University of California
Berkeley

EE290C: 28nm SoC for IoT

The Tapeout Class

Digital Filters Report

Channel-Selection Filter, Image-Rejection Filter, Matched-Filter Demodulator and Clock Synchronization

Author
Jonathan Wang

Instructor
Osama Khan
GSI
Edward Wang
Professor
Kristofer Pister

May 12, 2018

1 Overview

For EE290C, my main task was to design the digital filters that go between the analog receiver and the packet disassembler. These modules consist of a channel selection/image rejection filter, a matched-filter demodulator, and a clock-recovery block. This report will go over the fundamental math/algorithm behind each of these modules, as well as how I implemented it. For future students taking this course and undertaking this task, I recommend reading section 7 before skimming through the rest of this report.

2 System Specs

Our system is designed to be compatible with the Bluetooth Low-Energy (BLE) protocol. This means the receiver has to be able to handle 40 2MHz channels in the ISM band. In particular, the channels start at 2.402GHz, incrementing at 2MHz, and end at 2.48GHz, as seen in fig. 1. Inside each channel, the signal is Gaussian Frequency-Shift-Key (GFSK) modulated, with a modulation index of 0.5. This means that on top of the center of a channel's frequency, the ± 1 symbols correspond to a frequency of $\pm$250kHz. The data rate of the system is 1 Mb/s (or 1 mega symbol per second). We target a -17dB immediate side-channel (2MHz) rejection, and a -9dB image channel rejection.

Due to an earlier concern of not being able to sample fast enough for clock and data recovery (the concern itself is no longer valid), the filter module clock has been set at 40MHz, while the rest of the digital system runs at 10MHz. In addition, the analog team has decided to mix the incoming 2.4GHz RF signal down to an intermediate frequency (IF) of 2.5MHz using quadrature mixers. These downconverted I (in-phase) and Q (quadrature) signals are processed by a 5-bit ADC and ported into the digital filters. Low-side injection is used for the downconversion.

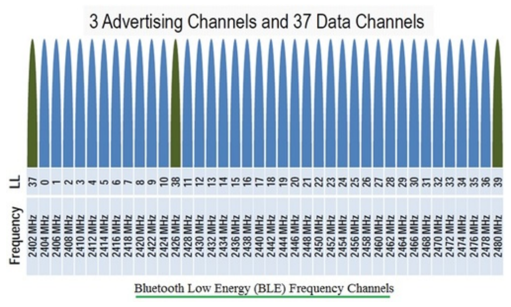

3 Advertising Channels and 37 Data Channels
LL
37 0 1 2 3 4 5 6 7 8 9 10 38 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 39
Frequency
2402 MHz 2404 MHz 2406 MHz 2408 MHz 2410 MHz 2412 MHz 2414 MHz 2416 MHz 2418 MHz 2420 MHz 2422 MHz 2424 MHz 2426 MHz 2428 MHz 2430 MHz 2432 MHz 2434 MHz 2436 MHz 2438 MHz 2440 MHz 2442 MHz 2444 MHz 2446 MHz 2448 MHz 2450 MHz 2452 MHz 2454 MHz 2456 MHz 2458 MHz 2460 MHz 2462 MHz 2464 MHz 2466 MHz 2468 MHz 2470 MHz 2472 MHz 2474 MHz 2476 MHz 2478 MHz 2480 MHz
Bluetooth Low Energy (BLE) Frequency Channels

Figure 1: BLE Frequency Channels

3 Image-Rejection and Channel Selection Filter

3.1 Problem of Image

In a heterodyne receiver, an "image" channel is defined as a frequency band that is equidistant from the LO as our signal band, but on the other side. Let's assume a sinusoidal input at the downconverted IF:

$$A\cos\omega_{IF}t = A\cos(\omega_{in} - \omega_{LO})t \tag{1}$$
$$= A\cos(\omega_{LO} - \omega_{in})t \tag{2}$$

From eq. (1) and eq. (2) we see that whether ω_{in} lies above or below ω_{LO}, it gets downconverted to the same ω_{IF}. This is problematic because for our low-side injection receiver, anything in the $-\omega_{in}$ band will get folded into the same input signal going into the digital filters. This unwanted signal could come from other RF signals at the same frequency bands, or could even come from BLE side-channels. Figure 2 shows an image representation of what was just discussed.

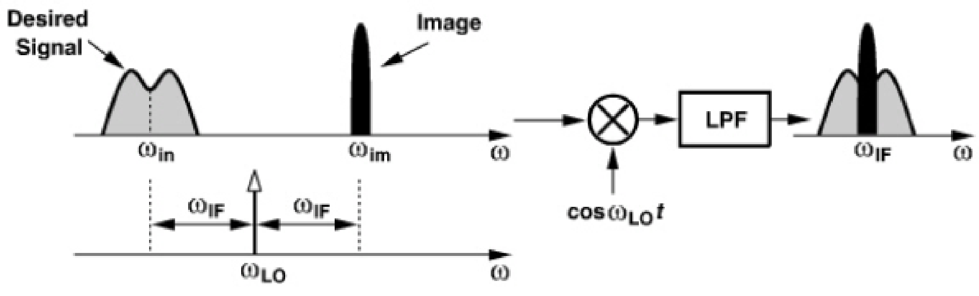


Figure 2: Problem of Image in a Heterodyne Receiver

3.2 Hartley Architecture and Shift-by-90°

In order to reject the unwanted image band, we must first be able to distinguish between negative and positive frequencies. To do that, we define a "shift-by-90°" operation, also known as the "Hilbert Transform". First, consider a tone, with impulses at $\pm w_c$ in the frequency domain (eq. (3)). Then we shift the waveform by 90°. This results in the impulse at $+\omega_c t$ to be multiplied by $-j$, and that at $-\omega_c t$ to be multiplied by $+j$. In the $z - plane$, it is as if we've rotated the $+\omega_c t$ impulse clockwise, and the $-\omega_c t$ impulse counter-clockwise Figure 3 illustrates this operation. Note that the math works out the same for a narrow-band modulated signal.

$$A\cos(\omega_c t) = \frac{A}{2}\left[e^{+j\omega_c t} + e^{-j\omega_c t}\right] \tag{3}$$

$$A\cos(\omega_c t - 90°) = \frac{A}{2}\left[e^{+j\omega_c t - 90°} + e^{-j\omega_c t - 90°}\right] \tag{4}$$

$$= \frac{A}{2}\left[-je^{+j\omega_c t} + je^{-j\omega_c t}\right] \tag{5}$$

$$= A\sin(\omega_c t) \tag{6}$$

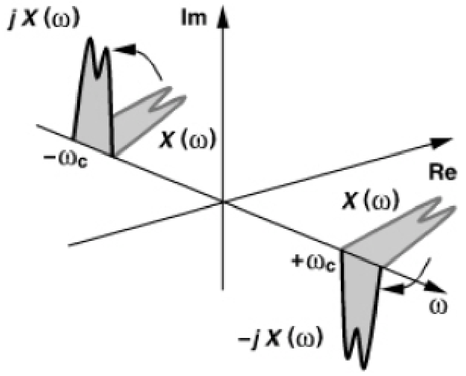


Figure 3: 90° Shift of a Modulated Signal

Now that we've defined the operation, we look into a practical application of it: the Hartley Image Reject Receiver. Figure 4 shows the receiver architecture, with a frequency-domain example signal spectra. The green bands are the wanted signal, and the red, the unwanted image. By passing these signals into the I-arm (fig. 5) and the Q-arm (fig. 6), and ignoring high-frequency mixing contents (filtered out by the LPF) we observe that the IF spectrum emerging from the Q-arm is already the Hilbert Transform of the I-arm. Now, by performing the Hilbert Transform again on the Q-arm (fig. 7), we get an image spectrum that is opposite in sign as that in the I-arm. Upon summing the two signals at points A and C, we obtain the image-free signal at the output (fig. 8)

