for the DAC

output to suppress this noise. The final step in feedback control is the separate Proportional-Integral (PI) control circuit that uses the error signal to produce corrective signals that can control the laser and adjust its frequency.

This system has similarities to other laser locking systems. Systems developed at Copenhagen University, the University of Rochester, and the University of Florence all use both use some form of digital laser control [3, 5, 6]. These systems consist in optical phase-locked loops and rely on digital phase-frequency discriminators. Systems at the University of Rochester and the University of Florence utilize both analog and digital phase detection to combine the speed and accuracy benefits of analog systems with the expanded range of digital detectors. The lock system at the University of Florence is particularly noteworthy for its use of FPGA based Phase-frequency discrimination. The system in this thesis contrasts with the above systems by relying soley on frequency detection. It also does not include any analog detection, similar to the system at Copenhagen University.

3 FPGA

The FPGA used in the laser locking system is the Altera Cyclone II mounted on the DE2 development board which includes a 50 MHz clock, memory, and many I/O interfaces, as well as providing power to the FPGA. The board is programmed via a dedicated USB cable and Altera's proprietary software Quartus. A program written in the Verilog hardware description language configures the FPGA to perform its frequency counting task. Figure 3 charts the function of this program. A previous researcher, Julia Stone, created the FPGA program. My contribution to the program consisted in debugging its communications with the DAC.

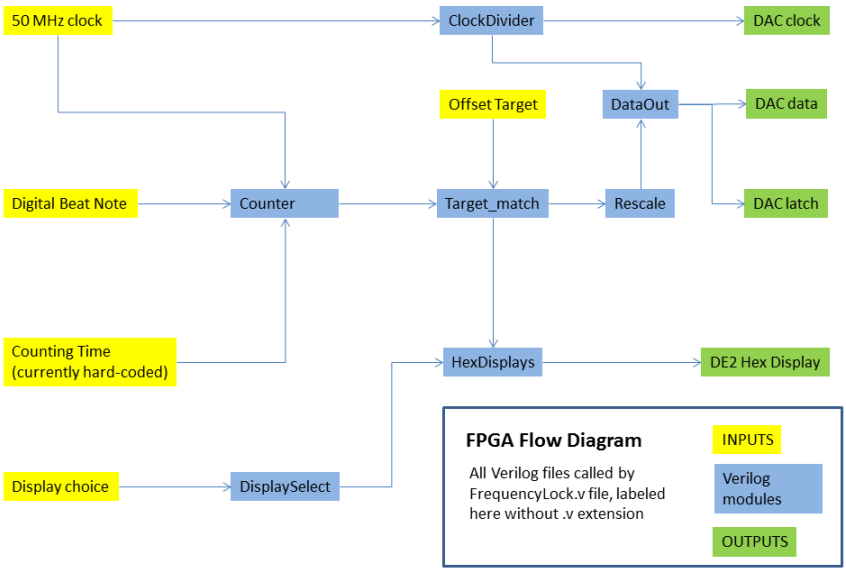


Figure 3: FPGA function. The arrows indicate the passage of digital information from the FPGA inputs and program modules to outputs for controlling the DAC. The beat note is first counted in reference to the system clock (50 MHz), then compared to the offset target. The difference is scaled to a specific voltage in the DAC's output range, then sent as serial bits to the DAC.

The frequency-divided signal is fed to the DE2 board through a bias-T as a square pulse train. the "counter.v" module counts these pulses as well as clock pulses from the on-board clock. After a certain number of clock pulses, the frequency count is passed forward and the count begins again from zero. Counting time is currently set in the "counter.v" file at 10ms, but this could be modified so that the counting time is set via external input. The frequency count is sent as a 16-bit number to the "Target_match.v" module which subtracts the counted frequency and the target frequency to arrive at the frequency error. "Target_match.v" feeds all three values to the board's hex display. The default display is the counted frequency. The other two values, the target and error frequencies, can be displayed by pressing the push-buttons on the bottom right of the DE2 board. The "Rescale.v" module takes the 16-bit error and creates a new 16-bit integer that is scaled to produce the desired DAC voltage. It is here that the ratio of the DAC's voltage response relative to its received integer is set. Finally, the "DataOut.v" module sends the error integer as serial bits to the DAC as well as a latch signal to cause the DAC to clear its register. This process is coordinated by clock signals divided down to the required 5 MHz by "ClockDivider.v". One 5 MHz signal triggers each serial bit sent from the FPGA, while the second signal, on a slight delay, instructs the DAC to record the sent pulse in its

memory.

In the original version of the frequency lock program, the clock signal, DAC latch, and data signals were not coordinated properly, thus resulting in an overlap of the latch and data signals. This error only occurred when the least significant bit of data after rescaling was logic high. The magnitude of the effect depends on the scaling but at the current setting the distortion amounted to a 70mV hop. The corrected "DataOut.v" file is included in the appendix.

4 Digital-to-Analog Conversion

As shown in figure 4, digital-to-analog conversion involves several components. The DAC circuit itself, LC filters, RC filters, and two amplifiers all contribute to a clean analog output.

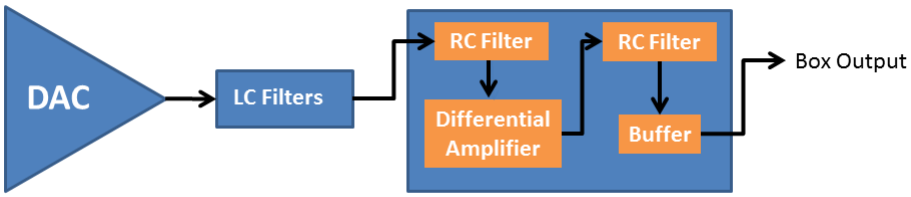


Figure 4: DAC output conditioning. The LC 'Chebyshev' filter attenuates high frequency noise, while the RC filters provide stable attenuation at lower frequencies. The differential amplifier takes the difference of the DAC signal and the DAC's ground voltage to eliminate noise common to both. op-amp buffer provides low output impedance to drive the following circuit.

Digital-to-Analog conversion is currently performed by the Analog Devices AD660 integrated circuit. The DAC outputs between -10V and 10V at a level proportional to the 16-bit integer it receives from the FPGA. This amounts to a maximum resolution of 0.3 mV per bit. Figure 5 displays the schema for wiring the DAC IC as well as an image of the DAC itself.

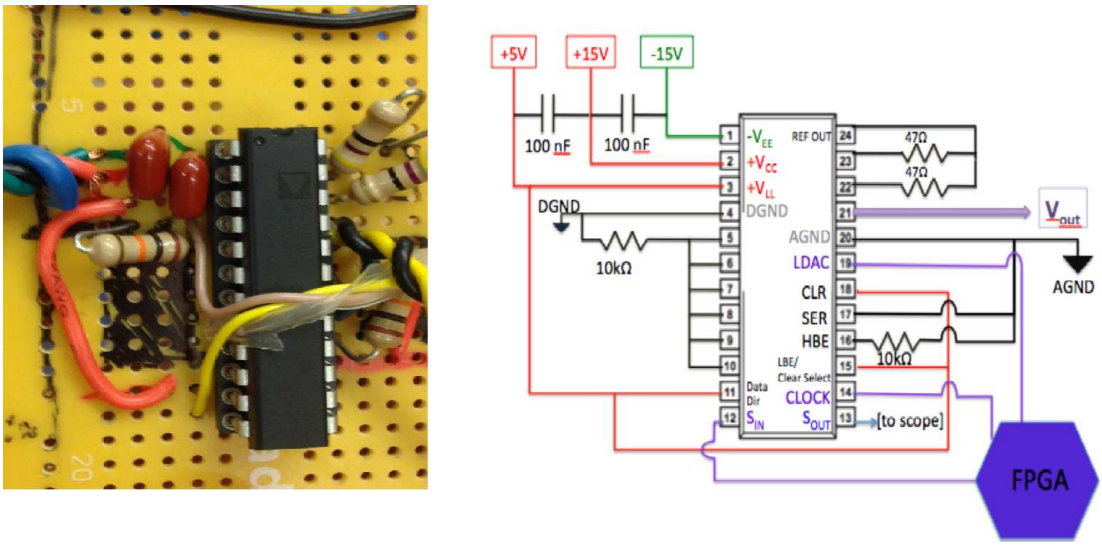


Figure 5: DAC circuit. Power input capacitors (100 uF) reduce noise from the power supply lines. Separate analog and digital grounds are connected in one location only to prevent ground loops. Serial bits (Sin), clock pulses (CLOCK), and the latch signal (LDAC) are routed from the FPGA to their respective pins. Vout is the analog signal output. +5V and grounded pins configure the DAC for serial input mode.

