Digital Front End (DFE)

Design and verification

By :
Hazem Yasser Mahmoud Mohamed
Mohamed Ahmed Mohamed Elsayed
Mohamed Anwar

Table of Contents

4. Verification & Test Results – Python (Page 33–55)

4.1 Polyphase Verification (Page 33–41) — Impulse/frequency tests; tones; MSE vs SciPy

4.2 Notch Verification (Page 41–47) — 2.4 MHz rejection; time & spectrum

4.3 Full DFE Verification (Page 47–55) — Composite chain tests; R=1/2/4/8/16 spectra; aliasing

---

5. Hardware Implementation – SystemVerilog (Page 58–92)

5.1 Polyphase Resampler (Page 58–76) — polyphase_filter; wrapper L=2, M=3; tb_rational_resampler; waveforms

5.2 Notch Filter (Page 76–84) — DFII-T biquad; testbench; waveform analysis

5.3 CIC Filter (Page 84–92) — Integrator/comb chain; 36-bit growth; tb_cic_filter

---

6. Top-Level Integration & FPGA Implementation (Page 95–111)

6.1 DFE Top-Level (Page 95–100) — dfe_top integration; pipelining; tb_dfe_top

6.2 FPGA Constraints (Page 100–104) — Clock/reset; switches; PMOD I/O

6.3 Implementation Results (Page 111) — Schematic; timing (pre/post-pipelining); power

---

Project Repositories

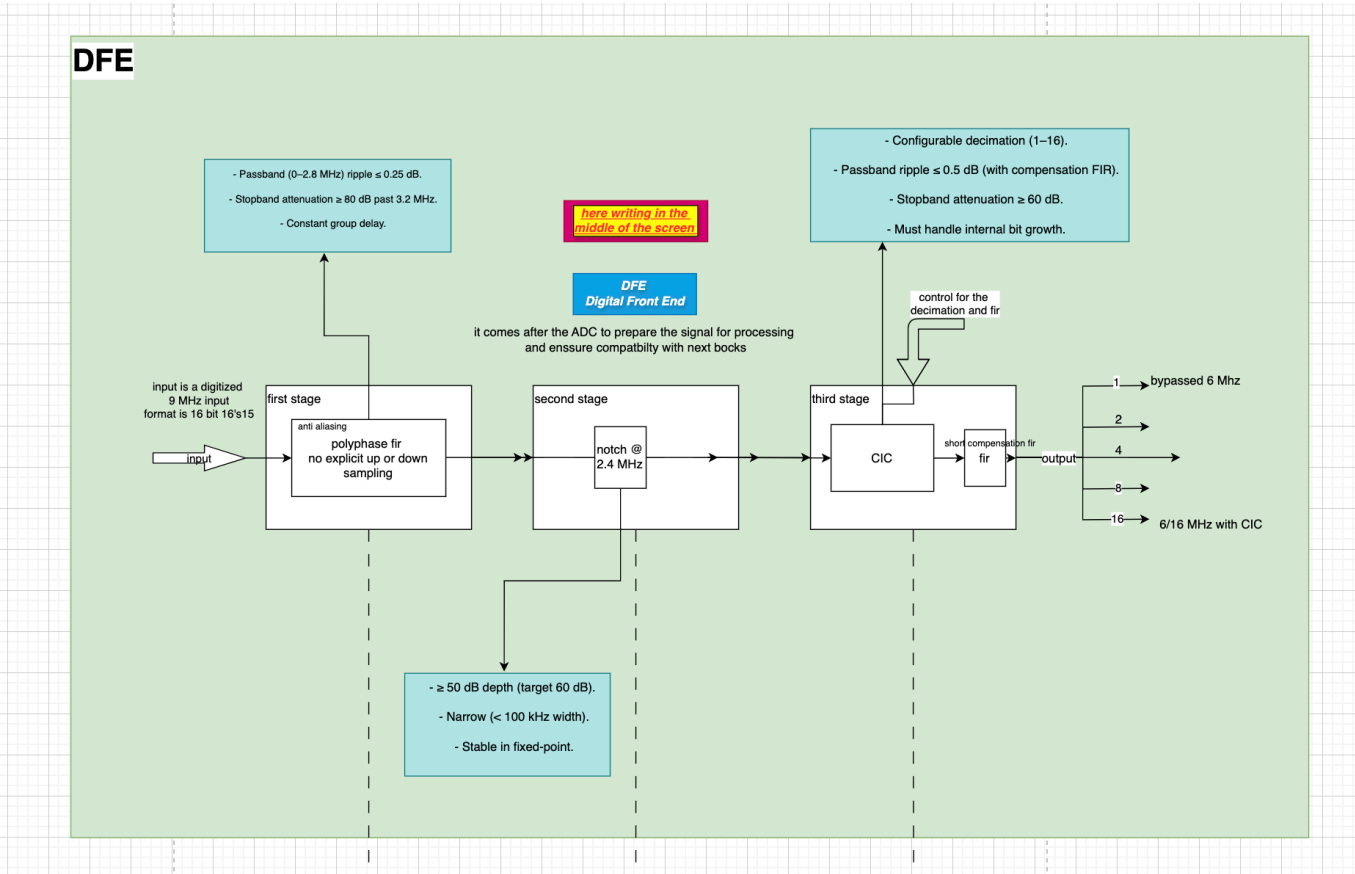
All RTL, Python models, VCD/VVP files, and logs: https://github.com/hazem-yasser

Repos: Configurable_CIC • Notch_Filter • DFE_FPGA • Polyphase_Resampler • DFE_Golden_Model

Golden Model report

By : Hazem Yasser mahmoud

Mohamed ahmed mohamed



the golden model is split into two Jupyter notebooks filterdesign.ipynb & goldenmodel.ipynb

general notes

> **problem_1** : document state an interference at 5 MHz while the sampling frequency is 9 MHz. so max frequency component is 4.5 and if present it will alias to 4 MHz.
> and with further downsampling to 6 MHz it will fold even around 3 MHz.

> **problem_2** : for rational resampling the prototype filter cuttof frequncy $w_c = \frac{1}{\max(L,M)} \cdot \frac{F_s}{2}$, so filter specs are wrong
>
> **resolution** : change specs from 3 to 1.5 so same specs but frequency is divided by two when implementing this prototype at polyphase structure it gives effective filtering at 3.

the golden model is split into two Jupyter notebooks filterdesign.ipynb and goldenmodel.ipynb

steps

1. first use python for design of the FIR IIR and CIC and export coefficients
2. use python to analyze stability of fixed point approximation of the coeff and use s1.23 instead of s1.15 if needed and for IIR coefficients you need to use s2.14 as coefficients are bigger than 1
3. design polyphase filter for rational resampling realizing a filter at a lower sampling freq is easier and less resource intensive .
4. use biquad IIR filter to realize the notch filter and DF II Transposed also use multiple cascaded IIR if needed we will filter only at 2.4 MHz.
5. design and visualize the proper CIC and short compensation filter
6. design a general abstract high level model
7. import coefficients to python and build the functions that do the same functions in low level without abstracting any details
8. test the model in Jupyter notebook and Matlab by resampling a signal at 9 MHz and injecting noise at 6 MHz.

fractional decimator

we will use polyphase fir filter to sample at only the needed ouput phase for efficent computaion

1. Naive way

The straightforward way is:

1. Upsample by 2 → insert zeros → sampling rate goes 9 → 18 MHz.
2. Low-pass filter at 1.5 MHz (to remove images).
3. Downsample by 3 → pick every 3rd sample → final rate is 6 MHz.

- Issue:
  - The filter runs at the high intermediate rate (18 MHz), so a lot of wasted multiply-accumulate (MAC) operations on zeros.
  - High computational cost, especially if the filter has many taps.

2. Polyphase way

With polyphase decomposition, you restructure the filter so you:

- Never actually insert zeros.

- Only compute the output samples that survive after downsampling.
- Exploit symmetry between the up/down steps.
  For L/M resampling (L=2, M=3):
- The polyphase filter has M = 3 branches (because of downsampling factor).
- Each branch only processes the needed input samples for one output stream.
- So instead of filtering at 18 MHz, the effective computation happens closer to the final 6 MHz rate.

notes

in implemneting the up and down sampler we use mux and demux

we are free to discard the zero-valued replacement samples. In reality we do not insert the zero-valued samples since they are immediately discarded.

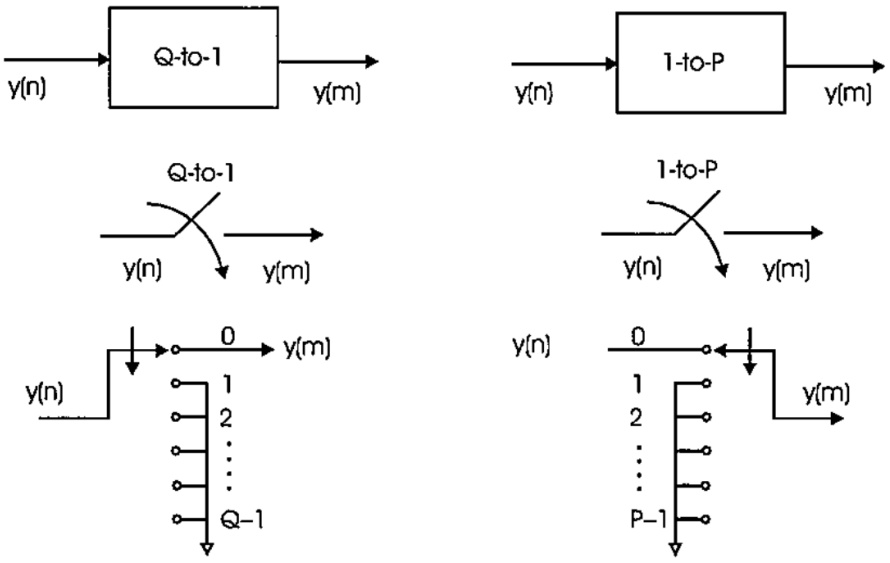
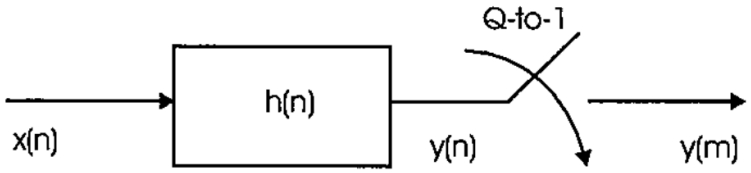


Figure 2.8 Symbols Representing Down Sampling and Up Sampling Elements



- identities used to compute polyphase filter efficently

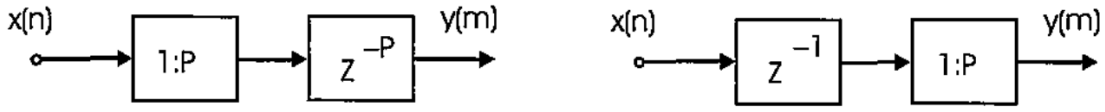


Figure 2.15 Q-units of Delay and Q-to-1 Down Sampler Is Equivalent to Q-to-1 Down Sampler and 1-unit of Delay.

Design

python scipy

we will use Python to design a FIR filter at 9 MHz sampling rate and we will optimise the taps to get the best fir that meets

- > -80 dBs stopband at 1.6 MHz.
- < .25 dBs passband at 1.4 MHz.
- a Gain of L = 2 to normalize after upsampling.

for best optmized LPF FIR with steep freq response we will use kaiser window

- Kaiser order estimate:

$N \approx \lceil \frac{A-8}{2.285\ \Delta\omega} \rceil = \lceil \frac{80-8}{2.285 \times 0.1396} \rceil \approx 226$

```
Number of taps = N+1 = 227
```

Problems

a prototype filter at 3 MHz passpand has effective filtering at 6 MHz when implemented in polyphase resampler

resolution: it was a specs error as all materials state to filter up to
$w_c = \frac{1}{\max(L,M)} \cdot \frac{F_s}{2}$ when resampling

- so filter should be at 1.5 MHz so the prototype filter was designed as such
- at 1.4 ripple less than .25dB and at 1.6 less than -80dB

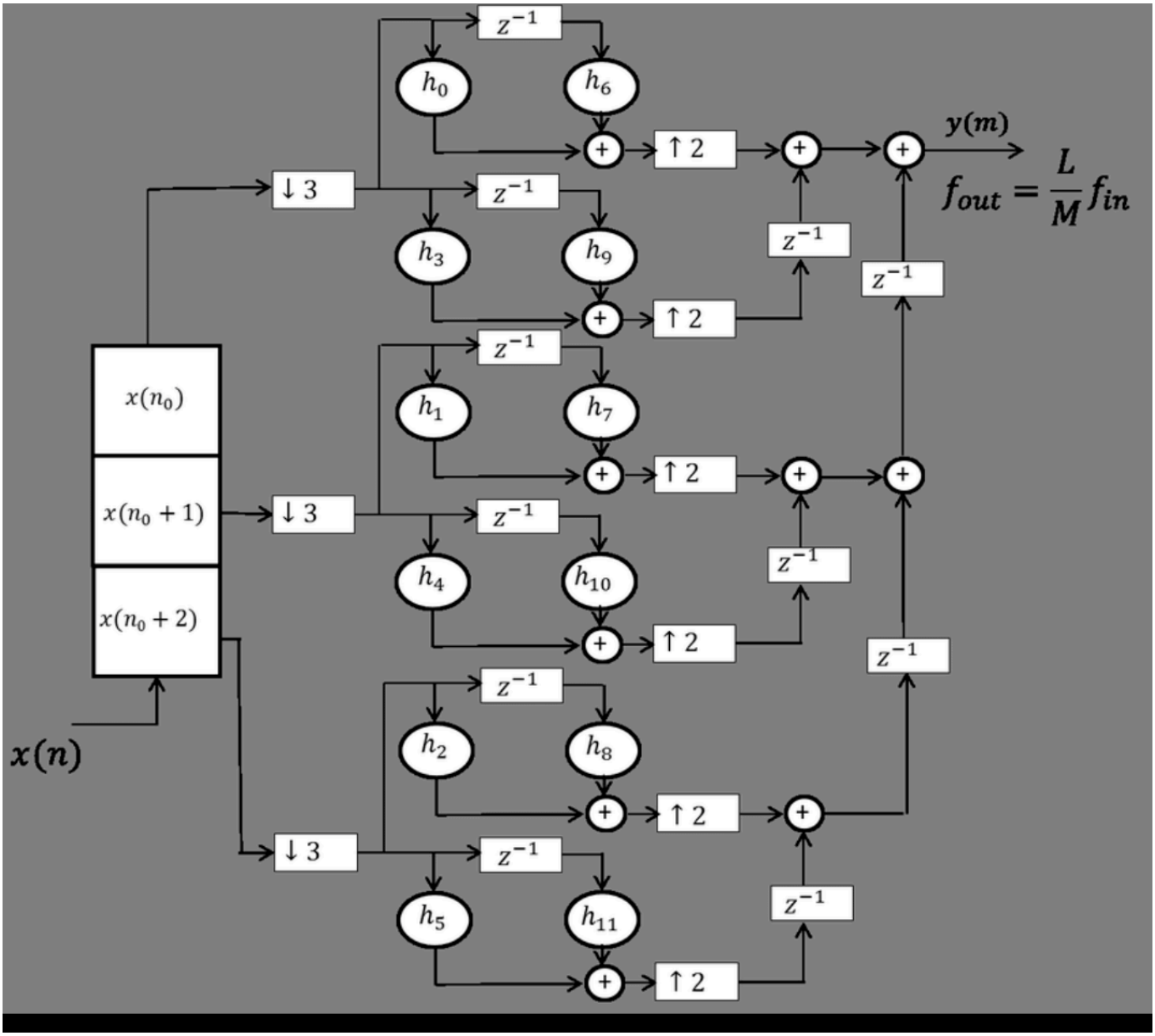
> observation : after implementing the fir in polyphase structure
> it is filtering upto 3 MHz it seems that the structure has a frequency doubling eeffect

implementaions

First approach

then we will implement it in polyphase structure to achieve resampling effects
we will use pipelined delay lines after multiplier or pipelined filter phases to achieve better timing.

this polyphase structure was proposed in a paper
https://www.researchgate.net/profile/Abhishek-Kumar-202/publication/319763113



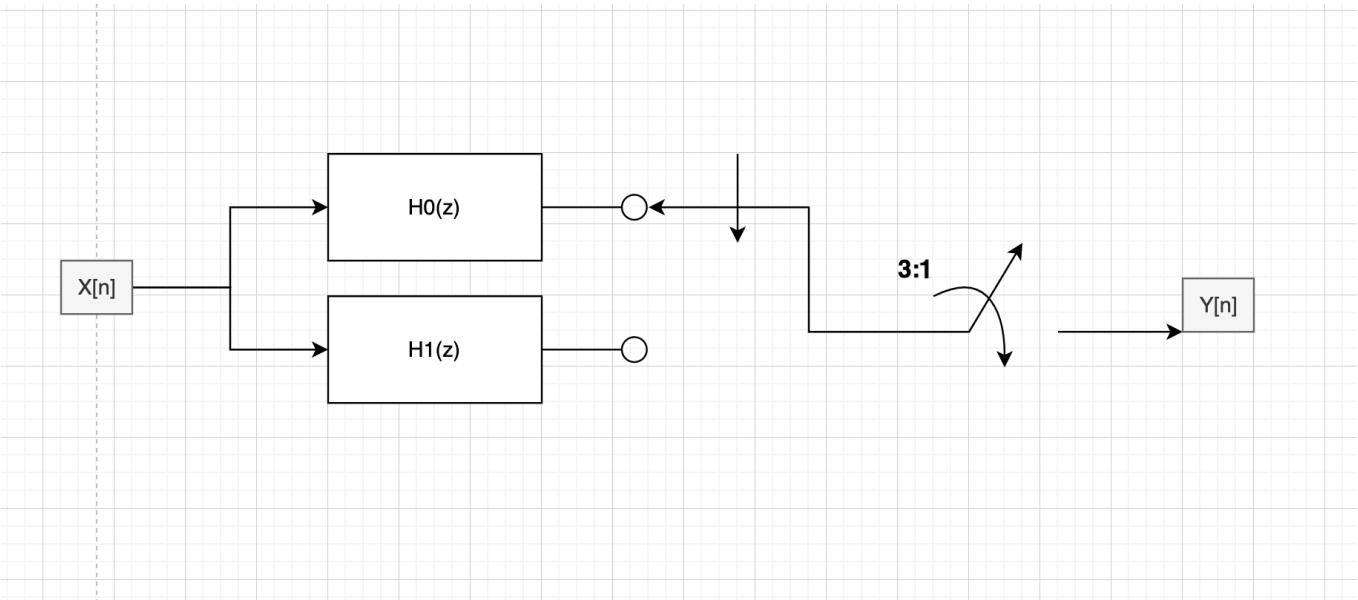
- three main branches for downsampling
- 2 secondary branches for upsampling
- can replace all the delays lines by muxes and demux

> note that each branch has more coeff than in the image each sybsequent two have 6 in difference so h0,h6,h12,h18....
> h1,h7,h13,h19....

while complex it uses more effiecnt computations and thus less delay considerably less
also much better when L > M than the traiditional polyphase structure such as the one used in matlab

second approach

using the traditional structure will be easier to convert to HDL with hdlcoder in matlab
not as optimized as the paper proposed but easier to implement



IIR Notch Filter Design at 6 MHz Sampling

> After resampling, the maximum frequency component is 3 MHz, so we only need to design a notch filter centered at 2.4 MHz to remove unwanted interference.

Design Steps

- We design a second-order IIR notch filter with $\Delta f = 100\,\text{kHz}, F_s = 6\,\text{MHz}, and F_0 = 2.4\,\text{MHz}$.
- The quality factor is chosen as Q = 30 to achieve a deep attenuation of approximately 60 dB, while maintaining stability after fixed-point quantization.
- The pole radius is given by $r = e^{-\pi \frac{\Delta f}{F_s}}$ ensuring a narrow bandwidth around F_0.
- The transfer function of the biquad notch filter is: $H(z) = \frac{1 - 2\cos(\omega_0) z^{-1} + z^{-2}}{1 - 2r\cos(\omega_0) z^{-1} + r^2 z^{-2}}$ where $\omega_0 = 2\pi \frac{2 * F_0}{F_s}$.
- The filter coefficients b_i and a_i are then quantized to Q2.14 format for hardware implementation the quantization is not in Q1.15 because values exceed one.

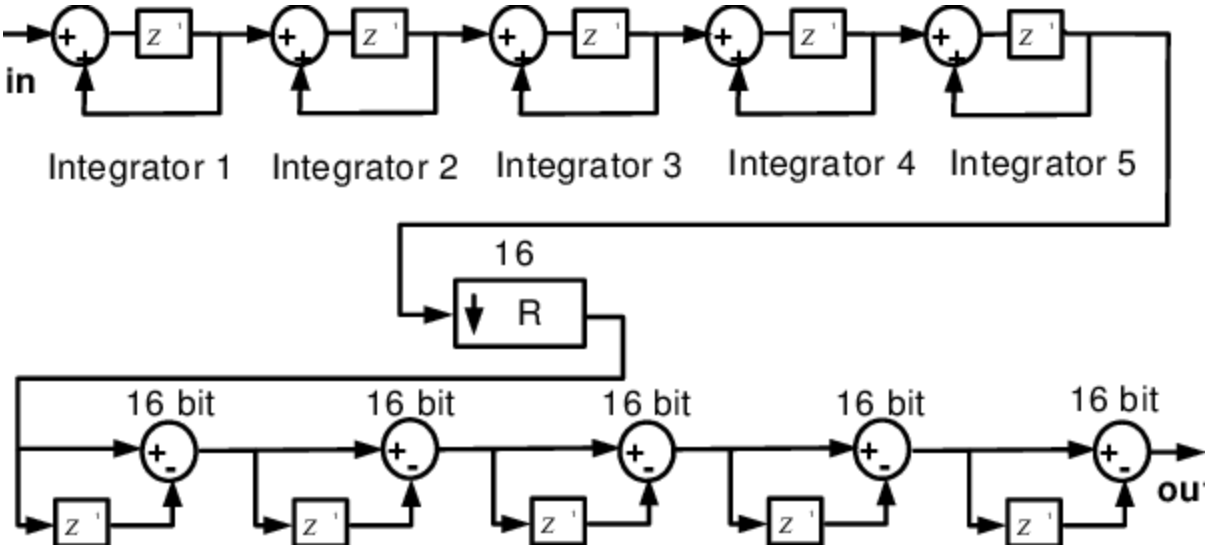
> later in implementation in HDL we need to account for change in format between first and second stage and design multipy and add accordingly

- After quantization, we evaluate:
  - The frequency response, to confirm the −60 dB notch depth near 2.4 MHz.
  - The impulse response, to verify correct transient behavior.
  - The pole-zero diagram, to ensure all poles remain inside the unit circle (stability).
- The filter is implemented using the Direct Form II Transposed (DFII-T) structure, chosen for numerical efficiency and good fixed-point behavior.


/Users/hazemysr/Desktop/projects/DFE/python/biquaad_DF2_T.PNG

.....

CIC Decimator

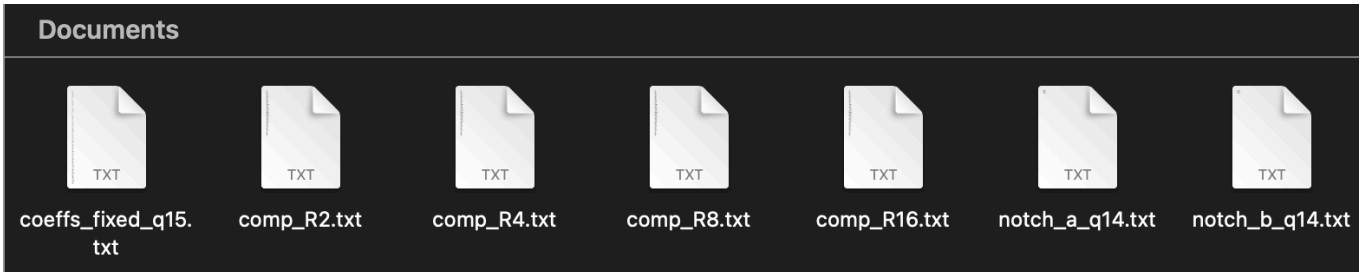


steps

- Design N-stage CIC filter with decimation factor R and differential delay M.
- Use only adders and delays (no multipliers) for efficiency.
- Compensate passband droop with a short FIR filter loaded from comp_R{R}.txt
- Scale output by theoretical gain G = (R*M)^N to normalize amplitude
- Implement in Python first, then test with single-tone and multi-tone signals

1. N and M chosen to be 5 and 1
2. R is configurable from 1,2,4,8,16
3. when R is 1 the signal bypass the CIC altogether
4. the CIC is followed by an fir for the droop

all filter coefficients are quantized and stored in txt files



Filter Design (FIR ,IIR ,CIC)

by :

Hazem Yasser Mahmoud , Group : 7

Mohamed Ahmed Elsayed

mohamed Naem

imports

In [ ]:
```
import numpy as np
from scipy.signal import firwin, remez, freqz, kaiserord, iirnotch, lfilter,
import matplotlib.pyplot as plt
```

Polyphase filter for rational resampling

Promplems and Solutions

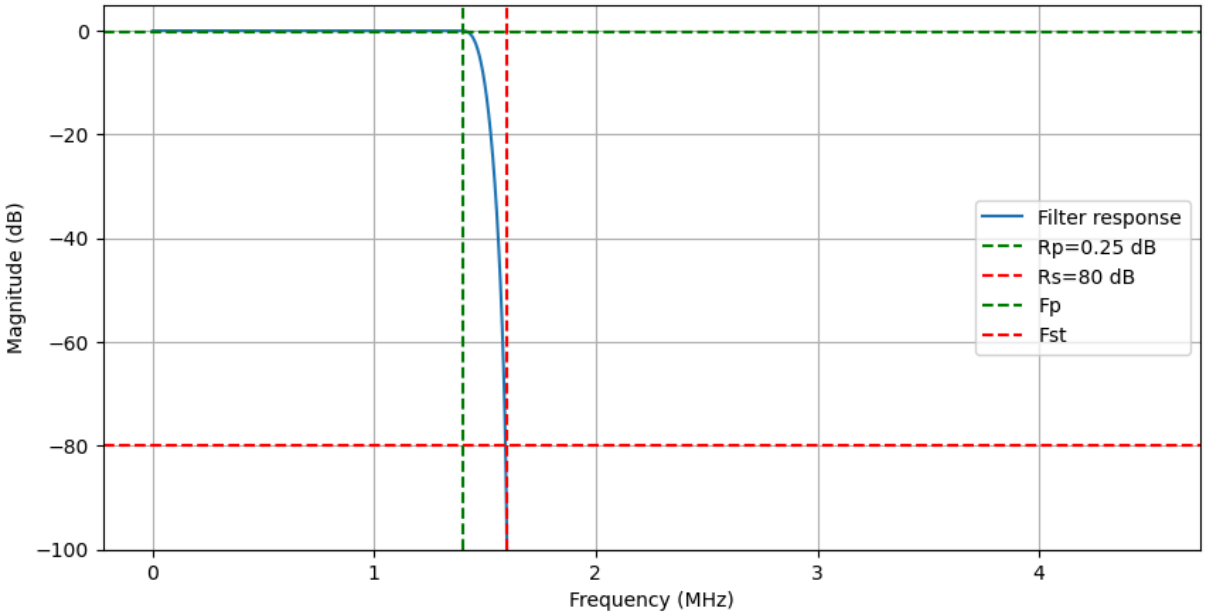
1. For prpoer resampling the fir filter need to be $w_c = \frac{1}{\max(L, M)} \cdot \frac{F_s}{2}$
2. so we Design a Prototype FIR at 1.5 wher 1.4 lesss than .25dB and after 1.6 less than -80dB
3. implementing this prototype FIR in polyphase structure give effective filtering at 3 MHz

DESIGN

In [ ]:
```
# Specs
Fs  = 9e6
Fp  = 1.4e6
Fst = 1.6e6

# Desired ripple/attenuation
Rp = 0.25   # passband ripple in dB
Rs = 80     # stopband attenuation in dB

# Convert to linear ripple/attenuation
delta_p = (10**(Rp/20) - 1) / (10**(Rp/20) + 1)   # passband ripple (linear)
delta_s = 10**(-Rs/20)                            # stopband ripple (linear)
```

```
# Relative weights: higher weight on passband or stopband as needed
# (choose ratio based on importance)
weights = [1/delta_p, 1/delta_s]

# Bands in Hz
bands   = [0, Fp, Fst, Fs/2]   # passband 0–Fp, stopband Fst–Fs/2
desired = [1, 0]               # gain in each band

# Estimate number of taps (Kaiser formula as starting point)

N, beta = kaiserord(Rs, (Fst-Fp)/(Fs/2))
numtaps = N if N % 2 else N+1  # odd length

print(f"Estimated filter order: {numtaps}")

# Design equiripple filter
h = remez(numtaps, bands, desired, weight=weights, fs=Fs)

# Frequency response
w, H = freqz(h, worN=8192, fs=Fs)
H_db = 20*np.log10(np.maximum(np.abs(H), 1e-12))

# Plot
plt.figure(figsize=(10,5))
plt.plot(w/1e6, H_db, label="Filter response")
plt.axhline(-Rp, color="g", linestyle="--", label="Rp=0.25 dB")
plt.axhline(-Rs, color="r", linestyle="--", label="Rs=80 dB")
plt.axvline(Fp/1e6, color="g", linestyle="--", label="Fp")
plt.axvline(Fst/1e6, color="r", linestyle="--", label="Fst")
plt.xlabel("Frequency (MHz)")
plt.ylabel("Magnitude (dB)")
plt.ylim([-100, 5])
plt.grid()
plt.legend()
plt.show()
```

```
Estimated filter order: 227
```



export to quantized

In [5]:
```
nbits = 16
scale = 2**(nbits - 1)

h_fixed = np.round(h * scale)
h_fixed = np.clip(h_fixed, -scale, scale - 1).astype(np.int16)

np.savetxt("coeffs_fixed_q15.txt", h_fixed, fmt="%d")
```

biquad IIR Notch at 2.4 MHz

Promplems and Solutions

1. original sampling frequency is 9 MHz so Max freq component Present is 4.5 MHz
2. any frequency higher would have been either filtered by the previous filter

or aliased around the 4.5 so an interfernce at 5 would be at 4 if not filtered 3. after the resampling filter the original interfernce at 4 (previously 5) is already filtered 4. so we design only for the IIR at 2.4

numerical stabilty

- hor numerical stabilty we design biquad iir and implement DFII

Design

In [8]:
```
# ============================================================
# 1. Design notch filter (floating point)
# ============================================================
```

```
fs = 6e6        # Sampling frequency
f0 = 2.4e6      # Notch frequency
Q = 30          # Quality factor

# Design IIR notch (2nd order)
b, a = iirnotch(f0/(fs/2), Q)

print("Floating-point coefficients:")
print("b =", b)
print("a =", a)
```

```
Floating-point coefficients:
b = [0.95977357 1.55294626 0.95977357]
a = [1.         1.55294626 0.91954714]
```

In [9]:

```
# ================================================================
# 2. Quantize to Q1.14 fixed-point (16-bit signed)
# ================================================================

frac_bits = 14
scale = 2**frac_bits

def float_to_q14(x):
    return np.round(x * scale).astype(int)

def q14_to_float(x):
    return x / scale

b_q14 = float_to_q14(b)
a_q14 = float_to_q14(a)

print("\nQ1.14 integer coefficients:")
print("b_q14 =", b_q14)
print("a_q14 =", a_q14)

# Convert back to float for simulation
b_fixed = q14_to_float(b_q14)
a_fixed = q14_to_float(a_q14)

print("\nQ1.14 quantized coefficients (float equivalent):")
print("b_fixed =", b_fixed)
print("a_fixed =", a_fixed)
```

