Channelization using RFNoC (GRCON 2017)

Phil Vallance PJVALLA@GMAIL.COM
Vallance Engineering LLC, 4235 Alta Vista Way, Knoxville, TN 37919 USA

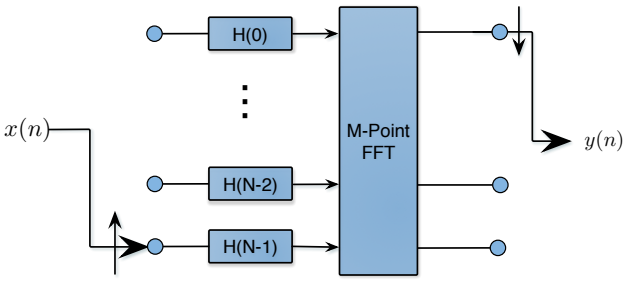
Abstract

This document will review the derivation of the $M/2$ channelizer structure found in (Harris F., 2010). The document will then provide the detailed FPGA implementation of the architecture. The FPGA implementation is specific to the Xilinx architecture since it utilizes the pipelining functionality of the DSP48 cores found in 7 series devices. The implementation is fully pipelined to achieve maximum FMax performance and achieves maximum throughput.

1. Channelizer Background

The Channelizer design is based on (Harris F., 2010). The derivation begins with the classic Nyquist rate channelizer shown in Figure 1. The fundamental relationship given in (1). (Harris F., 2010) uses this structure to derive a channelizer / analysis filter bank that generates channel sample rates at twice the Nyquist rate. The main advantages of this channelizer are that it relaxes the filter transition bandwidth requirements and that it allows perfect reconstruction of the channels when using the synthesis filter bank (dual of the analysis filter bank). Please refer to (Harris F., 2010) for a detailed description of the synthesis filter bank.

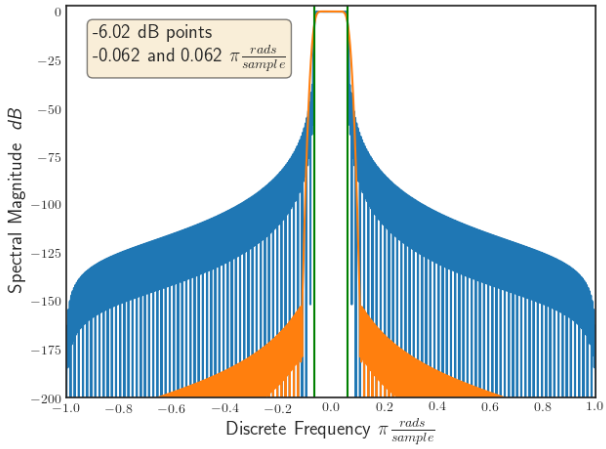
Figure 1. M Down-sample Channelizer



$$y(n,k) = \left[x(n)e^{-j\theta_k n}\right] * h(n) \quad (1)$$
$$= \sum_{r=0}^{N-1} x(n-r)e^{-j\theta_k(n-r)}h(r)$$

Figure 2 displays the impulse response for the M and $M/2$ down-sampled filter prototypes. Both filters exhibit the correct cutoff frequency, f_c, with the required 6.02 dB attenuation required for reconstruction. However the $M/2$ down-sample channelizer can relax the transition bandwidth of the filter yielding a significantly lower passband ripple stop-band attenuation. Both filter prototypes were designed for 16 sub-band channelizer using 24 taps per phase for a total of 384 filter coefficients.

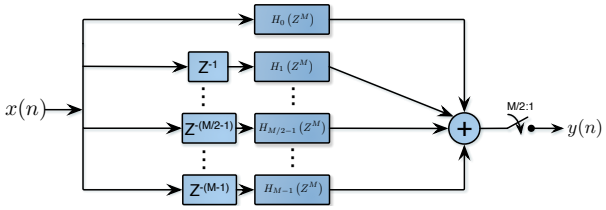
Figure 2. $M/2$ vs M Down-Sample Channelizer



The derivation of the new channelizer form is best shown through the use of a simple $M/2$ downsampling filter shown in 3. The functional polyphase filter relationship is given in (2). This is equivalent to extracting only the DC channel of the channelizer. It should be understood that the IFFT block implements the bank of phase rotation and summers to generate the full set of output channels.