Digital-to-analog conversion can be a noisy process. The sharp edges of digital signals result in high frequency noise that may be present in the DAC's output. Additional sources of noise include electromagnetic interference and any noise in the DAC's power lines.

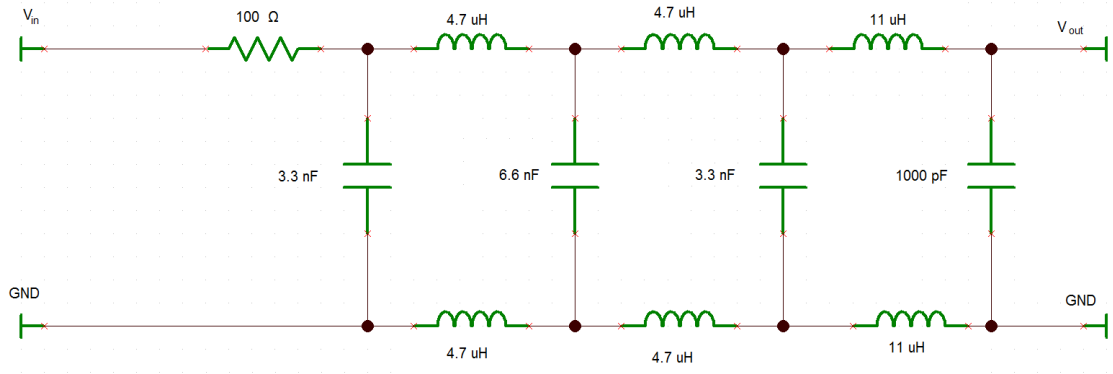
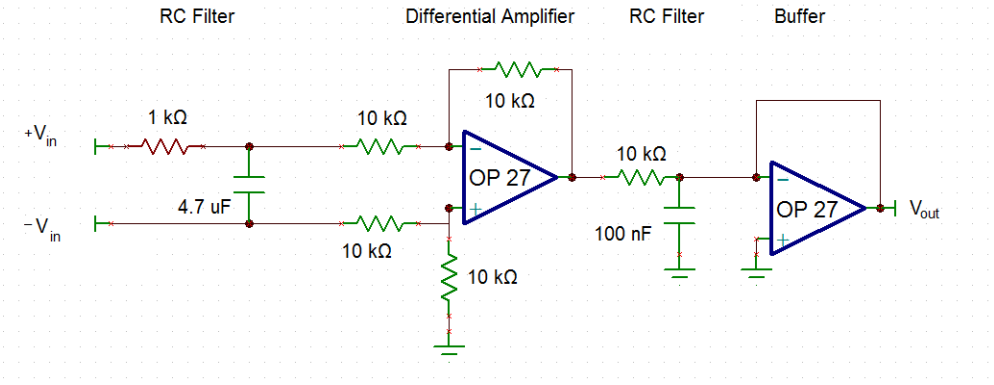


Figure 6: Low-pass Chebyshev filter. Design was modeled in SPICE to attenuate signals above 1 MHz. More filter elements could result in steeper roll-off but risk distorting signals at the knee.

Elimination of such noise is essential to maintain a stable error signal and thus a stable laser. To attenuate this noise, a low-pass Chebychev filter was added to the DAC output. A Chebychev filter

(Fig. 6) consists of inductors and capacitors in a ladder topology and is desirable for its steep roll off. Several filter designs were tested in SPICE and prototyped to arrive at a filter with no excessive distortion of the pass band.



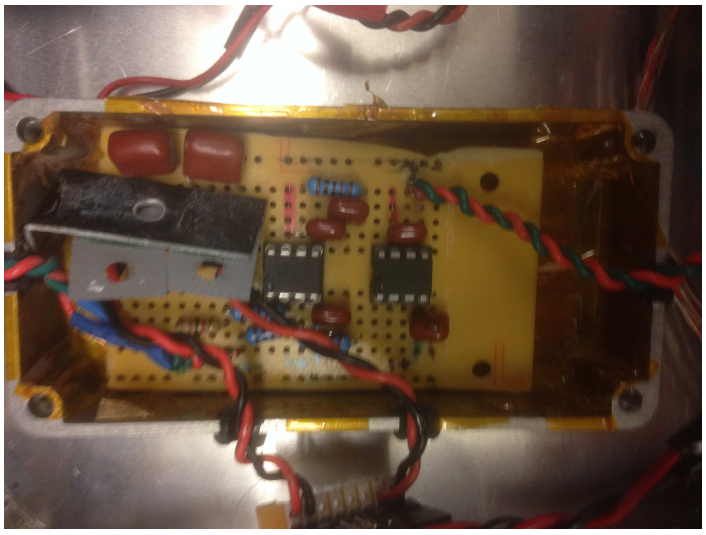


Figure 7: Differential amplifier and buffer circuit. $-V_{in}$ is the DAC reference voltage (ground) after filtering. The differential amplifier subtracts this from the higher voltage line to reduce any noise that is common to both lines. The ground and power of the op-amps are isolated from the other systems in the main box. Voltage regulators within the enclosure are thermally connected to it via an aluminium strip due to space considerations.

After passing through the high frequency LC filter, the signal enters a final circuit (Fig. 7) to further reduce noise and buffer the box's output. First the signal enters through a a resistor-capacitor (RC) filter before passing both the signal and 'ground' into a differential amplifier. By taking the difference, any noise that is common to both lines will cancel out, resulting in a cleaner signal. This difference is then filtered a final time before passing into an op-amp buffer to lower the output impedance. The op-amps that make up the buffer and differential amplifier are powered by local ± 15 V regulators.

Besides filtering, other steps were taken to reduce noise including placing circuit elements in metal enclosures to reduce electromagnetic interference and clasping ferrite magnets on various power lines to increase their inductance.

The lock box was tested by passing an RF signal from a Direct Digital Synthesizer (DDS) into the system's limiting amplifier. The FPGA was given a target frequency of 200 MHz (3125 kHz after division by 64). A range of frequencies from 100 MHz though 300 MHz were fed to the box via the limiting amplifier and the DAC output voltage was recorded after filtering (Fig. 8). As desired, there is a clear linear relationship between input frequency and DAC output. With the current FPGA and divider settings, every 64 kHz shift in DDS frequency will result in a 0.9mV change in DAC output. The maximum observed deviation from the linear model was 7mV, or 0.497 MHz in DDS frequency.There exists a small amount of rounding error on top of this measured deviation, due to the fact that the FPGA has no bits to record any frequency under 1kHz. This error is small, less than 32kHz in terms of DDS frequency, but if needed it can be reduced by changing the divider setting. Accuracy might also be improved in some applications by changing the scaling of the data sent to the DAC.