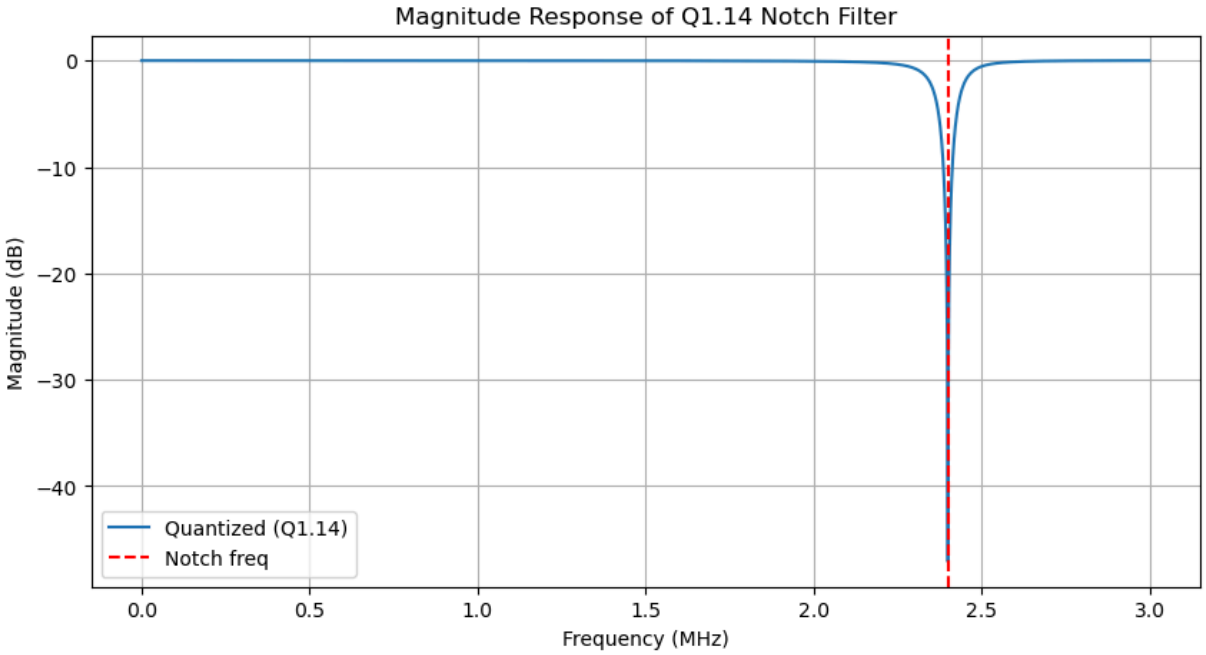
```
Q1.14 integer coefficients:
b_q14 = [15725 25443 15725]
a_q14 = [16384 25443 15066]

Q1.14 quantized coefficients (float equivalent):
b_fixed = [0.95977783 1.55291748 0.95977783]
a_fixed = [1.         1.55291748 0.91955566]
```

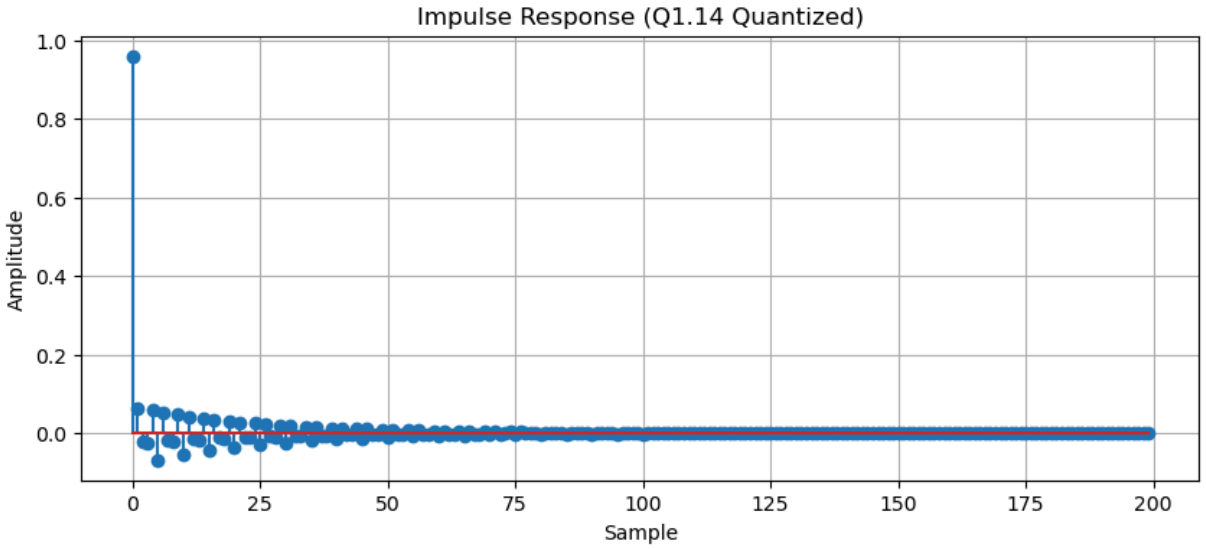
In [10]:

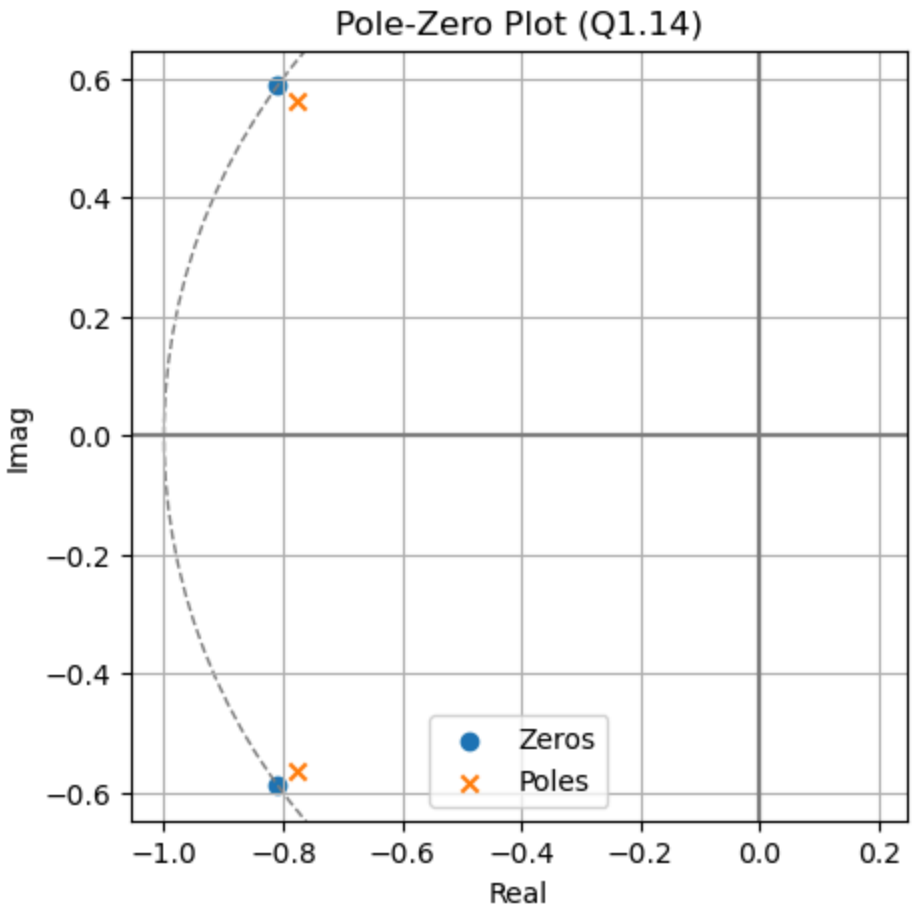
```
# ================================================================
# 3. Frequency response (quantized)
# ================================================================

w, h = freqz(b_fixed, a_fixed, worN=4096, fs=fs)
```

```
plt.figure(figsize=(10,5))
plt.plot(w/1e6, 20*np.log10(np.abs(h)), label='Quantized (Q1.14)')
plt.title('Magnitude Response of Q1.14 Notch Filter')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.axvline(f0/1e6, color='r', linestyle='--', label='Notch freq')
plt.legend()
plt.show()

# ================================================================
# 4. Impulse response (quantized)
# ================================================================

impulse = np.zeros(200)
impulse[0] = 1.0
resp = lfilter(b_fixed, a_fixed, impulse)

plt.figure(figsize=(10, 4))
plt.stem(resp)
plt.title('Impulse Response (Q1.14 Quantized)')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
# ================================================================
# 5. Pole–zero plot (quantized)
# ================================================================

z, p, k = tf2zpk(b_fixed, a_fixed)

plt.figure(figsize=(5,5))
plt.axhline(0, color='gray')
plt.axvline(0, color='gray')
plt.scatter(np.real(z), np.imag(z), marker='o', label='Zeros')
plt.scatter(np.real(p), np.imag(p), marker='x', label='Poles')
circle = plt.Circle((0,0), 1, color='gray', fill=False, linestyle='--')
plt.gca().add_artist(circle)
plt.title('Pole-Zero Plot (Q1.14)')
plt.xlabel('Real')
plt.ylabel('Imag')
plt.grid(True)
plt.axis('equal')
plt.legend()
plt.show()

# ================================================================
# 6. Check stability
# ================================================================

stable = np.all(np.abs(p) < 1)
print("\nFilter stability after quantization:", "Stable ✅" if stable else "

# ================================================================
# 7. Save fixed-point coefficients (for HDL)
```

```
# ============================================================

np.savetxt("notch_b_q14.txt", b_q14, fmt="%d")
np.savetxt("notch_a_q14.txt", a_q14, fmt="%d")

print("\nSaved coefficients to notch_b_q14.txt and notch_a_q14.txt")
```








```
Filter stability after quantization: Stable ✅

Saved coefficients to notch_b_q14.txt and notch_a_q14.txt
```

CIC decimation filter

CIC Filter Specs:

- Configurable decimation (1–16)

- Passband ripple ≤ 0.5 dB (with compensation FIR)

- Stopband attenuation ≥ 60 dB

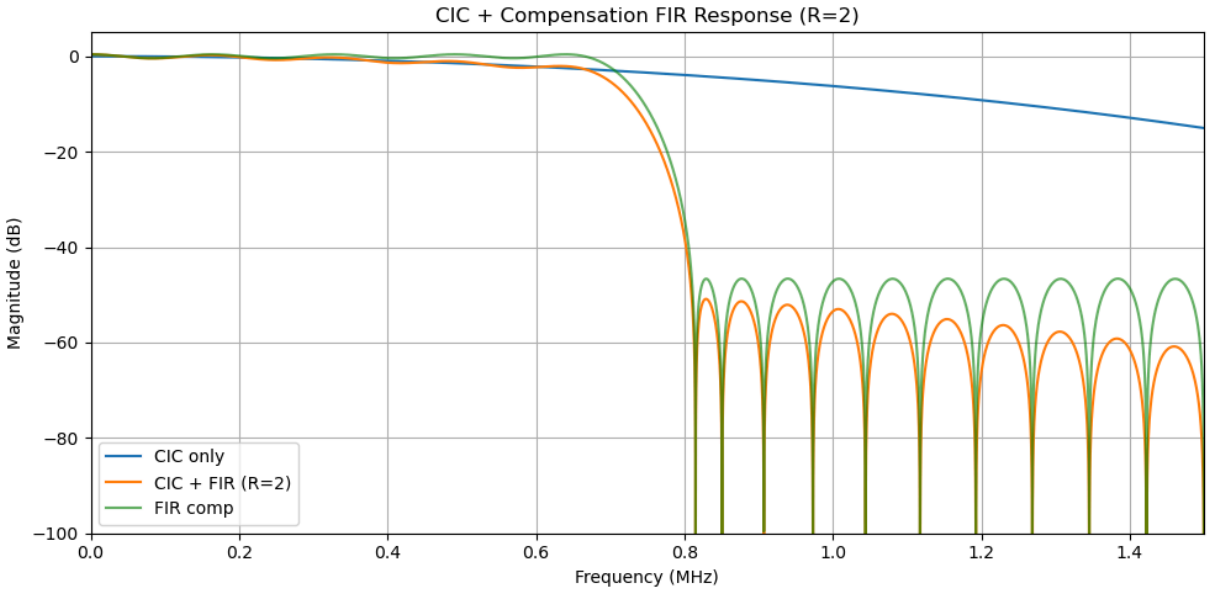
- Must handle internal bit growth

In [15]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import remez, freqz

# ----------------------------
# Parameters
# ----------------------------
fs_in = 6e6         # input sampling rate (Hz)
N = 5               # number of CIC stages
M = 1               # differential delay
numtaps = 40
```

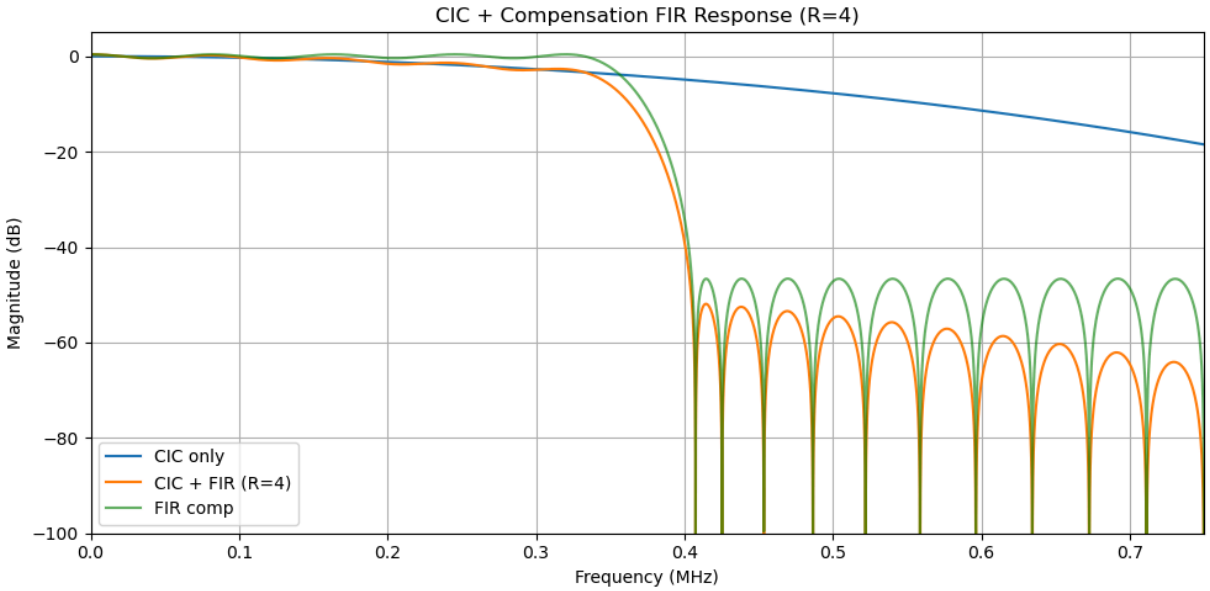
```
nfft = 16384
eps = 1e-12

# Quantization parameters (s.1.15)
N_FRAC = 15
SCALE = 2 ** N_FRAC
MAX_Q = (2 ** (N_FRAC + 1)) - 1

# ----------------------------
# Function to quantize coefficients to Q1.15
# ----------------------------
def quantize_q15(x):
    x = np.clip(x, -1, 1 - 1/SCALE)
    q = np.round(x * SCALE).astype(int)
    return q, q / SCALE

# ----------------------------
# Main Loop for different decimation factors
# ----------------------------
R_values = [2, 4, 8, 16]

for R in R_values:
    print("\n==============================")
    print(f"Designing for R = {R}")
    print("==============================")

    fs_out = fs_in / R
    w_out = np.linspace(0, np.pi, nfft)
    f_out = w_out * fs_out / (2 * np.pi)

    # ----------------------------
    # CIC Response (output-domain)
    # ----------------------------
    num = np.sin(M * w_out / 2.0)
    den = np.sin(w_out / (2.0 * R))
    den[np.abs(den) < eps] = eps
    H_cic_out = (num / (R * M * den)) ** N


    print("CIC diagnostics:")
    print("  CIC DC gain =", ((R * M) ** N))
    print("  finite check: any NaN/Inf? ", np.any(~np.isfinite(H_cic_out)))

    # ----------------------------
    # Compensation FIR Design
    # ----------------------------
    f_pass = 0.45 * (fs_out / 2.0)
    bands = [0, f_pass, f_pass * 1.2, fs_out / 2.0]
    desired = [1, 0]
    weights = [1, 10]

    comp = remez(numtaps, bands, desired, weight=weights, fs=fs_out)
    w_comp, H_comp = freqz(comp, 1, worN=w_out)
    H_total = H_cic_out * H_comp
```

```
    print("  finite check: any NaN/Inf? ", np.any(~np.isfinite(H_comp)))

    # ----------------------------
    # Quantize to Q1.15
    # ----------------------------
    q15_int, q15_float = quantize_q15(comp)
    print(f"  Quantized range: min={q15_int.min()} max={q15_int.max()}")

    # ----------------------------
    # Export coefficients
    # ----------------------------
    filename = f"comp_R{R}.txt"
    np.savetxt(filename, q15_int, fmt="%d")
    print(f"  Saved quantized coefficients to {filename}")

    # ----------------------------
    # Optional Plot
    # ----------------------------
    plt.figure(figsize=(10, 5))
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_cic_out)+1e-18), label="CIC c
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_total)+1e-18), label=f"CIC +
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_comp)+1e-18), color="green",
    plt.title(f"CIC + Compensation FIR Response (R={R})")
    plt.xlabel("Frequency (MHz)")
    plt.ylabel("Magnitude (dB)")
    plt.grid(True)
    plt.legend()
    plt.xlim(0, fs_out/2/1e6)
    plt.ylim(-100, 5)
    plt.tight_layout()
    plt.show()
```

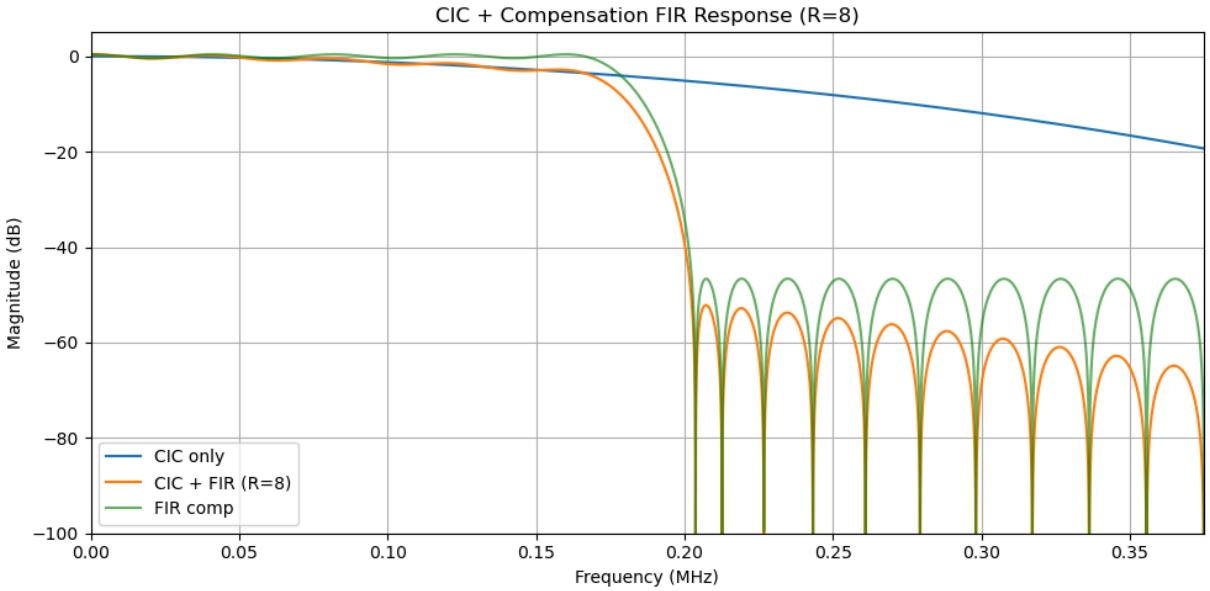
```
==============================
Designing for R = 2
==============================
CIC diagnostics:
  CIC DC gain = 32
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R2.txt
```



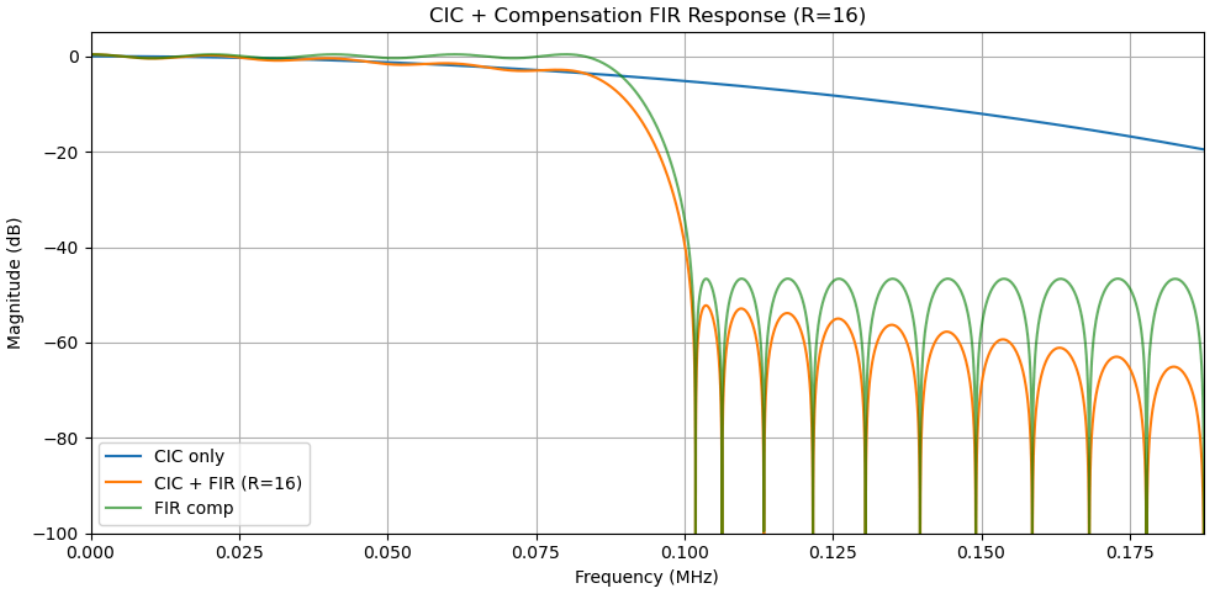

```
==============================
Designing for R = 4
==============================
CIC diagnostics:
  CIC DC gain = 1024
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R4.txt
```




```
==============================
Designing for R = 8
==============================
CIC diagnostics:
  CIC DC gain = 32768
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R8.txt
```




```
==============================
Designing for R = 16
==============================
CIC diagnostics:
  CIC DC gain = 1048576
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R16.txt
```



In [ ]:

Golden Model For DFE

by : Group 7

Hazem Yasser Mahmoud

Mohamed Ahmed Elsayed

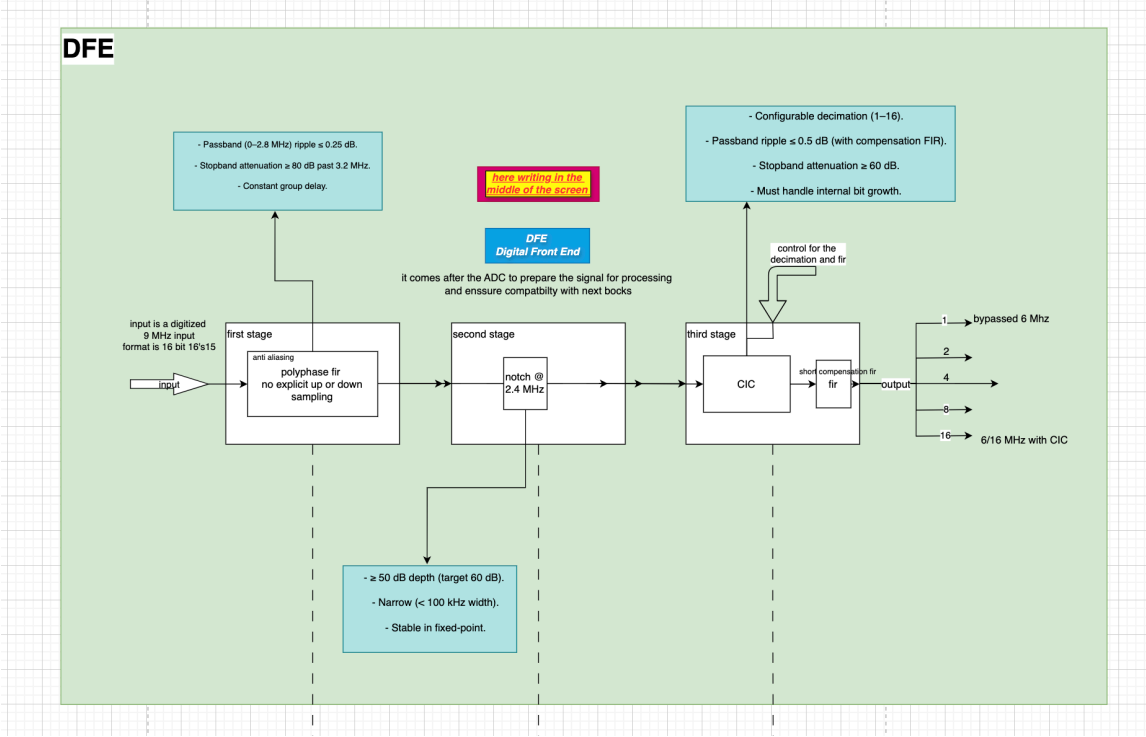
mohamed Naem

imports

In [108…

```
import numpy as np
from scipy.signal import firwin , freqz , resample_poly
import matplotlib.pyplot as plt
```

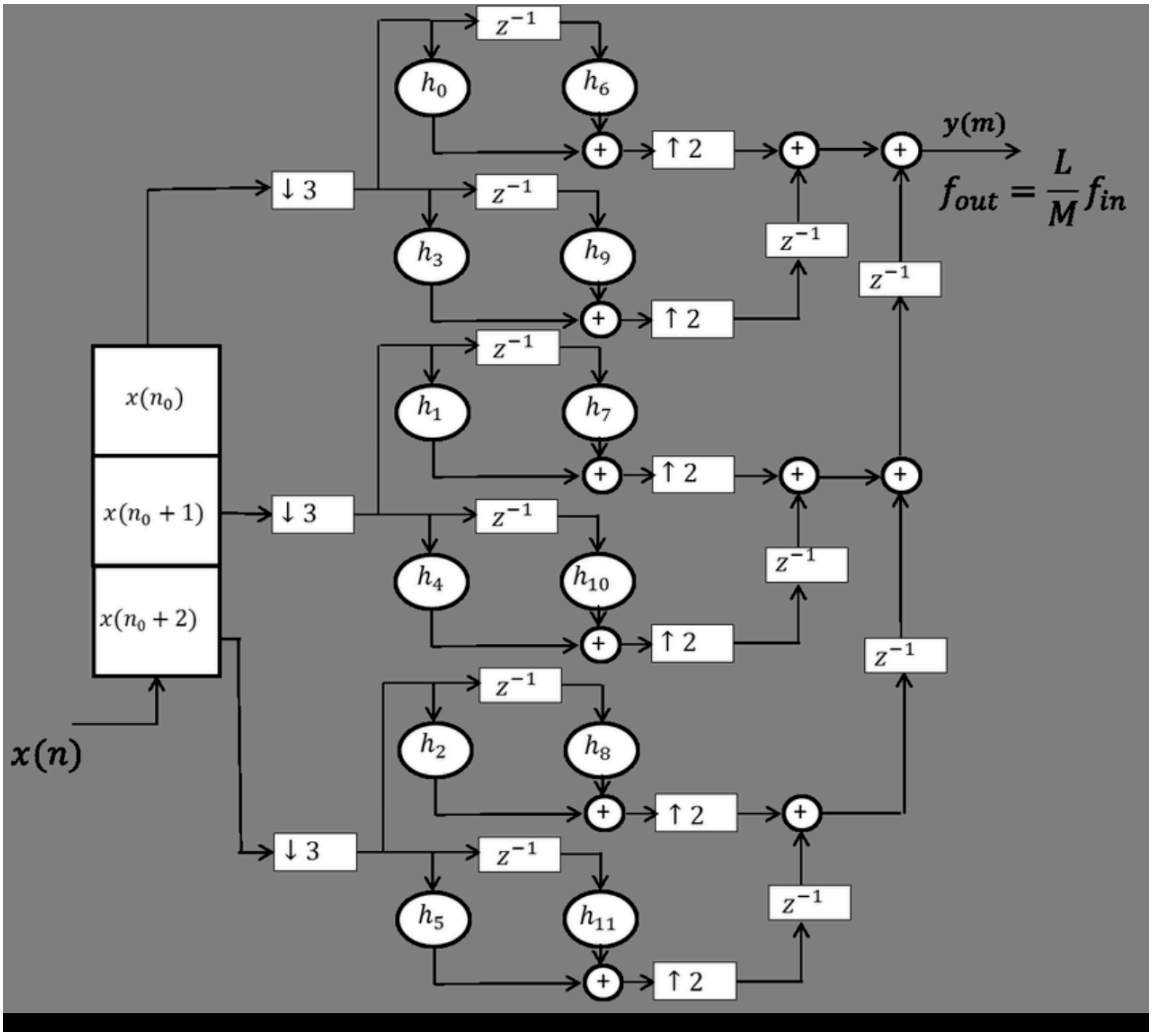
DFE stages



Paper Proposed rational Polyphase resampler structure

note that each branch has more coeff than in the image each sybsequent two have 6 in differnce so h0,h6,h12....

https://www.researchgate.net/profile/Abhishek-Kumar-202/publication/319763113

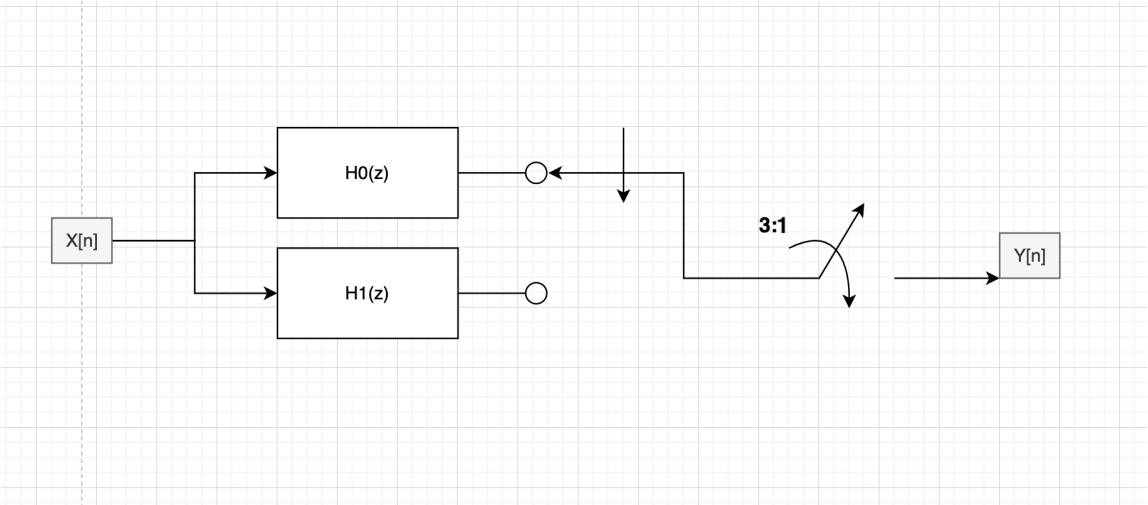


In [109…

```python
def polyphase_resample(x, L, M,h):
    """
    Rational resampler using explicit 3D polyphase structure with
    parallel main phases and interleaved subphases.
    """

    # 2. Polyphase decomposition into M x L branches
    phases = []
    for m in range(M):   # main phases
        row = []
        for l in range(L):  # sub-phases
            row.append(h[m + l*M :: L*M])
        phases.append(row)

    # 3. Split input into M decimated streams (one per main phase)
    x_p = [x[m::M] for m in range(M)]

    # 4. Process each main phase separately
    parallel_outputs = [[] for _ in range(M)]

    for m in range(M):       # main phase
        x_m = x_p[m]
```

```
        h_m = phases[m]

        parallel_outputs[m].extend([0.0] * m)

        for n in range(len(x_m)):
            subphase_samples = []
            for l in range(L):   # subphases
                acc = 0.0
                h_ml = h_m[l]
                for k in range(len(h_ml)):
                    if n - k >= 0:
                        acc += h_ml[k] * x_m[n - k]
                subphase_samples.append(acc)
            # interleave subphase results into one stream per main phase
            parallel_outputs[m].extend(subphase_samples)