We can also arrive at this result analytically. Assuming we have an input described by eq. (7), eq. (8)-eq. (10) shows the signal at points A-C, with the final output shown in eq. (11)

$$x(t) = A_{sig}\cos(\omega_c t + \phi_{sig}) + A_{im}\cos(\omega_{im} t + \phi_{im}) \tag{7}$$

$$x_A(t) = \frac{A_{sig}}{2}\cos((\omega_c - \omega_{LO})t + \phi_{sig}) + \frac{A_{im}}{2}\cos((\omega_{im} - \omega_{LO})t + \phi_{im}) \tag{8}$$

$$x_B(t) = -\frac{A_{sig}}{2}\sin((\omega_c - \omega_{LO})t + \phi_{sig}) - \frac{A_{im}}{2}\sin((\omega_{im} - \omega_{LO})t + \phi_{im}) \tag{9}$$

$$x_C(t) = \frac{A_{sig}}{2}\cos((\omega_c - \omega_{LO})t + \phi_{sig}) - \frac{A_{im}}{2}\cos((\omega_{im} - \omega_{LO})t + \phi_{im}) \tag{10}$$

$$x_A(t) + x_C(t) = A_{sig}\cos((\omega_c - \omega_{LO})t + \phi_{sig}) \tag{11}$$

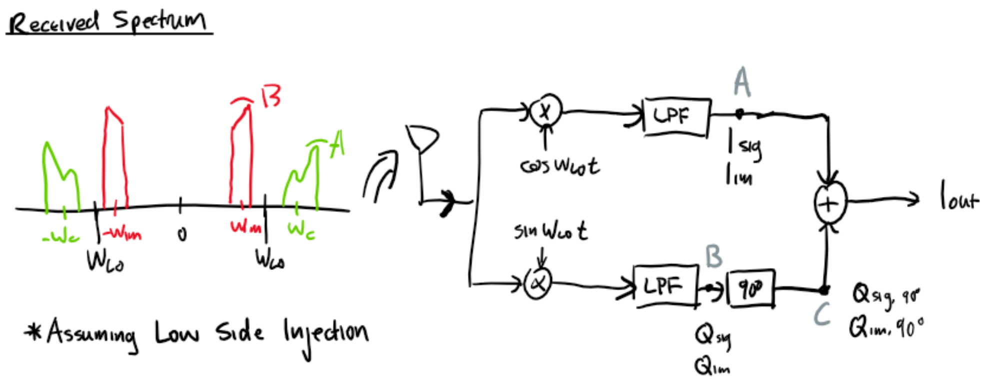
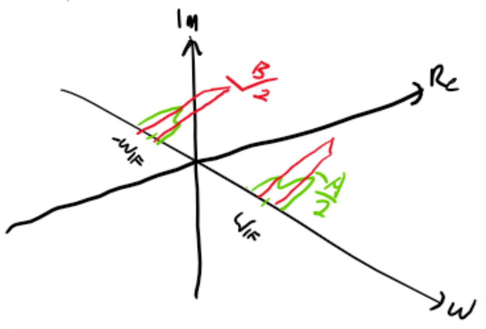


Figure 4: Hartley Image-Reject Receiver

3.3 Channel Selection Filter

As mentioned earlier, the channel's are spaced 2MHz apart, and a -17dB rejection is expected of the immediate side channel. Since the input to the filters is at $f_{IF} = 2.5\text{MHz} \pm 250\text{kHz}$, we need a bandpass filter that passes 2.25MHz to 2.75MHz.

At Point A



I_{sig}

$W_{LO} = (W_c - W_{IF})$

$W_{sig} = W_c - W_{LO} = W_{IF}$

I_{im}

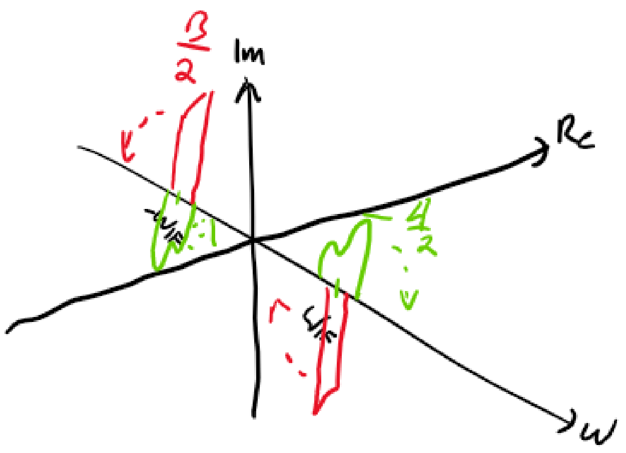
$W_{LO} = (W_{im} + W_{IF})$

$W_{im,new} = W_{im} - W_{LO} \simeq -W_{IF}$

At W_{IF}, you have the positive signal and the flipped image

Figure 5: Spectra at point A

At Point B



Q_{sig} sees low side injection

→ negative of Hilbert Transform

Q_{im} sees high side injection

→ positive of Hilbert Transform

Figure 6: Spectra at point B

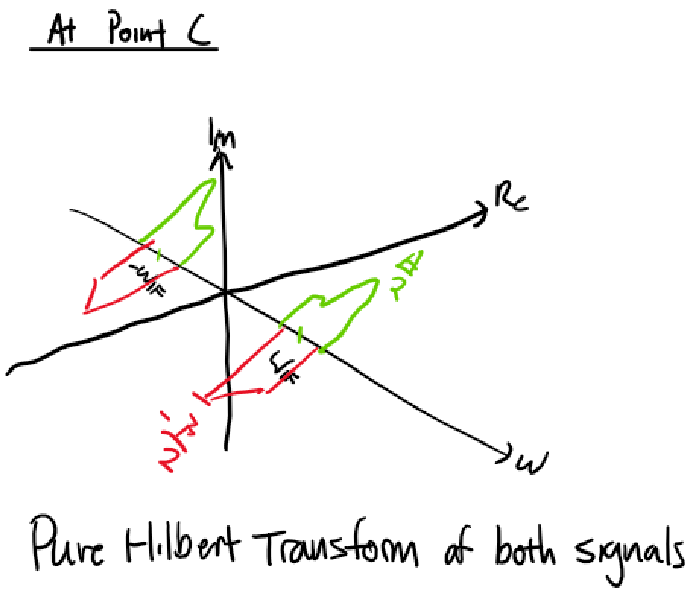


Figure 7: Spectra at point C

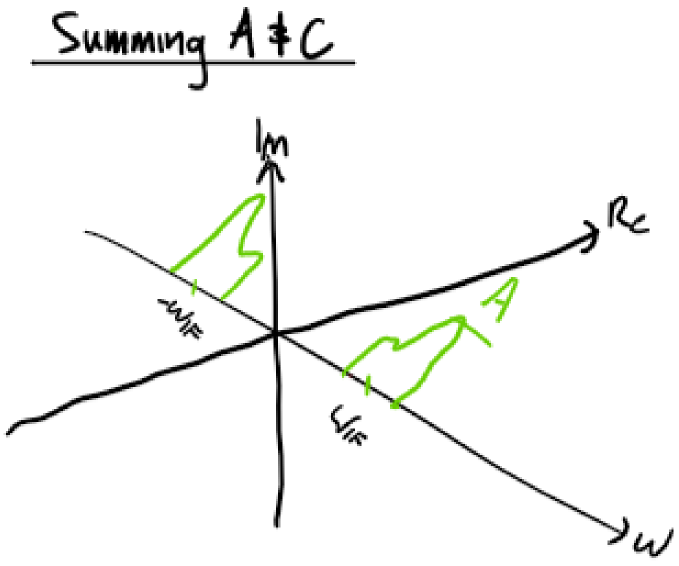


Figure 8: Summed Spectra at Output

3.4 Implementation

The implementation of the Hilbert Transform is done through a set of complex bandpass filters. The filters will perform the function of channel selection, which means the specs of the bandpass filter itself will be determined by the channel-selection specs. The Hilbert Transform in this sense is a simple multiplication ("heterodyne" with) of the filter coefficients with a complex exponential. As lines 22-42 in the MATLAB code (section 9) shows, we first create a lowpass filter with the desired transition band center frequency of 1MHz, a transition band width of 750kHz, passband ripple of 1dB, and stopband attenuation of 17dB. The resulting filter is shown in fig. 10. Then, we frequency shift the filter from a center frequency of 0Hz to 2.5MHz, while applying the complex exponential term. To implement this filter in hardware, we separate it into real and imaginary coefficients, resulting in two real bandpass filters. The architecture is shown in fig. 9, where the real filter is shown in fig. 11, and the imaginary filter is shown in fig. 12. While the architecture features four filters in total, since the matched-filter demodulator following this block does not need quadrature input signals, we do not need to output an image-free Q-channel. This means half of the architecture can be thrown out safely. Additionally, since the sampling frequency is the clock frequency of 40MHz, we are clearly above the Nyquist rate, and therefore do not have any aliasing issues.