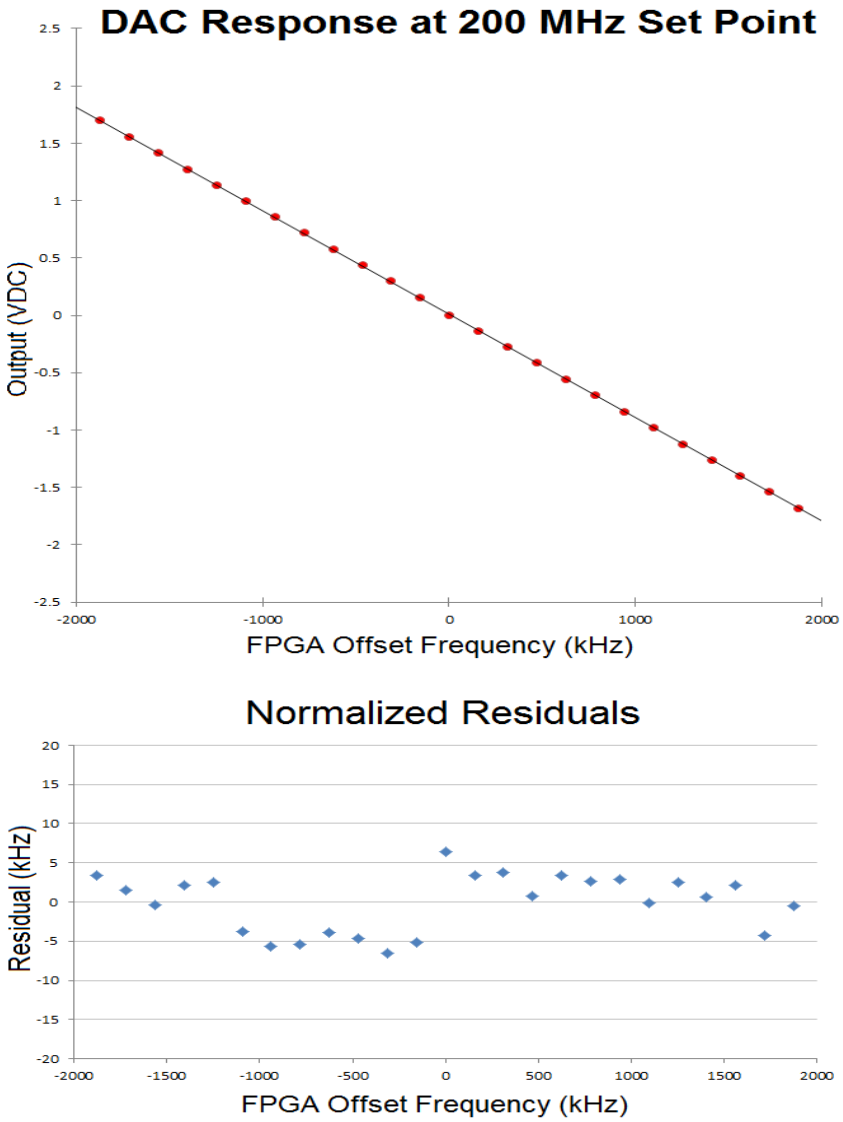


Figure 8: DAC response. Top: DAC output voltage versus FPGA frequency error. The Linear fit shows a 0.9mV DAC output voltage per 64 kHz change in input frequency. Bottom: Residuals of the linear fit. Residuals are normalized in terms of FPGA frequency (kHz).

A measurement of the response time of the DAC output was achieved by dropping the DDS frequency from 300 MHz to 100 MHz in a step function. A fall time of 6.6 ms was observed (Fig. 9). Because this time was recorded only with reference to the DAC output, it does not include the delay due to the 10 ms counting time of the FPGA.

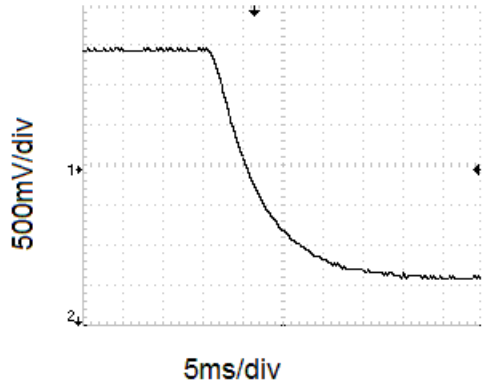


Figure 9: DAC response time. The DDS frequency was stepped from 300 MHz to 100 MHz. The respnse time of the DAC is about 6.6 ms. This must be considered in addition to the FPGA counter's 10 ms counting time.

To measure the effectiveness of the filter's noise reduction, noise traces of the DAC output before and after filtering were compared. The roughly 300 mV peak-to-peak DAC output resulted in around 20 mV of noise after filtering (Fig. 10). This leaves some room for improvement. Expanding the noise trace (Fig. 11) reveals that the loudest noise is nearly a 5 MHz pulse, likely the effect of the DAC's data clock signal from the FPGA. If this noise is not further reduced after ensuring that the DAC is not suffering from grounding issues, a commercial filter could be applied to the locking systems output.

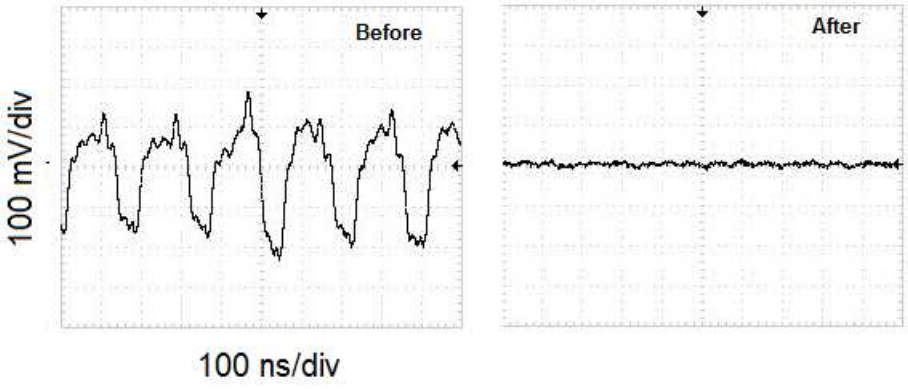


Figure 10: Noise filtering. Left: the DAC output noise before filtering. Right: DAC output noise after filters are applied. The DAC noise approximates a 5 MHz square pulse, and so is likely due to the FPGA data clock.

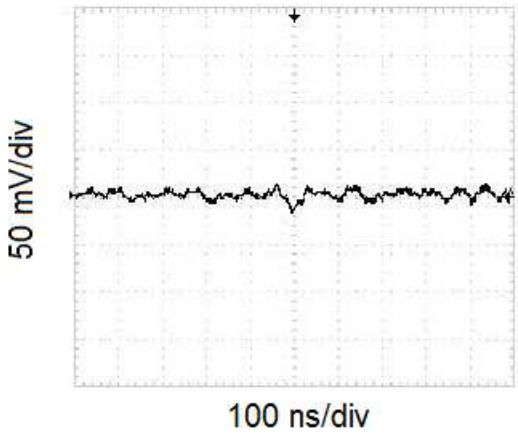


Figure 11: Expanded "after" noise trace. The plot shows the noise after filtering with a higher resolution scale (50 mV/div), the 5MHz noise persists in the DAC output after filters, but is suppressed.