    # 5. Sum the parallel streams (sample-wise)
    min_len = min(len(p) for p in parallel_outputs)
    summed_output = np.zeros(min_len)
    for m in range(M):
        summed_output += np.array(parallel_outputs[m][:min_len])

    return np.asarray(summed_output) #, parallel_outputs
```

traditional polyphase resampler structure structure

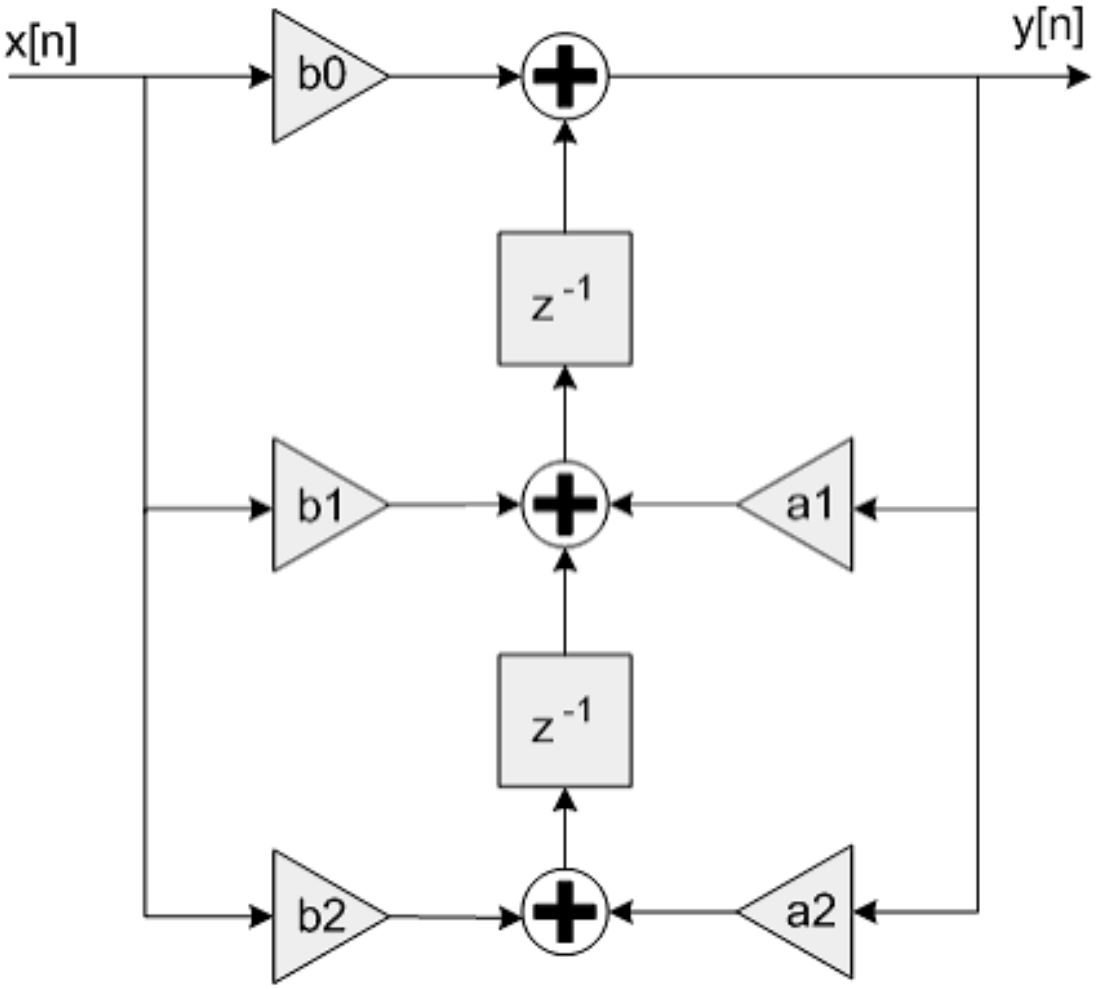


In [110…

```
def polyphase_traditional(x, L, M, h):

    phases = [h[l::L] for l in range(L)]
    y = []
    t = 0
    for n in range(len(x)):
        for l in range(L):
            acc = 0.0
            for k in range(len(phases[l])):
                if n - k >= 0:
                    acc += phases[l][k] * x[n - k]
            # output sample if timing hits a downsample point
            if t % M == 0:
```

```
                y.append(acc)
            t += 1
    return np.array(y)
```

biquad_IIR_DFII_T

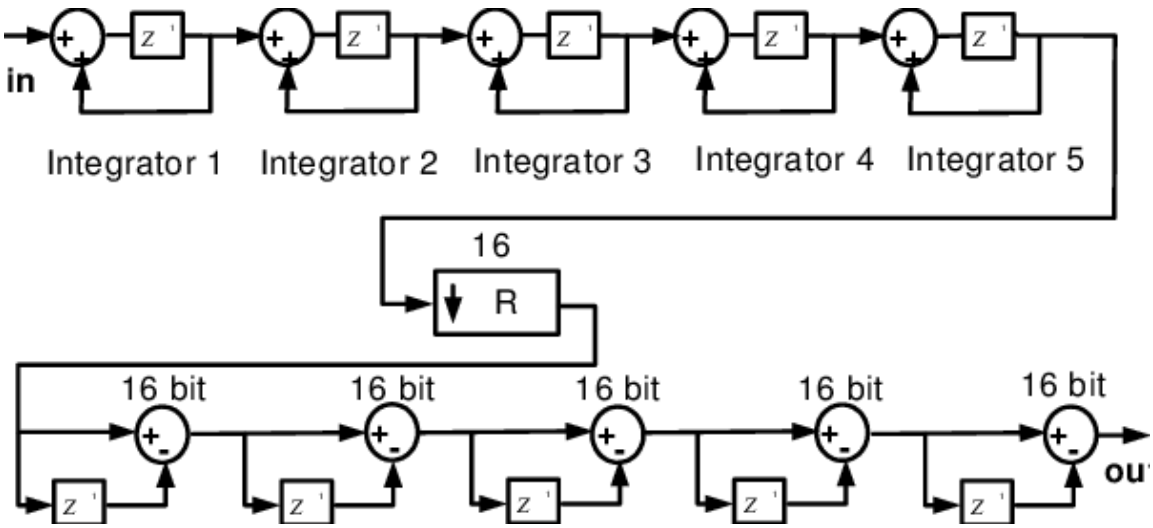


In [111…

```
def biquad_df2t(x, b, a):
    """
    Floating-point Direct Form II Transposed biquad filter
    x : input signal (float array)
    b : [b0, b1, b2]
    a : [1, a1, a2]  (a[0] must be 1)
    returns y : filtered output (float array)
    """
    N = len(x)
    y = np.zeros(N)
    s1, s2 = 0.0, 0.0  # delay elements

    b0, b1, b2 = b
    _, a1, a2 = a

    for n in range(N):
        # Direct Form II Transposed structure
        y_n = b0 * x[n] + s1
```

```
        s1 = b1 * x[n] - a1 * y_n + s2
        s2 = b2 * x[n] - a2 * y_n
        y[n] = y_n

    return y
```

CIC + comp FIR



CIC no compensation

using fixed point integers

more prone to overflow at high freq but accurate HW modelling. overflow happen for big frequency near nyqist

In [112…

```
def cic_decimator(x, R=2, M=1, N=5, B_in=16, B_out=16):
    """
    CIC decimator model with FPGA-style bit growth and truncation.

    Parameters:
        x : np.ndarray      - Input signal in range [-1, 1)
        R : int             - Decimation factor
        M : int             - Differential delay
        N : int             - Number of stages
        B_in : int          - Input bit width
        B_out : int         - Final output width

    Returns:
        y_out : np.ndarray  - Output after decimation and truncation
    """
    x = np.asarray(x, dtype=np.float64)

    # Convert to fixed-point integer
    x_int = np.round(x * (2**(B_in - 1))).astype(np.int64)

    n_in = len(x_int)
    integ = np.zeros((N, n_in), dtype=np.int64)
```

```
    # --- Integrator section ---
    integ[0, 0] = x_int[0]
    for i in range(1, n_in):
        integ[0, i] = integ[0, i-1] + x_int[i]
    for stage in range(1, N):
        integ[stage, 0] = integ[stage-1, 0]
        for i in range(1, n_in):
            integ[stage, i] = integ[stage, i-1] + integ[stage-1, i]
    y_int = integ[-1, :]

    # --- Decimate ---
    y_dec = y_int[::R]

    # --- Comb section ---
    n_out = len(y_dec)
    comb = np.zeros((N, n_out), dtype=np.int64)
    for stage in range(N):
        delay = np.zeros(M, dtype=np.int64)
        for i in range(n_out):
            cur_in = y_dec[i] if stage == 0 else comb[stage-1, i]
            cur_out = cur_in - delay[0]
            comb[stage, i] = cur_out
            delay = np.roll(delay, -1)
            delay[-1] = cur_in

    y_out = comb[-1, :]

    # --- Bit growth and truncation ---
    shift_bits = int(N * np.ceil(np.log2(R * M)))
    y_trunc = np.right_shift(y_out, shift_bits)
    y_float = y_trunc / (2**(B_out - 1))

    # Clip to [-1, 1)
    y_float = np.clip(y_float, -1, 1 - 1/(2**(B_out-1)))
    return y_float
```

using quantized float

no overflow at all

In [113…

```
def cic_decimator_2(x, R=2, M=1, N=5, B_in=16, B_out=16):
    """
    CIC decimator model (safe version, no overflow).
    Keeps same parameters and scaling behavior,
    but uses float64 arithmetic internally.

    Parameters:
        x : np.ndarray      - Input signal in range [-1, 1)
        R : int             - Decimation factor
        M : int             - Differential delay
        N : int             - Number of stages
        B_in : int          - Input bit width
        B_out : int         - Final output width
```

```
    Returns:
        y_out : np.ndarray  - Output after decimation and truncation
    """
    x = np.asarray(x, dtype=np.float64)

    # --- Simulate quantization (just for realism) ---
    x_int = np.round(x * (2**(B_in - 1))) / (2**(B_in - 1))

    n_in = len(x_int)
    integ = np.zeros((N, n_in), dtype=np.float64)

    # --- Integrator section ---
    integ[0, 0] = x_int[0]
    for i in range(1, n_in):
        integ[0, i] = integ[0, i-1] + x_int[i]
    for stage in range(1, N):
        integ[stage, 0] = integ[stage-1, 0]
        for i in range(1, n_in):
            integ[stage, i] = integ[stage, i-1] + integ[stage-1, i]

    y_int = integ[-1, :]

    # --- Decimation ---
    y_dec = y_int[::R]

    # --- Comb section ---
    n_out = len(y_dec)
    comb = np.zeros((N, n_out), dtype=np.float64)
    for stage in range(N):
        delay = np.zeros(M, dtype=np.float64)
        for i in range(n_out):
            cur_in = y_dec[i] if stage == 0 else comb[stage-1, i]
            cur_out = cur_in - delay[0]
            comb[stage, i] = cur_out
            delay = np.roll(delay, -1)
            delay[-1] = cur_in

    y_out = comb[-1, :]

    # --- Bit growth and truncation emulation ---
    shift_bits = int(N * np.ceil(np.log2(R * M)))
    y_scaled = y_out / (2**shift_bits)

    # Clip to output range
    y_float = np.clip(y_scaled, -1, 1 - 1/(2**(B_out - 1)))
    return y_float
```

CIC compensated

In [114…

```
def cic_comp(x, R=2):
    """
    CIC + Compensation FIR cascade.
    - If R == 1: bypass (return x unchanged)
    - Else: call cic_decimator_fpga() and convolve with compensation filter
    """
```

```
    if R == 1:
        return np.asarray(x, dtype=np.float64)

    # --- 1. Run CIC decimator ---
    # y_cic = cic_decimator(x, R=R)
    y_cic = cic_decimator_2(x, R=R)

    # --- 2. Load compensation filter coefficients ---
    coeff_path = f"comp_R{R}.txt"
    try:
        h_comp = np.loadtxt(coeff_path)/(2**15)
    except FileNotFoundError:
        raise FileNotFoundError(f"Compensation file '{coeff_path}' not found

    # --- 3. Manual convolution (low-level style) ---
    n = len(y_cic)
    L = len(h_comp)
    y_out = np.zeros(n + L - 1, dtype=np.float64)

    for i in range(n):
        for k in range(L):
            y_out[i + k] += y_cic[i] * h_comp[k]

    return y_out
```

DFE (integrated stages)

In [115…

```
def dfe_integrated(x, R=2):
    """
    Full DFE processing chain:
        1. Polyphase Resampler (L=2, M=3)
        2. Biquad Notch Filter
        3. CIC Decimator + Compensation FIR

    Loads coefficients from:
        - coeffs_fixed_q15.txt  (polyphase filter)
        - notch_b_q14.txt, notch_a_q14.txt  (biquad filter)
        - comp_R{R}.txt  (CIC compensation FIR)
    """

    # --- Stage 1: Polyphase Resampler ---
    L , M = 2,3
    h_15 = np.loadtxt("coeffs_fixed_q15.txt",dtype=int)/(2**15) *L
    P = int(np.ceil(len(h_15)/M/L))
    h = np.concatenate([h_15, np.zeros(M * P * L - len(h_15))])
    y = polyphase_resample(x, L=2, M=3, h=h)

    # --- Stage 2: Biquad Notch Filter ---
    b = np.loadtxt("notch_b_q14.txt")
    a = np.loadtxt("notch_a_q14.txt")
    frac_bits = 14
    b = b / (2**frac_bits)
    a = a / (2**frac_bits)
    y = biquad_df2t(y, b, a)
```

```
    # --- Stage 3: CIC + Compensation ---

    y_out = cic_comp(y, R=R)
    return y_out
```

Helping Routines

Quantize

In [116…

```
def fixed_point_quantize(x, n_int=1, n_frac=15):
    """
    Quantize a value (or numpy array) x to signed fixed-point s.I.F format.

    Parameters:
        x: float or np.ndarray – input value(s)
        n_int: int – number of integer bits (excluding sign bit)
        n_frac: int – number of fractional bits

    Returns:
        qx: quantized value in float (after rounding)
        q_int: quantized integer representation (int)
    """
    import numpy as np

    # Compute limits
    scale = 2 ** n_frac
    max_val = (2 ** n_int) - (1 / scale)
    min_val = - (2 ** n_int)

    # Clamp input
    x_clamped = np.clip(x, min_val, max_val)

    # Quantize
    q_int = np.round(x_clamped * scale).astype(int)

    # Handle wrap-around if overflowed
    q_int = np.clip(q_int, -int(2 ** (n_int + n_frac)), int(2 ** (n_int + n_

    # Convert back to float
    qx = q_int / scale

    # return qx, q_int
    return qx
```

Filter inspection

In [117…

```
def inspect_filter(h, fs):
    # DC gain: Sum of filter coefficients (valid for FIR filters)
    dc_gain = np.sum(h)
```

```
    print(f"Filter length: {len(h)}, DC gain (sum of taps) = {dc_gain:.6f}")

    # Impulse response
    n = np.arange(len(h))  # Time indices for impulse response

    # Frequency response
    w, H = freqz(h, worN=4096)
    f = w * fs / (2 * np.pi)  # Convert rad/sample to Hz

    # Create side-by-side plots
    plt.figure(figsize=(12, 4))

    # Subplot 1: Impulse response
    plt.subplot(1, 2, 1)
    plt.stem(n, h, basefmt=" ")
    plt.xlabel('Sample index (n)')
    plt.ylabel('Amplitude')
    plt.title('Impulse Response')
    plt.grid(True)

    # Subplot 2: Frequency response
    plt.subplot(1, 2, 2)
    plt.plot(f / 1e6, 20 * np.log10(np.abs(H) + 1e-12))
    plt.axvline(3.0, color='r', linestyle='--', label='3 MHz (output Nyquist
    plt.xlabel('Frequency (MHz)')
    plt.ylabel('Magnitude (dB)')
    plt.title('Frequency Response')
    plt.grid(True)
    plt.legend()
    plt.xlim(0, fs / 2 / 1e6)

    plt.tight_layout()  # Adjust spacing between subplots
    plt.show()
```

plotting I/O

In [118…

```
def plot_time_freq(x, y, fs_in, fs_out, title,samples=200,L=1,M=1,step = 200
    fig, axes = plt.subplots(2,1, figsize=(12,6))

    # Trim to same duration for time plot
    min_samples = min(len(x), int(len(y)*fs_in/fs_out))
    x = x[:min_samples]
    y = y[:int(min_samples*fs_out/fs_in)]

    n_in = np.arange(len(x))/fs_in*1e6
    n_out = np.arange(len(y))/fs_out*1e6

    # ---- Time-domain ----
    axes[0].plot(n_in[samples-step:samples], x[samples-step:samples], label=
    axes[0].plot(n_out[int((samples-step)*(L/M)):int(samples*(L/M))], y[int(
    axes[0].set_title(f"{title} - Time domain ( {step} samples)")
    axes[0].set_xlabel("Time (µs)")
    axes[0].legend()

    # ---- Frequency spectrum ----
```

```
    Nfft = 16384*64  # power of 2 for clean FFT
    X = np.fft.fftshift(np.fft.fft(x, Nfft))
    Y = np.fft.fftshift(np.fft.fft(y, Nfft))
    f_in = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_in))/1e6
    f_out = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_out))/1e6

    axes[1].plot(f_in, 20*np.log10(np.abs(X)+1e-12),label="input",color="blu
    axes[1].plot(f_out, 20*np.log10(np.abs(Y)+1e-12),label="output", color="
    axes[1].set_xlim([-max(fs_out/(2*1e6),fs_in/(2*1e6)), max(fs_out/(2*1e6)
    axes[1].set_title("Output Spectrum")
    axes[1].set_xlabel("Frequency (MHz)")
    axes[1].set_ylabel("Magnitude (dB)")
    axes[1].legend()

    plt.tight_layout()
    plt.show()
```

Testing & Graphs

Polyphase Filter

Promplems and Solutions

1. For prpoer resampling the fir filter need to be $w_c = \frac{1}{\max(L, M)} \cdot \frac{F_s}{2}$
2. so we Design a Prototype FIR at 1.5 wher 1.4 lesss than .25dB and after 1.6 less than -80dB
3. implementing this prototype FIR in polyphase structure give effective filtering at 3 MHz

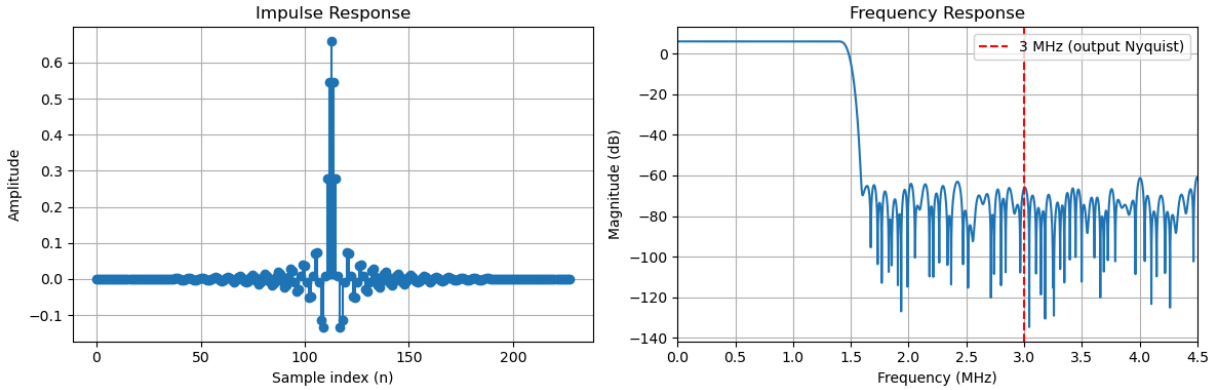
Paramaters definition & Filter Testing

In [119…

```
# Example usage
fs_in = 9e6
L, M = 2, 3
fs_out = fs_in * L/M
N = 8192   # number of samples
t = np.arange(N) / fs_in

#  # 1. Design prototype filter at upsampled rate

h_win = firwin(403, 1/ max(L,M) ) * L
h_15 = np.loadtxt("coeffs_fixed_q15.txt",dtype=int)/(2**15) *L
P = int(np.ceil(len(h_15)/M/L))
h = np.concatenate([h_15, np.zeros(M * P * L - len(h_15))])
```

```
inspect_filter(h,9e6)


# Example input: 1 MHz tone
t = np.arange(0, 9000) / fs_in
x = np.sin(2 * np.pi * 1e6 * t)

y = polyphase_resample(x, L, M,h)
print("Input length:", len(x))

print("Final output length:", len(y))
```

```
Filter length: 228, DC gain (sum of taps) = 1.998474
```

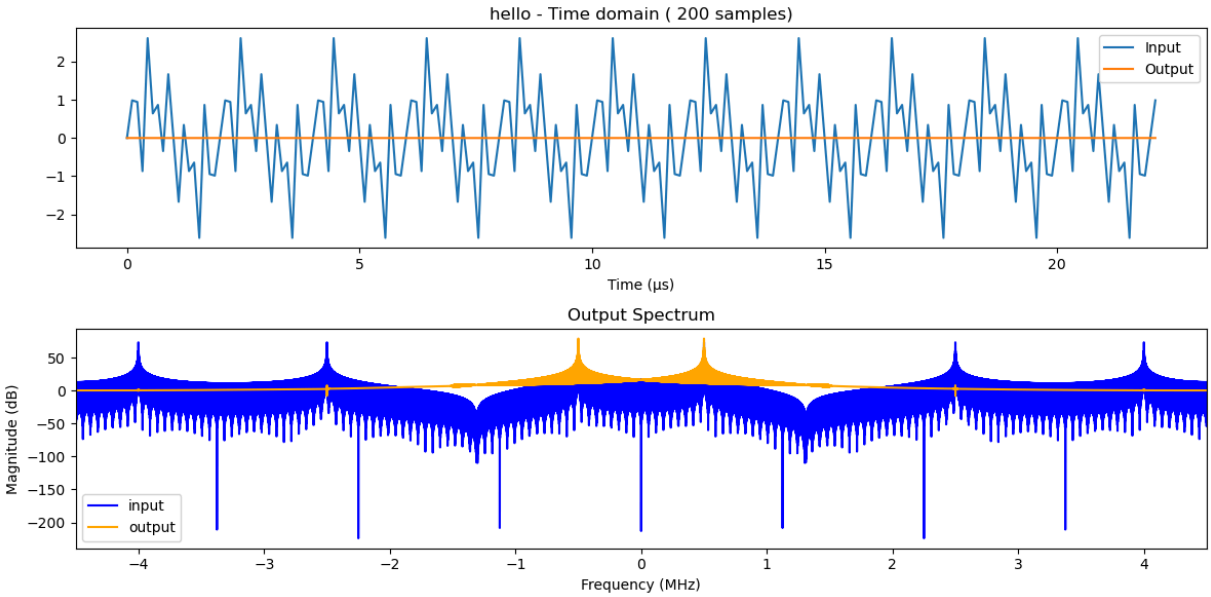



```
Input length: 9000
Final output length: 6000
```

prototype filter when not in polyphase strucure only filter at 1.5 MHz

In [120…

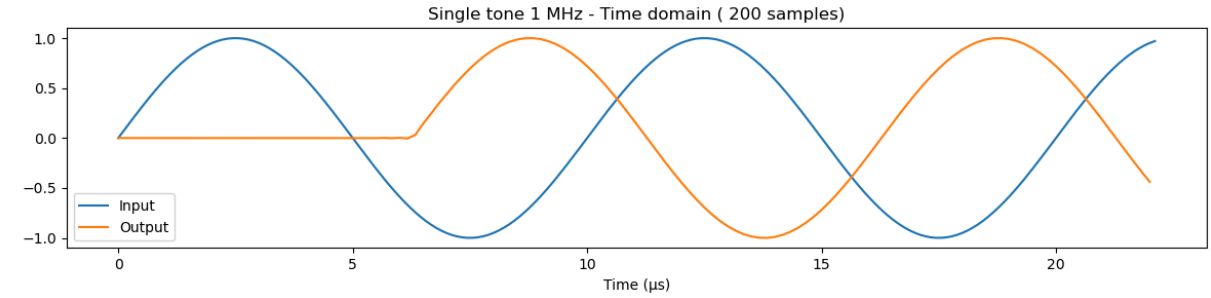
```
y = []
x = np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.sin(2*np.pi*5e6*t
h_ml = firwin(603, 1/ max(2,3) ) * 2
for n in range(len(x)):
        acc = 0.0
        for k in range(len(h_ml)):
            if n - k >= 0:
                acc += h_ml[k] * x[n - k]
        y.append(acc)
plot_time_freq(x, y, fs_in, fs_in,"hello")
```

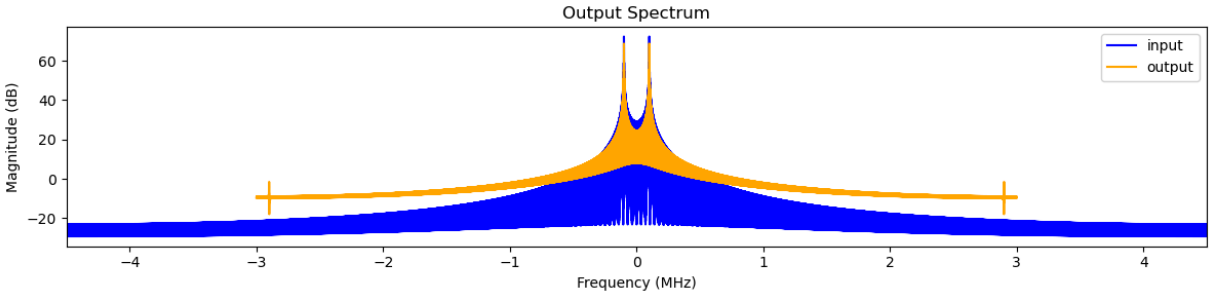


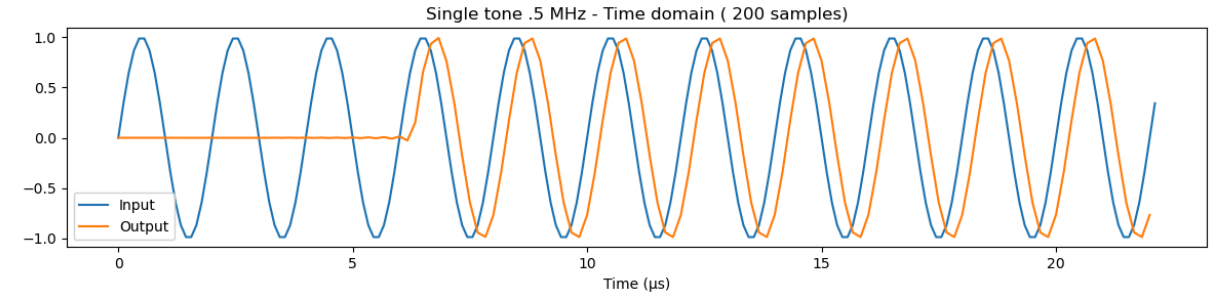
I/O Testing

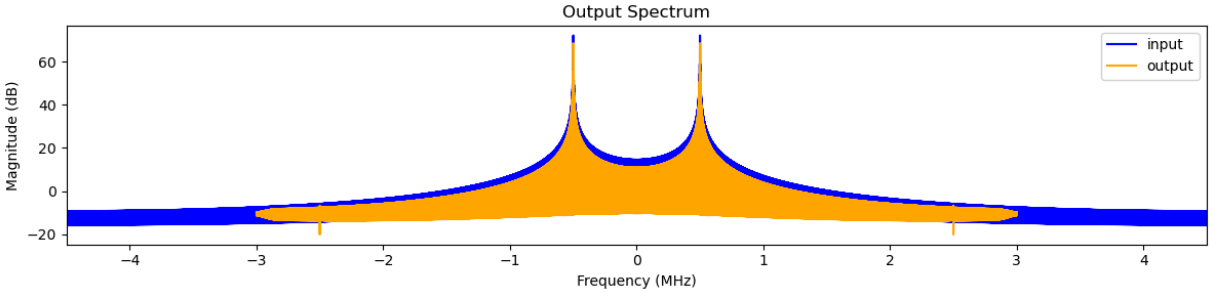
In [121…

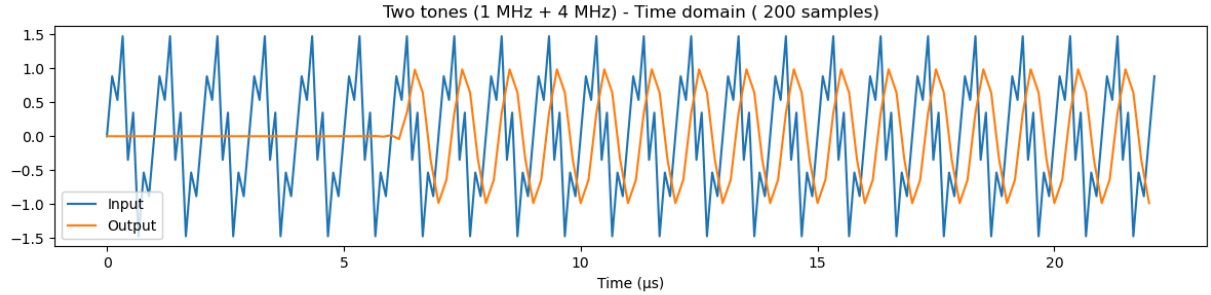
```
# ----------------- TESTING -----------------
fs_in = 9e6
L, M = 2, 3
fs_out = fs_in * L/M
N = 8192   # number of samples
t = np.arange(N) / fs_in

# Different input signals
signals = {
    "Single tone 1 MHz": np.sin(2*np.pi*1e5*t),
    "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi*
    "Wideband noise": np.random.randn(N),
    "Multi-tone": np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.sir
}