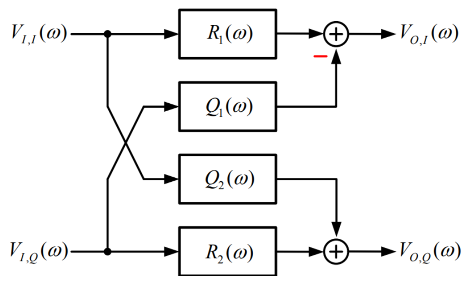


Figure 9: Decompose a complex transfer function to four real transfer functions

3.5 Chisel3 Implementation

The main Chisel module consists of several instances of parameterizeable FIR filters (see *FIRfilter.scala* in the *channel-and-image-reject* GitHub repo). The tap-length, input bit-width, coefficient bit-width, as well as the coefficients themselves can be parameterized. For this design, we have 5-bit inputs, and 8-bit filter coefficients. The filter coefficients are quantized and exported to CSV files in lines 44-54 of section 9, and imported directly by Scala.

The result from the initial lowpass filter specifications is a 47-tap FIR filter, which means each of the two CSVs, representing the real and imaginary filters, have 47 8-bit terms. The *IRCSfilter.scala* module takes the I/Q inputs, instantiates the two corresponding *FIRfilter*s, and combines/downscales the result back to a 5-bit width. Both the inputs and the outputs are signed, so the 5-bit unsigned-to-signed conversion needs to happen between the ADC and the inputs of this module. The output of the module is a 5-bit signed signal,

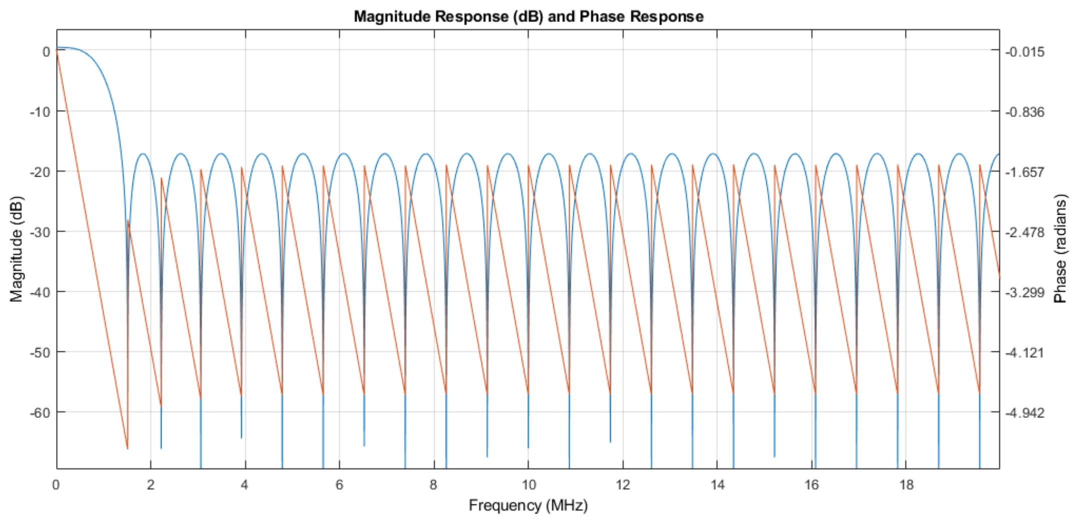


Figure 10: Lowpass Filter Magnitude and Phase Response

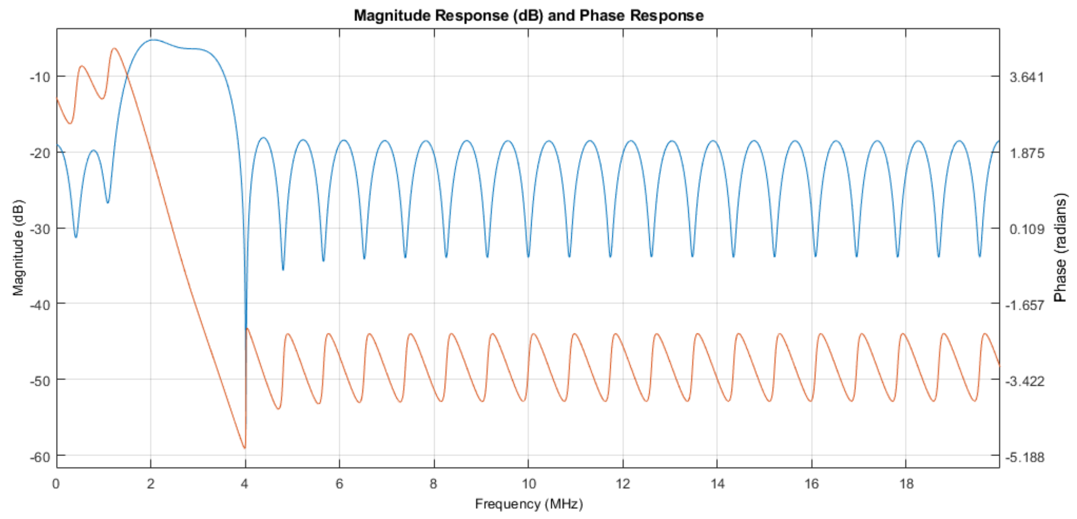


Figure 11: Real Bandpass Filter Magnitude and Phase Response

also operating at 40MHz. This is an image-free I-channel signal, and is directly sent to the demodulator.

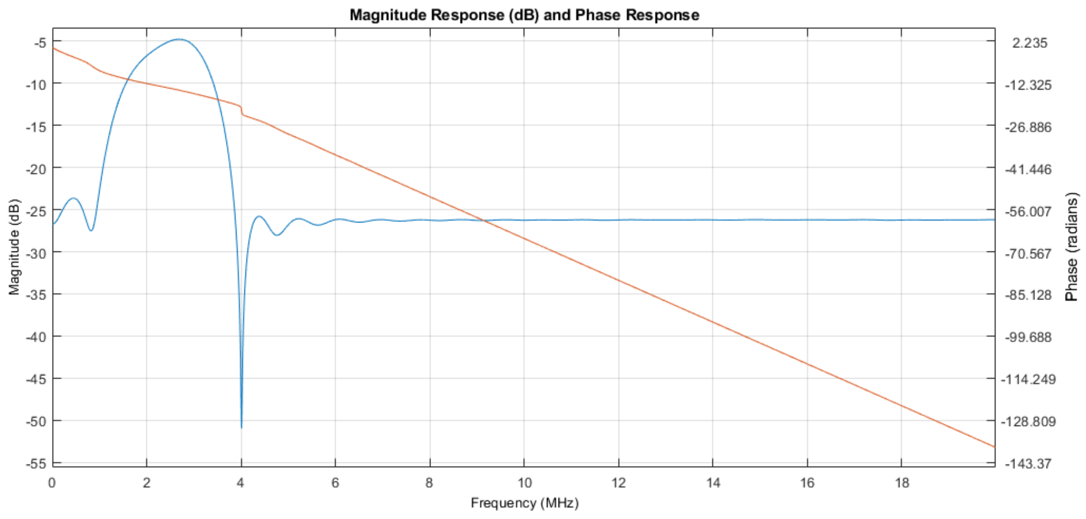


Figure 12: Imaginary Bandpass Filter Magnitude and Phase Response

4 Matched-Filter Demodulator

A brief overview of the math behind the matched-filter demodulator will be given here for reporting purposes. However, if the reader wishes to truly understand, and improve upon the performance of the current implementation, a thorough read of external resources, such as chapter 7.5 of *Contemporary Communication Systems Using MATLAB, 3rd Edition* (Proakis) is recommended. In addition, the following math is given for FSK signals. When used on GFSK, performance deterioration is expected, but should be minimal.