5 PI Control

A large portion of the thesis work has involved the construction of the Proportional Integral (PI) feedback control circuit. The control circuit is housed separately from the other lock components in its own enclosure. The role of the PI circuit is to transform and condition the error signal generated by the FPGA and the DAC into a signal that can be fed back to the laser to control its optical frequency. This laser control signal is the sum of a signal proportional to the error and a signal proportional to the time integral of the error. Some designers of similar systems involving an FPGA have elected to conduct PI control digitally on the same chip as the counter [4]. The primary benefits of such an arrangement is precision control, but at the cost of a significant digital user interface. Laser control circuits in the Ultra-cold AMO lab are uniformly analog. The circuit diagram for the PI circuit is shown in figure 12 below. The proportional (P) signal is generated by a simple inverting amplifier, while the integration (I) is performed by an op-amp integrator. The equations governing these configurations are the well known result of Kirchoffs laws and the operational amplifier rules. The gain of the inverting amplifier is controlled by the ratio of the feedback resistance and the input resistance, whereas the integrator gain is proportional to the inverse product of the feedback capacitance and input resistance.

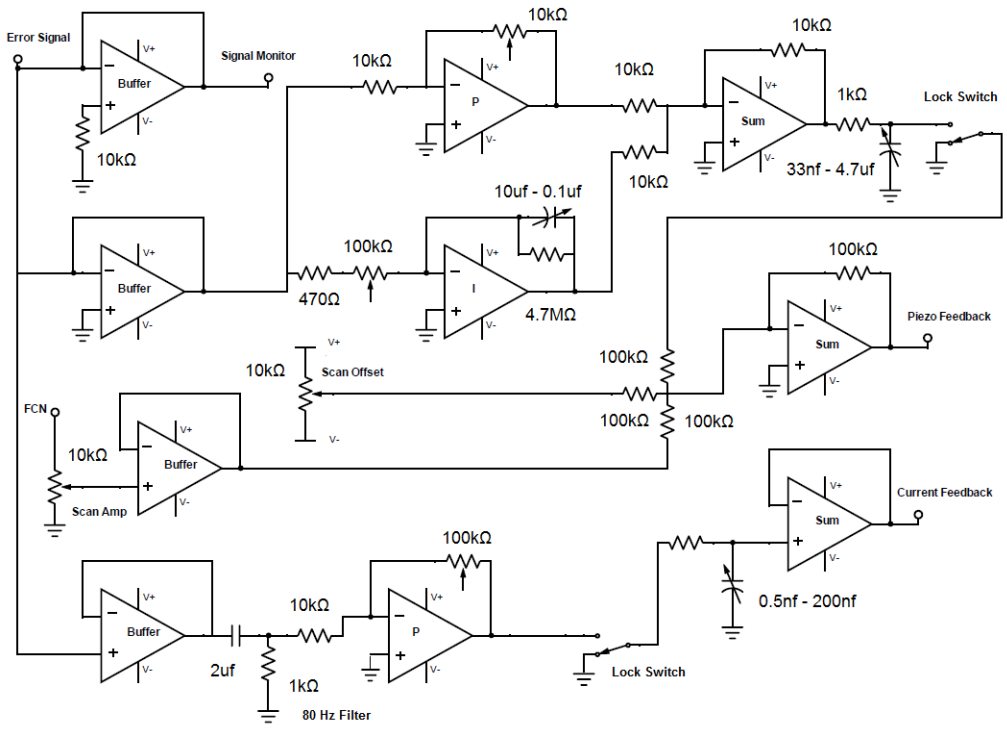


Figure 12: PI circuit diagram. Op-amps are marked 'P' and 'I' to label proportional and integral amplification. Piezo feedback is controlled by the upper part of the circuit, while current control occurs on the bottom "row". Selectable capacitor banks, excluding the one controlling integration gain, are used to select the PI circuit's response speed. V+ is +10V, and V- is -10V.

There are two methods by which a diode laser's optical frequency may be controlled by the PI circuit. The first is to simply alter the electric current poweringg the laser diode. This changes the index of refraction of the diode, and thus changes the laser frequency [2]. This method of control is beneficial for making quick corrections, as the control is executed purely electronically. For larger frequency excursions current feedback cannot be relied on because of the limited range of current-frequency tuning. To correct for large noise excursions, the preferred method is to alter the length of the extended Fabry-Perot cavity via piezo-electronics. Here, the control signal causes a proportional change in cavity length to alter the laser frequency. This process has a large dynamic range than current control but is slower due to its mechanical nature. The PI control circuit produces both current control and piezo control signals through their own sub-circuits. To accomplish this the error signal is fed to both sub-circuits and a high pass filter ensures that only fast excursions (80 Hz and above) reach the current controller. Because the current control is limited to high speed corrections, it is unnecessary to include an integrator, as any errors that persist over time can be handled by the piezo

controller. The piezo controller has infrastructure which allows it to accept an external user signal for scannign laser frequency. This also allows the user to check the integrity of the lock by introducing a known error.

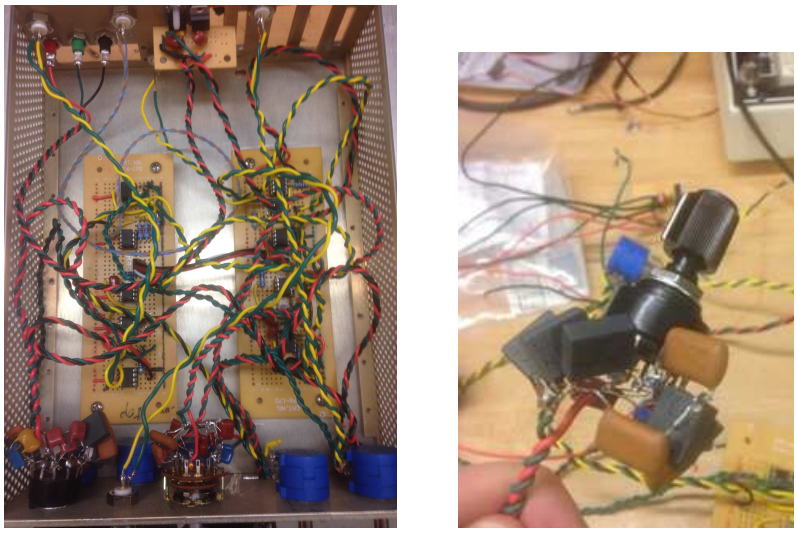


Figure 13: PI feedback control circuitry. Left: PI circuit. Right: Selectable capacitor bank. Red-green lines supply power while yellow-green lines carry the signal.

Because the lock must function with a number of possible laser systems, the control component of the lock must be widely tunable. The inverting amplifiers and integrator of the control circuitry are constructed with potentiometers and capacitor banks to individually control the gain of each component. Capacitor banks were also constructed to create tunable filters on the outputs. The circuit also features a signal monitor so that the DAC output can be easily read for debugging. Figure 13 shows the completed PI circuit and a selectable capacitor bank.

Low noise control is extremely important for a stable lock. Filtered voltage regulators on a separate board power the circuit to avoid noise from the power supply. OP27 op-amps were used for their low noise characteristics. The exception is the final summing amplifiers at the circuits output, which are constructed from LT1498 operational amplifiers for their rail-to-rail capability and larger output current.

Although a true assessment of the PI circuit's effectiveness requires testing with an actual laser, a preliminary test of the circuit was conducted. A square wave 'error signal' was fed into the circuit, and the output was recorded on an oscilloscope. For a mock square wave error signal, the proportional control signal should yield a square wave, while the integral control signal should yield a triangle-like waveform. As figure 14 confirms, these expected patterns can be observed via the oscilloscope traces,

and the gains can be varied such that either the proportional or integral signal dominates.