Proceedings of the 7^{th} *GNU Radio Conference*, Copyright 2017 by the author(s).

Figure 3. $M/2$ Downsampling Filter



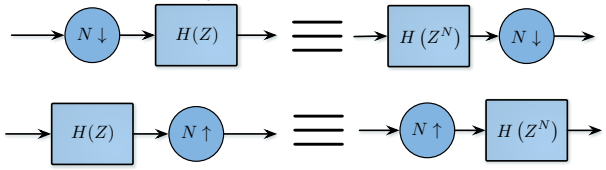
$$H(Z) = \sum_{r=0}^{M/2-1} Z^{-r} H_r\left(Z^M\right) + Z^{-(r+M/2)} H_{r+M/2}(Z^M)$$

where

$$H_r\left(Z^M\right) = \sum_{n=0}^{(N/M)-1} h\left(r+nM\right) Z^{-nM} \qquad (2)$$

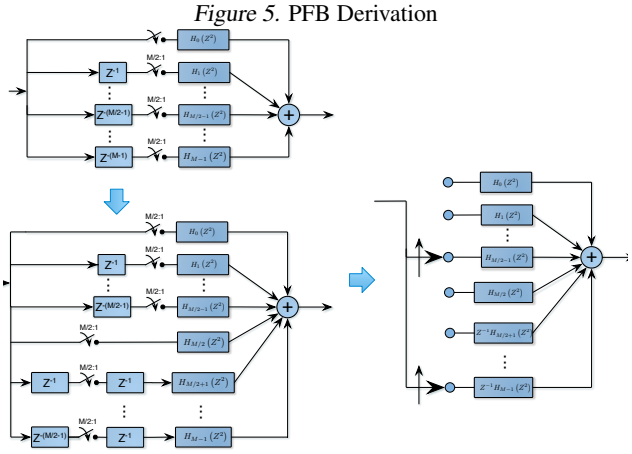
The derivation of the $2f_s/M$ channelizer is performed by successively applying the Noble Identity to Figure 3. The Noble Identities are given in Figure 4. The filter derivation is shown in Figure 5.

Figure 4. Noble Identities



The derivation results in the two PFB structures shown in Figure 6. The translation of these filter structures to the Xilinx DSP48 architecture will be discussed in 2. An $M/2$ channelizer outputs samples from each frequency bin at a rate of $2f_s/M$. To accomplish this, the IFFT is run every $M/2$ input samples. Note that the new PFB filter architecture performs data shifting of $M/2$ samples. This is seen by the Z^{-1} delay present in the bottom-half of the filter arms. Since each filter arm is fed a new sample every $M/2$ samples, this generates a $M/2$ sample shift. This data shifting into the polyphase filter stages causes a frequency dependent phase shift that is summarized in equation set (3).

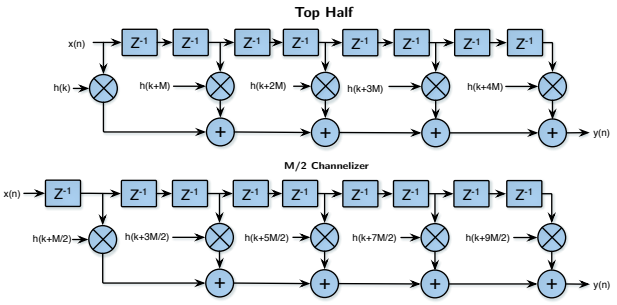
Figure 5. PFB Derivation



$$\begin{aligned}
\theta\left(\omega\right) &= \Delta t \omega \\
\theta\left(\omega_k\right) &= nTk\frac{1}{M}\frac{2\pi}{T} = \frac{nk}{M}2\pi \qquad (3)\\
\theta\left(\omega_k\right)|_{n=M} &= \frac{nk}{M}2\pi|_{n=M} = k2\pi \\
\theta\left(\omega_k\right)|_{n=M/2} &= \frac{nk}{M}2\pi|_{n=M/2} = k\pi
\end{aligned}$$

From this we see that the odd indexed frequency terms experience a phase shift of π radians for each successive $M/2$ shift of input data. The channelizer must compensate for this phase shift by applying the appropriate phase correction to the output of the IFFT. For an $M/2$ down-sampled channelizer, the correction is a circular shift of the filter output data.