# Run tests
for name, sig in signals.items():
    y = polyphase_resample(sig, L, M,h)
    # y_2 = polyphase_traditional(sig, L, M,h)
    plot_time_freq(sig, y, fs_in, fs_out, name,200,L,M)
    # plot_time_freq(sig, y_2, fs_in, fs_out, name+" matlab",200,L,M)
```

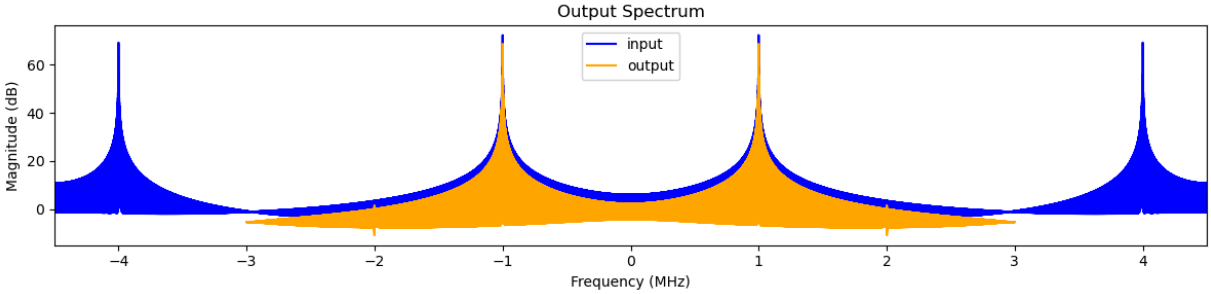

Single tone 1 MHz - Time domain (200 samples)
Input
Output
Time (µs)

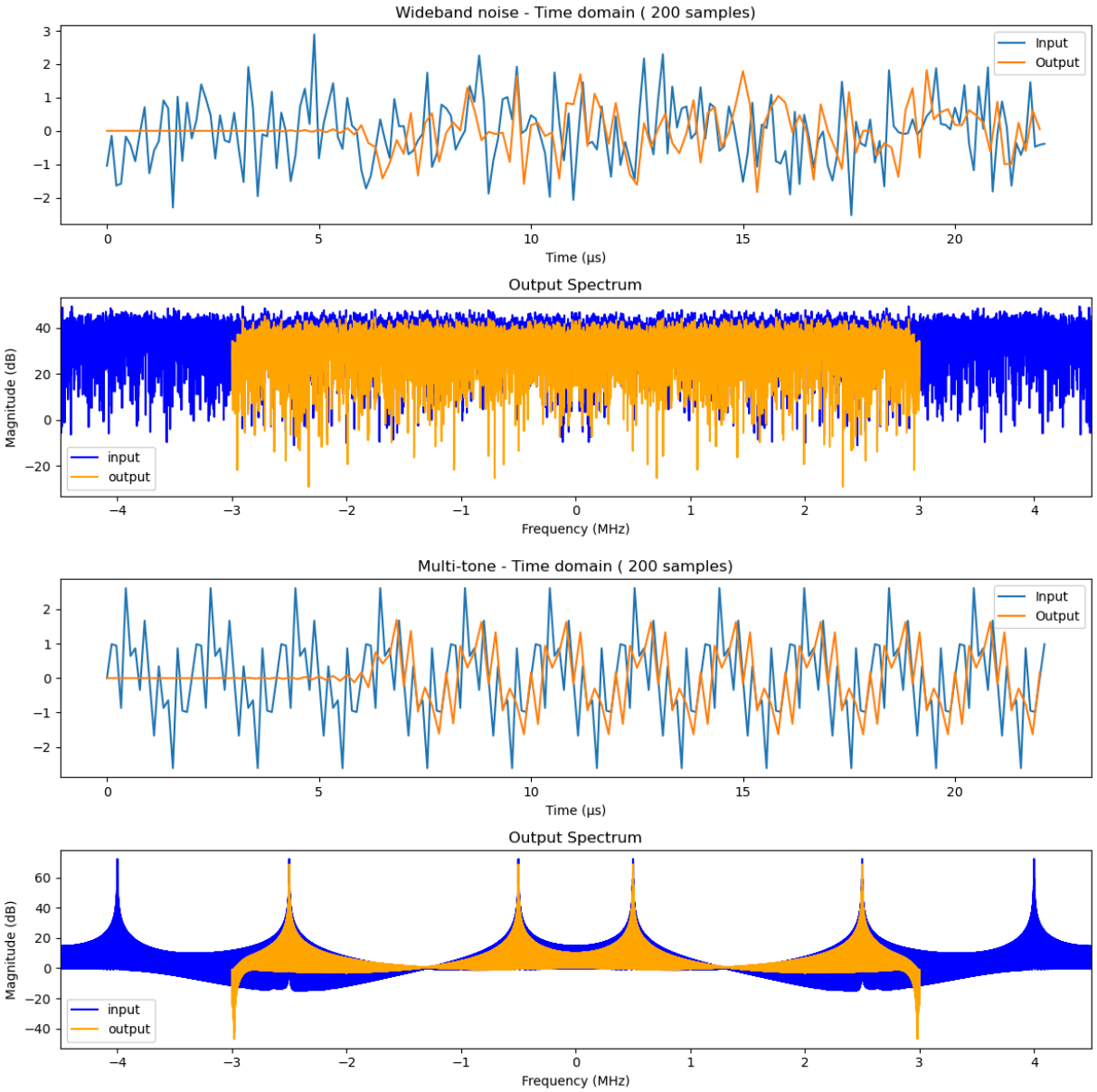

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)


Single tone .5 MHz - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)


Two tones (1 MHz + 4 MHz) - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)



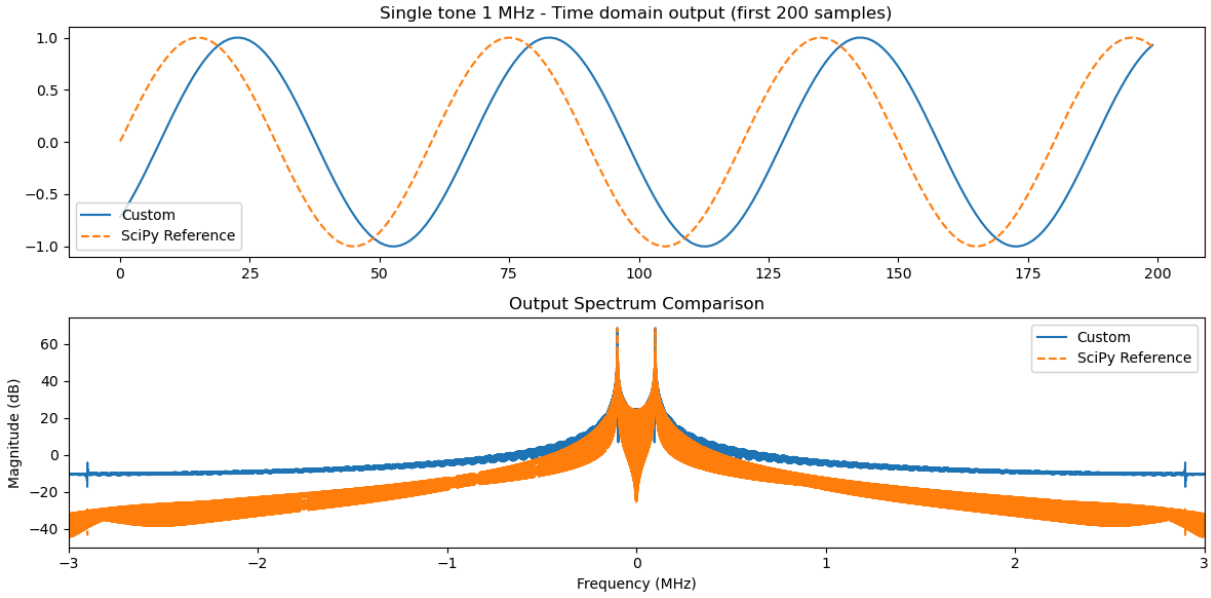
reference comparison

In [122…

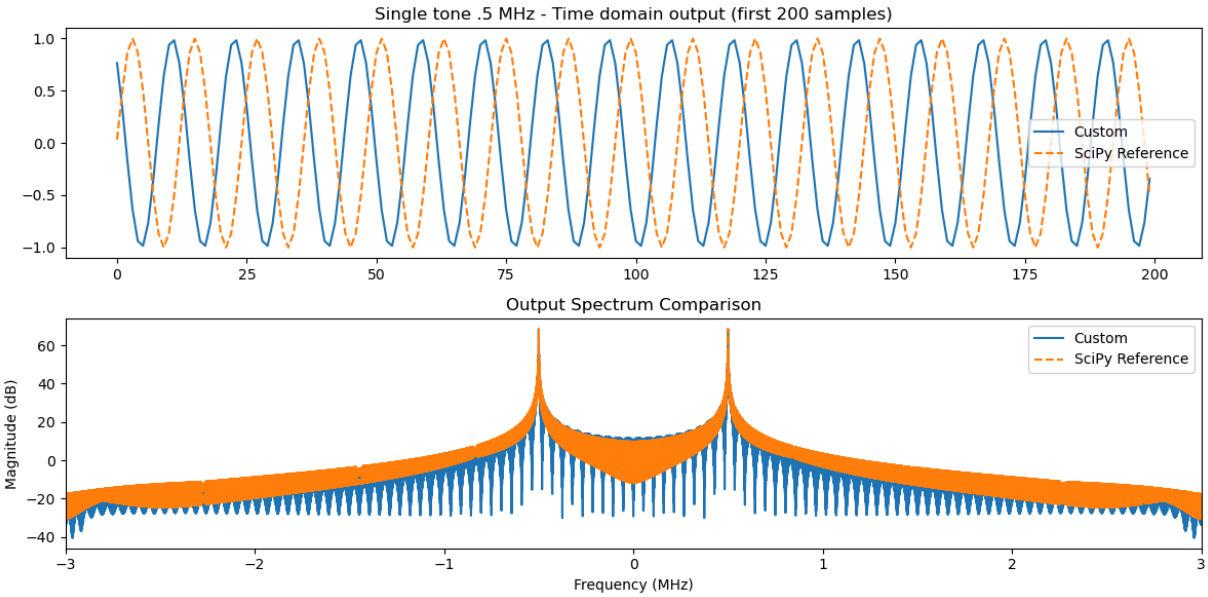
```
# ----------------- COMPARISON BLOCK -----------------
for name, sig in signals.items():
    # Your implementation
    y_custom = polyphase_resample(sig, L, M,h)

    # Reference implementation (SciPy)
    y_ref = resample_poly(sig, L, M, window=('kaiser', 8.6))  # default Kais

    # --- Match lengths for fair comparison ---
    min_len = min(len(y_custom), len(y_ref))
    y_custom = y_custom[:min_len]
    y_ref = y_ref[:min_len]

    # --- Time-domain difference ---
    mse = np.mean((y_custom - y_ref)**2)
    print(f"{name}: MSE vs. SciPy = {mse:.2e}")
```

```
    # --- Plot comparison ---
    plt.figure(figsize=(12,6))

    # First 200 samples (time domain)
    plt.subplot(2,1,1)
    plt.plot(y_custom[150:350], label="Custom")
    plt.plot(y_ref[:200], '--', label="SciPy Reference")
    plt.title(f"{name} - Time domain output (first 200 samples)")
    plt.legend()

    # Spectrum comparison
    Nfft = 16384
    Yc = np.fft.fftshift(np.fft.fft(y_custom, Nfft))
    Yr = np.fft.fftshift(np.fft.fft(y_ref, Nfft))
    f = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_out))/1e6

    plt.subplot(2,1,2)
    plt.plot(f, 20*np.log10(np.abs(Yc)+1e-12), label="Custom")
    plt.plot(f, 20*np.log10(np.abs(Yr)+1e-12), '--', label="SciPy Reference"
    plt.xlim([-fs_out/(2*1e6), fs_out/(2*1e6)])
    plt.title("Output Spectrum Comparison")
    plt.xlabel("Frequency (MHz)")
    plt.ylabel("Magnitude (dB)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

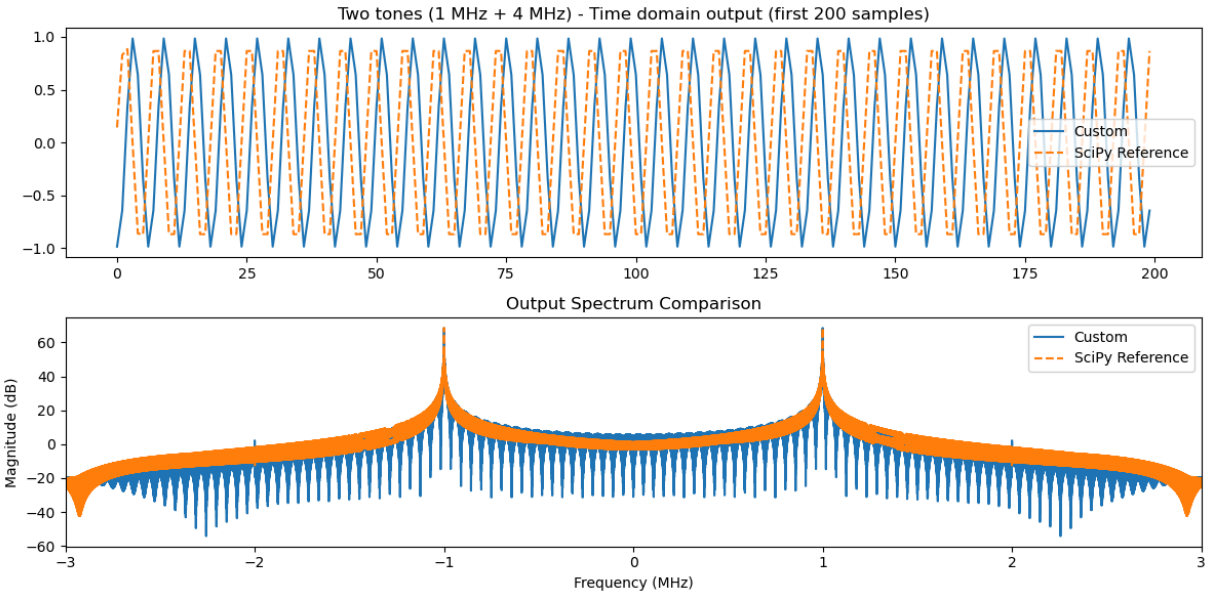
```
Single tone 1 MHz: MSE vs. SciPy = 1.69e+00
```




```
Single tone .5 MHz: MSE vs. SciPy = 3.58e-01
```

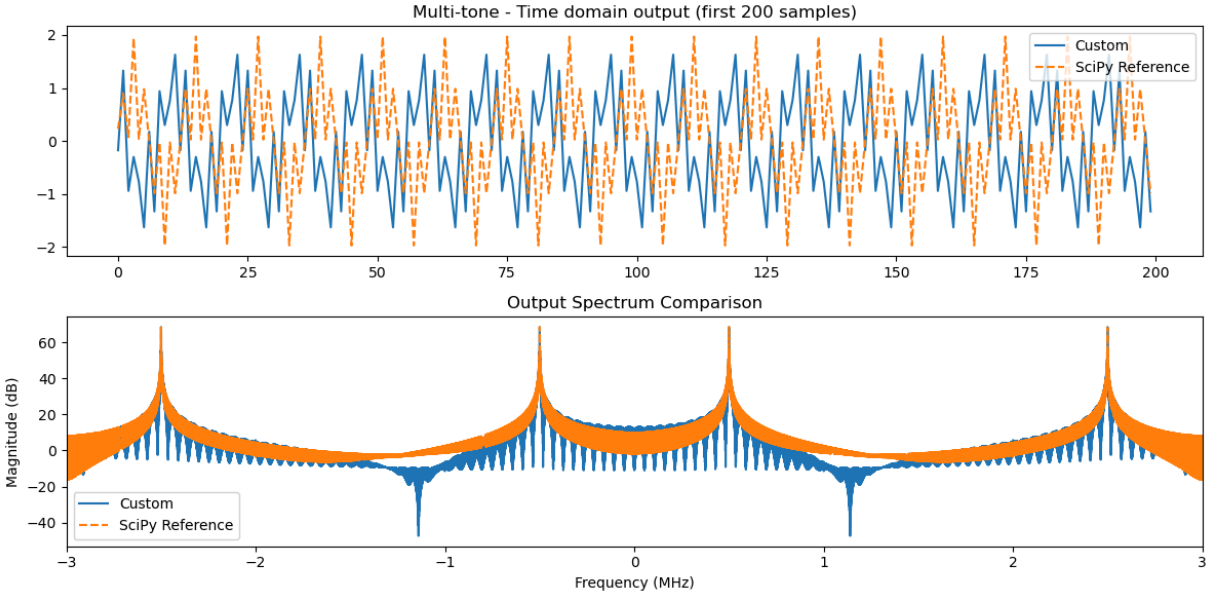


Two tones (1 MHz + 4 MHz): MSE vs. SciPy = 1.17e+00



Wideband noise: MSE vs. SciPy = 1.25e+00

```
Multi-tone: MSE vs. SciPy = 1.65e+00
```



IIR Notch

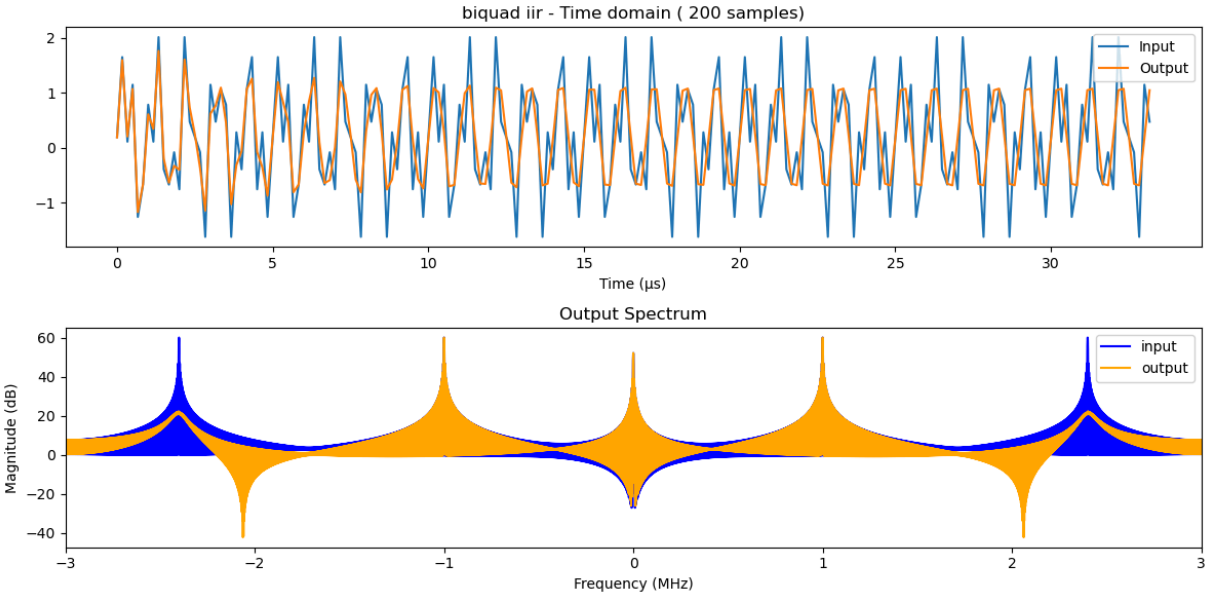
I/O + Filter Testing

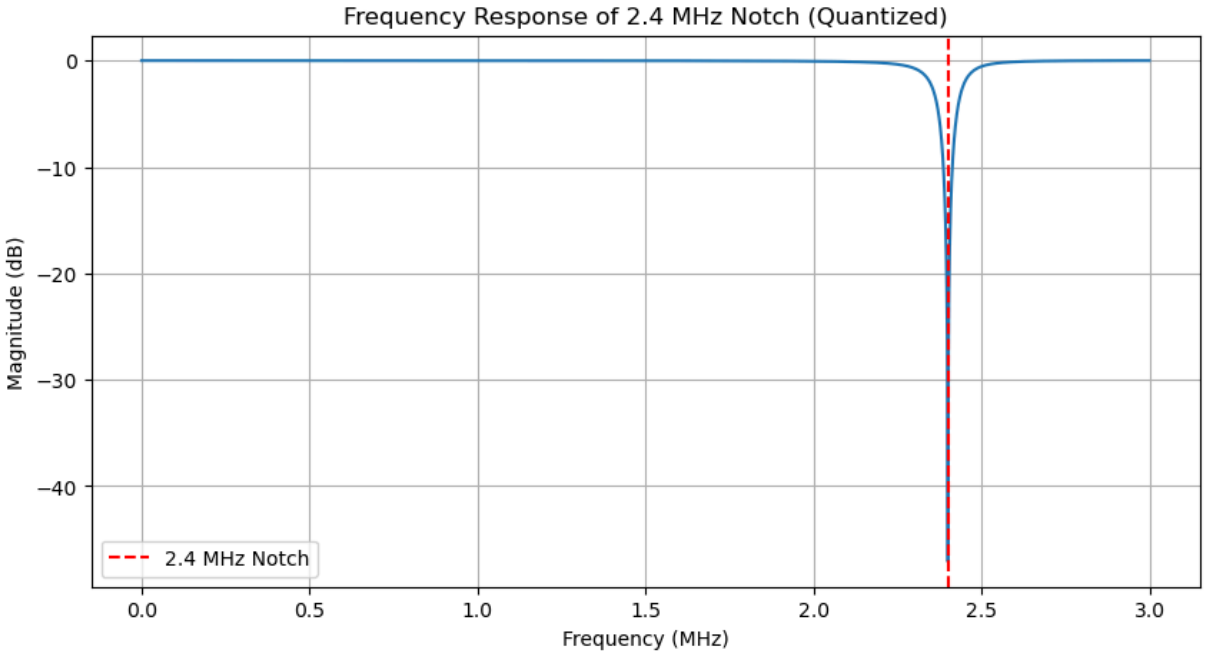
In [123…

```
b_q14 = np.loadtxt("notch_b_q14.txt", dtype=int)
a_q14 = np.loadtxt("notch_a_q14.txt", dtype=int)

frac_bits = 14
b_fixed = b_q14 / (2**frac_bits)
a_fixed = a_q14 / (2**frac_bits)

print("Loaded Q1.14 coefficients:")
print("b_q14 =", b_q14)
print("a_q14 =", a_q14)
print("b_fixed =", b_fixed)
print("a_fixed =", a_fixed)

# ================================================================
# 2. Generate test signal (1 MHz + small DC)
# ================================================================

fs = 6e6
t_2 = np.arange(0, 2048) / fs
# t = t* fs_in / fs
x = np.sin(2*np.pi*1e6*t_2) + 0.2*np.ones_like(t_2) + np.sin(2*np.pi*2.4*1e6


# ================================================================
# 4. Run both filters
# ================================================================

y = biquad_df2t(x, b_fixed, a_fixed)
```

```python
plot_time_freq(x, y, fs, fs, "biquad iir",samples=200,L=1,M=1)
# ================================================================
# 6. Frequency response
# ================================================================

w, h = freqz(b_fixed, a_fixed, worN=4096, fs=fs)
plt.figure(figsize=(10,5))
plt.plot(w/1e6, 20*np.log10(np.abs(h)))
plt.axvline(2.4, color='r', linestyle='--', label='2.4 MHz Notch')
plt.title('Frequency Response of 2.4 MHz Notch (Quantized)')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.legend()
plt.show()
```

```
Loaded Q1.14 coefficients:
b_q14 = [15725 25443 15725]
a_q14 = [16384 25443 15066]
b_fixed = [0.95977783 1.55291748 0.95977783]
a_fixed = [1.         1.55291748 0.91955566]
```



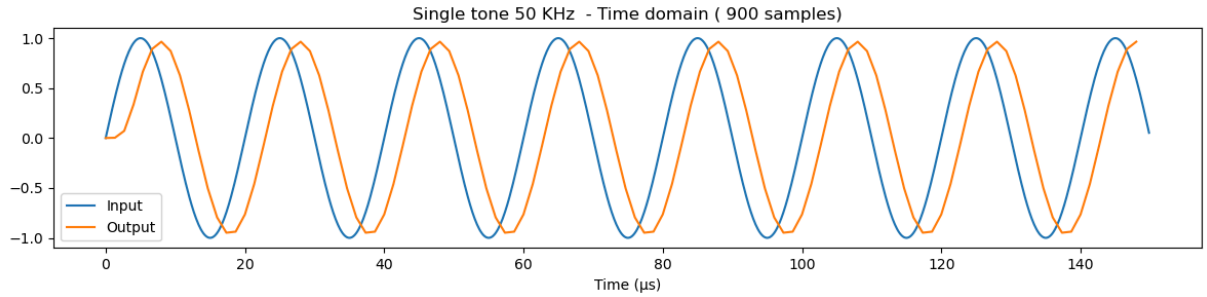


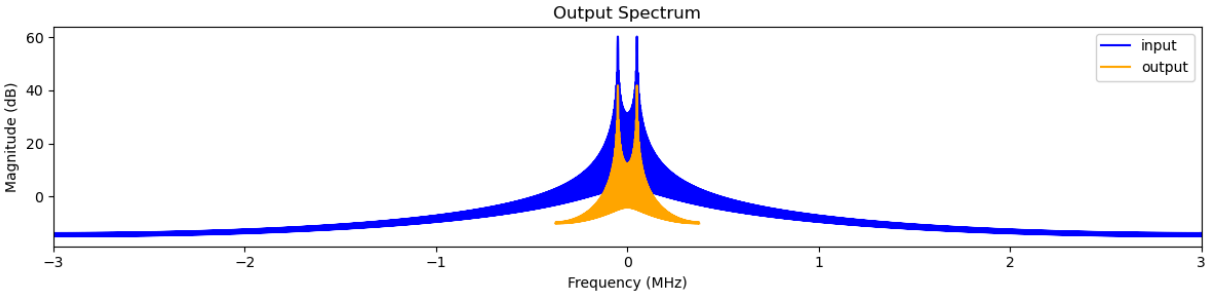
CIC + Compensation FIR

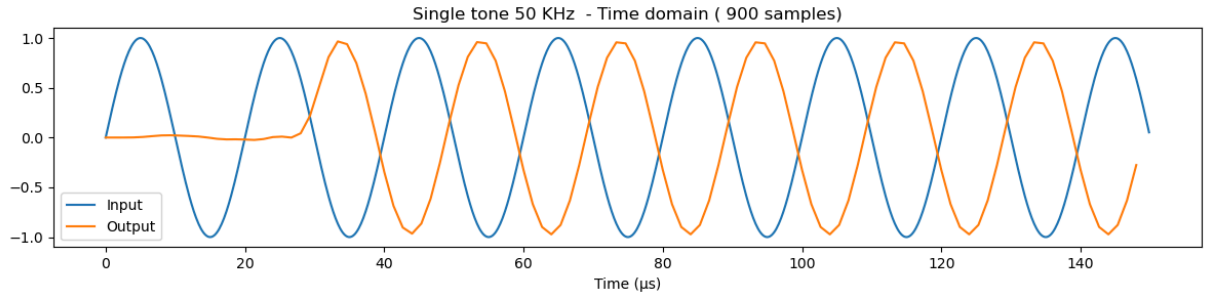
I/O Testing

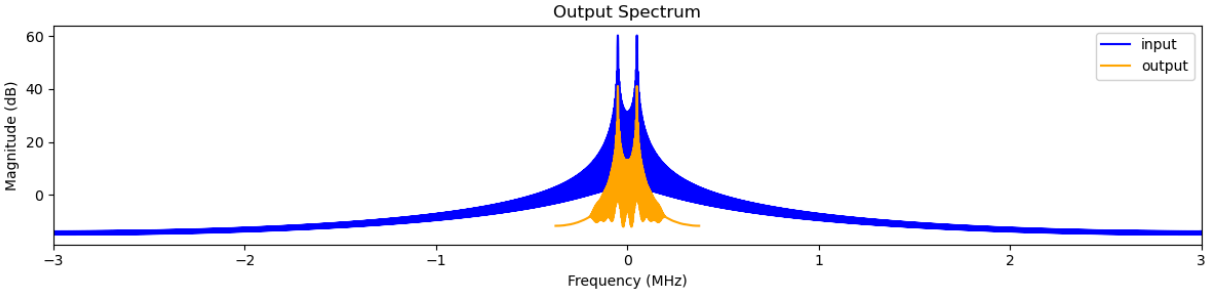
In [124…

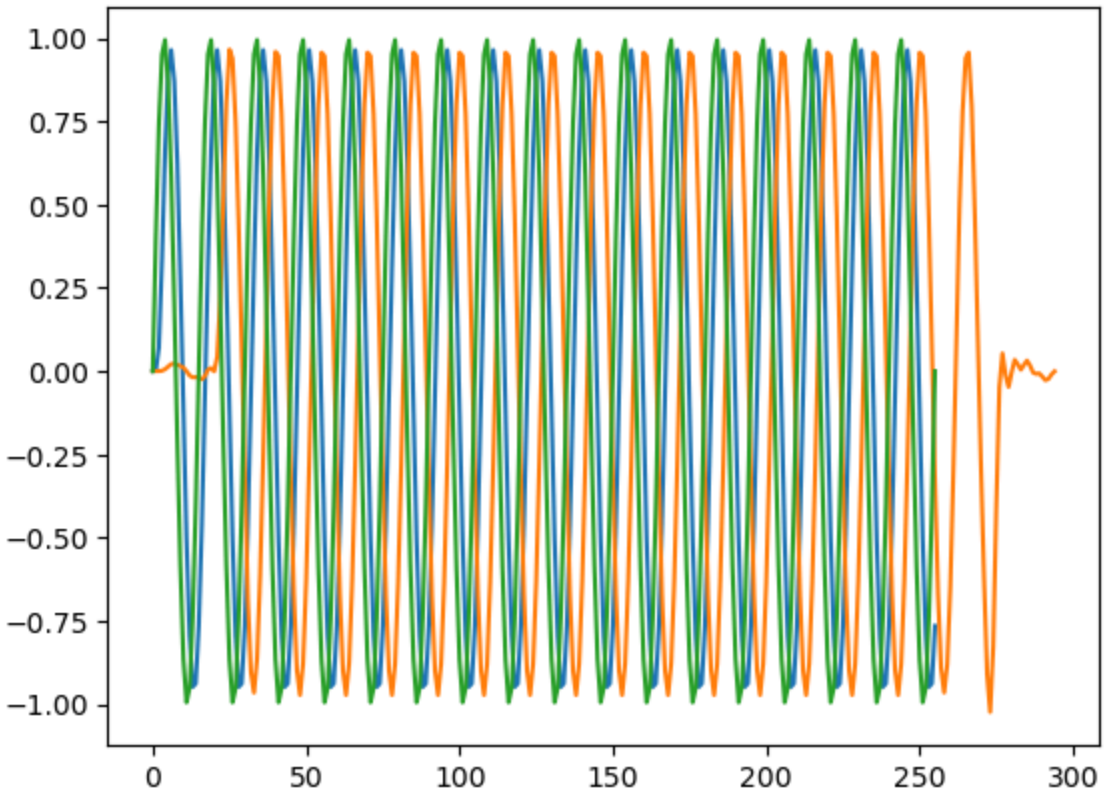
```
# ----------------- TESTING -----------------
fs_cic = 6e6
R = 8
# Different input signals
signals = {
    "Single tone 50 KHz ": np.sin(2*np.pi*5e4*t_2),
#     "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
#     "Two tones (1 MHz + 4 MHz)": np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.p
    #  "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e4*t_2) + np.sin(2*np.pi*2.5e4*t_2) + r
}

# Run tests
for name, sig in signals.items():
    y_1= cic_decimator(sig,R)
    y_2= cic_comp(sig,8)
    plot_time_freq(sig, y_1, fs_cic, fs_cic /R, name,900 ,1  , R  , 900)
    plot_time_freq(sig, y_2, fs_cic, fs_cic /R, name,900 , 1 , R  , 900)
    plt.plot(y_1,label="no comp")
    plt.plot(y_2,label=" comp")
    plt.plot(sig[::R],label="original")
input_cic_1 = (np.sin(2*np.pi*0.5e4*t) + np.sin(2*np.pi*2.5e4*t) + np.sin(2*
bypassed = cic_comp(input_cic_1,1)
plot_time_freq(input_cic_1, bypassed,fs_cic,fs_cic,"bypassed", 900,1,1,900)
plt.plot(bypassed)
plt.plot(input_cic_1* 1.08)
```

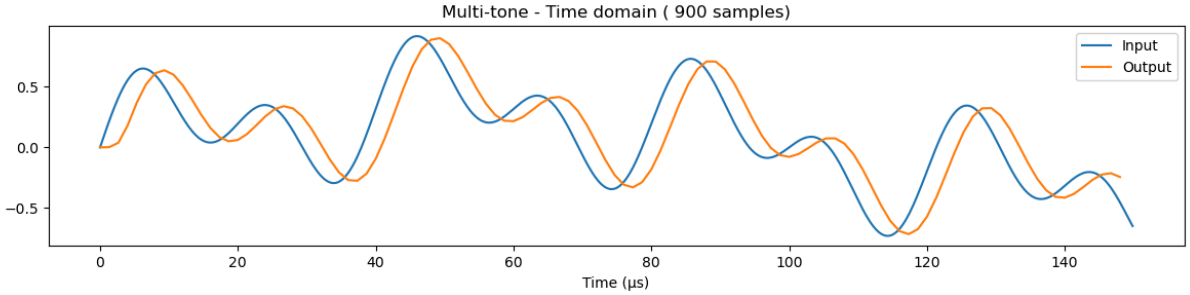

Single tone 50 KHz - Time domain (900 samples)
Input
Output
Time (µs)

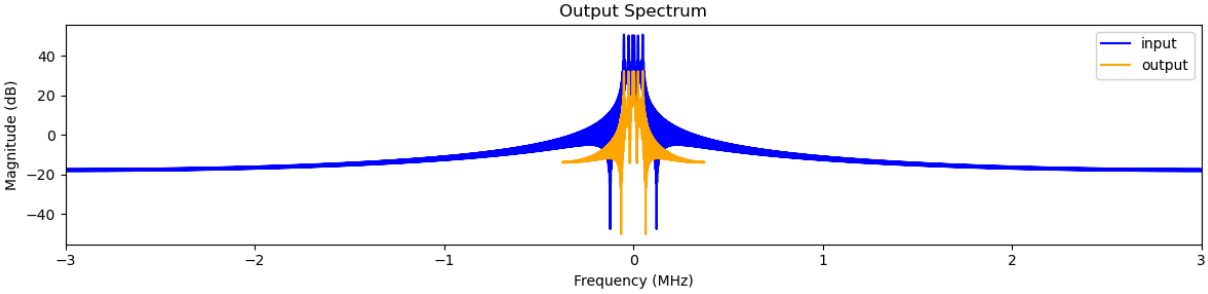

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

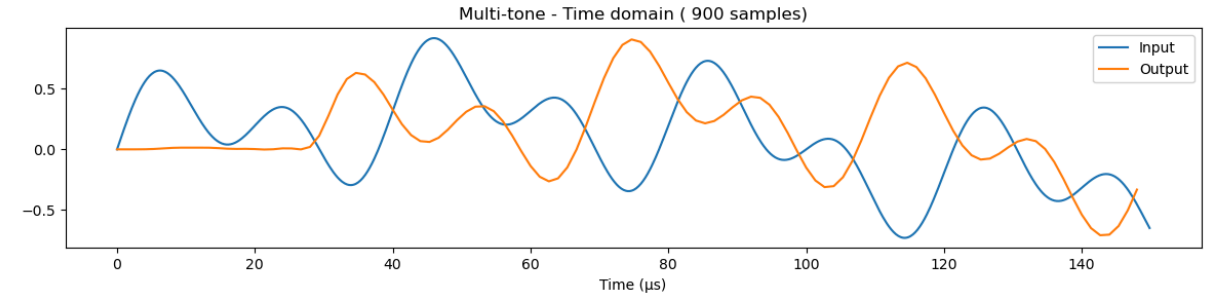

Single tone 50 KHz - Time domain (900 samples)
Input
Output
Time (µs)