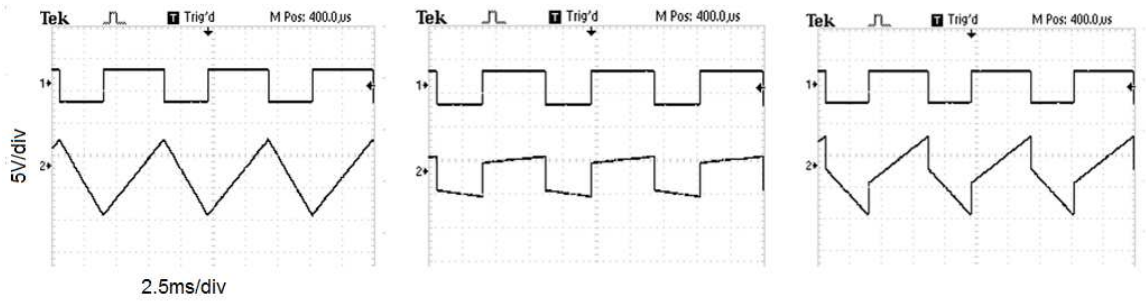


Figure 14: PID response examples. From left to right: integrated signal, proportional signal, mixed signal. These various gain settings all produce the expected response for PI control.

The rightmost image is a trace of roughly mixed integral and proportional gain. We can conclude from this check that the correct PI feedback signal is produced and that the gain of each sub-circuit is tunable. Checks on the filters and monitoring components demonstrate that all parts of the circuit are functional. The final task in assembling a useful PI controller is fixing the circuit within a rack-compatible enclosure (Fig. 15) with independent power. The circuit may be tuned and monitored from the front panel of the enclosure.

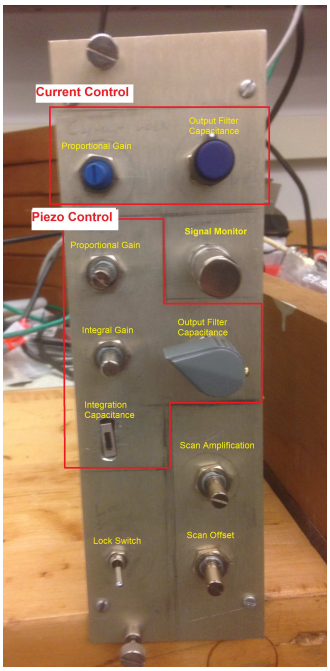


Figure 15: PI front panel. The signal monitor outputs the DAC error signal input to the PI circuit. Potentiometers and the integration capacitance switch provide gain adjustment of the various amplifiers. Rotary capacitor selector banks filter the circuit's output. Scan amplification and offset allow for an added scanning signal to be adjusted. The lock switch allows the circuit to begin or cease feedback control.

6 Lock System Assembly

The box that formerly contained the limiting amplifier, divider, and supporting electronics was determined to be too small for the final locking system. Credit must be given to a previous researcher, Lauren White, for assemling this earlier enclosure as well as configuring and testing its components. A new 17" x 17" box was selected to enclose the laser locking system. Figure 16 below displays the current lock system installed in the new box (compare with Fig. 2). Power is supplied to the box via an external +18V source. This power is distributed via three voltage regulator banks mounted inside the box. The largest of these, located on the back wall of the enclosure supplies power to the radio frequency systems, inducing the limiting amplifier, cooling fan, frequency divider, ROSA, and RF amplifiers. The regulator bank inside a metal enclosure (grey) provides power to the DAC as well as the buffer system. Because the buffers require $\pm$15V, which is beyond the +18V supply, a DC-to-DC converter is required.

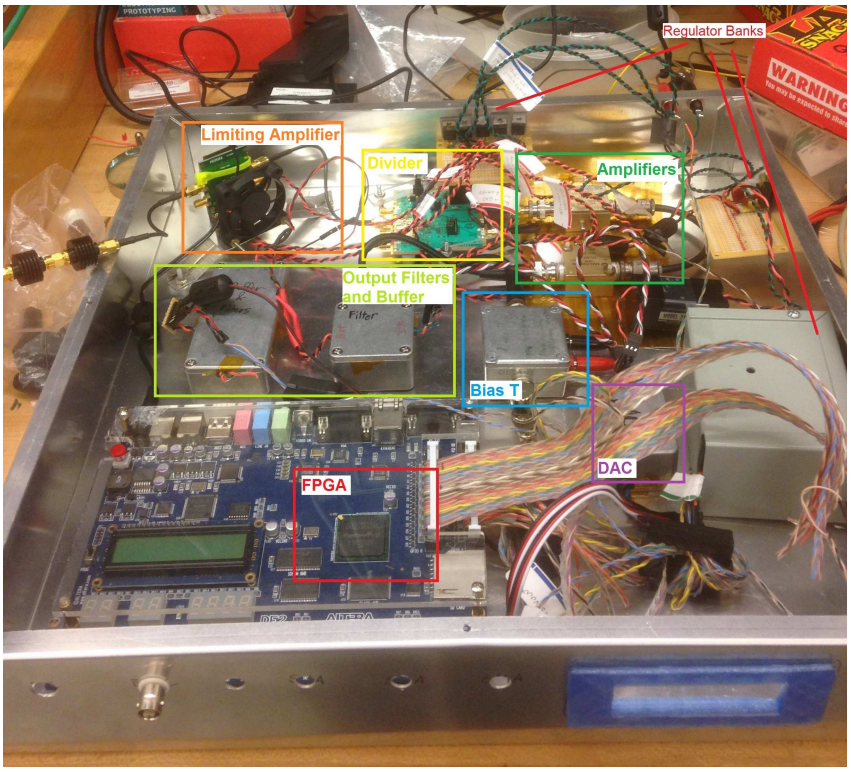


Figure 16: Image of locking system. Displayed are functional components and power supplies. The holes in the front of the box allow for future input/output ports and an Arduino LCD screen. Signal flow is conformed with figure 2. The beat note entersl the limiting amplifier, then the divider, RF amplifiers and bias-T prepare the signal for the FPGA. The FPGA output is converted to analog by the DAC and then conditioned by the output filters and buffer.

The switching noise created by the converter is a major source of electromagnetic interference, which is the reason for enclosing this regulator bank. To further reduce noise, the output of the converter is equipped with a "Pi filter" which greatly attenuates noise at the converter's switching frequency without substantially reducing voltage. The final and smallest regulator bank powers the bias-T and provides extra space for future regulators to power additional components if needed. The aluminum lock box has no electrical contact with the circuit elements, including the regulators which are in thermal contact with the box via non-conducting pads. Tape on the floor of the box prevents the grounded enclosures of the two RF amplifiers (Mini-Circuits) from contacting the box. The outside of the box included a power switch, LED power indicator, and a cut out space for a future Arduino LCD screen, which is discussed further in the next section.

7 Future Work and Conclusion

The optical receiver of the lock system has not yet been installed or tested. Once this is complete, the system can be tested with a laser to determine its effectiveness. An operational lock that meets the lab's noise and speed standards will likely require laser-specific debugging, tuning, and tweaking to succeed. Once basic laser control is achieved, several options exist to increase the lock system's functionality and ease of use. For a more robust lock the current 10 ms counting time of the FPGA is appropriate for slow excursions, but could be augmented by a second frequency counter implemented on the same FPGA. This counter could run in parallel with the current counter, but with a much faster counting time. This faster count would be sent to a separate DAC to generate a faster error signal that could be sent to the PI controller. This could greatly improve the lock's ability to respond to high speed noise.