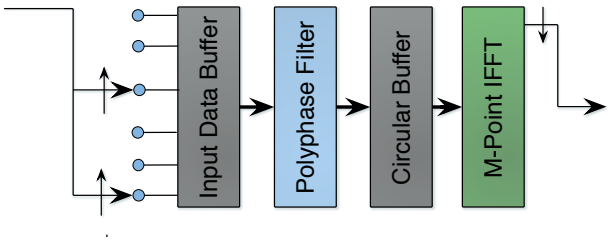
Figure 6. Final Structures



The final channelizer structure is presented in Figure 7. This shows the input buffer, PFB, circular buffer, and the IFFT module connected to create an analysis filter bank.

The final implementation uses the block floating point option of the IFFT block from Xilinx. This option requires the implementation of a final exponent shifting logic to normalize and enforce amplitude consistency between successive IFFT blocks.

Figure 7. Final Structure



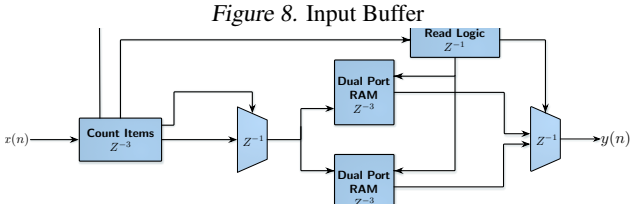
2. Hardware Implementation

This section provides implementation details for each block in Figure 7. The level of detail provided is adequate to directly translate the design to FPGA resources. The implementation of each block shown in Figure 7 will be reviewed and a logical block diagram will be presented. The block diagrams include all important pipelining and logical operations. All modules produce and consume 16 bit I/Q samples. Rounding and truncation is performed by the PFB and IFFT modules. Finally, the number of sub-bands, M, of the channelizer is run-time configurable. This capability requires that all blocks include logic to handle the variable block sizes and that software is responsible for loading new coefficients. The one restriction on the number of sub-bands is that only powers of 2 are valid with an upper bound of 512. The 512 limit is simply due to the prototyping nature of the current development. Depending on the available resources, the upper limit is bound to the Xilinx FFT module (currently 65536 is the upper bound).

2.1. Input Buffer

The input buffer implemented in logic performs three functions. First, it utilizes AXI flow control to ensure data reliability and to conform to the interface provided by the RFNoC infrastructure. Second, it provides a ping-pong buffer interface so that read data is not corrupted by new input data. Finally, the input buffer reads each dual-port RAM twice. This produces the time domain sequence required by the PFB as shown in Figure 5. The current logic implementation is shown in Figure 8.

Figure 8. Input Buffer



The muxing shown allows the input buffer to read one dual port ram while the other is being written. The fixed counters provide the appropriate addressing for both reading and writing RAMs. Note that the input data rate must be $<= \frac{1}{2}$ the clock rate. The current design ensures this data rate by including a half-band filter before the input buffer. The output of the input buffer does not implement the AXI interface. This was done to reduce logic complexity and to treat the entire channelizer as one single block with AXI interfaces on both master and slave ports. To ensure data reliability a FIFO is used internally that gates the slave tready, *s_tready*, signal of the input buffer with the almost full flag. This ensures that the channelizer is able to ingest another block of samples from the input buffer after the *s_tready* is asserted.

2.2. Polyphase Filter Bank

The mathematical filter derivation is shown below in (4). Notice that the second column must be offset in both sample and coefficient addressing to ensure $y(n)$ is computed correctly. This is a consequence of the systolic implementation. The delay registers at the top of Figure 9 perform the offsetting operation. Adding columns to the filter arm simply requires duplicating the column structure. The DSP48s and the memory architectures utilize a large degree of pipelining to grant the Place and Route (PAR) tools enough flexibility to ensure success.