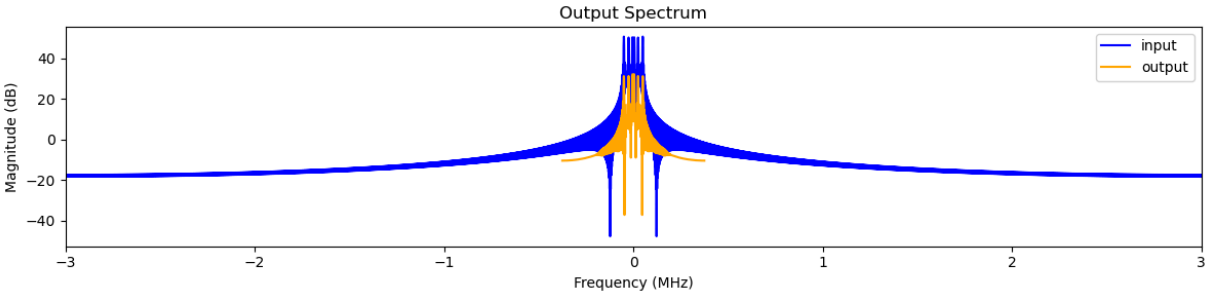

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

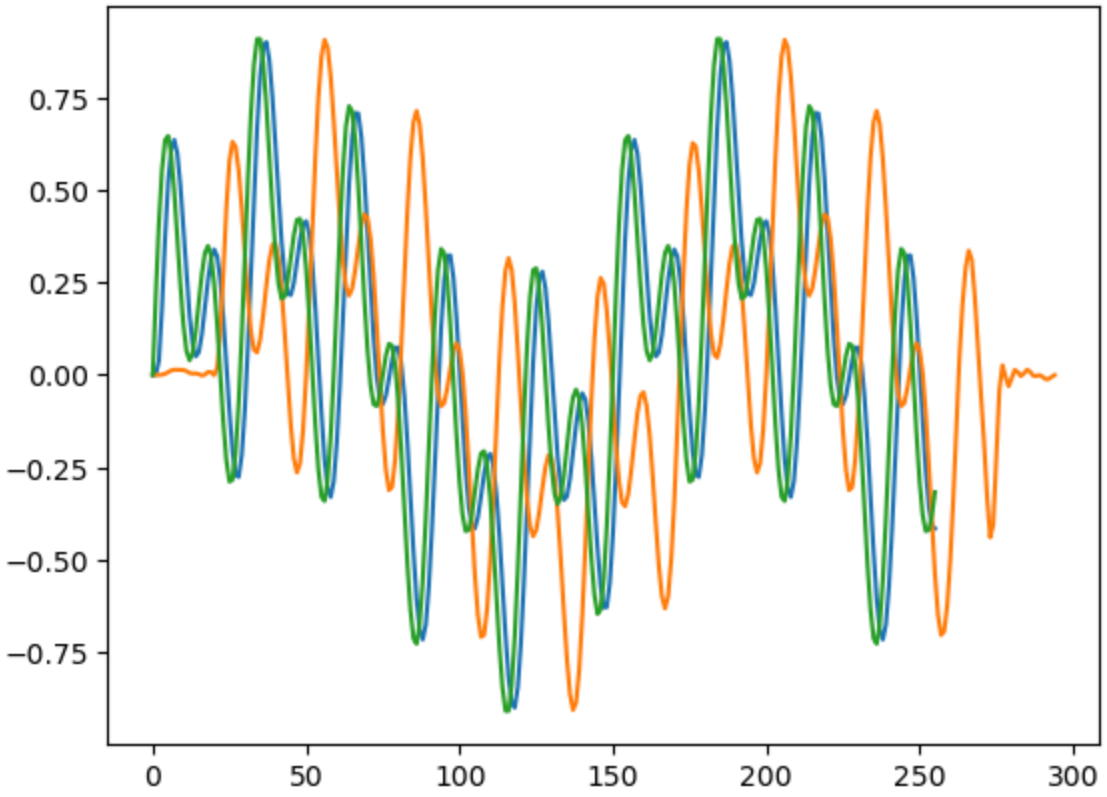

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
50
100
150
200
250
300

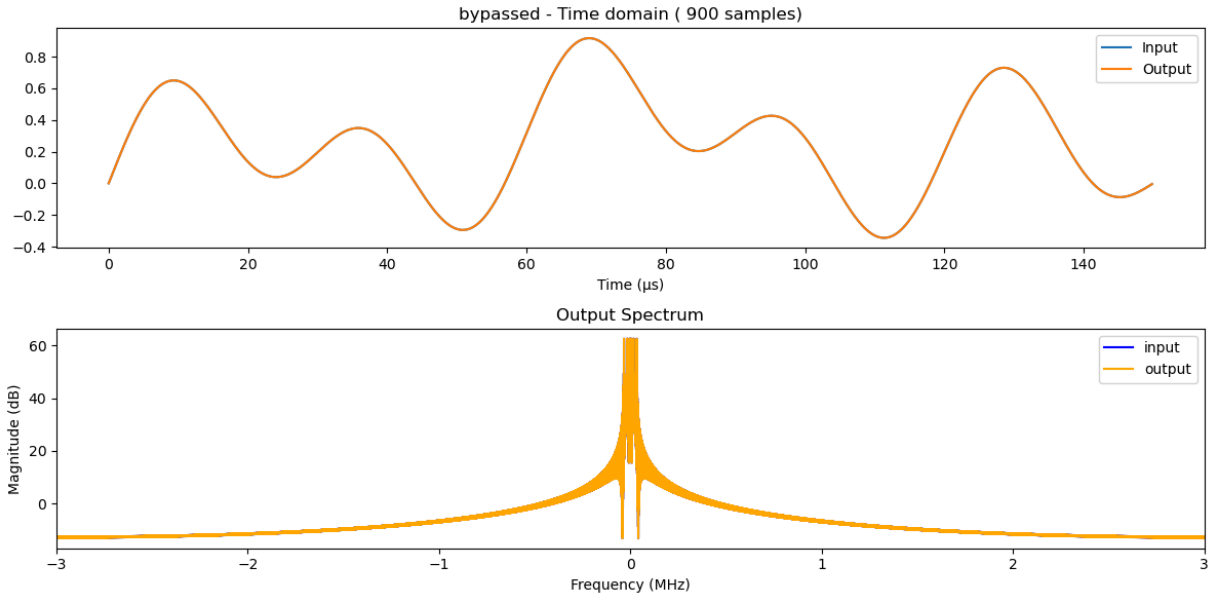

Multi-tone - Time domain (900 samples)
Input
Output
0.5
0.0
−0.5
0
20
40
60
80
100
120
140
Time (µs)


Output Spectrum
input
output
40
20
0
−20
−40
Magnitude (dB)
−3
−2
−1
0
1
2
3
Frequency (MHz)

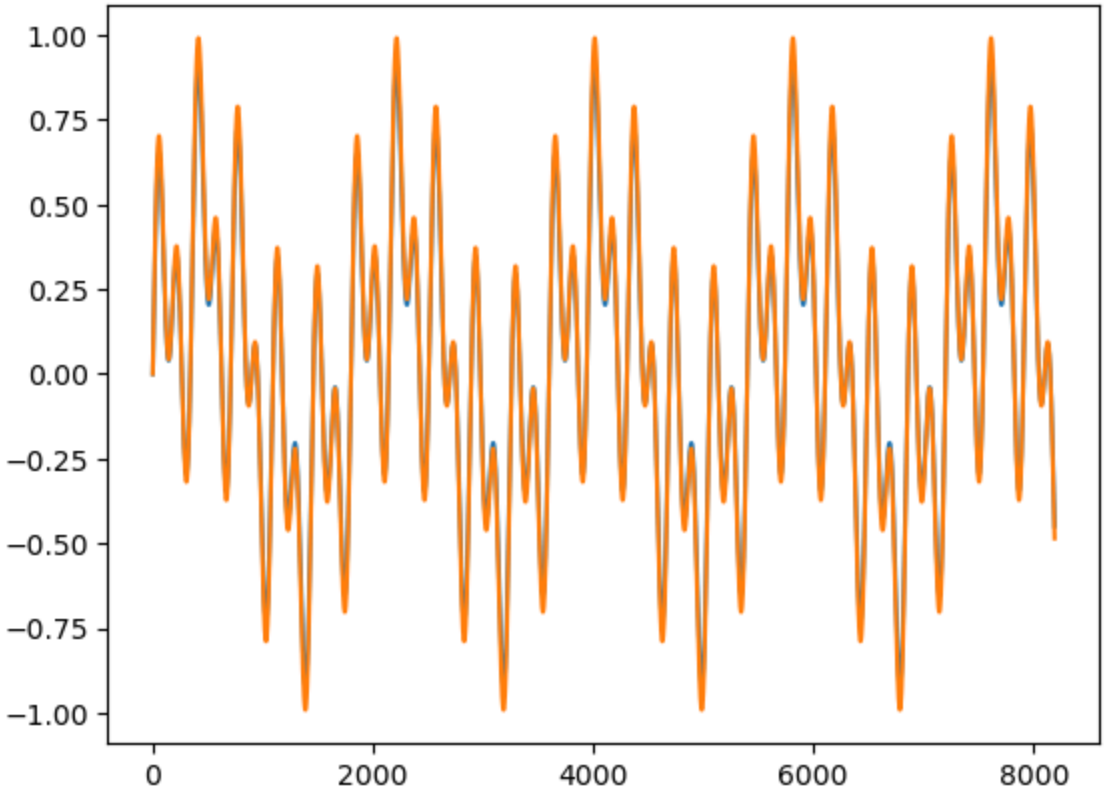

Multi-tone - Time domain (900 samples)
Input
Output
0.5
0.0
−0.5
0
20
40
60
80
100
120
140
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
40
20
0
−20
−40
−3
−2
−1
0
1
2
3
Frequency (MHz)


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
50
100
150
200
250
300



Out[124… [<matplotlib.lines.Line2D at 0x1641e4f10>]



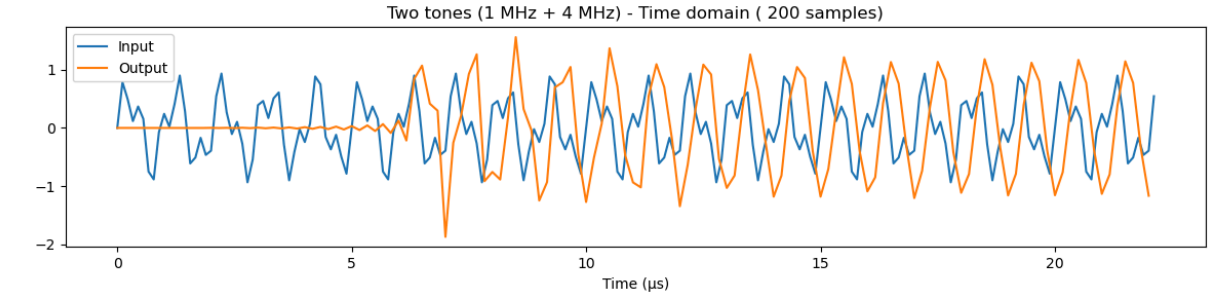
DFE (integrated)

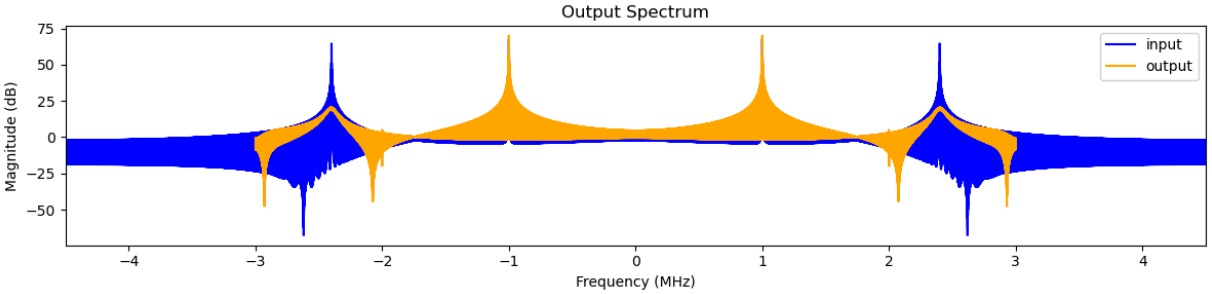
I/O Testing

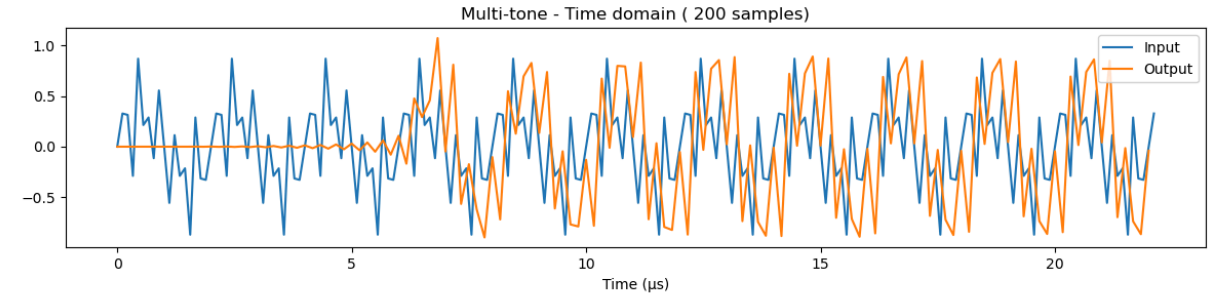
note: that when we use resampler be it interpolator or decimator the frequency DFT of the output is scaled be L/M of that resampling ratio so when we decimate by 4 it will be 1/4 of the input spectrum

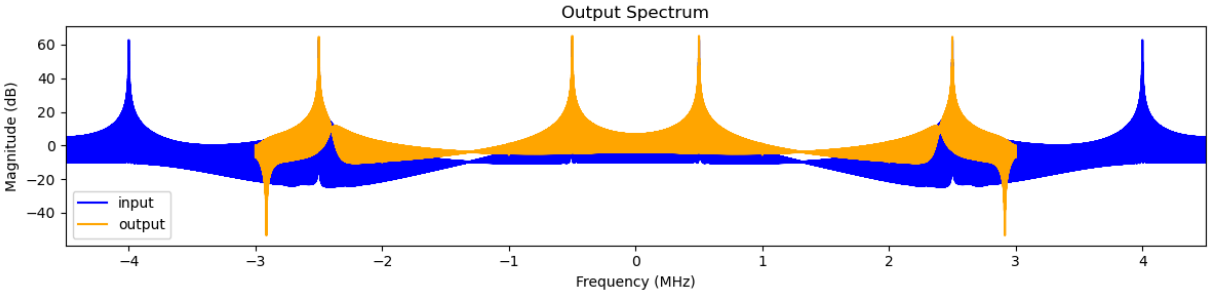
In [125…

```
fs_in = 9e6
fs_out = 6e6
R = 1
# Different input signals
signals = {
    # "Single tone 50 KHz ": np.sin(2*np.pi*5e4*t),
#     "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": (np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi
    #  "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.si
}

# Run tests
for name, sig in signals.items():
    y = dfe_integrated(sig ,R)
    plot_time_freq(sig, y*2, fs_in, fs_out /R, name,200 , 2 , M*R  , 200)
```





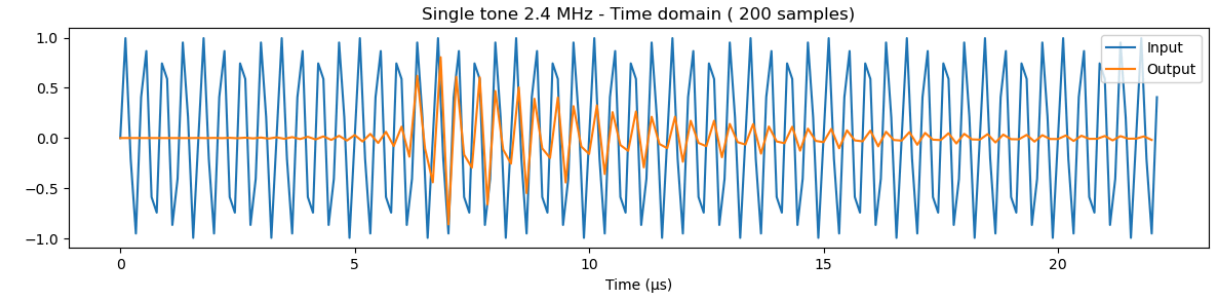


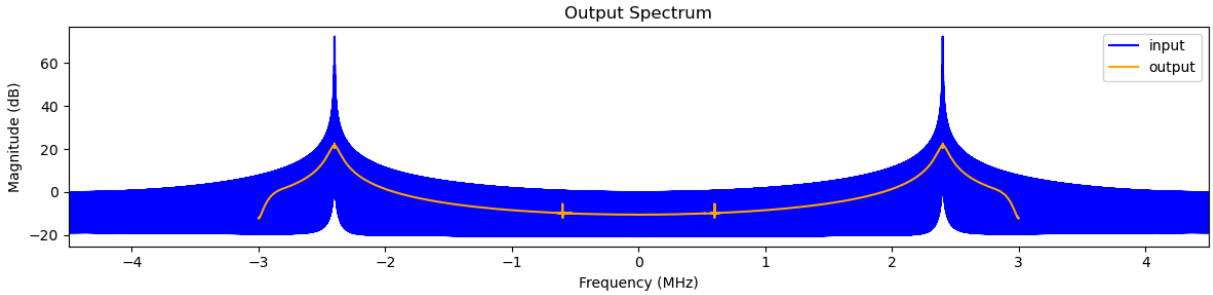


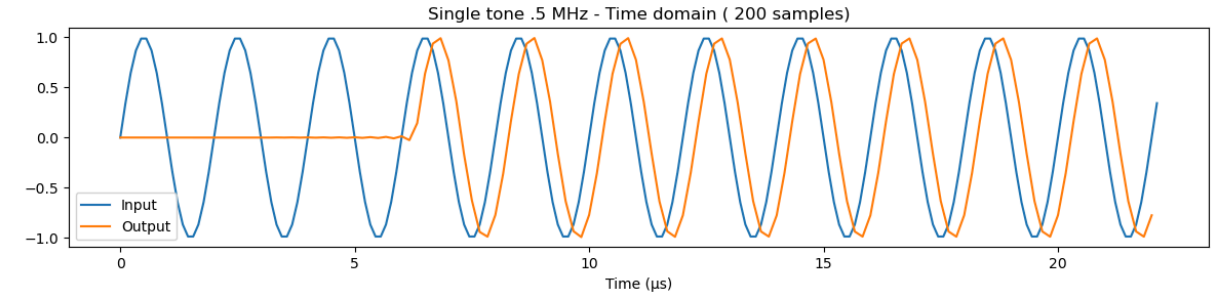
In [126…

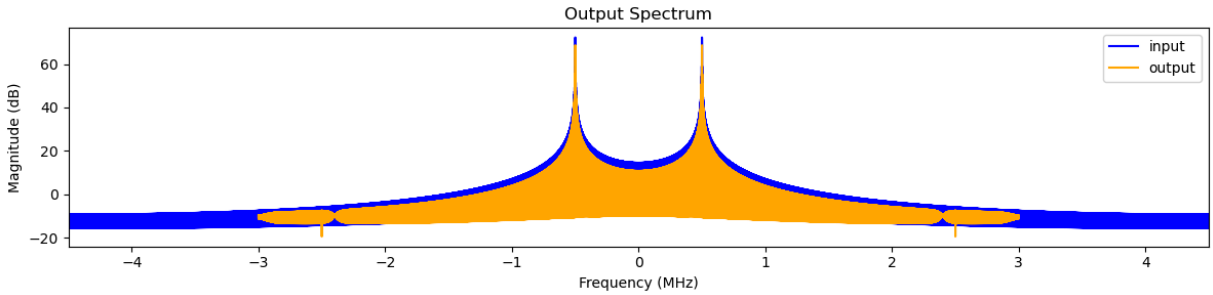
```
# ----------------- TESTING -----------------
```

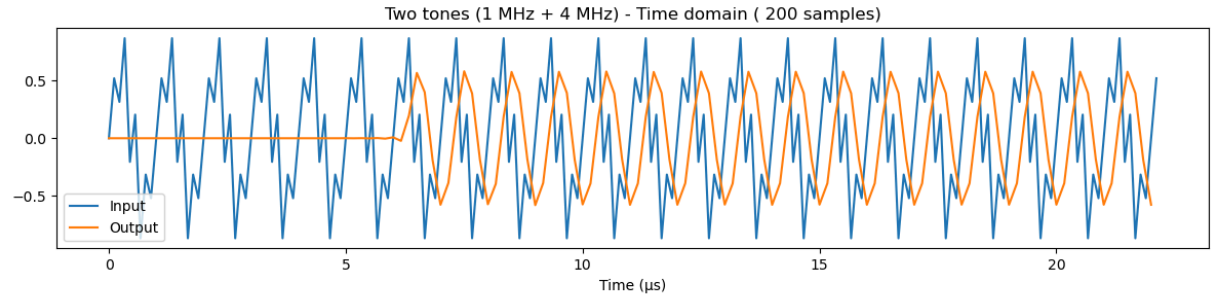
```
# Different input signals
signals = {
    "Single tone 2.4 MHz": np.sin(2*np.pi*2.4e6*t),
    "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": (np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi
    # "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.si
}

# Run tests
for name, sig in signals.items():
    y = dfe_integrated(sig, 1)
    plot_time_freq(sig, y, fs_in, fs_out, name,200,L,M)



signals_2 = {
    "Single tone 2.4 KHz": np.sin(2*np.pi*2.4e3*t),
    # "Single tone .5 MHz": np.sin(2*np.pi*2e4*t),
    "Two tones (10 khz + 40 khz)": (np.sin(2*np.pi*1e4*t) + 0.7*np.sin(2*np.
    # "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e4*t) + np.sin(2*np.pi*2.5e4*t) + np.si
}

print("stop_2 low freq sig at R = 2")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 2)
    plot_time_freq(sig, y, fs_in, fs_out/2, name+"at R = 2",20000,L,M*2,2000

print("stop_3 low freq sig at R = 4")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 4)
    plot_time_freq(sig, y, fs_in, fs_out/4, name+"at R = 4",20000,L,M*4,2000

print("stop_4 low freq sig at R = 8")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 8)
    plot_time_freq(sig, y, fs_in, fs_out/8, name+"at R = 8",20000,L,M*8,2000

print("stop_5 low freq sig at R = 16")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 16)
    plot_time_freq(sig, y, fs_in, fs_out/16, name+"at R = 16",20000,L,M*16,2
```

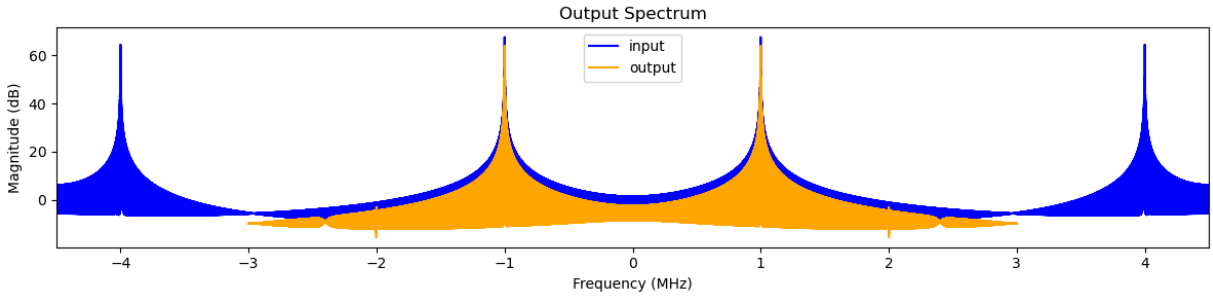

Single tone 2.4 MHz - Time domain (200 samples)
Input
Output
Time (µs)

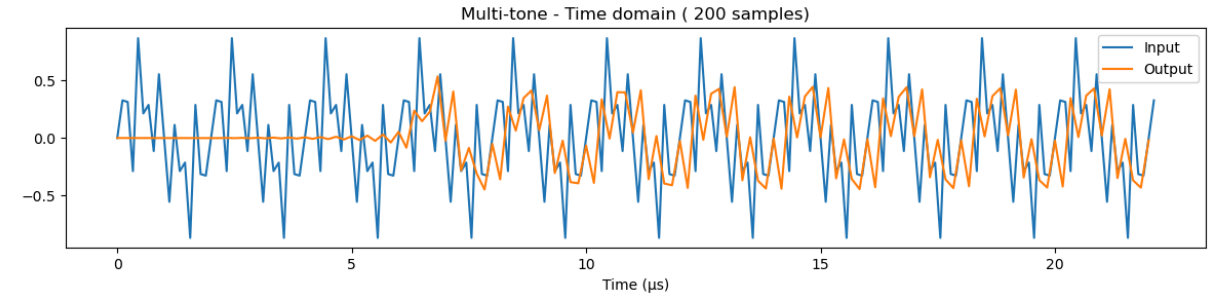

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

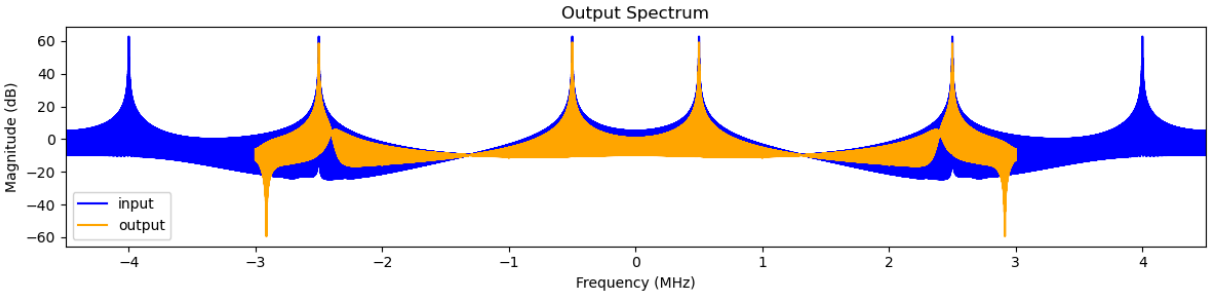

Single tone .5 MHz - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

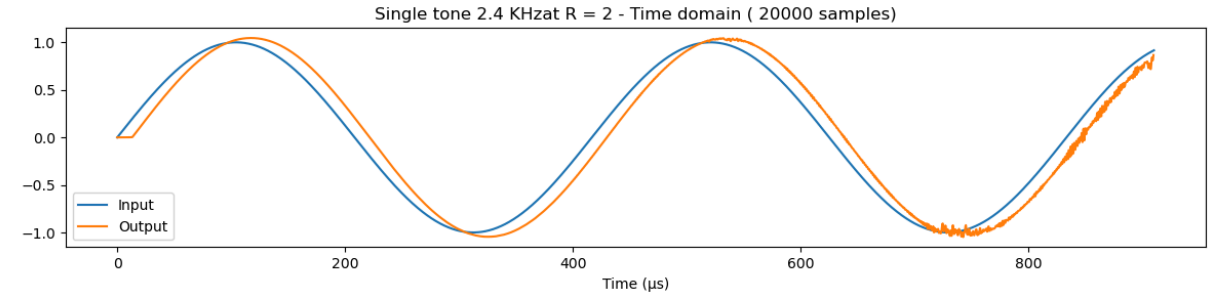

Two tones (1 MHz + 4 MHz) - Time domain (200 samples)
Input
Output
Time (µs)