In addition to improving the lock's effectiveness, several changes could be made to make the lock system easier to use. Currently the frequency offset must be input manually. Ideally, the target frequency offset could be changed in real time by the lab's computer control system. This could be accomplished by an Ethernet compatible Arduino micro-controller programmed to translate the frequency offset sent via ethernet into a bit-wise digital signal for the FPGA. A further advantage of this approach is the Arduino's ability to display relevant information, such as target frequency and measured frequency, on the front of the box via a small screen. Ambitiously, an Arduino could control every user input including counting time and divider setting. Options that could supplement or replace a software based input include a numerical keypad on the box, or if time becomes an issue, simple switches on the front panel of the box for an easier binary input. The divider setting and FPGA counting time could also be set by external switches if the Arduino control route is not pursued. The final useful way that the locking system could receive its target frequency is via an external reference signal of the same frequency. The FPGA is already equipped to count frequency, so this part of the code could be duplicated to count the target frequency as well. The supporting electronics could not be used however, and more components may have to be added to ensure compatibility if the external reference waveform is not compatible with the FPGA's TTL logic. The goal of these proposed changes is to make the lock closer to fully configurable without any need to open the enclosure. The result of these efforts should be a flexible and useful tool for future atomic physics experiments.

References

[1] Y. Zhang, Z. Tian, Z. Sun, S. Fu, IEEE Journal of Quantum Electronics. 50, (802pp), 2014 (10.1109/JQE.2014.2345552).

[2] H. Ludvigsen, C. Holmlund, E. Ikonen, IEEE Transactions on Instrumentation and Measurement. 42, (426pp), 1993 (10.1109/19.278597).

[3] J. Appel, A. MacRae, A.I. Lvovsky, Meas. Sci. Technol. 20, (5pp) 2009 (10.1088/0957-0233/20/5/055302).

[4] Z. Xu, K. Huang, X. Lu, 2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE), 2, (pp.795) 2014.

[5] A. M. Marino and C. R. Stroud, Jr., Rev. Sci. Instrum. 79, 2007 (10.1063/1.2823330).

[6] L. Cacciapuoti, M. de Angelis, M. Fattori, G. Lamporesi, T. Petelski, M. Prevedelli, J. Stuhler, and G. M. Tino, Rev. Sci. Instrum. 76, 2005 (10.1063/1.1914785).

Appendix

```
                                        DataOut
/*sends data out in a series of 16 single bit pulses and then sets LDAC hi when done
*/
module DataOut (data_16bit, clk_data, output_1bit, LDAC);
        input [15:0] data_16bit;
        input clk_data;

        output reg output_1bit;
        output reg LDAC;

        reg [4:0] num; //5-bits binary


        initial begin
                output_1bit = data_16bit[15];
                num = 5'b00000;
                LDAC = 1'b0;
                end
        assign clk_data_test = clk_data;

        always @ (posedge clk_data) begin
                num = num + 1;
                if (num <= 16) begin
                        LDAC <= 0;
                        output_1bit <= data_16bit[16-num]; //cycles through data in
from MSB to LSB
                        end
                else if (num > 16) begin
                        output_1bit <= 0; //Ian's Correction - fixes LDAC conflict /
output jumps. Formerly:"data_16bit[0]" instead of 0
                        LDAC <=1;
                        if (num == 17) begin
                                num <= 0;
                                end
                        end
                end //end always block

                endmodule
```