The logic shown in Figure 9 has been mapped to the Xilinx architecture. This represents a fully pipelined design that maximizes the potential FMax. The references next to the delay registers indicate the current sample, x, and coefficient, h. The subscript for x indicates the corresponding phase or arm of the filter bank and n represents the nth input sample loaded into the relevant arm. The first subscript for the h value represents the phase arm value and the second subscript represents the column position within the arm. These coefficient values correspond to those given in Figure 6. It is important to understand that filter arms are loaded sequentially. This is referenced in the diagram by the incremental changes in the phase subscript through each subsequent delay register. The nth index is only updated once per revolution of the filter bank.

Reads are always $2M$ samples behind the writes into the

sample memories. This ensures that there are no collisions between reads and writes of memory. Addressing is also delayed between successive columns of the PFB architecture to align correctly align samples with the pipelining of the follow-on DSP48 logic. Each sample buffer shown must be $4M$ words long.

Also note on the control logic in upper-left of the figure. This logic performs the extra delay operation required by the bottom-half of the PFB. A single PFB implementation is used for both the top and bottom-half filters referenced in Figure 6.

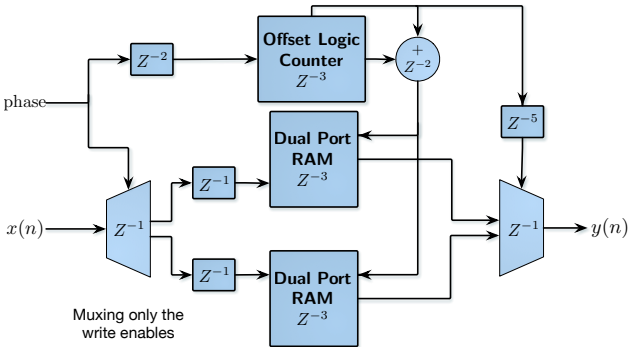
$$\begin{aligned} H(z) &= \left(\left(x_0(n)h_{0,0}z^{-2}\right)z^{-2} + \left(x_{255}(n-3)h_{255,3}z^{-2}\right)z^{-1}\right)z^{-1} \\ &= \left(\left(x_{254}(n-1)h_{254,1}\right)z^{-2} + \left(x_{253}(n-3)h_{253,3}\right)z^{-1}\right)z^{-1} \\ &= \left(x_{252}(n-1)h_{252,1} + x_{252}(n-3)h_{252,3}\right)z^{-1} \\ &= x_{251}(n-1)h_{251,1} + x_{251}(n-3)h_{251,3} \end{aligned} \quad (4)$$

The actual implementation is 24 taps per PFB arm. This yields a filter that has relatively narrow transition bandwidths, good passband ripple, and significant stop-band attenuation.

2.3. Circular Buffer

The implementation of the Circular Buffer is shown below in Figure 10. The design is similar to the input buffer in that the block is able to ping-pong between different memories. Again, this allows full streaming operation without risk of data corruption from simultaneous reads and writes. The offset counter logic offsets the read pointer by $M/2$ samples every other block of M samples. The counter is still modulo M resulting in a circular shift of the data input to the follow on IFFT block.

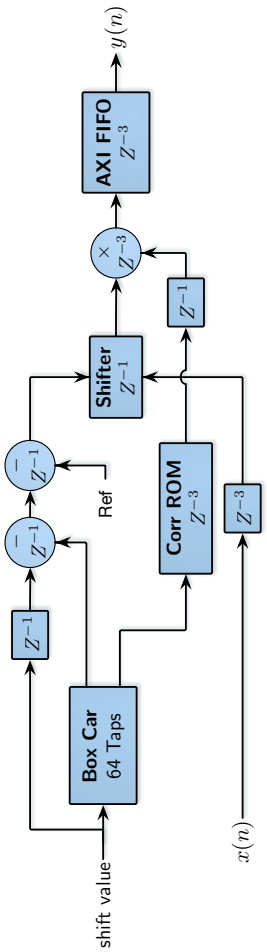
Figure 10. Circular Buffer Implementation



2.4. Exponent Shifter

The output of the IFFT block produces a block of samples and the common exponent of the samples is passed through the status interface. The Exponent Shifter module averages the exponent of 64 consecutive IFFT frames. The entire block of samples is then shifted by the integer difference between the current exponent and the moving average. The sample block is also scaled by the remainder to linearly interpolate the difference. The module also enforces a fixed offset of the exponent to allow for headroom for loud and bursty transmissions. This ensures that signal amplitude differences are captured and not removed by the block-floating point operation of the IFFT module. The flowgraph of the module is shown below in Figure 11. Finally, an AXI FIFO block is used to implement the flow control of the AXI master interface.