4.1 The Math

Suppose we have a binary FSK system that uses the following two signals

$$\begin{aligned} u_1(t) &= \cos(2\pi f_1 t), \quad 0 \le t \le T_b \\ u_2(t) &= \cos(2\pi f_2 t), \quad 0 \le t \le T_b \end{aligned}$$

Where $f_1 = 2.25kHz$, $f_2 = 2.75kHz$, and $T_b = 1/1Mbps$. Assuming that the channel imparts a phase shift of $\phi = 45°$ phase shift on each of the transmitted signal. The received signal, assuming noiseless, is therefore:

$$r(t) = \cos\left(2\pi f_i t + \frac{\pi}{4}\right), \quad i = 1,2, \quad 0 \le t \le T_b$$

Since at the receiver end, we do not have the PLL to estimate this phase difference, we utilize an incoherent receiver scheme, where two correlators per signal waveform (therefore a total of 4 correlators for binary FSK) are used. The received signal is correlated with quadrature signals (cos and sin of each frequency f_1 and f_2), sampled, and passed to a square-law detector. The detector selects the signal that, at the sampling time, gives the largest correlation. The entire architecture is seen in fig. 13. (Note that in our case, $f_c = f_{IF} =$ 2.5MHz, and $f_d =$ 250kHz)

Continuing with our example, since our receiver is operating at F_s=40MHz while our T_b=1/1Mbps, we sample 40 times per bit-interval. The correlation demodulator multiplies $r(n/F_s)$ by four "templates", which are sampled versions of:

$$\begin{aligned} u_1(t) &= \cos(2\pi f_1 t), \quad 0 \le t \le T_b \\ v_1(t) &= sins(2\pi f_1 t), \quad 0 \le t \le T_b \\ u_2(t) &= \cos(2\pi f_2 t), \quad 0 \le t \le T_b \\ v_2(t) &= \sin(2\pi f_2 t), \quad 0 \le t \le T_b \end{aligned}$$

and so the correlator outputs are:

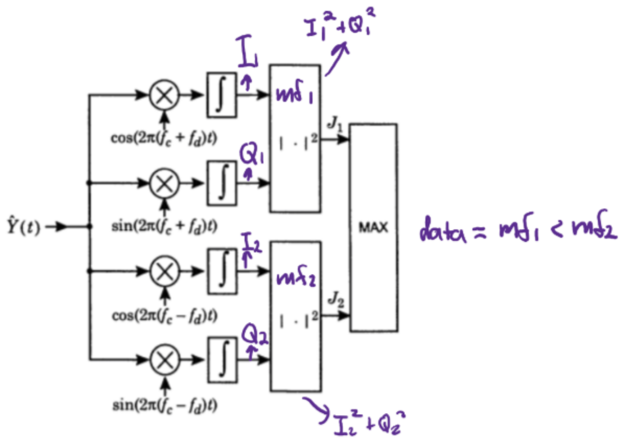


Figure 13: Incoherent Matched-Filter Demodulator for Binary FSK

$$r_{1cos}(k) = \sum_{n=0}^{k} r\left(\frac{n}{F_s}\right) u_1\left(\frac{n}{F_s}\right), \quad k = 1, 2, ..., 40$$
$$r_{1sin}(k) = \sum_{n=0}^{k} r\left(\frac{n}{F_s}\right) v_1\left(\frac{n}{F_s}\right), \quad k = 1, 2, ..., 40$$
$$r_{2cos}(k) = \sum_{n=0}^{k} r\left(\frac{n}{F_s}\right) u_2\left(\frac{n}{F_s}\right), \quad k = 1, 2, ..., 40$$
$$r_{2sin}(k) = \sum_{n=0}^{k} r\left(\frac{n}{F_s}\right) v_2\left(\frac{n}{F_s}\right), \quad k = 1, 2, ..., 40$$

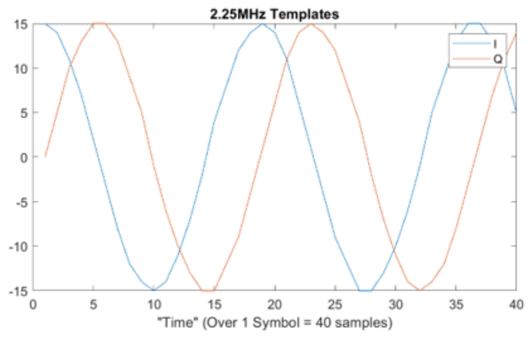
The detector then compares the two decision variables:

$$r_1 = {r_{1cos}}^2(40) + {r_{1sin}}^2(40)$$
$$r_2 = {r_{2cos}}^2(40) + {r_{2sin}}^2(40)$$

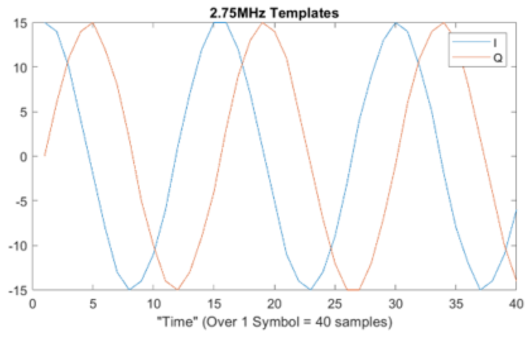
and selects the larger decision variable as the demodulated bit.

4.2 Implementation

Lines 223-262 of section 9 shows the MATLAB implementation of this algorithm. A correlation, in terms of DSP, is simply the convolution of the input signal with a time-reversed sampled signal. The templates are one bit-period of the out-of-phase versions of the two modulation frequencies, as described earlier. This is portrayed in fig. 14. Since the signals are sinusoids, time-reversal was omitted. As for the Chisel implementation, since we're again using convolution, the same *FIRfilter.scala* is used, this time with the four templates exported in lines 281-286 of section 9 as the filter coefficients. The actual implementation can be found in *MF.scala* in the *cdr* GitHub repo. The *MF* module takes in a 5-bit I-channel input, instantiates 4 *FIRfilter*s, and performs the square-law detection on the outputs of the filters. The output of the module is a 1-bit UInt at 40MHz, indicating whether the the received signal was a 1 or a 0. One difference to note between the math example given above and my actual implementation is that, in my implementation, I have opted to let the square-law detector run continuously (as opposed to once every 40 clock cycles, or bit period). This also means that the sums seen earlier get replaced by running-sums. The reason is that this simplifies the clock recovery module, and allows me to attach it as a separate entity after the demodulator. If I were to sample the square-law detector output, the clock-synchronizer would have to be in the loop of the demodulator itself (explained in the next section).



(a) 2.25MHz I/Q Templates



(b) 2.75MHz I/Q Templates

Figure 14: Templates for the Incoherent Correlators

5 Clock Synchronization

The data coming out of the demodulator module is a bitstream at 40MHz, but the actual symbol rate is 1Mbps. This means that the clock synchronizer has make one decision every 40 samples it takes in. The timing of this decision is important. Since the matched-filter demodulator performs correlation of the input signal with two sets of sampled waveforms, one can imagine that the actual output will be inaccurate at the transition (when the input frequency shifts between 2.25MHz and 2.75MHz), most accurate at the middle of the symbol. Therefore, if the sample was taken at the transition, the error rate would skyrocket. Furthermore, if the sampling time is fixed (i.e. not adjustable), then due to Murphy's Law you would end up with an unusable receiver.