Figure 17: DataOut.v

```
1  /*Top level module for calculating the difference between a target frequency and a
   calculated frequency of an
2     input signal and sending output to the DAC (AD660)
3     target_freq: input target frequency, for now use switches on FPGA
4              goes from top level module to Target_match.v where the signal freq is
   subtracted from it
5              also have option to display it using DisplaySelect.v which goes to HexDisplays.v
6     display_select: input pushbutton to toggle between display options, for now set at
   target frequency and signal frequency\
7     clock: 50 MHz internal clock
8     input_signal: signal to be measured, represents beat note
9     display_register: data to be displayed on HEX displays, goes from top level module to
   HexDisplays.v
10    extra_HEX: used to set extra HEX LEDs high, turning them off
11
12    counter.v : calculates frequency of input signal by counting rising edges for 10 ms
   (possibly change this sample time in the future)
13             output is counter_output = frequency of input_signal
14    Target_match.v : subtracts target_freq - counter_output to give freq_diff (frequency
   difference)
15    DisplaySelect.v : switches HEX display options, for now set between target_freq and
   input signal freq
16    HexDisplays.v : turns data into instructions for which HEX LEDs to turn on (note 0=on,
   1=off)
17    ClockDivider.v: divides 50 MHz clock down to 5 MHz -- two 5 MHz clocks with a phase
   offset created for DAC timing
18    Rescale.v: reformates freq_diff from 16-bit number with extra bit for sign into a 16bit
   number with bipolar info encoded
19    DataOut.v: sends reformatted number to DAC serially
20
21    freq_diff is ultimate goal of these modules, will be sent to DAC
22    */
23 module FrequencyLock(display_select, clock, target_freq, input_signal, display_register,
   extra_HEX, LDAC, DAC_clk, output_1bit);
24    input [1:0] display_select; //use two buttons to select Hex Display option
25
26    input clock;         //use internal 50MHz clock to synchronize circuit
27    input [17:0] target_freq; //need 18-bits binary to input target freq of up to 250MHz
28                         //goal is to be able to handle 200MHz, should bits be mult. of 8?
29    input input_signal;     //laser or function generator for testing
30
31    //output [15:0] freq_diff; //16-bit binary output to send to DAC (AD660) -- still need
   module to transmit data to DAC
32    output [48:0] display_register; //6-digit decimal number on Hex display with an extra
   digit to display the sign of freq_diff
33    output [6:0] extra_HEX; //used to turn off extra hex displays
34
35    output output_1bit; //data output to DAC (SIN, pin12)
36    output DAC_clk; //DAC clock(CS, pin 14)
37    output LDAC; //tells DAC latch is loaded (LDAC pin 19)
38
39    wire [19:0] counter_output;
40    wire [19:0] signal_freq;
41    assign signal_freq = counter_output;//used to connect counter.v to Target_match.v
42
43    wire [15:0] freq_diff;
44    wire [15:0] freq_diff_output;
45    assign freq_diff_output = freq_diff; //used to display frequency difference
46    wire [15:0] data_input;
47    assign data_input = freq_diff; //connect freq diff calculation to DAC rescaling
48
49    wire [19:0] input_data;
```

```
50      wire [19:0] input_data_choice;
51      assign input_data_choice = input_data; //used to connect data to DisplaySelect.v
52
53      wire extra_bit;
54      wire sign_disp;
55      assign sign_disp = extra_bit; //used to make freq_diff bipolar
56      wire sign;
57      assign sign = extra_bit;
58
59      wire data_clk;
60      wire clk_data;
61      assign clk_data = data_clk; //used to time data output from FPGA to S-in on DAC
62
63      wire [15:0] data_to_DAC;
64      wire [15:0] data_16bit;
65      assign data_16bit = data_to_DAC;//used to connect Rescale to DataOut
66
67      counter counter_result(clock, input_signal, counter_output);
68      //call the counter module with the instance "counter result"
69      //to measure frequency of the input signal
70
71      Target_match target_result(target_freq, signal_freq, freq_diff, extra_bit);
72      //measure difference in input signal freq and target freq
73
74
75      DisplaySelect DisplayChoice(display_select,target_freq, signal_freq, freq_diff_output,
    input_data);
76      HexDisplays HexOut1(input_data_choice, display_register, extra_HEX, sign_disp);
77
78
79      ClockDivider clock_5M(clock, DAC_clk, data_clk);//right now at 2.5MHz for testing
80
81      Rescale rescale_data(data_input, sign, data_to_DAC); //reformats data to be compatible
    with DAC
82
83      DataOut data_output(data_16bit, clk_data, output_1bit, LDAC); //sends data out serially
    to DAC
84
85
86      endmodule
87
88
```

```
1   //selects display for HEX using pushbuttons
2   //00=measured frequency, 01 = target frequency, 10=calculated frequency difference between
    target and signal
3   module DisplaySelect(display_select, target_freq, signal_freq, freq_diff_output, input_data);
4      input [1:0] display_select; //use switch to select display option
5      input [17:0] target_freq; //input in top level module
6      input [19:0] signal_freq; //input in top level module
7      input [15:0] freq_diff_output; //freq diff calculated by target_match
8
9      output reg [19:0] input_data; //will represent choice from display select to be
    displayed by Hex Displays
10
11
12     initial begin
13        input_data = 20'b00000000000000000000;
14        end
15
16     always begin
17        case (display_select)
18        2'b11: input_data <=  signal_freq;
19        2'b01: input_data <= target_freq;
20        2'b10: input_data <= freq_diff_output;
21        endcase
22
23     end //end 2nd always block
24
25     endmodule
26
```

```
1  // The HexDisplay module converts a 19-bit binary number into a 6-digit decimal display
2  module HexDisplays(input_data, display_register, extra_HEX, sign);
3     input [19:0] input_data;
4     input sign; //sign of freq diff
5
6     output reg [48:0] display_register; //sign for first digit and then a 6 digit number
7     output reg [6:0] extra_HEX; //used to set extra displays off (1=off, 0=on)
8     reg [6:0] display0;
9     reg [6:0] display1;
10    reg [6:0] display2;
11    reg [6:0] display3;
12    reg [6:0] display4;
13    reg   [6:0] display5;
14
15    reg [6:0] sign_display;
16
17    reg [22:0] tmp_division; //dummy var
18    reg [3:0] tmp_remainder0, tmp_remainder1, tmp_remainder2, tmp_remainder3, tmp_remainder4,
       tmp_remainder5;
19
20 // initialize the display (i.e. turn all the LEDs off: High = off, Low = on)
21    initial
22       begin
23       display_register = 49'b1111111111111111111111111111111111111111111111111;
24       extra_HEX = 7'b1111111;
25       end
26    always begin
27       case (sign)
28       1'b0: sign_display <= 7'b1111111; //display nothing for positive signal
29       1'b1: sign_display <= 7'b0111111; //display minus sign for negative signal
30       endcase
31       end //end 1st always block  //used for displaying sign of freq_difference
32    always
33       begin
34       // convert input_data in units of counts/1.3ms to KHz
35       //tmp_division = ((input_data*10)/13);
36
37       //convert input_data in units of counts/10ms to KHz
38       // conversion moved to counter module, tmp_divison now just dummy var
39       tmp_division = input_data;
40
41       // 1s display
42       tmp_remainder0 = tmp_division % 4'b1010;
43       display0 = output_singledisplay_7bit(tmp_remainder0);
44
45       // 10s display
46       tmp_division = tmp_division / 4'b1010;
47       tmp_remainder1 = tmp_division % 4'b1010;
48       display1 = output_singledisplay_7bit(tmp_remainder1);
49
50       // 100s display
51       tmp_division = tmp_division / 4'b1010;
52       tmp_remainder2 = tmp_division % 4'b1010;
53       display2 = output_singledisplay_7bit(tmp_remainder2);
54
55       // 1000s display
56       tmp_division = tmp_division / 4'b1010;
57       tmp_remainder3 = tmp_division % 4'b1010;
58       display3 = output_singledisplay_7bit(tmp_remainder3);
59
60       // 10,000s display
61       tmp_division = tmp_division / 4'b1010;
```

```
62          tmp_remainder4 = tmp_division % 4'b1010;
63          display4 = output_singledisplay_7bit(tmp_remainder4);
64
65          //100,000s display
66          tmp_division = tmp_division/ 4'b1010;
67          tmp_remainder5 = tmp_division % 4'b1010;
68          display5 = output_singledisplay_7bit(tmp_remainder5);
69
70          // eliminate zeros on the left
71          if(tmp_remainder5 == 4'b0000)
72             begin
73             display5 = 7'b1111111;
74             if (tmp_remainder4 == 4'b0000)
75                begin
76                display4 = 7'b1111111;
77                if (tmp_remainder3 == 4'b0000)
78                   begin
79                   display3 = 7'b1111111;
80                   if (tmp_remainder2 == 4'b0000)
81                      begin
82                      display2 = 7'b1111111;
83                      if (tmp_remainder1 == 4'b0000)
84                         begin
85                         display1 = 7'b1111111;
86                         end
87                      end
88                   end
89                end
90             end
91          // Generate Display output
92          display_register = {sign_display,display5,display4,display3,display2,display1,display0
    };
93
94          end
95
96       function [6:0] output_singledisplay_7bit;
97          input [3:0] input_number_4bit;
98          begin
99          output_singledisplay_7bit = 7'b1111111;
100
101         if (input_number_4bit == 4'b0000)
102            output_singledisplay_7bit = 7'b1000000;
103
104         if (input_number_4bit == 4'b0001)
105            output_singledisplay_7bit = 7'b1111001;
106
107         if (input_number_4bit == 4'b0010)
108            output_singledisplay_7bit = 7'b0100100;