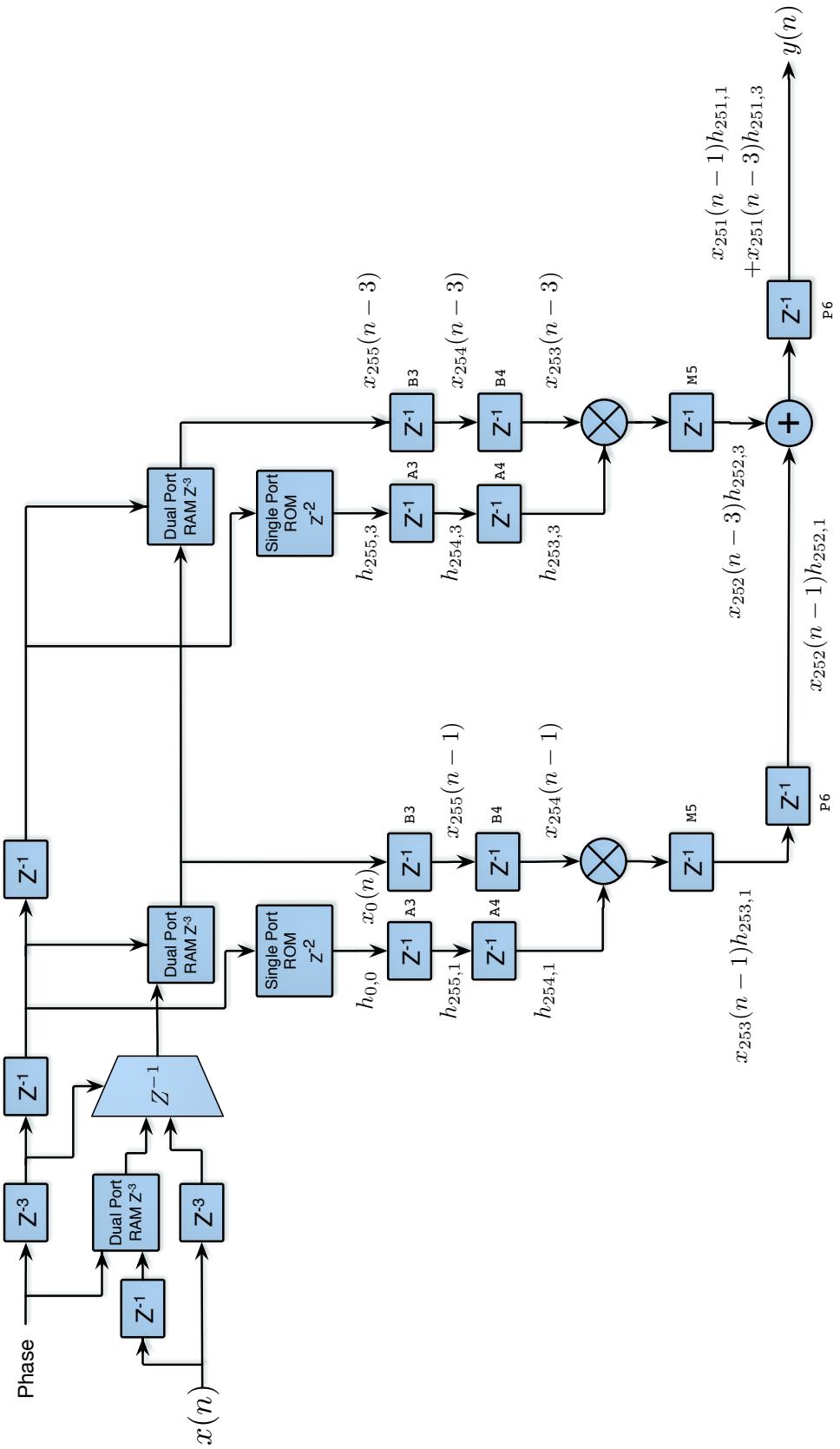
Figure 11. Exponent Shift Implementation