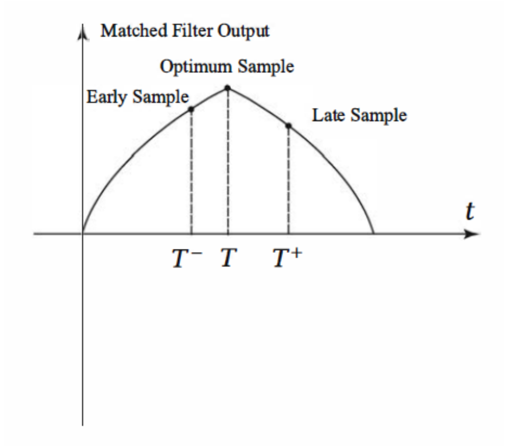


Figure 15: Early-Late Gate

To overcome this, a popular design is to use an *early-late gate*, where three samples are taken within one bit-period, and the sampling time is adjusted to ensure that the middle sample has the largest magnitude. In this design, I have taken a slightly different approach, adapted from the paper *A Low-Power All-Digital GFSK Demodulator with Robust Clock Data Recovery* (P. Chen). Instead of sampling at three different timestamps of the matched-filter output, I decided to use a shift register counter based approach.

5.1 The Algorithm

The exact flowchart of the algorithm is shown in fig. 16. It is somewhat convoluted, so I will explain it with several examples.

At the input of the clock recovery *CR.scala*, we have the 1-bit output operating at 40MHz *data_noclk*. Since we have a 40 to 1 ratio between f_{clk} and f_{bit}, we use a 40-bit shift register *noclk_shiftreg*. We use a pointer to point to the first bit of this register *noclk_shiftreg[0]*. Every clock cycle, th current value of *data_noclk* is shifted into *noclk_shiftreg[0]*. We also use

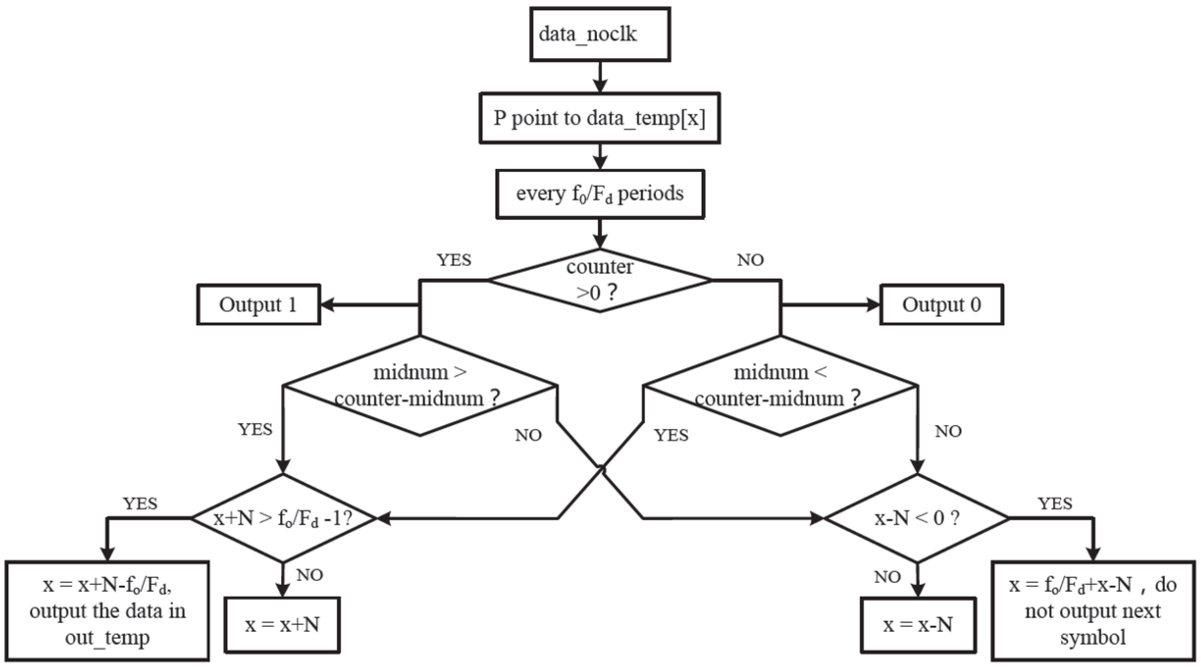


Figure 16: Clock Recovery Flowchart

two accumulators, *data_sum* and *mid_sum*. At every clock cycle, the accumulators read the data pointed to by the pointer, and adds or subtracts 1 from its current sum based on if the input was a 1 or a 0. *mid_sum*, as the name implies, only sums for 20 clock cycles, whereas *data_sum* will sum the entire 40 clock cycles. At the end of 40 clock cycles, a decision is made based on *data_sum*. This means the decision is based on a majority function of the last 40 bits sampled. However, in an ideal case the sum at the end should be either 40 or -40. In the case that it isn't, the sampling window would have to be adjusted, depending on which side is closer to the ideal case.

Figure 17 gives an example of such a case. Here we shorten the *noclk_shiftreg* to 6-bits long for ease of portrayal. In this case, after 6 clock cycles, we have *data_sum*=2 and *mid_sum*=-1. Since data_sum>0, we decide that *data_out*=1. However we have sampled 2 bits of the previous symbol in our current symbol. Since *mid_sum<data_sum-mid_sum*, there are more 0 than 1 in the first half, while the current databit is 1. We have therefore sampled "ahead of time" and will move the pointer back by N (a parameterizable constant).

There are four such cases to consider in total:

- *data_sum*>0

  mid_sum<data_sum-mid_sum

  mid_sum>data_sum-mid_sum

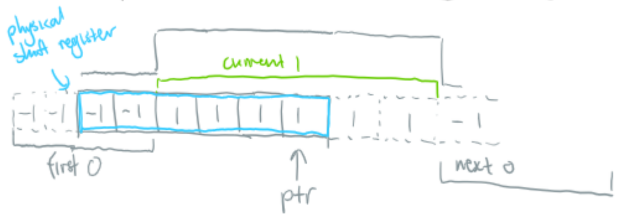
- *data_sum*<0

  mid_sum<data_sum-mid_sum

  mid_sum>data_sum-mid_sum

In addition, there are cases when moving the pointer would cause the pointer to wrap around. In these cases, depending on the direction of the wrapping, we either output no bits for the next 40-clock-cycle round, or two bits. The exact algorithm for each of these cases are shown in fig. 16. Due to the bit-counting nature of this algorithm, it was not simulated in MATLAB. The Chisel code can be found in *CR.scala*, and also observed in the signal of *data_sum[6:0]* in fig. 22. The sawtooth-like waveform shows that the window has been optimized, since within each 40-clock-cycle round, *data_sum* only accumulates in one direction.

Example: Data sequence 010, currently sampling the 1



Continuing with this example, after this clock cycle,

data_sum = 2

mid_sum = -1

mid_sum < data_sum - mid_sum

- there are more 0 than 1 in the first half, while the overall data is 1
- sampling interval is "ahead of time"
  - → move ptr back by N (customizeable)

Figure 17: Clock Recovery Example 1

6 Results

In this section I will go through the MATLAB simulation of the entire filter+CDR process, as well as show what the actual output of the Chisel simulation looks like. We assume an LO frequency $f_{LO} = 2.403\text{GHz}$. Lines 56-85 of section 9 simulates the generation of a random FSK modulated RF signal at $f_{sig} = 2.406\text{GHz}$. We add a side-channel tone at 2.408GHz, as well as an image tone at 2.401GHz. This is done in lines 87-97. Then we use MATLAB's *awgn()* function to add noise to this signal such that the SNR is 4dB (lines 102-111). The resulting time-domain plot is shown in fig. 18, and the frequency plot is shown in fig. 19. We can zoom in on the positive side of the frequency plot to find the signal spectrum, as well as the side-channel and image tone (fig. 19b). Next, we down-convert and filter out the high frequency content with a Butterworth filter, with a cutoff frequency of 40MHz, and downsample to 40MHz (lines 112-130). We quantize the signal to 5-bits (lines 144-154), and he signal at this point represents what we'd expect coming out of the ADCs. The frequency domain plot of this signal is fig. 20a. Clearly, the image tone has now folded itself onto the signal spectrum, and the side-channel still exists.