109
110         if (input_number_4bit == 4'b0011)
111            output_singledisplay_7bit = 7'b0110000;
112
113         if (input_number_4bit == 4'b0100)
114            output_singledisplay_7bit = 7'b0011001;
115
116         if (input_number_4bit == 4'b0101)
117            output_singledisplay_7bit = 7'b0010010;
118
119         if (input_number_4bit == 4'b0110)
120            output_singledisplay_7bit = 7'b0000010;
121
122         if (input_number_4bit == 4'b0111)
```

```
123             output_singledisplay_7bit= 7'b1111000;
124
125         if (input_number_4bit == 4'b1000)
126             output_singledisplay_7bit = 7'b0000000;
127
128         if (input_number_4bit == 4'b1001)
129             output_singledisplay_7bit = 7'b0011000;
130
131         end
132
133       endfunction
134
135   endmodule
136
137
138
```

```
/* divides 50 MHz clock into two 5 MHz clocks with a phase difference
One for DAC clock, the other for data output timing in DataOut.v */
module ClockDivider(clk50, DAC_clk, data_clk);
   input clk50; // internal 50MHz clock
   output reg DAC_clk; //send out 5MHz clock
   output reg data_clk; //5 MHz clock with phase offset to time data entry

   reg [3:0] counter; //register which contains timing for slow clock

   initial begin
      counter = 4'b0000;
      DAC_clk = 1'b0;
      data_clk = 1'b0;
      end

   always @ (posedge clk50) begin
      counter = counter + 1;
      if (counter < 2) begin
         DAC_clk <= 1;
         data_clk <=0;
         end
      if ((counter <5) && (counter >=2)) begin
         data_clk <= 1;
         DAC_clk <=1;
         end
      if ((counter >= 5)&&(counter<7)) begin
         DAC_clk <=0;
         data_clk <=1;
         end
      if ((counter >= 7) && (counter < 9))begin
         data_clk <= 0;
         DAC_clk <=0;
         end
      else if (counter ==9) begin
            counter <= 0; //resets counter
         end
      end //end always block
      endmodule

/*module ClockDivider(clk50, DAC_clk, data_clk);
   input clk50; // internal 50MHz clock
   output reg DAC_clk; //send out 5MHz clock
   output reg data_clk; //5 MHz clock with phase offset to time data entry

   reg [4:0] counter; //register which contains timing for slow clock

   initial begin
      counter = 5'b00000;
      DAC_clk = 1'b0;
      data_clk = 1'b0;
      end

   always @ (posedge clk50) begin
      counter <= counter + 1;
      if (counter < 10) begin
         DAC_clk <= 1;
         if (counter >= 4) begin
         data_clk <= 1;
         end
         end
      else if (counter >= 10) begin
         DAC_clk <=0;
```

```
63            if (counter >= 13) begin
64               data_clk <= 0;
65               end
66            if (counter == 19) begin
67               counter <= 0; //resets counter
68               end
69            end
70         end //end always block
71         endmodule*/
72
```

```
1  //Measures frequency of input signal
2  //to change sample time, change values in 1st always block
3  module counter(clock, input_signal, counter_output);
4     input clock;   //reference clock -- for now at least, use internal 50 MHz clock
5     input input_signal;  //beat note frequency input to beat note counter
6
7  //For 10ms sample time:
8     output [19:0] counter_output;    // beat note counter output, 20-bits binary
9
10    reg [22:0] counttime_test_in;
11    //beat note input counter, gives number of counts/10ms, 23-bits binary
12
13    reg [19:0] counttime_test_out;
14    //beat note output counter, 20-bits binary
15    //gives number of counts/10ms counted by counttime_test_in
16
17    reg [19:0] counttime_ref;
18    //reference clock counter, gives number of counts/10ms, 20-bits binary
19
20    reg ref_count_complete;
21    //ref clock counter has completed count to 10ms, 1 bit binary
22    // 5E5 oscillations (19bits binary)
23
24
25    assign counter_output[19:0] = counttime_test_out[19:0];
26    //attach so beat note count total is sent to output display
27
28    initial
29       begin
30       counttime_test_in = 23'b00000000000000000000000;
31       counttime_test_out = 20'b00000000000000000000;
32       counttime_ref = 20'b00000000000000000000;
33       ref_count_complete = 1'b0;
34       end
35
36
37    always @ (posedge clock) begin //begin at low to high transition of 50 MHz clock
38       counttime_ref = counttime_ref + 1;
39       ref_count_complete = 0;
40       //if (counttime_ref == 16'b1111111111111111) //"stop count" at 65,535 beats of ref
   clock
41       //if (counttime_ref == 19'b1111010000100100000) //"stop count" at 5E5 beats of ref
   clock (10ms)
42       if (counttime_ref == 500000) //count to 10 ms with 50 MHz clock
43          begin
44          counttime_test_out <= (counttime_test_in/10); //counttime_test_out gives number of
   counts by counttime_test_in/ms
45          counttime_ref <= 0;
46          ref_count_complete <= 1; //set high to indicate temp counter is full
47          end
48       end //end 1st always block
49
50    always @ (posedge input_signal or posedge ref_count_complete) begin
51       //begin at low to high transition of beat note signal
52       // or begin at low to high transition of complete reference count indicator
53
54       if (ref_count_complete == 1) begin
55          counttime_test_in <= 0;
56          end
57       else begin
58          counttime_test_in <= counttime_test_in + 1;
59          end
```

```
60
61          end //end second always block
62      endmodule
63
```

```
1  /*sends data out in a series of 16 single bit pulses and then sets LDAC hi when done */
2  module DataOut (data_16bit, clk_data, output_1bit, LDAC);
3     input [15:0] data_16bit;
4     input clk_data;
5
6     output reg output_1bit;
7     output reg LDAC;
8
9     reg [4:0] num; //5-bits binary
10
11
12    initial begin
13       output_1bit = data_16bit[15];
14       num = 5'b00000;
15       LDAC = 1'b0;
16       end
17    assign clk_data_test = clk_data;
18
19    always @ (posedge clk_data) begin
20       num = num + 1;
21       if (num <= 16) begin
22          LDAC <= 0;
23          output_1bit <= data_16bit[16-num]; //cycles through data in from MSB to LSB
24          end
25       else if (num > 16) begin
26          output_1bit <= 0; //Ian's Correction - fixes LDAC conflict / output jumps.
   Formerly:"data_16bit[0]" instead of 0
27          LDAC <=1;
28          if (num == 17) begin
29             num <= 0;
30             end
31          end
32       end //end always block
33
34       endmodule
35
```

```
1   //Reformats data for the DAC
2   //every value of 10000 in this module refers to the choice of 10 MHz frequency difference
    for maximum voltage output to DAC
3   //Refer to final paper for rescaling algorithm
4   module Rescale(data_input, sign, data_to_DAC);
5      input [15:0] data_input; //15 bit number from switches
6      input sign; //extra bit for sign of input
7      output reg [15:0] data_to_DAC;
8
9      reg [15:0] scaling_data; //register used to scale data_input
10     reg [24:0] prescale_calc; //target value times 10,000 used to avoid decimals in verilog
11                        //25 bits binary to fit values above 10 MHz
12     initial begin
13        data_to_DAC = 16'b0000000000000000;
14        scaling_data = 16'b0000000000000000;
15        end
16
17     always begin
18        if (data_input > 10000) begin //set anything over the max threshold equal to 10,000
    (ie. 10 MHz)
19                               //to give max output
20           scaling_data <= 10000;  //anything about 10MHz will get max output (absolute
    value)
21           case(sign)
22           1'b0: data_to_DAC <= positive_conversion(scaling_data); //scale positive number
23           1'b1: data_to_DAC <= negative_conversion(scaling_data); //scale negative number
24           endcase
25           end //end if statement
26        else if (data_input <=10000) begin
27           scaling_data <= data_input;
28           case(sign)
29           1'b0: data_to_DAC <= positive_conversion(scaling_data); //scale positive number
30           1'b1: data_to_DAC <= negative_conversion(scaling_data); //scale negative number
31           endcase
32           end //end else statement
33        end //end ALWAYS block
34
35
36     function [15:0] positive_conversion;
37        input [15:0] data_input;
38        begin
39        prescale_calc = (data_input*32767)/10000+ 32768;
40        positive_conversion = prescale_calc[15:0];
41        end
42        endfunction
43
44     function [15:0] negative_conversion;
45        input [15:0] data_input;
46        begin
47        prescale_calc = (32767*(10000 - data_input))/10000;
48        negative_conversion = prescale_calc[15:0];
49        end
50        endfunction
51
52  endmodule
```

```
1  //Calculates difference between target frequency and measured signal frequency
2  module Target_match(target_freq, signal_freq, freq_diff, sign);
3     input [17:0] target_freq; //desire frequency for laser, input externally with switches,
   18-bit binary number
4     input [19:0] signal_freq; //measured frequency of laser in counter module
5
6     output [15:0] freq_diff;
7     output sign;
8     reg [15:0] subtraction; //register for subtraction operation
9     reg extra_bit; //1-bit register to keep track of sign of the frequency difference
   (0=positive, 1=negative)
10
11       initial begin
12       subtraction = 16'b0000000000000000;
13       extra_bit = 1'b0;
14       end
15
16    assign freq_diff[15:0] = subtraction[15:0]; //attach output counter to output wires
17    assign sign = extra_bit;
18
19    always begin
20       if (signal_freq >= target_freq) begin
21          subtraction <= signal_freq - target_freq;
22          extra_bit <= 0;
23          end
24       else begin
25          subtraction <= target_freq - signal_freq; //how to add in sign information?
26          extra_bit <= 1;
27          end
28
29       end //end always block
30
31
32
33    endmodule
34
```