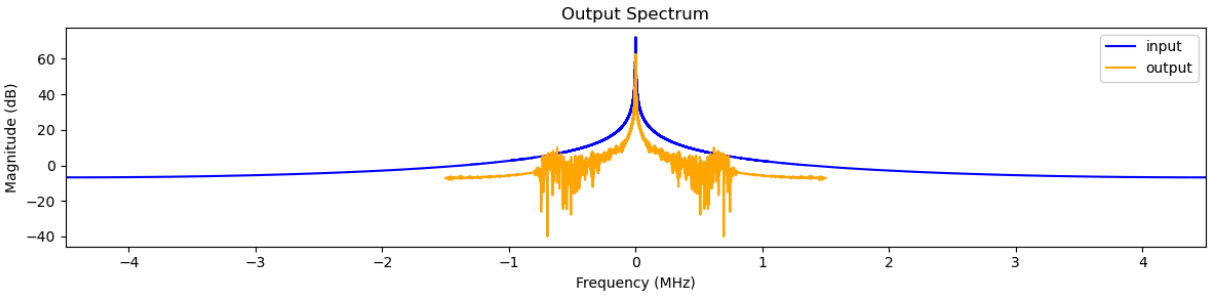

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

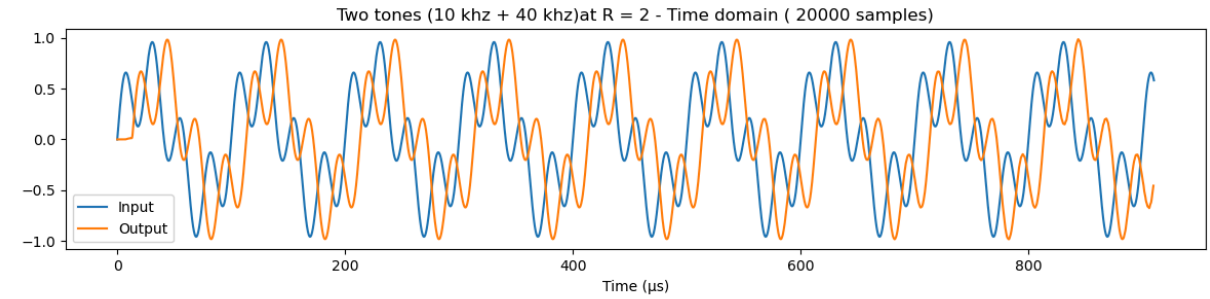


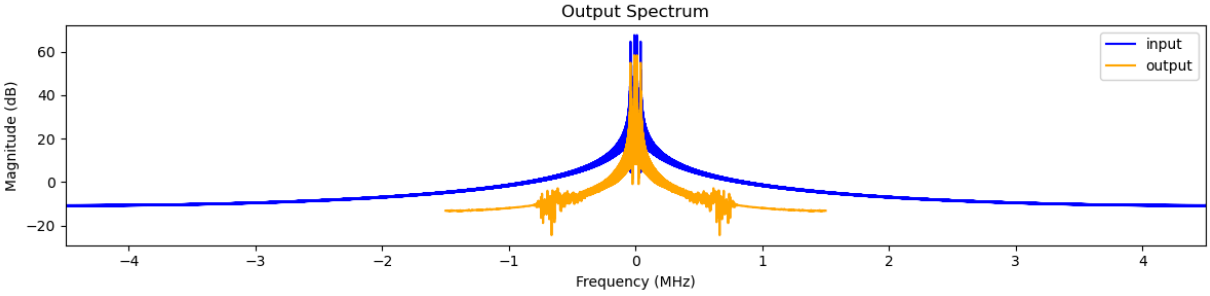


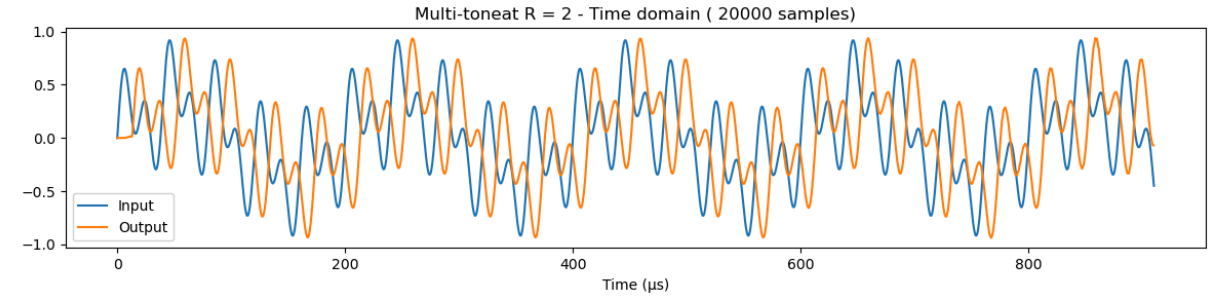
stop_2 low freq sig at R = 2

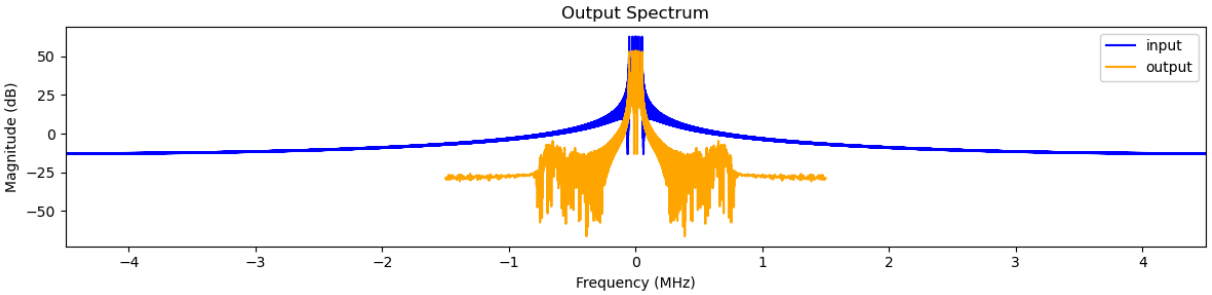




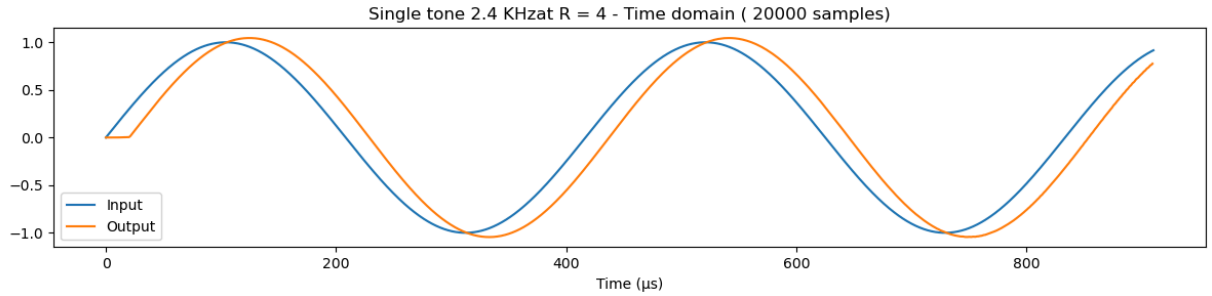


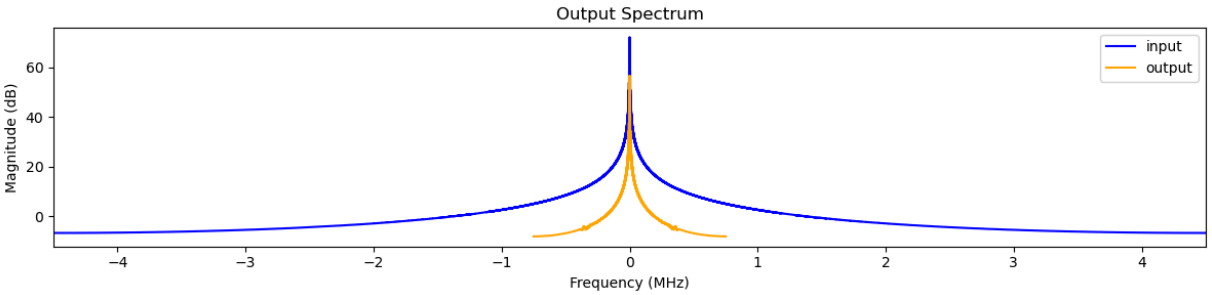


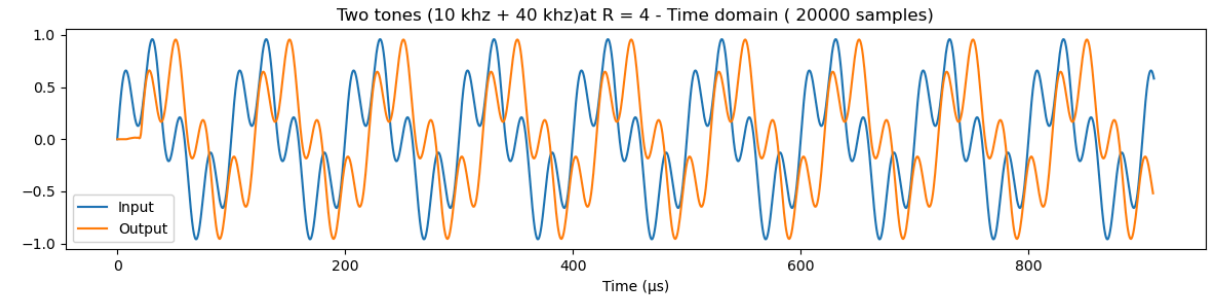


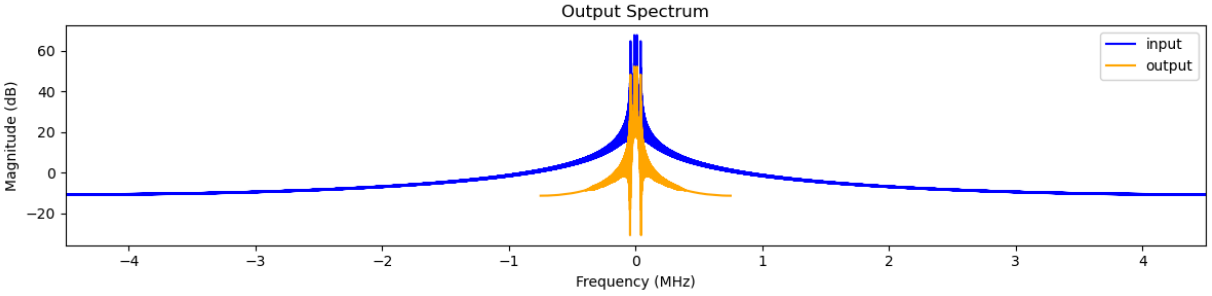


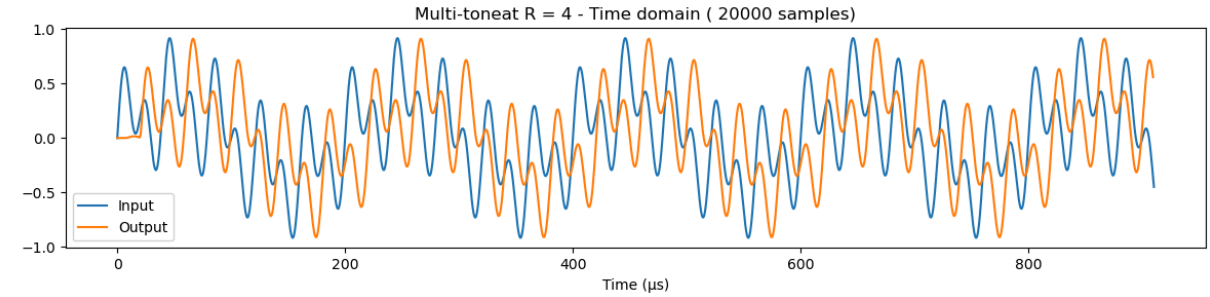
stop_3 low freq sig at R = 4

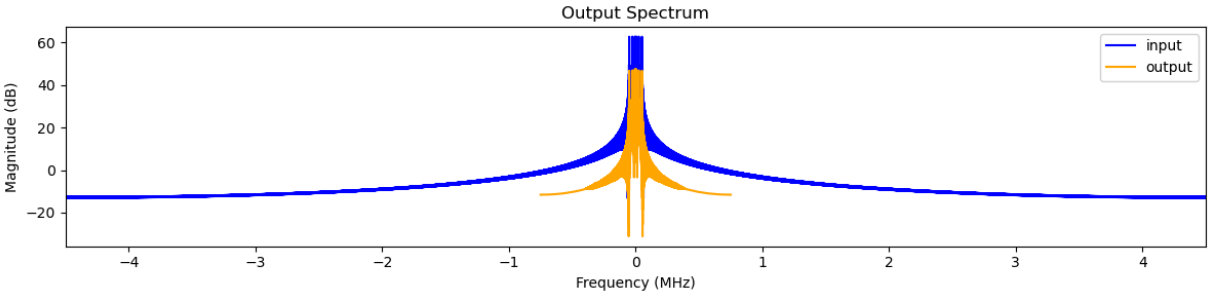




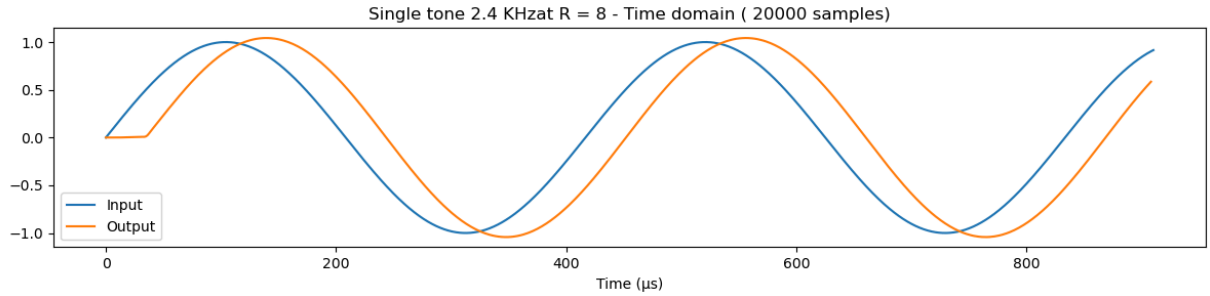


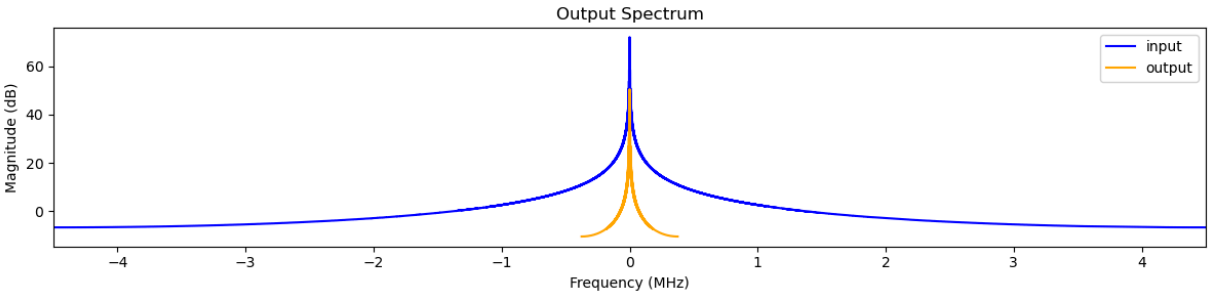


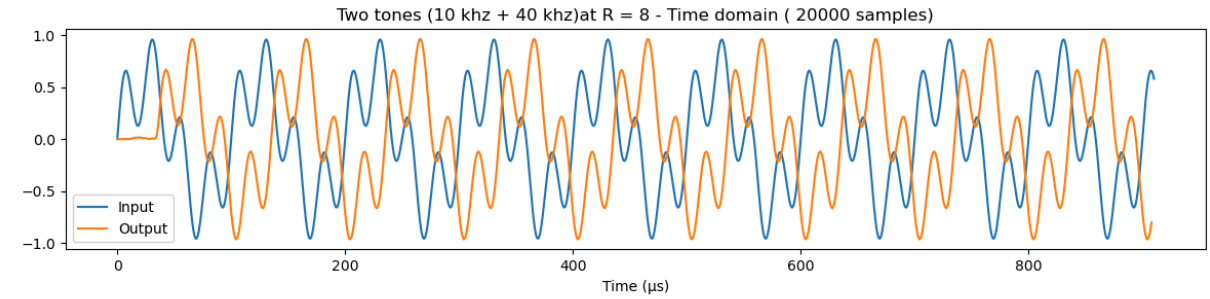


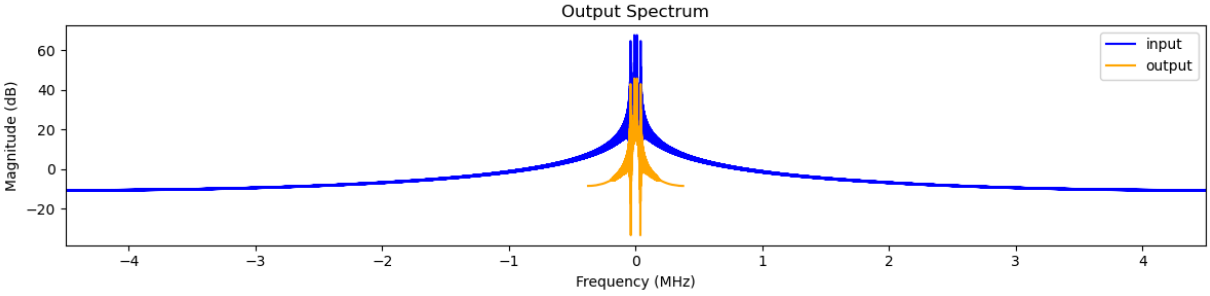


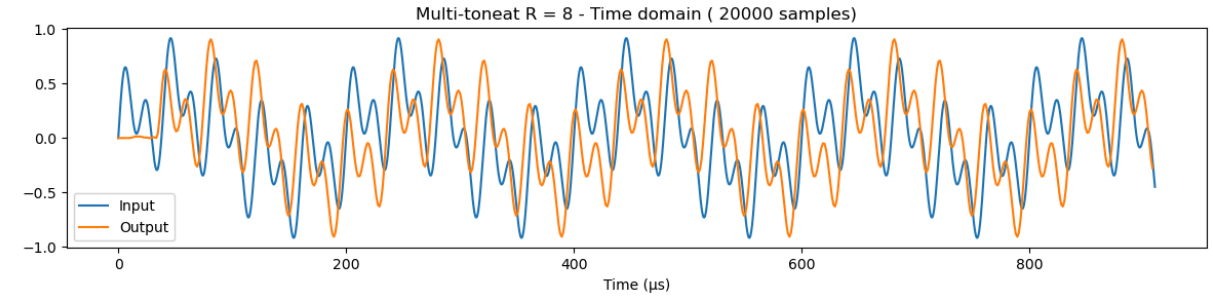
stop_4 low freq sig at R = 8

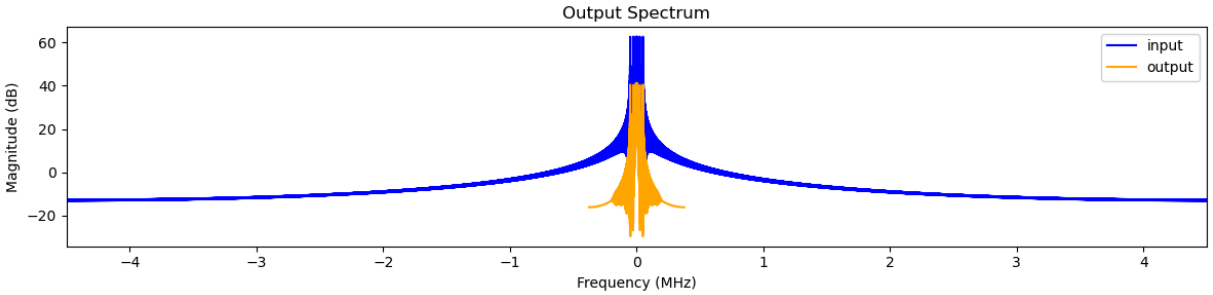




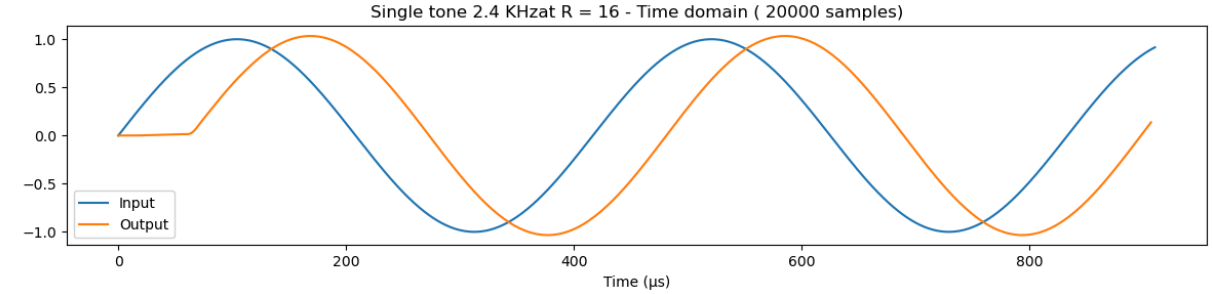


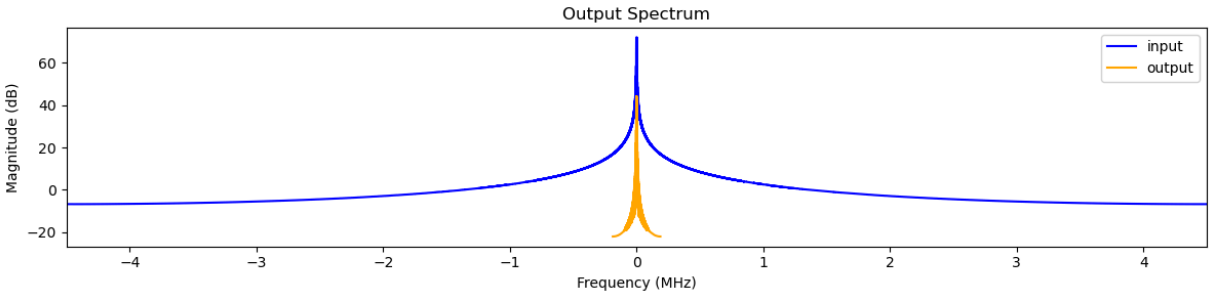


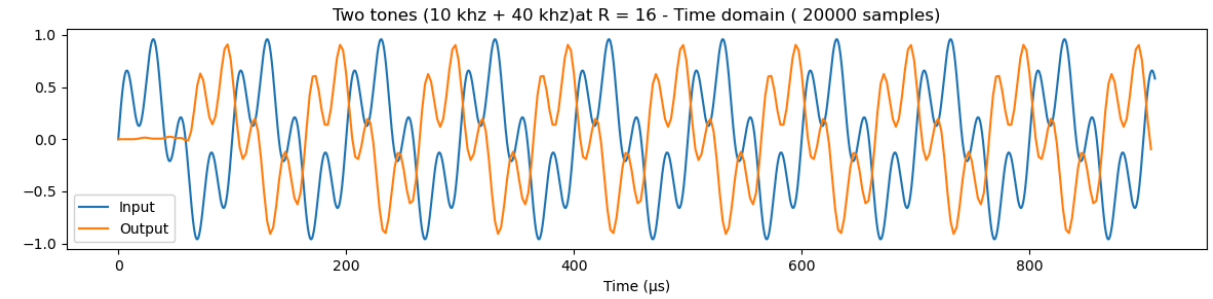


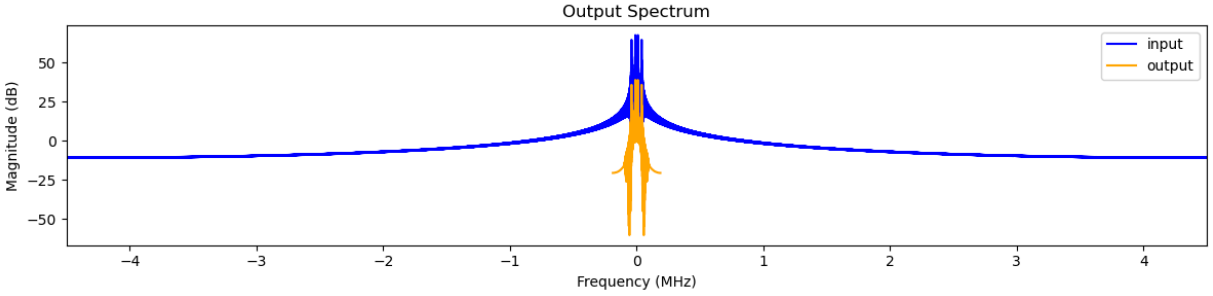


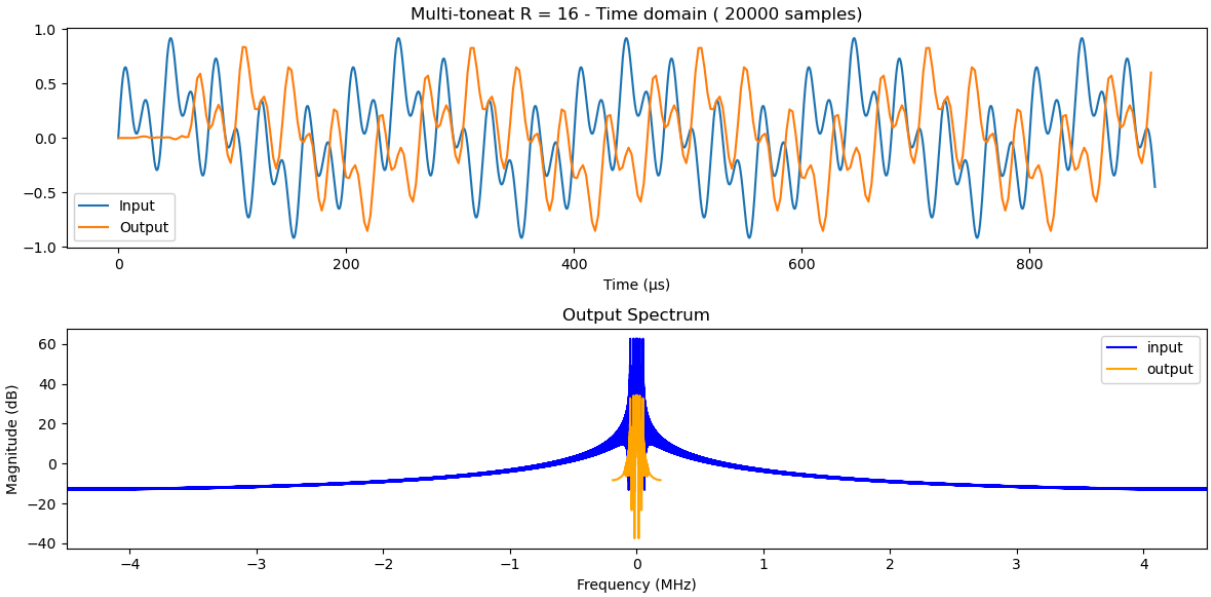
stop_5 low freq sig at R = 16











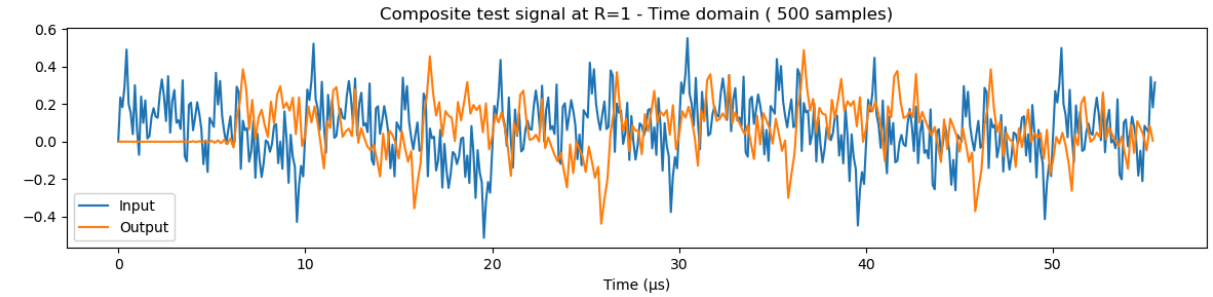
exhastive testbench

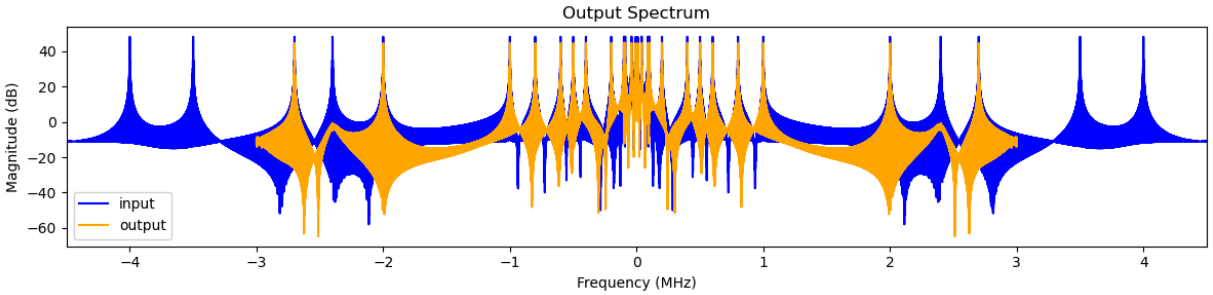
In [127…

```
# ----------------- TESTING: Composite signal -----------------

# Frequencies to test (Hz)
# - Low freq: below final output
# - High freq: near CIC nulls / aliasing points
test_freqs = [
    2.4e3,    # low
    1e4, 4e4, 9e4 ,# low
    1e5, 2e5, 4e5, 5e5, 6e5, 8e5,   # mid
    1e6, 2.4e6, 2.7e6, 3.5e6, 5e6 , 7e6 # high / near CIC attenuation points
]

# Generate single composite signal
sig = sum(np.sin(2*np.pi*f*t) for f in test_freqs) / len(test_freqs)

# Decimation factors to test
R_list = [1, 2, 4, 8, 16]

# Expected behavior dictionary for annotation
expected_behavior = {
    1: "No decimation, all frequencies above 3 MHz and at 2.4 MHz attenuated
    2: "Frequencies above 1.5 MHz attenuated by CIC, others pass",
    4: "Frequencies above 0.75 MHz attenuated, aliasing may occur",
    8: "Frequencies above 0.375 MHz attenuated, strong aliasing for >0.75 MH
    16:"Frequencies above 0.1875 MHz strongly attenuated, only very low freq
}

# Loop over R
for R in R_list:
    print(f"\n--- Decimation R = {R} ---")
    print("Expected:", expected_behavior[R])

    y = dfe_integrated(sig, R)
```

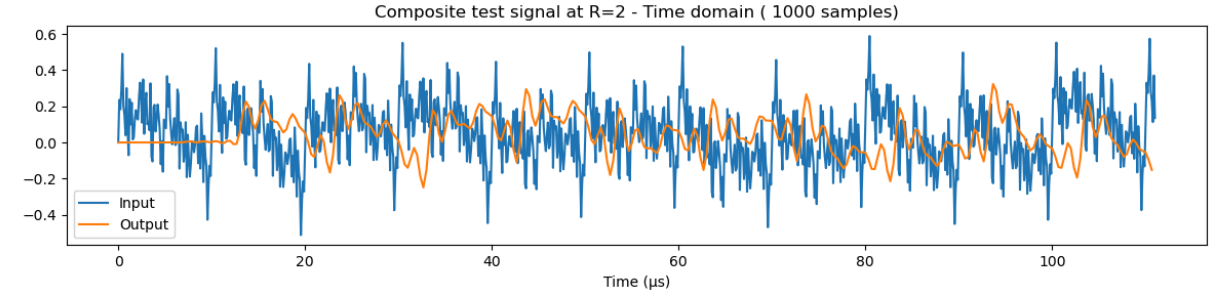
```
    fs_out_R = 6e6 / R
    plot_time_freq(sig, y, 9e6, fs_out_R, f"Composite test signal at R={R}",
```

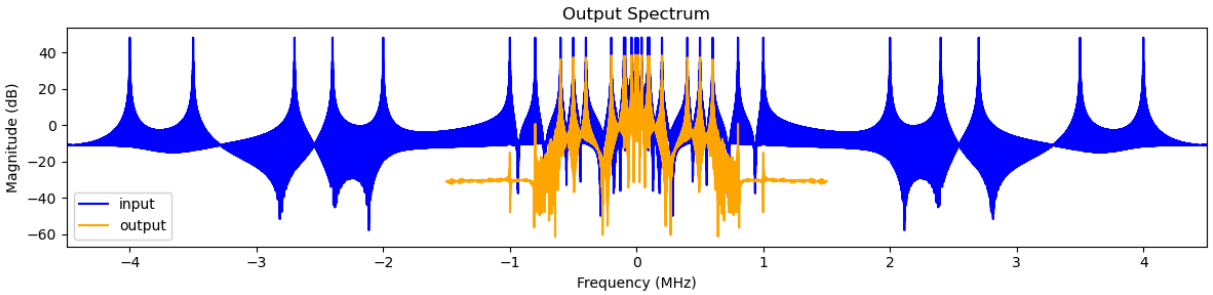
```
--- Decimation R = 1 ---
Expected: No decimation, all frequencies above 3 MHz and at 2.4 MHz attenuat
ed by polyphase resampler pass
```



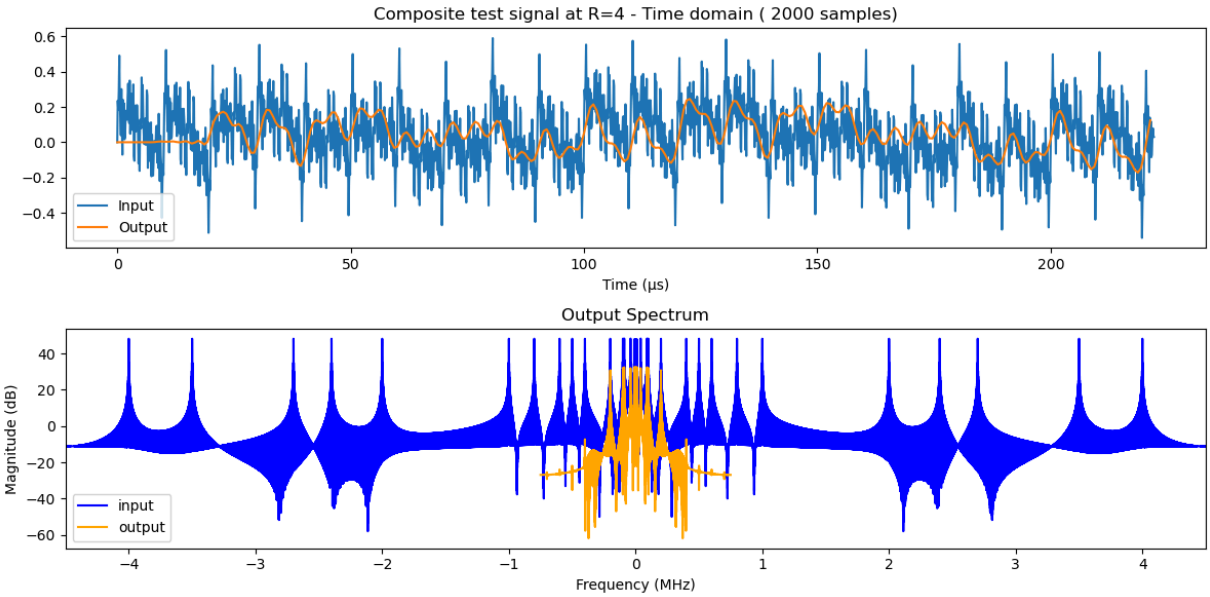



```
--- Decimation R = 2 ---
Expected: Frequencies above 1.5 MHz attenuated by CIC, others pass
```

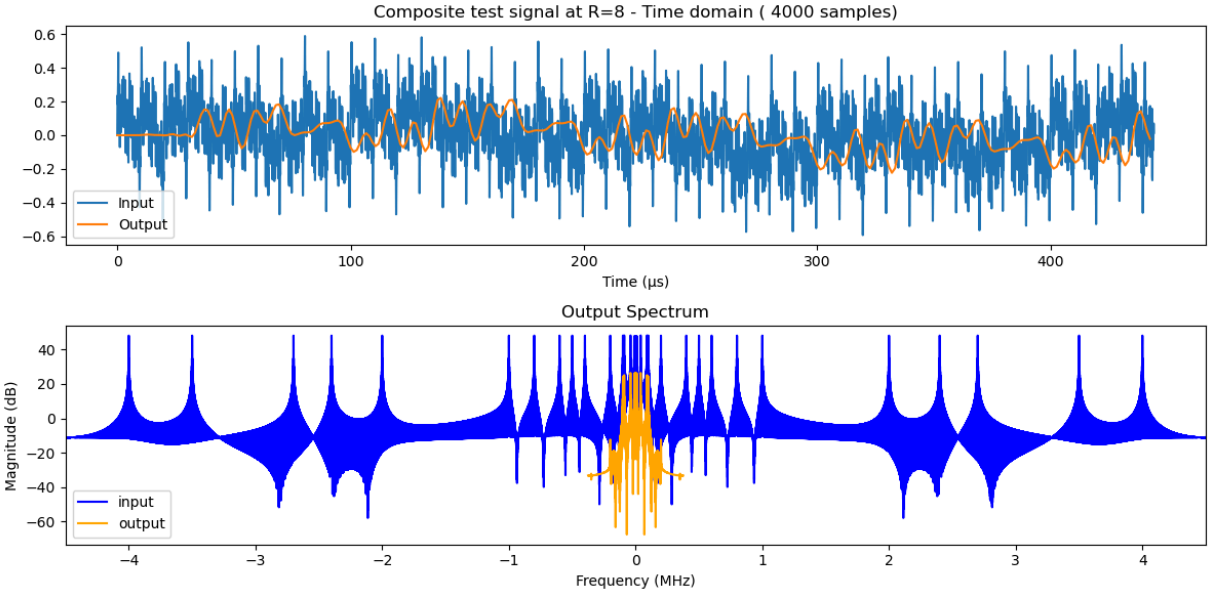




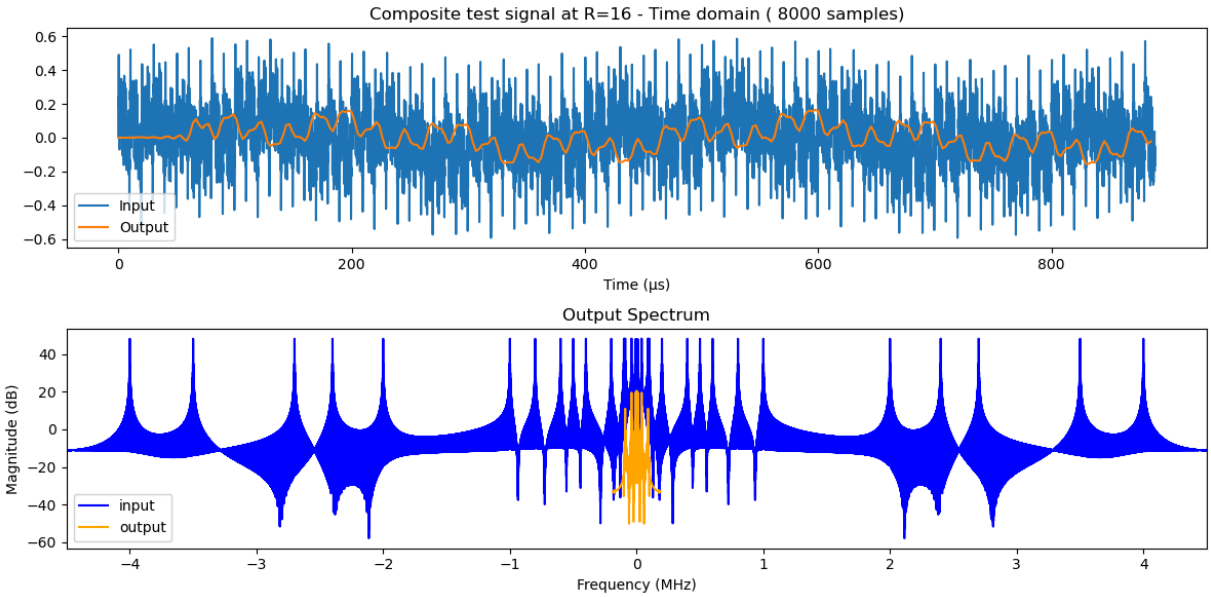

```
--- Decimation R = 4 ---
Expected: Frequencies above 0.75 MHz attenuated, aliasing may occur
```