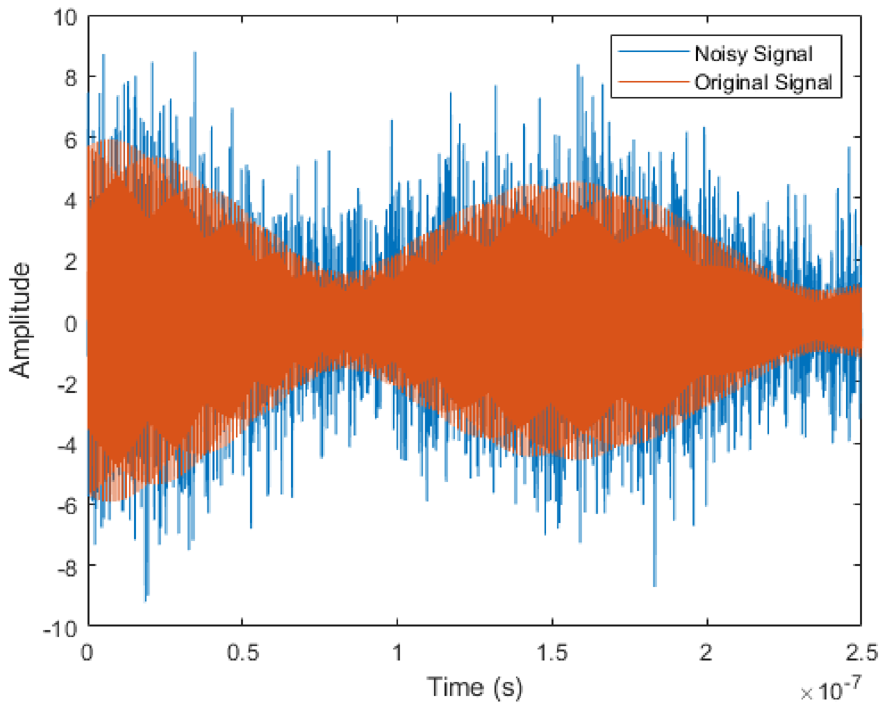
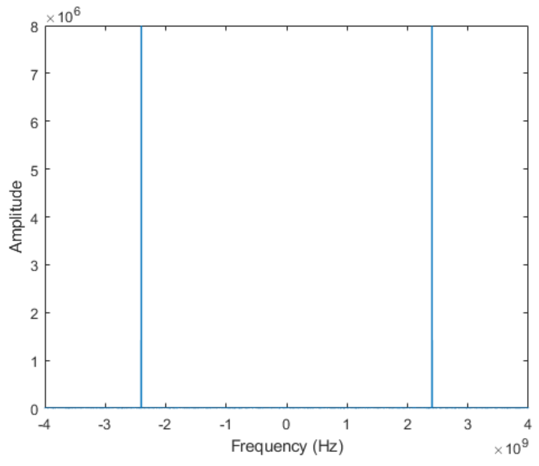
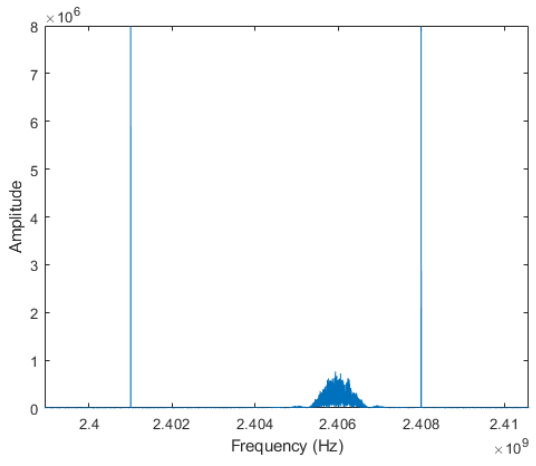


Figure 18: 0.25µs of RF Signal with Image and Sideband Added(Noisy and Noiseless)

The complex BPF is created in lines 22-42, and quantized to 8-bits in lines 44-49. We filter this signal and obtain fig. 20b (lines 173-197). Both the side-channel and the image tone have been greatly attenuated. Now, passing this filtered signal through demodulation (lines 223-262), we obtain fig. 21. We can clearly observe that at the beginning,

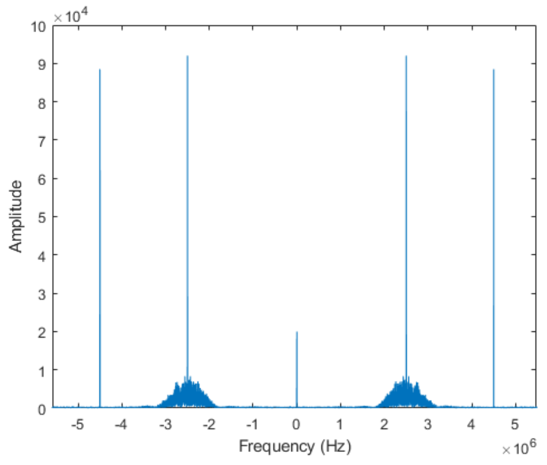


(a) Frequency-Domain Plot of RF

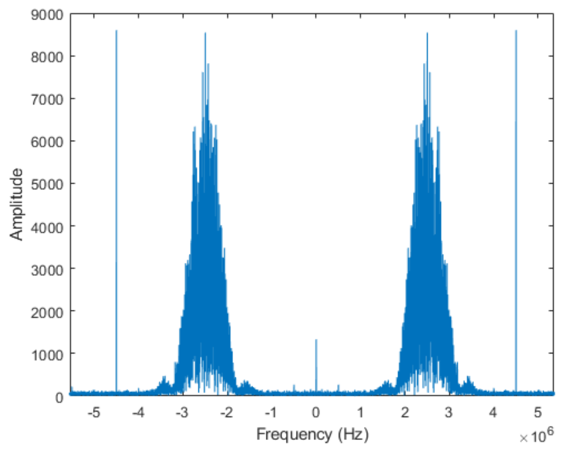


(b) Frequency-Domain Plot of RF, Zoomed at +2.4GHz

Figure 19: RF Frequency Domain Plots



(a) Frequency-Domain Plot of IF



(b) Frequency-Domain Plot of IF, Post-Filters

Figure 20: IF Frequency Domain Plots

a "minipreample" exists, and that at all times (except for the transition regions), one matched-filter output is much greater than the other. The same result can be observed in fig. 22.