3. Filter Generation

The filter generation is based on the work of Wessel Lubberhuizen found in (Lubberhuizen, 2010). Using root raised erf functions to perform filter coefficient calculation has two distinct advantages over the commonly used Remez algorithm. First, the algorithm is stable even for very narrow filter bandwidths. This makes it an effective tool

Figure 9. PFB Implementation



for calculating filters for channelizers with large numbers of sub-bands. The second advantage is the ease of computation. The outline of the procedure is shown below in Algorithm 1.

Algorithm 1 Channelizer Filter Coefficient Generation

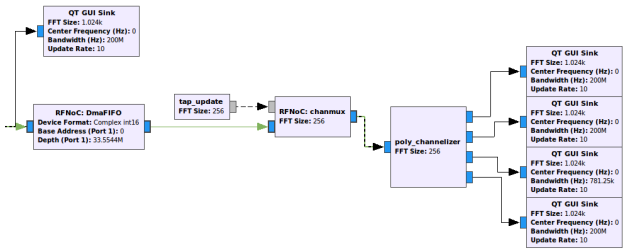
function TAPEQUATION(M, K)
▷ *Create normalized frequency vector*
▷ *Error function is created in the frequency domain and then*
▷ *transformed in the time domain*
$F = range(M * taps\_per\_path)$
$F = F/len(F)$
$x = K * (M * F - .5)$
$A = \sqrt{0.5 * erfc(x)}$
$N = len(A)$
$idx = range(N/2)$
$A[N - idx - 1] = conj(A[1 + idx])$
$A[N//2] = 0$
▷ *this sets the appropriate -6.02 dB cut-off point required for the channelizer*
db_diff $= -6.02 - 10 * log10(.5)$
$exponent = 10^{\frac{-db\_diff}{10.}}$
$A = A^{exponent}$
$b = ifft(A)$
$b = fft\_shift(b)$
▷ *normalize filter magnitude to 0 dB passband*
Return $b/sum(b)$
end function

Notice that there are two arguments to the procedure, M and K. M is the number of channelizer sub-bands. K is a shaping constant of the procedure. Increasing K results in a low-pass filter with narrower transition bandwidth with the trade-off of increased pass-band ripple and stop-band attenuation. The complete tap generation procedure includes a hill-climbing routine to iteratively increment K until the desired transition bandwidth is achieved. The relaxed transition bandwidth used by the $M/2$ down-sampled channelizer results in a filter with extremely low pass-band ripple (¡.01 dB) and very good stop-band attenuation (¡ -150 dB to nearest adjacent sub-band).

4. GNURadio Software Component

There are three components to the GNURadio based controller for the channelizer. The first component is the block controller, *chanmux*. The module *chanmux* is an RFNoC block controller. It utilizes the gr_ettus library to pass both data and parameters between the GNURadio and RFNoC infrastructure. Specifically, it updates the FFT size of the channelizer, accepts a message containing the filter coefficients and then passes them to the FPGA resources, and provides the basic mechanism for moving data from the FPGA to the GNURadio infrastructure.

Figure 12. Test System Diagram



The second component is a pure python module, *tap_update*. This block generates the fixed point PFB coefficients based on the user specified FFT size. The PFB coefficients are computed in floating point, then translated to fixed point representation ensuring that the slicing and rounding settings of the PFB module are consistent across all FFT sizes. Finally, the fixed point coefficients are passed to the block controller, *chanmux*, via a message interface.

The final component is a traditional GNURadio block written in C++. This module, *poly_channelizer*, is responsible for consuming blocks of samples from *chanmux* and then de-interlacing each sub-band and appending the sub-band samples to the corresponding output port. The user specifies a mask of bins that they want returned. This mask determines the number of output ports. The mask is a simple integer vector that where each integer corresponds to the desired sub-bands. The mask does not have to be in ascending sub-band order and repeated sub-band values are allowed.