--- Decimation R = 8 ---
Expected: Frequencies above 0.375 MHz attenuated, strong aliasing for >0.75 MHz



--- Decimation R = 16 ---
Expected: Frequencies above 0.1875 MHz strongly attenuated, only very low frequencies survive


Composite test signal at R=16 - Time domain (8000 samples)
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Input
Output
0
200
400
600
800
Time (µs)

Output Spectrum
40
20
0
−20
−40
−60
Magnitude (dB)
input
output
−4
−3
−2
−1
0
1
2
3
4
Frequency (MHz)

RTL & TESTBENCH

by: Hazem Yasser Mahmoud Mohamed

Mohamed Ahmed Mohamed

mohamed anwar

GROUP 7

Rational Resampler

faced a lot of issues to achieve the novel architecture proposed in the golden model
so we falled back to a more tradiotional approach

**a polyphase upsampler with 226 filter coeffs that satisfy the specs given followed by a down simpler with only 3 coeffs full pass as the full filtering happened in the UPSAMPLER

> note : proposed filter in golden model has frequency doubling effect inherent to its structure not in this filter
> so filter coeffs need to be adjusted for twice the cuttoff just by changing the mem file

> **note : not all files are mentioned in this report but there are mem files for easy exchange of filters and vcd files for visualizing the result along with vvp files and txt log files along with python script to visualize and plot in python**

> polyphase resampler can further be optimized for area and resources by sharing DSP blocks instead of making it fully parallel it is a tradeoff between delay and utilization

> the Notch takes most data delay to solve timing issues it was pipelined it went from -5~6 ns to less than 1 ns setup violation
> because it is DFII iir filter it has FB so only piplined to two sections so no
> further pipelining possible but still the DFE can be ran at 50 MHz so it is not a big deal

> to fix input ouput wire delay we register I/O ports and modify DFE_top

RTL

polyphase_filter

```
`timescale 1ns / 1ps
```

```
module polyphase_filter #(
    parameter  COEFF_FILE     = "decim_coeffs.mem", // File path for
coefficients
    parameter int DATA_WIDTH        = 16,
    parameter int COEFF_WIDTH       = 16,
    parameter int PHASES            = 8,   // Previously CONVERSION_FACTOR
    parameter int TAPS_PER_PHASE    = 16,
    parameter bit IS_DECIMATION     = 0    // 0 for Interpolation, 1 for
Decimation
)(
    input  logic                          clk,
    input  logic                          rst_n, // Added Reset
    input  logic                          valid_i,
    input  logic [DATA_WIDTH-1:0]         data_i,

    output logic                          valid_o,
    output logic [DATA_WIDTH-1:0]         data_o
);

    // Calculate gain shift based on phases (ceil(log2(phases)))
    localparam int GAIN_BITS = $clog2(PHASES);
    localparam int TOTAL_TAPS = PHASES * TAPS_PER_PHASE;

    // ----------------------------------------------------------------
    // Coefficient Memory
    // ----------------------------------------------------------------
    logic signed [COEFF_WIDTH-1:0] coeff_rom [0:TOTAL_TAPS-1];

    initial begin
        $readmemh(COEFF_FILE, coeff_rom);
    end

    // ----------------------------------------------------------------
    // Internal Signals
    // ----------------------------------------------------------------
    // Phase Counters
    int phase_counter;

    // Registers
    logic signed [DATA_WIDTH-1:0]         a_reg [0:TAPS_PER_PHASE-1]; // Input
Pipeline
    logic signed [COEFF_WIDTH-1:0]        b_reg [0:TAPS_PER_PHASE-1]; // Coeff
Latch

    // Pipeline Registers (MAC chain)
```

```
    // Size matches VHDL: CONVERSION_FACTOR * TAPS_PER_PHASE
    logic signed [2*DATA_WIDTH-1:0]     p_reg [0:TOTAL_TAPS-1];

    // Accumulator for Decimation
    logic signed [2*DATA_WIDTH-1:0]     product_sum;

    // FSM State for Interpolation
    typedef enum logic [1:0] {IDLE, GAP, PULSE} state_t;
    state_t state;

    logic active_cycle;

    // ---------------------------------------------------------------
    // Main Process
    // ---------------------------------------------------------------
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            valid_o         <= 1'b0;
            data_o          <= '0;
            phase_counter   <= 0;
            state           <= IDLE;
            product_sum     <= '0;
            active_cycle    <= 1'b0;

            // Reset Arrays
            for (int i = 0; i < TAPS_PER_PHASE; i++) a_reg[i] <= '0;
            for (int i = 0; i < TAPS_PER_PHASE; i++) b_reg[i] <= '0;
            for (int i = 0; i < TOTAL_TAPS; i++)     p_reg[i] <= '0;

        end else begin

            // Default Assignments
            valid_o <= 1'b0;
            active_cycle <= 1'b0;

            // ===========================================================
            // MODE 1: INTERPOLATION (IS_DECIMATION = 0)
            // ===========================================================
            if (!IS_DECIMATION) begin

                // --- FSM Control ---
                case (state)
                    IDLE: begin
                        phase_counter <= 0;
                        if (valid_i) begin
                            active_cycle    <= 1'b1; // Process Phase 0
```

```
immediately
                            phase_counter   <= 1;    // Next is Phase 1
                            state           <= GAP;
                        end
                    end

                    GAP: begin
                        active_cycle <= 1'b0; // Wait state
                        state        <= PULSE;
                    end

                    PULSE: begin
                        active_cycle <= 1'b1;
                        if (phase_counter == PHASES - 1) begin
                            state           <= IDLE;
                            phase_counter   <= 0;
                        end else begin
                            phase_counter   <= phase_counter + 1;
                            state           <= GAP;
                        end
                    end
                endcase

                // --- Processing Logic ---
                if (active_cycle) begin
                    valid_o <= 1'b1;

                    // Filter Structure
                    for (int m = 0; m < PHASES; m++) begin
                        for (int t = 0; t < TAPS_PER_PHASE; t++) begin

                            // 1. Input Latching (Only on Phase 0)
                            // Note: Logic logic uses current_phase derived
from FSM previous state,
                            // simpler to use the 'active_cycle' trigger
logic implied by VHDL.
                            // If we are starting IDLE->GAP, phase is
effectively 0 for math.
                            // If we are in PULSE, use current
phase_counter.

                            // We use temporary variable for current
processing phase for clarity
                            int current_p;
                            if (state == IDLE) current_p = 0;
                            else current_p = phase_counter; // In PULSE
```

```
state

                            if (current_p == 0) begin
                                a_reg[t] <= $signed(data_i);
                            end

                            // 2. Load Coefficient
                            // Index = t * PHASES + current_p
                            b_reg[t] <= coeff_rom[t*PHASES + current_p];

                            // 3. MAC Operation
                            // Index Mapping: t*PHASES + (PHASES-1) is the
"Top" of the column for tap 't'
                            if (t == TAPS_PER_PHASE - 1) begin
                                p_reg[t*PHASES + (PHASES-1)] <= a_reg[t] *
b_reg[t];
                            end else begin
                                p_reg[t*PHASES + (PHASES-1)] <= (a_reg[t] *
b_reg[t]) + p_reg[(t+1)*PHASES];
                            end

                            // 4. Shift Pipeline
                            if (m < PHASES - 1) begin
                                p_reg[t*PHASES + m] <= p_reg[t*PHASES + m +
1];
                            end
                        end
                    end

                    // Output Scaling
                    data_o <= p_reg[0][2*DATA_WIDTH-2-GAIN_BITS -:
DATA_WIDTH];
                end
            end

            // ============================================================
            // MODE 2: DECIMATION (IS_DECIMATION = 1)
            // ============================================================
            else begin

                // --- Control Logic ---
                if (valid_i) begin
                    active_cycle <= 1'b1;
                    if (phase_counter > 0)
                        phase_counter <= phase_counter - 1;
                    else
```

```
                    phase_counter <= PHASES - 1;
                end

                // --- Processing Logic ---
                if (active_cycle) begin
                    // Note: In decimation, we use the phase_counter state
*before* the update
                    // inside the math loop effectively, but since we
updated it above non-blocking,
                    // we need the logic to align.
                    // The VHDL used variables to use "current" value before
update.
                    // To match VHDL: if phase was 0, it wraps to 7.

                    // We need the value BEFORE the decrement above took
effect?
                    // No, standard coding: use a temporary or derived
logic.
                    // Let's rely on the previous cycle value logic by using
immediate logic if needed.
                    // Actually, simpler: Recalculate 'current'
conceptually.

                    int current_p;
                    // Reverse the decrement logic to find what the phase IS
for this data
                    if (phase_counter == PHASES - 1) current_p = 0;
                    else current_p = phase_counter + 1;

                    // Wait, simpler approach:
                    // If we just entered valid_i, the 'phase_counter'
register holds the CURRENT phase.
                    // We decrement it for the NEXT cycle.
                    // So we use 'phase_counter' (the value before the clock
edge update).

                    // But in non-blocking assignments, reading
'phase_counter' reads the OLD value.
                    // So simply using phase_counter here works perfectly.

                    for (int m = 0; m < PHASES; m++) begin
                        for (int t = 0; t < TAPS_PER_PHASE; t++) begin

                            // 1. Always latch input
                            a_reg[t] <= $signed(data_i);
```

```
                            // 2. Load Coefficient
                            b_reg[t] <= coeff_rom[t*PHASES + phase_counter];

                            // 3. MAC
                            if (t == TAPS_PER_PHASE - 1) begin
                                p_reg[t*PHASES + (PHASES-1)] <= a_reg[t] *
b_reg[t];
                            end else begin
                                p_reg[t*PHASES + (PHASES-1)] <= (a_reg[t] *
b_reg[t]) + p_reg[(t+1)*PHASES];
                            end

                            // 4. Shift
                            if (m < PHASES - 1) begin
                                p_reg[t*PHASES + m] <= p_reg[t*PHASES + m +
1];
                            end
                        end
                    end

                    // Accumulator Logic
                    if (phase_counter > 0) begin
                        product_sum <= p_reg[0] + product_sum;
                    end else begin
                        // Phase 0 reached (End of Block)
                        // Output Result
                        // (p_reg[0] + product_sum)
                        logic signed [2*DATA_WIDTH-1:0] final_val;
                        final_val = p_reg[0] + product_sum;

                        data_o <= final_val[2*DATA_WIDTH-2 -: DATA_WIDTH];
                        valid_o <= 1'b1;
                        product_sum <= '0;
                    end
                end
            end
        end
    end

endmodule
```

polyphase_resampler

```
`timescale 1ns / 1ps

module polyphase_resampler (
    input  logic        clk,
    input  logic        rst_n,
    input  logic        i_valid,
    input  logic signed [15:0] i_data,
    output logic        o_valid,
    output logic signed [15:0] o_data
);

    //
================================================================================
    // Parameters
    //
================================================================================
    localparam int DATA_WIDTH = 16;
    localparam int COEFF_WIDTH = 16;

    // Stage 1: Upsampler (Interpolate by 2)
    localparam int L_FACTOR = 2;
    localparam int L_TAPS_TOTAL = 226;
    localparam int L_TAPS_PER_PHASE = L_TAPS_TOTAL / L_FACTOR; // 64

    // Stage 2: Downsampler (Decimate by 3)
    localparam int M_FACTOR = 3;
    localparam int M_TAPS_TOTAL = 3;
    localparam int M_TAPS_PER_PHASE = M_TAPS_TOTAL / M_FACTOR; // 5

    //
================================================================================
    // Interconnect Signals
    //
================================================================================
    logic signed [DATA_WIDTH-1:0] s1_data;
    logic                         s1_valid;

    //
================================================================================
    // 1. Upsampler (x2)
    //    Input: 9 MHz -> Output: 18 MHz
    //    Uses 128 Coeffs (64 taps per phase)
    //
================================================================================
    polyphase_filter #(
        .COEFF_FILE     ("interp_l2_226.mem"),
```

```
        .DATA_WIDTH     (DATA_WIDTH),
        .COEFF_WIDTH    (COEFF_WIDTH),
        .PHASES         (L_FACTOR),        // 2
        .TAPS_PER_PHASE (L_TAPS_PER_PHASE),// 64
        .IS_DECIMATION  (0)                // Interpolation
    ) u_upsampler (
        .clk        (clk),
        .rst_n      (rst_n),
        .valid_i    (i_valid),
        .data_i     (i_data),
        .valid_o    (s1_valid),
        .data_o     (s1_data)
    );

    //
================================================================================
    // 2. Downsampler (/3)
    //    Input: 18 MHz -> Output: 6 MHz
    //    Uses 15 Coeffs (5 taps per phase), Full Pass
    //
================================================================================
    polyphase_filter #(
        .COEFF_FILE     ("decim_m3_pass.mem"),
        .DATA_WIDTH     (DATA_WIDTH),
        .COEFF_WIDTH    (COEFF_WIDTH),
        .PHASES         (M_FACTOR),        // 3
        .TAPS_PER_PHASE (M_TAPS_PER_PHASE),// 5
        .IS_DECIMATION  (1)                // Decimation
    ) u_downsampler (
        .clk        (clk),
        .rst_n      (rst_n),
        .valid_i    (s1_valid), // Chained from Stage 1
        .data_i     (s1_data),
        .valid_o    (o_valid),
        .data_o     (o_data)
    );

endmodule
```

TestBench

tb_rational_resampler

```
`timescale 1ns / 1ps

module tb_rational_resampler;

    //
================================================================================
    // PARAMETERS & CONSTANTS
    //
================================================================================
    localparam int DATA_WIDTH = 16;
    localparam time CLK_PERIOD = 10ns; // 100 MHz Clock

    // Rational Resampling Config: 2/3
    // Stage 1: Interpolation (L=2)
    localparam int L_FACTOR = 2;
    localparam int L_TAPS_TOTAL = 226;
    localparam int L_TAPS_PER_PHASE = L_TAPS_TOTAL / L_FACTOR; // 64

    // Stage 2: Decimation (M=3)
    localparam int M_FACTOR = 3;
    localparam int M_TAPS_TOTAL = 3;
    localparam int M_TAPS_PER_PHASE = M_TAPS_TOTAL / M_FACTOR; // 5

    //
================================================================================
    // SIGNALS
    //
================================================================================
    logic                           clk;
    logic                           rst_n;

    // Stage 0: Input
    logic                           s0_valid_i;
    logic signed [DATA_WIDTH-1:0] s0_data_i;

    // Stage 1: Intermediate (Output of x2 Upsampler)
    logic                           s1_valid;
    logic signed [DATA_WIDTH-1:0] s1_data;

    // Stage 2: Final Output (Output of /3 Downsampler)
    logic                           s2_valid;
    logic signed [DATA_WIDTH-1:0] s2_data;

    // File Handle
    integer fd;
```

```
    //
================================================================================
    // COMPONENT INSTANTIATION
    //
================================================================================

    // 1. Upsampler (x2) -> Uses 128 Coeffs (64 per phase)
    polyphase_filter #(
        .COEFF_FILE     ("interp_l2_226.mem"),
        .DATA_WIDTH     (DATA_WIDTH),
        .PHASES         (L_FACTOR),        // 2
        .TAPS_PER_PHASE (L_TAPS_PER_PHASE),// 64
        .IS_DECIMATION  (0)                // Interpolation
    ) u_upsampler (
        .clk(clk),
        .rst_n(rst_n),
        .valid_i(s0_valid_i),
        .data_i(s0_data_i),
        .valid_o(s1_valid),
        .data_o(s1_data)
    );

    // 2. Downsampler (/3) -> Uses 15 Coeffs (5 per phase), Full Pass
    polyphase_filter #(
        .COEFF_FILE     ("decim_m3_pass.mem"),
        .DATA_WIDTH     (DATA_WIDTH),
        .PHASES         (M_FACTOR),        // 3
        .TAPS_PER_PHASE (M_TAPS_PER_PHASE),// 5
        .IS_DECIMATION  (1)                // Decimation
    ) u_downsampler (
        .clk(clk),
        .rst_n(rst_n),
        .valid_i(s1_valid), // Chained from Stage 1
        .data_i(s1_data),
        .valid_o(s2_valid),
        .data_o(s2_data)
    );

    //
================================================================================
    // CLOCK GENERATION
    //
================================================================================
    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
```

```
    end

    //
================================================================================
    // STIMULUS GENERATION
    //
================================================================================
    initial begin
        // Simulation Constants
        real FS_IN = 9.0e6;      // 9 MHz
        real F1    = 1.0e6;      // 1 MHz Tone
        real F2    = 4.0e6;      // 4 MHz Tone (Near Nyquist)
        real SCALE_FACTOR = 15000.0;
        int  N_SAMPLES = 200;    // Number of input samples to generate

        real theta1 = 0.0;
        real theta2 = 0.0;
        real step1;
        real step2;
        real val_raw;
        int  val_int;
        real PI = 3.141592653589793;

        // Reset Sequence
        rst_n = 0;
        s0_valid_i = 0;
        s0_data_i = 0;
        #100;
        rst_n = 1;
        @(posedge clk);

        $display("-------------------------------------------------");
        $display("Generating Two-Tone Signal");
        $display("Tone 1: 1 MHz");
        $display("Tone 2: 4 MHz");
        $display("Fs In : 9 MHz");
        $display("L=2 (128 Taps), M=3 (15 Taps Pass)");
        $display("-------------------------------------------------");

        // Open Log File
        fd = $fopen("resampler_output.txt", "w");

        // Calculate Steps
        step1 = 2.0 * PI * F1 / FS_IN;
        step2 = 2.0 * PI * F2 / FS_IN;
```

```
        // Main Loop
        for (int i = 0; i < N_SAMPLES; i++) begin

            // 1. Math Generation
            val_raw = $sin(theta1) + $sin(theta2);
            val_int = int'(val_raw * SCALE_FACTOR);

            // Saturation
            if (val_int > 32767) val_int = 32767;
            if (val_int < -32768) val_int = -32768;

            // 2. Drive Input
            s0_valid_i <= 1'b1;
            s0_data_i  <= val_int[15:0];

            @(posedge clk);
            // 3. Pipeline Gaps (Matching VHDL "wait for k in 1 to 15")
            // This slows down data input to allow processing time
            s0_valid_i <= 1'b0;
            // s0_data_i  <= '0;  // if you remove it . it will cause like
zero hold
            repeat(15) @(posedge clk);

            // 4. Update Phase
            theta1 = theta1 + step1;
            if (theta1 > 2.0*PI) theta1 = theta1 - 2.0*PI;

            theta2 = theta2 + step2;
            if (theta2 > 2.0*PI) theta2 = theta2 - 2.0*PI;
        end

        // End Simulation
        #2000;
        $fclose(fd);
        $display("Simulation Finished.");
        $finish;
    end

    //
================================================================================
    // LOGGING PROCESS
    //
================================================================================
    always @(posedge clk) begin
        if (rst_n) begin
            // Log INPUT
```

```
            if (s0_valid_i) begin
                $fdisplay(fd, "IN:  %d", $signed(s0_data_i));
            end

            // Log INTERMEDIATE
            if (s1_valid) begin
                $fdisplay(fd, "MID: %d", $signed(s1_data));
            end

            // Log OUTPUT
            if (s2_valid) begin
                $fdisplay(fd, "OUT: %d", $signed(s2_data));
            end
        end
    end
        //
================================================================================
    // WAVEFORM DUMPING (Required for Icarus Verilog)
    //
================================================================================
    initial begin
        $dumpfile("waveform_rational.vcd"); // Must match Makefile VCD2
variable
        $dumpvars(0, tb_rational_resampler);
    end


endmodule
```

Makefile

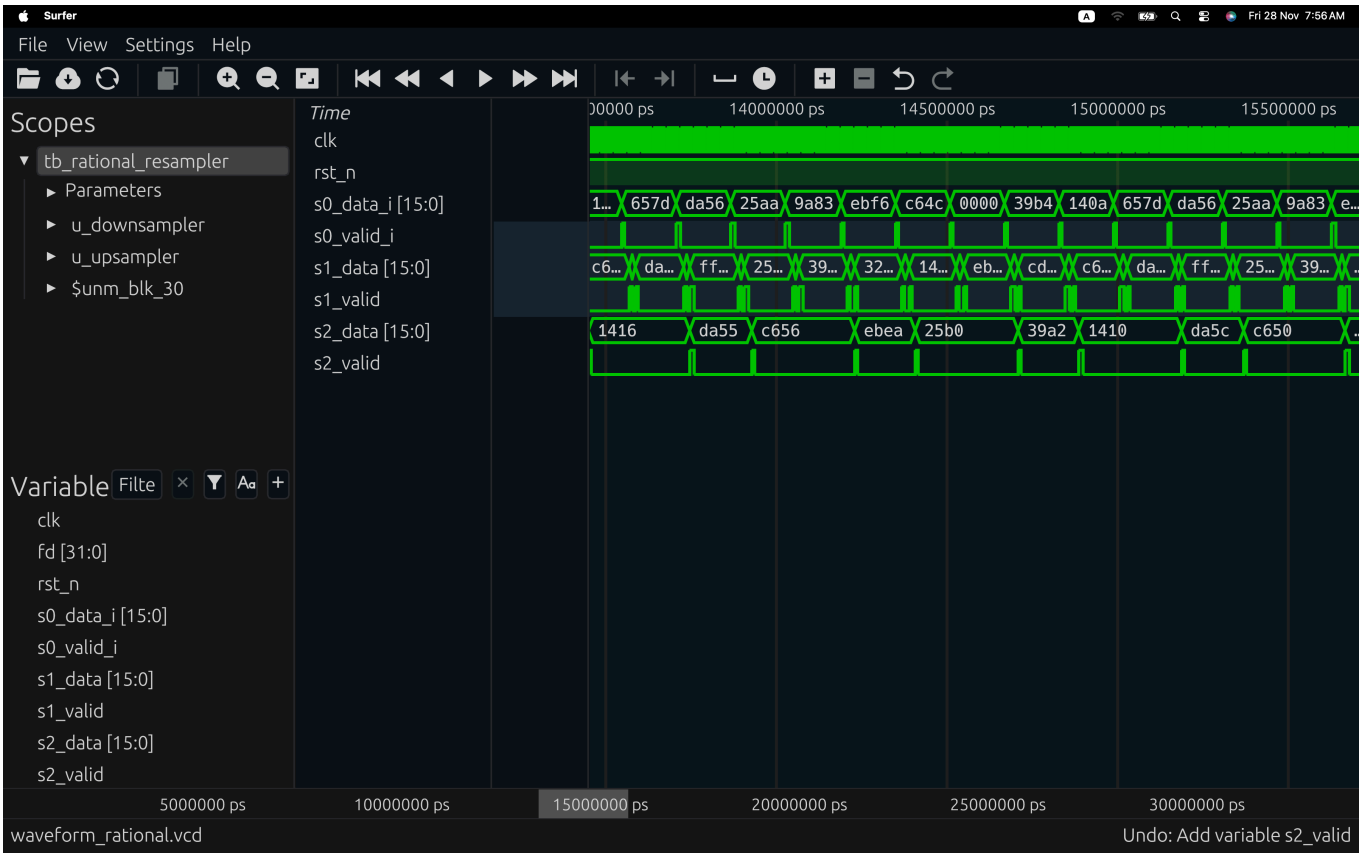
```
# Variables
CC = iverilog
SIM = vvp
VIEWER = surfer
PYTHON = python3

# -g2012 is required for SystemVerilog
FLAGS = -g2012 -Wall

# Common Design File
DUT = polyphase_filter.sv

# --- CONFIGURATION 2: Rational Resampler (Cascaded) ---
```

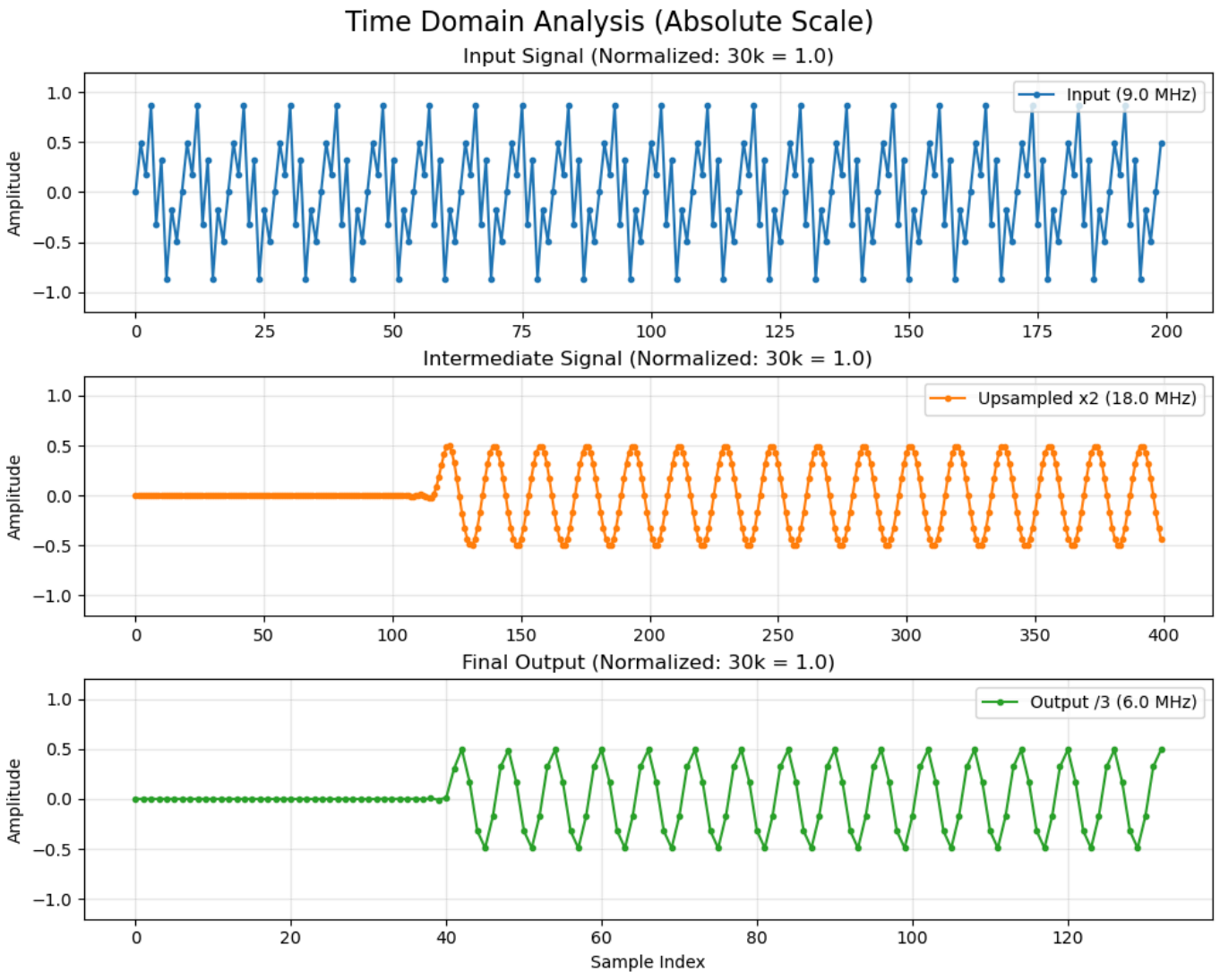
```
TB2_FILE = tb_rational_resampler.sv
# Output executable name
OUT2 = rational.vvp
# The VCD filename MUST match what is in $dumpfile inside the SV Testbench
VCD2 = waveform_rational.vcd
# Python script for plotting (Assumes script reads resampler_output.txt)
SCRIPT = plot_resampler.py

.PHONY: all help clean rational view_rational plot_rational

# Default Target
all: plot_rational view_rational

help:
    @echo "-----------------------------------------------------------------"
    @echo "Available Commands:"
    @echo "  make rational         -> Compile and Run Simulation"
    @echo "  make view_rational    -> Run Sim & View Waveforms (Surfer)"
    @echo "  make plot_rational    -> Run Sim & Plot Data (Python)"
    @echo "  make clean            -> Delete all compiled files"
    @echo "-----------------------------------------------------------------"

#
================================================================================
==
# OPTION 2: Rational Resampler (Cascaded)
#
================================================================================
==
# 1. Compile
$(OUT2): $(DUT) $(TB2_FILE)
    @echo "--- Compiling Rational Resampler ---"
    $(CC) $(FLAGS) -o $(OUT2) $(DUT) $(TB2_FILE)

# 2. Run
rational: $(OUT2)
    @echo "--- Running Rational Simulation ---"
    $(SIM) $(OUT2)

# 3. View
view_rational: rational
    @echo "--- Opening Surfer (Rational) ---"
    @if [ -f $(VCD2) ]; then \
        $(VIEWER) $(VCD2) & \
    else \
        echo "Error: $(VCD2) not found. Did you add \$$dumpfile to the
```

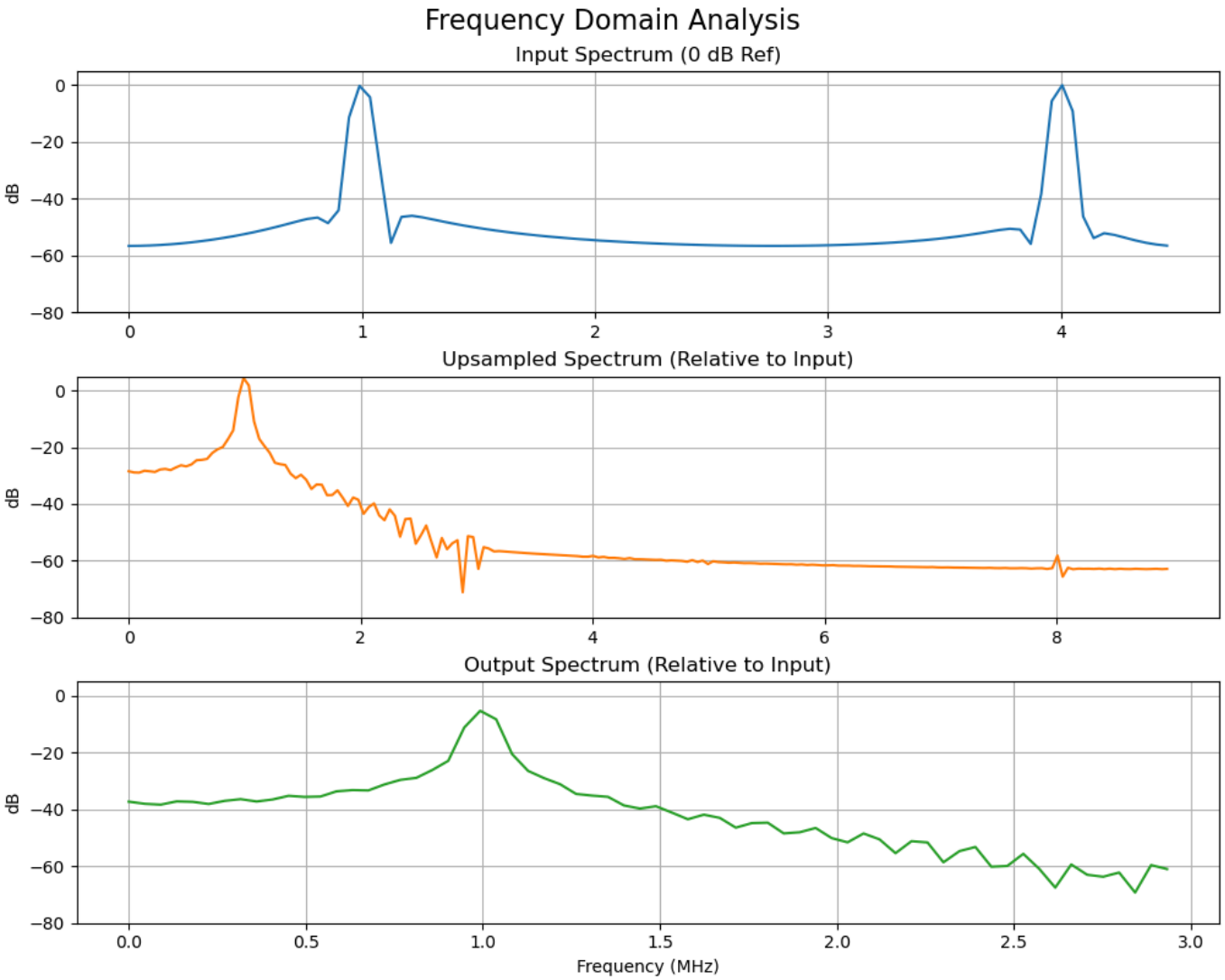
```
testbench?"; \
    fi

# 4. Plot (New Target)
plot_rational: rational
    @echo "--- Running Python Analysis ---"
    $(PYTHON) $(SCRIPT);

#
==============================================================================
==
# Cleanup
#
==============================================================================
==
clean:
    @echo "--- Cleaning ---"
    rm -f *.vvp *.vcd *.out resampler_output.txt
```