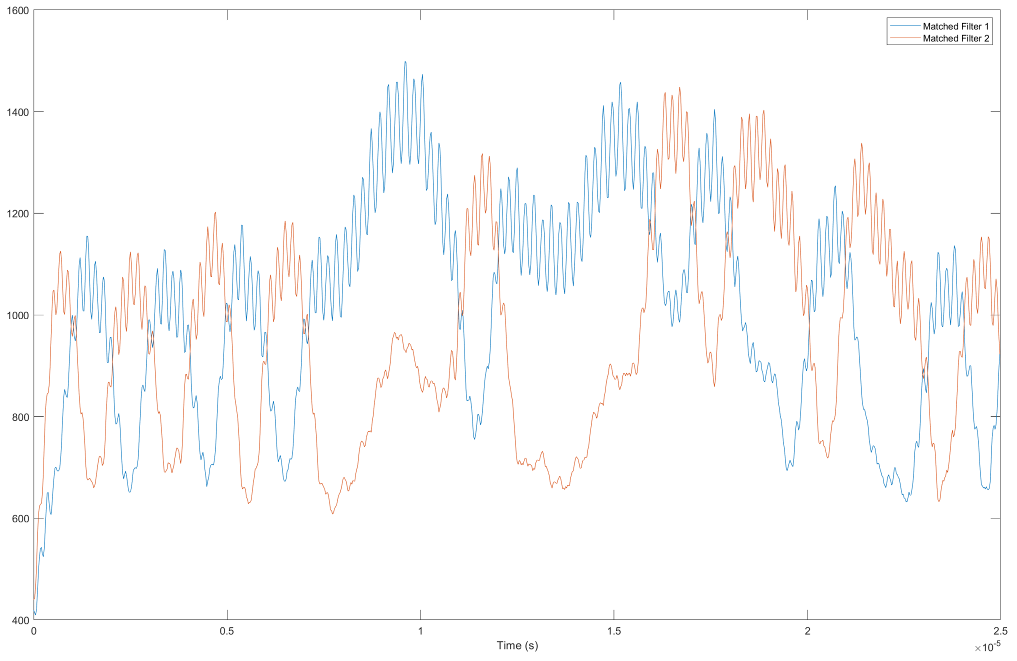
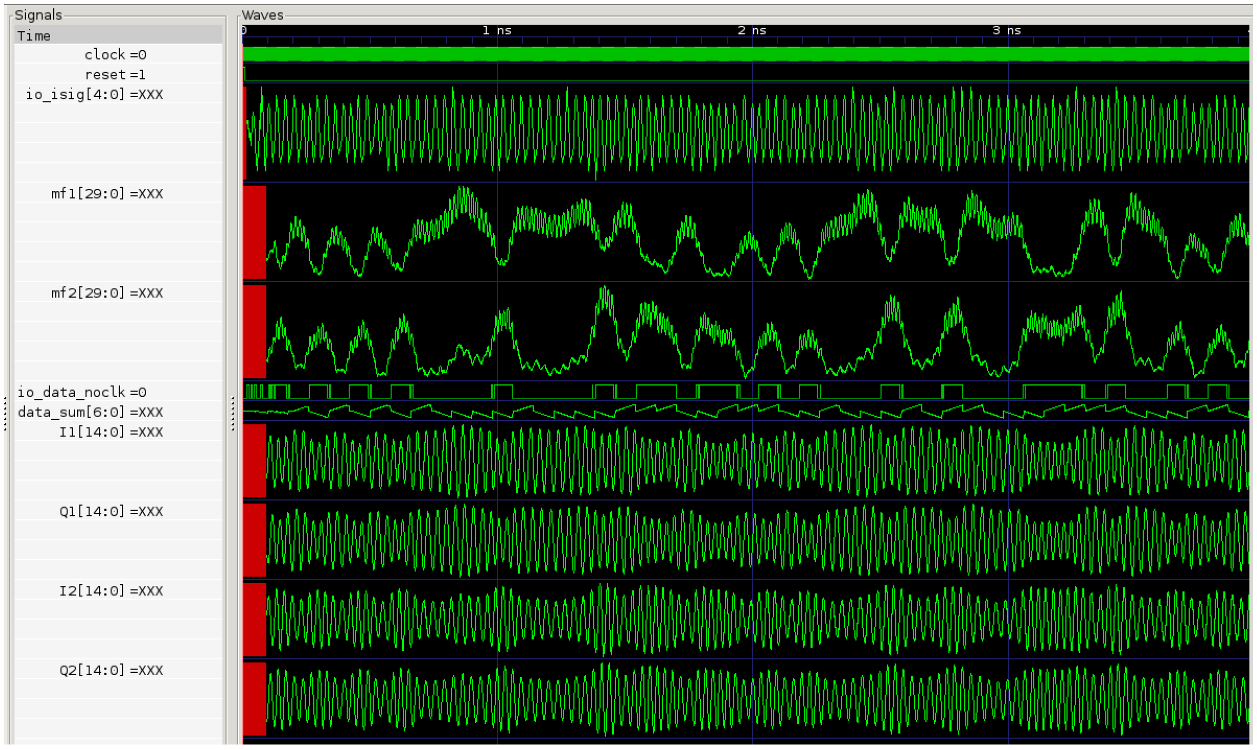


Figure 21: Matched-Filter Output of Post-Filtered Signal



```
clock =0
reset =1
io_isig[4:0] =XXX
mf1[29:0] =XXX
mf2[29:0] =XXX
io_data_noclk =0
data_sum[6:0] =XXX
I1[14:0] =XXX
Q1[14:0] =XXX
I2[14:0] =XXX
Q2[14:0] =XXX
```

Figure 22: Matched-Filter Output of Post-Filtered Signal, Chisel Simulation

7 Future Work

At the end of the semester, I have a functioning filter and CDR module. However, the absence of any performance specs is glaringly obvious. Since I began the semester with very little RF and DSP knowledge, it took me a long time to understand and implement, first in MATLAB, then in Chisel, of what was required for the system to work. For future students undertaking these tasks, I recommend starting from what I have, and exploring more in-depth into the image rejection ratio (IRR) and acceptable signal to noise ratio (SNR) of the filters. I know for a fact that the 40MHz clock over-constrained many aspects of the design. It is very likely that you could lower that by a half and still be within specs. Likewise, it is very likely that the bit-width and tap-length of the filters (at least the complex BPF) could be reduced with little impact on performance. Up until the point where this report was written, I still had no idea how much area my modules actually take on the chip, so I had no target to optimize to. At the end of the day, the limitation is likely on the analog receiver side, so communicate with the analog team and make sure you're on the same page. If you can get a better estimate of the noise characteristics coming from the analog receiver, you can adjust the simulation to match it.

8 GitHub Links

Channel Select/Image Reject Filter:
https://github.com/tapeout/channel-and-image-reject
Clock and Data Recovery:
https://github.com/tapeout/cdr/