5. Real Time Results

The GNURadio Companion flow graph is shown below in Figure 12. The input stages are omitted to save space. This setup is using a 256 channel channelizer. There are four sub-bands specified in the mask vector argument to the *poly_channelizer* block. The *tap_update* python block and *chanmux* block controller are functionally required.

Figure 13 shows the PSD plot of the channelizer input. The are three narrow-band pulse shaped QAM16 waveforms present. The four sub-band mask values extract the three channels plus another unoccupied sub-band. The PSD plots of these sub-bands are shown below in Figure 14.

Figure 13. Input Spectrum

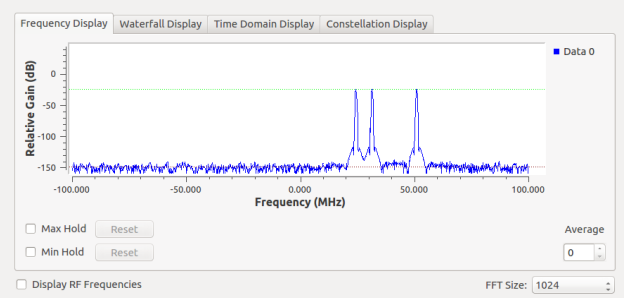
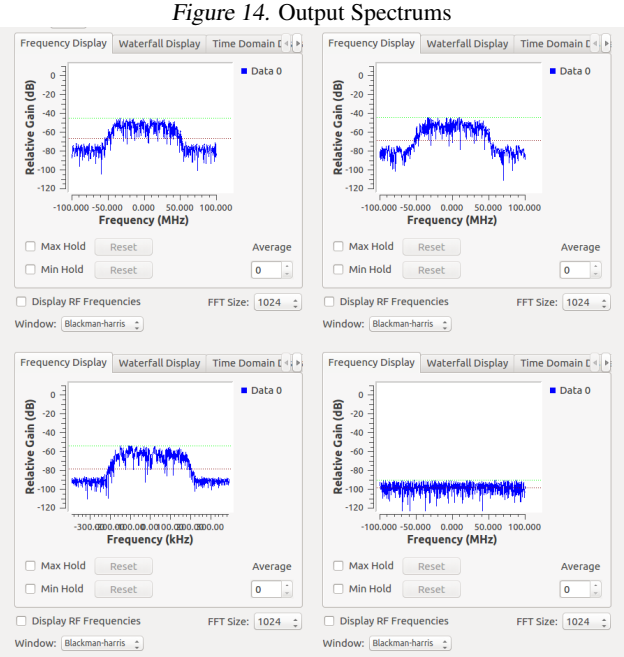


Figure 14. Output Spectrums



6. Future Work

There are two significant tasks that should be worked moving forward. First, the output of the channelizer requires word sizes larger than 16 bit I/Q samples. This is driven primarily by the fact that the Exponent Shifter module is scaling the output of the IFFT module to avoid clipping strong bursty signals. This reduces the effective dynamic range of the system since the upper bits are left unoccupied during normal operation. Narrow-band signals would also benefit from increased dynamic range. A system design may use a large channelizer to simultaneously extract many narrow-band signals. In the instance, the bit growth associated with the filtering operation could be significant. The current system is effectively discarding a large portion of these gains. These gains would be recovered by moving to a channelizer with larger output word size.

The second task is to convert the python code found in the *tap_update* module in to C++ and include it in the *chanmux* block controller. This would simplify the setup of the channelizer and remove ambiguity as to when the coefficients may be updated on startup.

References

Harris F., McGwier R., Egg B. A versatile multichannel filter bank with multiple channel bandwidths. In *CrownCom 5th International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, pp. 1–5, Cannes, France, 2010.

Lubberhuizen, Wessel. Near perfect reconstruction polyphase filterbank. `http://www.mathworks.com/matlabcentral/fileexchange/15813-near-perfect-reconstruction-polyphase-filterbank`, 2010.