9 MATLAB Code

```
close all
clear all
clc

%% Some parameters

Fs_dig = 40e6;        % This is the clock frequency of the digital
    module
dT_dig = 1/Fs_dig;
downrate = 200;       % This is used for the MATLAB filters
Fs_matlab = downrate * Fs_dig;
dT_matlab = 1/Fs_matlab;
RF_ampl = 2;
LOI_ampl = 1;         % LO ampl for In-phase arm
LOQ_ampl = 1;         % LO ampl for Quadrature arm
RF_freq = 2.406e9;    % RF frequency. Refer to BLE specs
IF = 2.5e6;           % Intermediate frequency

SNR_dB = 4;          % Pretty obvious what this is

LO_freq = RF_freq - IF; % Low-side injection: LO < RF

%% Complex BPF with Custom Specs
samplingFrequency = 40e6;       % 40MHz
centerFrequency = 1e6   ;       % 1MHz
transitionWidth = 750e3;        % 750kHz
passbandRipple = 1;             % 1dB Ripple
stopbandAttenuation = 17;       % 17dB stopband attenuation

designSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',...
    centerFrequency-transitionWidth/2, ...
    centerFrequency+transitionWidth/2, ...
    passbandRipple,stopbandAttenuation, ...
    samplingFrequency);
```

```
LPF = design(designSpec,'equiripple','SystemObject',true);

Hbp = LPF.Numerator;
N = length(Hbp)-1;
Fc = 0.125;
j = complex(0,1);
Hbp = Hbp.*exp(j*Fc*pi*(0:N));
Hbpreal = real(Hbp);
Hbpimag = imag(Hbp);

%% Quantize the filter to 8-bits
scalingfactorreal = min(abs(-128/min(real(Hbp))), abs(127/max(
    real(Hbp))));
scalingfactorimag = min(abs(-128/min(imag(Hbp))), abs(127/max(
    imag(Hbp))));
maxscalingfactor = min(scalingfactorreal, scalingfactorimag);
Hbpreal = round(real(Hbp) * maxscalingfactor);
Hbpimag = round(imag(Hbp) * maxscalingfactor);

%% Export filter coeffs
% These are the image rejection/channel selection filter
    coefficients
csvwrite("rcoeffs.csv", Hbpreal)
csvwrite("icoeffs.csv", Hbpimag)

%% Generate Data and modulate as RF MSK
num_bits = 220;
T=1e-6;              % This is the symbol rate
freq_shift = 250e3; % This is the frequency deviation or tone
    spacing for fsk
% Generate some random data
data = randi([0 1],num_bits-6,1);
data = [1 0 1 0 1 0 data']';
data_exp = data;

%Convert data to +/- 1
data = data*2-1;

% Generate first cycle of RF
t = 0:1/Fs_matlab:T-1/Fs_matlab;
phase = 2*pi*(RF_freq+data(1)*freq_shift)*t;
phase_last = phase(end);

% Generate remaining modulated RF cycles
t = 1/Fs_matlab:1/Fs_matlab:T;
```

```matlab
for x=2:1:length(data)
    % The phase of each cycle starts where the last one ended
    phase = [phase 2*pi*(RF_freq+data(x)*freq_shift)*t +
       phase_last];
    phase_last = phase(end);
end

% Scale amplitude to get desired power
FSK = RF_ampl * sin(phase);
t = 0:dT_matlab:(length(FSK)-1)*dT_matlab;

FSK_pure = FSK;

%% Generate side channel tone
sidechannel_freq = 2e6;
side = RF_ampl * sin(2*pi*(RF_freq + sidechannel_freq)*t);
%% Add side channel to main signal
FSK = FSK + side;

%% Generate image channel tone
RF_freq_image = LO_freq-IF;
image = RF_ampl * sin(2*pi*(RF_freq_image)*t);
%% Add image channel to main signal
FSK = FSK + image;

%% Mix and match zone
% FSK_pure, side, image can be added at will to form waveform-of-
   interest

%% Add RF Noise
FSK_noiseless = FSK;
FSK = awgn(FSK, SNR_dB, 'measured');

% To verify snr
FSKsnr = snr(FSK_noiseless, FSK-FSK_noiseless)
plot(t(1:2000), [FSK(1:2000)' FSK_noiseless(1:2000)'])
legend("Noisy Signal", "Original Signal")
xlabel("Time (s)")
ylabel("Amplitude")
%% Create LO

t = 0:dT_matlab:(length(FSK)-1)*dT_matlab;
LOI = LOI_ampl * sin(2*pi*LO_freq*t) + LOI_ampl;            % In
   phase: carrier frequency
```

```
LOQ = LOQ_ampl * sin(2*pi*LO_freq*t -90*pi/180) + LOQ_ampl; %
    Quadrature: carrier frequency - 90degrees

%% Downconvert with LO
IFI = LOI .* FSK;
IFQ = LOQ .* FSK;

% Remove high frequency mixing content
[b,a] = butter(5, Fs_dig/(Fs_matlab/2));
IFI_filt = filter(b,a,IFI);     % These should be in analog,
    dealt with by the mixers
IFQ_filt = filter(b,a,IFQ);

% Downsample to 40 MHz (Fs_dig) before further processing
Iout = IFI_filt(1:downrate:end);
Qout = IFQ_filt(1:downrate:end);
t = t(1:downrate:end);

%% Add IF Noise
% I_noiseless = Iout;
% Q_noiseless = Qout;
% Iout = awgn(Iout, SNR_dB, 'measured');
% Qout = awgn(Qout, SNR_dB, 'measured');

% To find power dB of input signal
% pow2db(rms(I_noiseless)^2)
% To verify snr
% Isnr = snr(I_noiseless, Iout-I_noiseless)
% Qsnr = snr(Q_noiseless, Qout-Q_noiseless)

%% Quantize to 5-bit resolution
Iout = Iout/max(abs(Iout));
Qout = Qout/max(abs(Qout));

Iout = floor(Iout*15);
Qout = floor(Qout*15);

Iout = max(Iout,-16);
Iout = min(Iout,15);
Qout = max(Qout,-16);
Qout = min(Qout,15);

%% 2-Sided FFT for checking purposes
X=fft(Iout);
X_shift=fftshift(X);
```

```
N=length(Iout);
w=fftshift((0:N-1)/N*2*pi);
w(1:N/2)=w(1:N/2)-2*pi;
f=Fs_dig/(2*pi)*w;
plot(f, abs(X_shift))
%% CSV Write
% This is the downconverted signal that needs to be filtered, and
% demodulated
csvwrite("I_out.csv",Iout);
csvwrite("Q_out.csv",Qout);
csvwrite("data_exp.csv",data_exp);

%% If you implemented I/Q compensation, it would go here

%% Filter signal with the complex bandpass filters

% Original
% Rcoeff = real(Hbp);
% Icoeff = imag(Hbp);

% Quantized
Rcoeff = Hbpreal;
Icoeff = Hbpimag;

x1 = conv(Iout, Rcoeff, 'same');
x2 = conv(Qout, Icoeff, 'same');

x3 = conv(Iout, Icoeff, 'same');
x4 = conv(Qout, Rcoeff, 'same');

Iout_prefilter = Iout;
Iout = x1 + x2;
Qout_prefilter = Qout;
Qout = x3 - x4;

% Plot the I and Q channels after filtering
plot(Iout);
figure;
plot(Qout);

%% 2-Sided FFT for checking purposes
X=fft(Iout);
X_shift=fftshift(X);
N=length(Iout);
w=fftshift((0:N-1)/N*2*pi);
```

```matlab
w(1:N/2)=w(1:N/2)-2*pi;
f=Fs_dig/(2*pi)*w;
plot(f, abs(X_shift))

%% If we want to export post-filter signals, do so here

%% Downscale back to 5-bit resolution
scalingfactor = 2^11;            % This is set to a hard value so that when there's noise, the remaining signal is less
I = round(Iout/2^11);
Q = round(Qout/2^11);

%% 2-Sided FFT for checking purposes
X=fft(I);
X_shift=fftshift(X);
N=length(I);
w=fftshift((0:N-1)/N*2*pi);
w(1:N/2)=w(1:N/2)-2*pi;
f=Fs_dig/(2*pi)*w;
plot(f, abs(X_shift))
%% Matched Filter Demodulation

% Using these previous variables:
% T is symbol rate
% IF is IF frequency
% dT_dig is digital sampling period
tt = 0:dT_dig:T-dT_dig;

% m = round(T/dT_dig);
tau = 7*ones(1,num_bits);

% -1
template1Q_v = round(15*sin(2*pi*((IF-250e3)*tt)));
template1I_v = round(15*cos(2*pi*((IF-250e3)*tt)));

% +1
template2Q_v = round(15*sin(2*pi*((IF+250e3)*tt)));
template2I_v = round(15*cos(2*pi*((IF+250e3)*tt)));

% Matched Filter 1
I1_v = conv(I,template1I_v);
Q1_v = conv(I,template1Q_v);
I1_v = I1_v(0.5*T/dT_dig:end-0.5*T/dT_dig);
Q1_v = Q1_v(0.5*T/dT_dig:end-0.5*T/dT_dig);
mf1_v = sqrt(I1_v.^2 + Q1_v.^2);
```

```
% Matched Filter 2
I2_v = conv(I,template2I_v);
Q2_v = conv(I,template2Q_v);
I2_v = I2_v(0.5*T/dT_dig:end-0.5*T/dT_dig);
Q2_v = Q2_v(0.5*T/dT_dig:end-0.5*T/dT_dig);
mf2_v = sqrt(I2_v.^2 + Q2_v.^2);
compared_v = mf1_v < mf2_v;

data_v = zeros(1,num_bits);

for k = 2:num_bits-4
    idx = 40*k + tau(k);
    data_v(k) = compared_v(idx);
end

%% Plot, for debug
plot(t(1:1000), [mf1_v(1:1000)' mf2_v(1:1000)'])
legend("Matched Filter 1", "Matched Filter 2")
xlabel("Time (s)")
%% Compare with initial bitstream
data_orig = (data(2:end)+1)/2;
data_orig = [data_orig' 0]';
plot([data_v' data_orig])
legend('Demodulated', 'Original')
figure;plot(data_v'-data_orig);

%% Compare with output generated by chisel
data_chisel = csvread("data_out_demod.csv");
data_chisel = data_chisel(2:end);
data_exp = csvread("data_exp.csv");
data_exp = data_exp(1:end-1);
plot(data_chisel-data_exp)
%% CSV Write to export the templates

csvwrite("template1I.csv",template1I_v);
csvwrite("template1Q.csv",template1Q_v);
csvwrite("template2I.csv",template2I_v);
csvwrite("template2Q.csv",template2Q_v);
```