Results

```
--- Running Rational Simulation ---
vvp rational.vvp
VCD info: dumpfile waveform_rational.vcd opened for output.
------------------------------------------------
Generating Two-Tone Signal
Tone 1: 1 MHz
Tone 2: 4 MHz
Fs In : 9 MHz
L=2 (128 Taps), M=3 (15 Taps Pass)
------------------------------------------------
Simulation Finished.
tb_rational_resampler.sv:163: $finish called at 34105000 (1ps)
--- Running Python Analysis ---
python3 plot_resampler.py;
Reading resampler_output.txt...
Samples Read -> In: 200, Mid: 400, Out: 133
Normalizing all signals by Fixed Factor: 30000.0
```


Surfer
Fri 28 Nov 7:56 AM
File View Settings Help
Scopes
tb_rational_resampler
Parameters
u_downsampler
u_upsampler
$unm_blk_30
Time
clk
rst_n
s0_data_i [15:0]
s0_valid_i
s1_data [15:0]
s1_valid
s2_data [15:0]
s2_valid
14000000 ps
14500000 ps
15000000 ps
15500000 ps
1... 657d da56 25aa 9a83 ebf6 c64c 0000 39b4 140a 657d da56 25aa 9a83
c6... da... ff... 25... 39... 32... 14... eb... cd... c6... da... ff... 25... 39...
1416 da55 c656 ebea 25b0 39a2 1410 da5c c650
Variable
Filte
clk
fd [31:0]
rst_n
s0_data_i [15:0]
s0_valid_i
s1_data [15:0]
s1_valid
s2_data [15:0]
s2_valid
5000000 ps
10000000 ps
15000000 ps
20000000 ps
25000000 ps
30000000 ps
waveform_rational.vcd
Undo: Add variable s2_valid


Time Domain Analysis (Absolute Scale)
Input Signal (Normalized: 30k = 1.0)
Input (9.0 MHz)
Amplitude
Intermediate Signal (Normalized: 30k = 1.0)
Upsampled x2 (18.0 MHz)
Amplitude
Final Output (Normalized: 30k = 1.0)
Output /3 (6.0 MHz)
Amplitude
Sample Index



NOTCH filter

RTL

```
`timescale 1ns / 1ps

module notch_filter #(
    parameter int DATA_WIDTH = 16
)(
    input  logic                        clk,
    input  logic                        rst_n,
    input  logic                        valid_i,
    input  logic signed [DATA_WIDTH-1:0] data_i,

    output logic                        valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);

    //
```

```
//============================================================================
    // Coefficients (Q2.14)
    //
//============================================================================
    localparam signed [15:0] B0 = 16'd15725;
    localparam signed [15:0] B1 = 16'd25443;
    localparam signed [15:0] B2 = 16'd15725;
    localparam signed [15:0] A1 = 16'd25443;
    localparam signed [15:0] A2 = 16'd15066;

    //
//============================================================================
    // Internal State
    //
//============================================================================
    logic signed [31:0] s1, s2;
    logic signed [DATA_WIDTH-1:0] x_reg; // Latch input
    logic signed [31:0] y_reg;           // Latch intermediate output

    // State Machine
    typedef enum logic { ST_CALC_OUT, ST_UPDATE_STATE } state_t;
    state_t state;

    //
//============================================================================
    // Processing Logic (Split into 2 Cycles)
    //
//============================================================================
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            valid_o <= 0;
            data_o  <= 0;
            s1      <= 0;
            s2      <= 0;
            x_reg   <= 0;
            y_reg   <= 0;
            state   <= ST_CALC_OUT;
        end else begin
            valid_o <= 0; // Default low

            case (state)
                // ------------------------------------------------------------
-----
                // State 0: Wait for Input -> Calculate Output (y)
                // ------------------------------------------------------------
-----
```

```
                ST_CALC_OUT: begin
                    if (valid_i) begin
                        // 1. Latch Input
                        x_reg <= data_i;

                        // 2. Calculate Feed-Forward (Part A)
                        // y[n] = b0*x[n] + s1[n-1]
                        // We register this result to break the timing path.
                        y_reg <= (data_i * B0) + s1;

                        // Move to next step immediately
                        state <= ST_UPDATE_STATE;
                    end
                end

                // ------------------------------------------------------------
----
                // State 1: Calculate Feedback -> Update States (s1, s2)
                // ------------------------------------------------------------
----
                ST_UPDATE_STATE: begin
                    // 1. Output the result calculated in previous cycle
                    data_o  <= y_reg[29:14]; // Scale Q2.29 -> Q1.15
                    valid_o <= 1;            // Signal valid output

                    // 2. Calculate Feedback Terms (Using registered y_reg)
                    // This splits the math load: Multiply A1/A2 happens
here.
                    // s1[n] = b1*x - a1*y + s2[n-1]
                    s1 <= (x_reg * B1) - ((y_reg >>> 14) * A1) + s2;

                    // s2[n] = b2*x - a2*y
                    s2 <= (x_reg * B2) - ((y_reg >>> 14) * A2);

                    // Done, go back to wait for next sample
                    state <= ST_CALC_OUT;
                end
            endcase
        end
    end

endmodule
```

TestBench

```
`timescale 1ns / 1ps

module tb_notch_filter;

    //
================================================================================
    // Header
    //
================================================================================
    // Author: Hazem Yasser Mahmoud Mohamed
    // Description: Testbench for Notch Filter with Valid every 4 cycles

    //
================================================================================
    // Parameters
    //
================================================================================
    localparam int DATA_WIDTH = 16;
    localparam real FS = 6.0e6;       // Sampling Frequency: 6 MHz
    localparam int NUM_SAMPLES = 2048; // Number of samples to simulate

    //
================================================================================
    // Signals
    //
================================================================================
    logic                       clk;
    logic                       rst_n;
    logic                       valid_i;
    logic signed [DATA_WIDTH-1:0] data_i;
    logic                       valid_o;
    logic signed [DATA_WIDTH-1:0] data_o;

    // File Handle
    int fd;

    // Simulation Variables
    real t;
    real val_1_0m, val_2_4m, val_total;
    integer i;

    // Temporary integer for conversion
    int temp_val;
```

```
    //
================================================================================
    // DUT Instantiation
    //
================================================================================
    notch_filter #(
        .DATA_WIDTH(DATA_WIDTH)
    ) u_dut (
        .clk     (clk),
        .rst_n   (rst_n),
        .valid_i (valid_i),
        .data_i  (data_i),
        .valid_o (valid_o),
        .data_o  (data_o)
    );

    //
================================================================================
    // Clock Generation
    //
================================================================================
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100 MHz System Clock
    end

    //
================================================================================
    // VCD Dump (Waveform Generation)
    //
================================================================================
    initial begin
        $dumpfile("notch_filter.vcd");
        $dumpvars(0, tb_notch_filter);
    end

    //
================================================================================
    // Stimulus Generation
    //
================================================================================
    initial begin
        // 1. Initialize
        rst_n   = 0;
        valid_i = 0;
        data_i  = 0;
```

```
        fd      = $fopen("notch_io.txt", "w");

        if (fd == 0) begin
            $display("Error: Could not open output file.");
            $finish;
        end

        // 2. Reset Sequence
        repeat(10) @(posedge clk);
        rst_n = 1;
        repeat(10) @(posedge clk);

        $display("Starting Simulation...");

        // 3. Drive Data (Valid High for 1 cycle, Low for 3 cycles)
        for (i = 0; i < NUM_SAMPLES; i++) begin

            // Calculate time 't'
            t = real'(i) / FS;

            // Tone 1: 1.0 MHz (Passband)
            val_1_0m = $sin(2.0 * 3.14159 * 1.0e6 * t);

            // Tone 2: 2.4 MHz (Notch Frequency)
            val_2_4m = $sin(2.0 * 3.14159 * 2.4e6 * t);

            // Combine: 1MHz + 0.2 DC + 2.4MHz
            val_total = val_1_0m + 0.2 + val_2_4m;

            // Scale to fit Q1.15 Fixed Point
            // Scaling factor 0.4 to keep within range (-1.0 to 1.0)
            temp_val = $rtoi(val_total * 0.4 * 32767.0);

            // Clamp to 16-bit range
            if (temp_val > 32767) temp_val = 32767;
            if (temp_val < -32768) temp_val = -32768;

            // --- DRIVE DATA ---
            valid_i <= 1'b1;
            data_i  <= temp_val[15:0];

            // Wait 1 clock cycle (Active Cycle)
            @(posedge clk);

            // --- IDLE GAP ---
            valid_i <= 1'b0;
```

```
            // Wait 3 clock cycles (Idle Cycles) -> Total period = 4 clocks
            repeat(3) @(posedge clk);
        end

        @(posedge clk);
        valid_i = 0;
        data_i  = 0;

        // Allow pipeline to flush
        repeat(50) @(posedge clk);

        $display("Simulation Finished. Waveform dumped to
notch_filter.vcd");
        $display("Data written to notch_io.txt");
        $fclose(fd);
        $finish;
    end

    //
================================================================================
    // Text Output (CSV Style)
    //
================================================================================
    always @(posedge clk) begin
        if (rst_n) begin
            // Only write to file when valid_i or valid_o is active to save
space/readability
            // Or keep it continuous to see the gaps. Keeping continuous
based on previous code.
            $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
        end
    end

endmodule
```

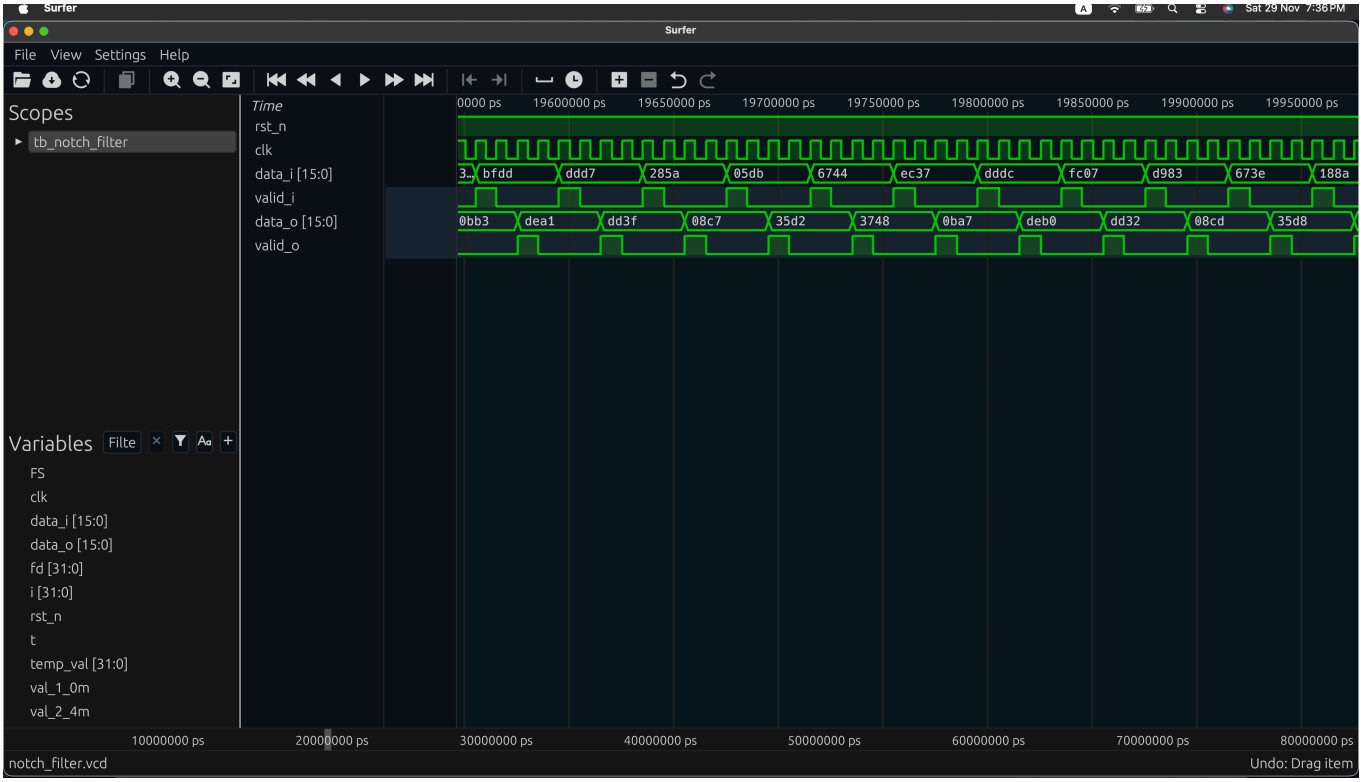
Makefile

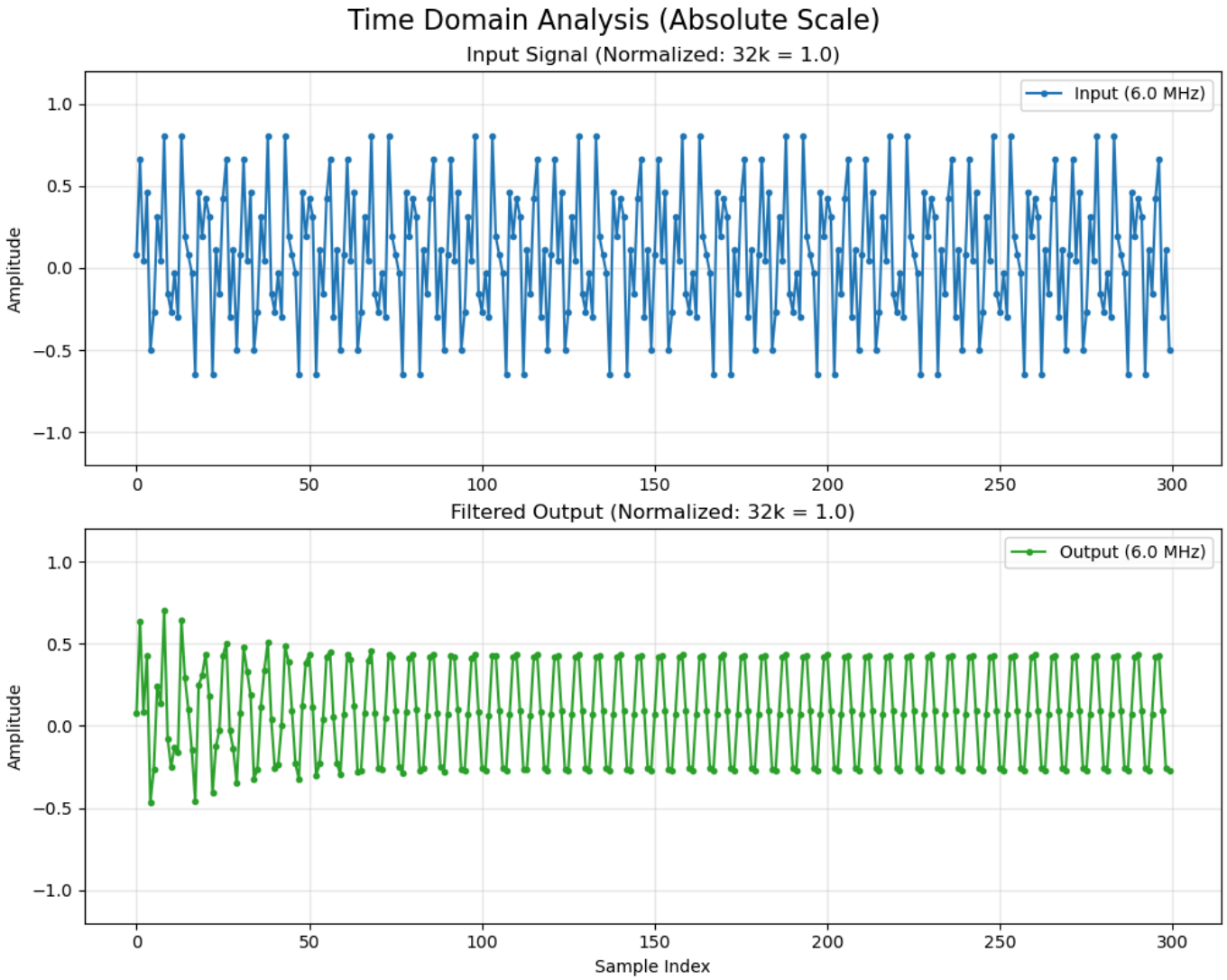
```
#
================================================================================
==
# Makefile for Notch Filter Simulation
# Tools: Icarus Verilog (iverilog), VVP, GTKWave
```

```
#
================================================================================
==

# Tools
COMPILER = iverilog
SIMULATOR = vvp
VIEWER = surfer
PYTHON = python3

# Files
# Assuming notch_filter.sv is in the same directory
SRC = notch_filter.sv tb_notch_filter.sv
OUT = notch_sim.out
VCD = notch_filter.vcd
TXT = notch_io.txt
SCRIPT = plot_notch_io.py

# Flags
# -g2012 enables SystemVerilog 2012 support
FLAGS = -g2012 -Wall

# Targets ------------------------------------------------------------------
----

.PHONY: all compile run view plot clean

all: compile run

# 1. Compile Verilog
compile:
    @echo "Compiling SystemVerilog files..."
    $(COMPILER) $(FLAGS) -o $(OUT) $(SRC)

# 2. Run Simulation
run: compile
    @echo "Running Simulation..."
    $(SIMULATOR) $(OUT)

# 3. View Waveform (GTKWave)
view: run
    @echo "Opening Waveform Viewer..."
    $(VIEWER) $(VCD) &

# 4. Run Python Analysis
plot: run
```

```
	@echo "Running Python Analysis..."
	$(PYTHON) $(SCRIPT)

# 5. Clean up
clean:
	@echo "Cleaning up..."
	rm -f $(OUT) $(VCD) $(TXT) *.png
```

Results





CIC

RTL

```
`timescale 1ns / 1ps

module cic_filter #(
    parameter int DATA_WIDTH = 16
)(
    input  logic                        clk,
    input  logic                        rst_n,
    input  logic [4:0]                  rate_i,  // Decimation Rate: 1, 2, 4, 8,
16
    input  logic                        valid_i,
    input  logic signed [DATA_WIDTH-1:0] data_i,

    output logic                        valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);
```

```
    //
================================================================================
    // Parameters & Derived Types
    //
================================================================================
    // Max Decimation R = 16, Stages N = 5.
    // Max Gain G = R^N = 16^5 = 1,048,576 (approx 2^20).
    // Required Internal Width = Input Width + ceil(log2(G)) = 16 + 20 = 36
bits.
    localparam int INTERNAL_WIDTH = 36;
    localparam int STAGES = 5;

    //
================================================================================
    // Internal Signals
    //
================================================================================

    // Integrator Signals
    // We sign-extend input to internal width
    logic signed [INTERNAL_WIDTH-1:0] int_in;
    logic signed [INTERNAL_WIDTH-1:0] integrators [0:STAGES-1];

    // Comb Signals
    logic signed [INTERNAL_WIDTH-1:0] comb_in;
    logic signed [INTERNAL_WIDTH-1:0] combs [0:STAGES-1];
    logic signed [INTERNAL_WIDTH-1:0] comb_delays [0:STAGES-1];

    // Decimation Control
    logic [4:0] count;
    logic       comb_pulse; // Valid signal for the comb section (rate / R)

    // Output Scaling
    logic signed [INTERNAL_WIDTH-1:0] scaled_data;
    int shift_amount;

    //
================================================================================
    // 1. Integrator Section (Running at High Rate)
    //
================================================================================

    // Sign extend input
    assign int_in = {{ (INTERNAL_WIDTH-DATA_WIDTH){data_i[DATA_WIDTH-1]} },
data_i};
```

```
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (int i = 0; i < STAGES; i++) integrators[i] <= '0;
        end else if (valid_i) begin
            // Stage 0 accumulates Input
            // Standard CIC Integrator: y[n] = y[n-1] + x[n]
            // Relies on modulo arithmetic wrapping (2's complement)
            integrators[0] <= integrators[0] + int_in;

            // Subsequent stages accumulate previous stage
            for (int i = 1; i < STAGES; i++) begin
                integrators[i] <= integrators[i] + integrators[i-1];
            end
        end
    end

    //
================================================================================
    // 2. Decimator (Rate Change)
    //
================================================================================

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            count <= 0;
            comb_pulse <= 0;
        end else if (valid_i) begin
            if (rate_i <= 1) begin
                // If rate is 1, we treat it as always valid (Bypass logic
handles data)
                // But for comb logic, we don't pulse to save power/logic
                count <= 0;
                comb_pulse <= 0;
            end else begin
                // Count from 0 to R-1
                if (count == rate_i - 1) begin
                    count <= 0;
                    comb_pulse <= 1;
                end else begin
                    count <= count + 1;
                    comb_pulse <= 0;
                end
            end
        end else begin
            comb_pulse <= 0;
```

```
        end
    end

    //
================================================================================
    // 3. Comb Section (Running at Low Rate)
    //
================================================================================
    // Input to combs comes from the last integrator
    assign comb_in = integrators[STAGES-1];

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (int i = 0; i < STAGES; i++) begin
                combs[i] <= '0;
                comb_delays[i] <= '0;
            end
            valid_o <= 0;
        end else if (comb_pulse) begin
            // Stage 0 Comb
            // y[m] = x[m] - x[m-1] (M=1 differential delay)
            combs[0]       <= comb_in - comb_delays[0];
            comb_delays[0] <= comb_in;

            // Subsequent Stages
            for (int i = 1; i < STAGES; i++) begin
                combs[i]       <= combs[i-1] - comb_delays[i];
                comb_delays[i] <= combs[i-1];
            end
            valid_o <= 1; // Pulse output valid
        end else begin
            // For Rate=1, valid logic is handled in the final block
            if (rate_i > 1) valid_o <= 0;
        end
    end

    //
================================================================================
    // 4. Output Scaling & Mux (Bypass Logic)
    //
================================================================================
    // Gain G = R^N. To normalize, we right shift by log2(G) = N * log2(R).
    // N=5.

    always_comb begin
        case (rate_i)
```

```
            5'd2:    shift_amount = 5;  // 2^5 = 32  (Shift 5)
            5'd4:    shift_amount = 10; // 4^5 = 1024 (Shift 10)
            5'd8:    shift_amount = 15; // 8^5 = 32k (Shift 15)
            5'd16:   shift_amount = 20; // 16^5 = 1M (Shift 20)
            default: shift_amount = 0;
        endcase
    end

    // Truncation/Rounding logic could go here. For simplicity, we use
truncation (>>>).
    // Note: We use the output of the last comb stage.
    assign scaled_data = combs[STAGES-1] >>> shift_amount;

    // Final Output Mux
    always_comb begin
        if (rate_i <= 1) begin
            // Bypass Mode
            data_o  = data_i;
            // valid_o follows valid_i directly in bypass
            // (Note: The sequential block above drives valid_o for
decimation,
            //  we override it here for bypass combinationally or we need
distinct signals)
        end else begin
            // Decimation Mode
            // Clamp to output width (Saturation)
            // Although unity gain suggests it fits, transients might
overflow slightly.
            // Simple truncation:
            data_o  = scaled_data[DATA_WIDTH-1:0];
        end
    end

    // Fix valid_o for Bypass:
    // We need a clean valid signal.
    // Let's create a specific `final_valid` wire
    logic final_valid;
    assign final_valid = (rate_i <= 1) ? valid_i : valid_o;

    // Because valid_o is a reg in the always block, we can't assign it
continuously.
    // Let's rename the register to `decim_valid` and assign valid_o.

    // RE-WRITING Output Logic for clarity/correctness

endmodule
```

TestBench

```
`timescale 1ns / 1ps

module tb_cic_filter;

    // Parameters
    localparam int DATA_WIDTH = 16;
    localparam real FS_IN = 6.0e6; // 6 MHz Input
    localparam int R = 4;          // Testing Decimation by 4 (Output 1.5
MHz)

    // Signals
    logic clk;
    logic rst_n;
    logic [4:0] rate;
    logic valid_i;
    logic signed [DATA_WIDTH-1:0] data_i;
    logic valid_o;
    logic signed [DATA_WIDTH-1:0] data_o;

    int fd;
    real t;
    real val_pass, val_stop, val_total;
    int i;

    // DUT
    cic_filter #(
        .DATA_WIDTH(DATA_WIDTH)
    ) u_dut (
        .clk(clk),
        .rst_n(rst_n),
        .rate_i(rate),
        .valid_i(valid_i),
        .data_i(data_i),
        .valid_o(valid_o),
        .data_o(data_o)
    );

    // Clock
    initial begin
        clk = 0;
```

```
        forever #5 clk = ~clk; // 100 MHz System Clock
    end

    // Simulation
    initial begin
        $dumpfile("cic_filter.vcd");
        $dumpvars(0, tb_cic_filter);

        fd = $fopen("cic_io.txt", "w");

        // Init
        rst_n = 0;
        valid_i = 0;
        data_i = 0;
        rate = R; // Set decimation rate

        repeat(10) @(posedge clk);
        rst_n = 1;
        repeat(10) @(posedge clk);

        $display("Starting CIC Simulation (R=%0d)...", R);

        // Drive 2048 Samples
        for (i = 0; i < 2048; i++) begin
            @(posedge clk);
            t = real'(i) / FS_IN;

            // 100 kHz (Passband)
            val_pass = $sin(2.0 * 3.14159 * 0.1e6 * t);

            // 2.0 MHz (Stopband for R=4, aliasing zone)
            // CIC should attenuate this significantly (sinc nulls at
multiples of fs_out=1.5M?)
            // First null is at fs_in/R = 1.5MHz.
            // 2.0MHz is in the side lobe, should be attenuated.
            val_stop = $sin(2.0 * 3.14159 * 2.0e6 * t);

            val_total = (val_pass + val_stop) * 0.5; // Scale to fit

            data_i <= $rtoi(val_total * 32767.0);
            valid_i <= 1;
        end

        @(posedge clk);
        valid_i = 0;
```

```
        repeat(100) @(posedge clk);
        $fclose(fd);
        $display("Simulation Finished.");
        $finish;
    end

    // File Write
    always @(posedge clk) begin
        if (rst_n) begin
            $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
        end
    end

endmodule
```

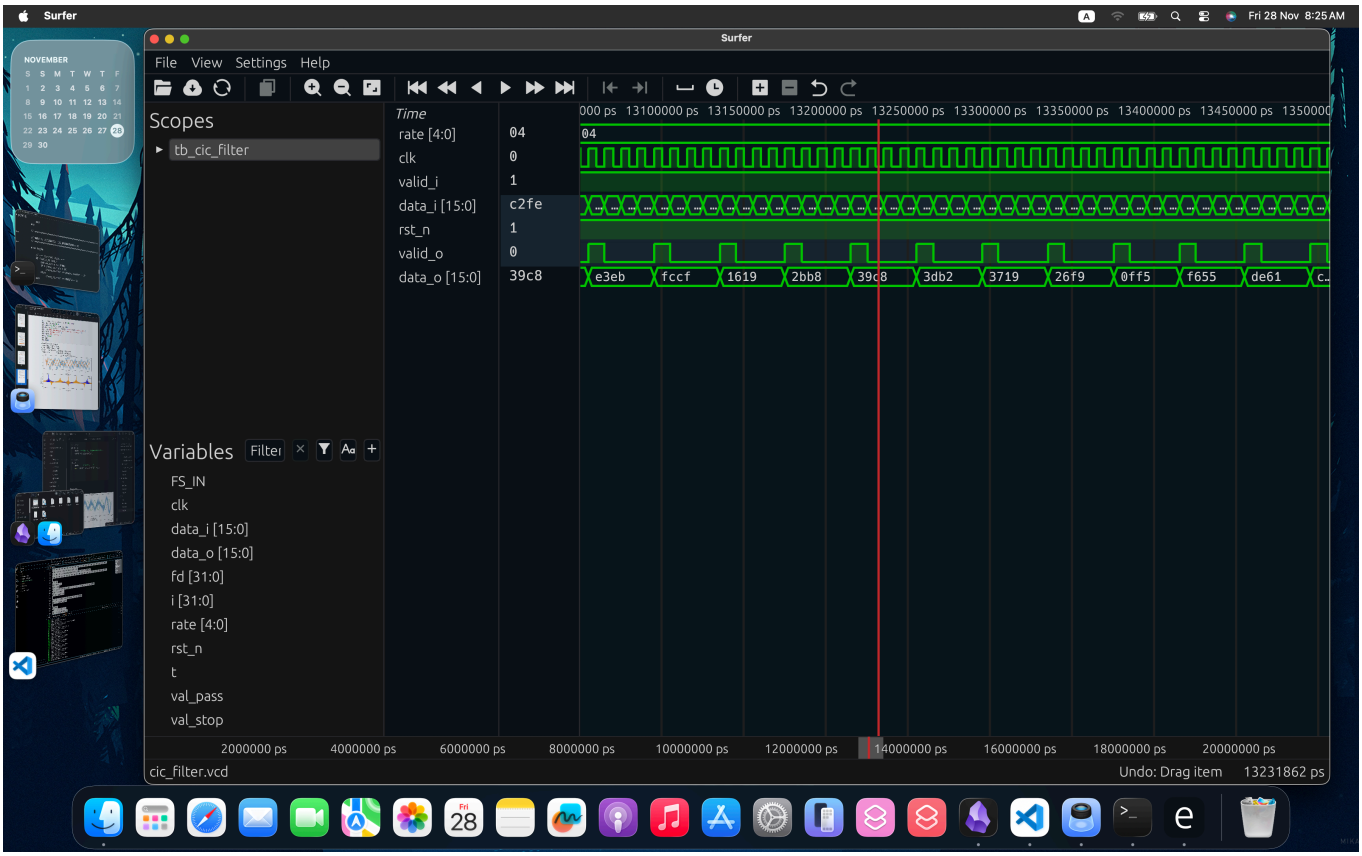
Makefile

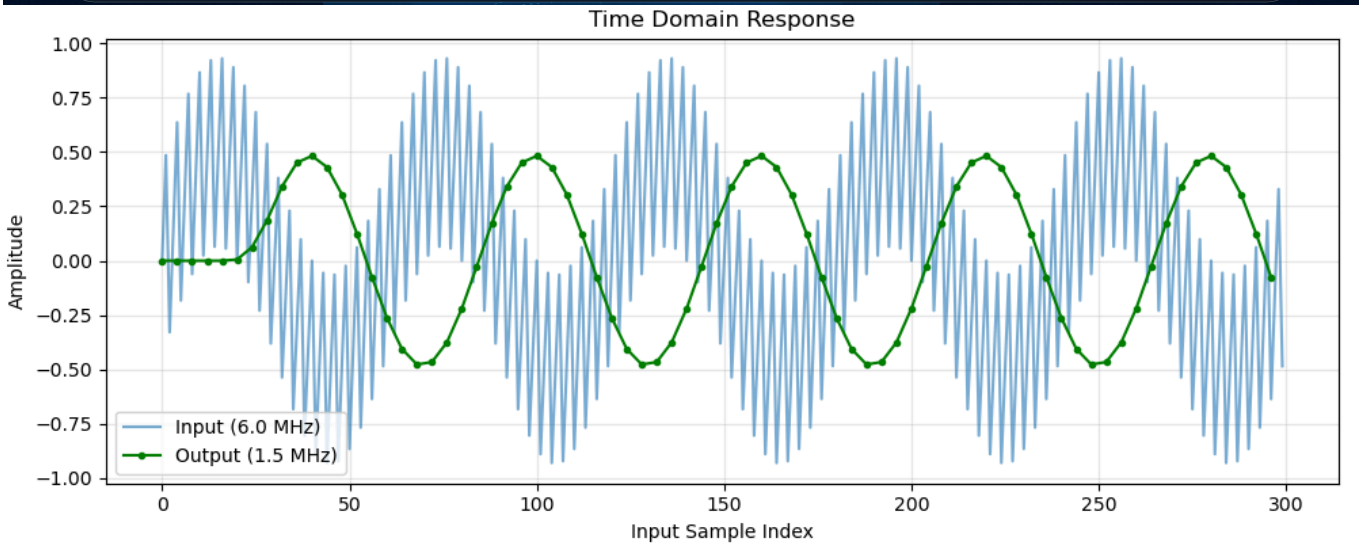
```
#
==============================================================================
==
# Makefile for CIC Filter Simulation
# Tools: Icarus Verilog (iverilog), VVP, Surfer/GTKWave
#
==============================================================================
==

# Tools
CC = iverilog
SIM = vvp
# Change VIEWER to gtkwave if you don't have surfer installed
VIEWER = surfer
PYTHON = python3

# Flags (-g2012 for SystemVerilog support)
FLAGS = -g2012 -Wall

# Files
DUT = cic_filter.sv
TB = tb_cic_filter.sv
OUT = cic_sim.out
VCD = cic_filter.vcd
TXT = cic_io.txt
```

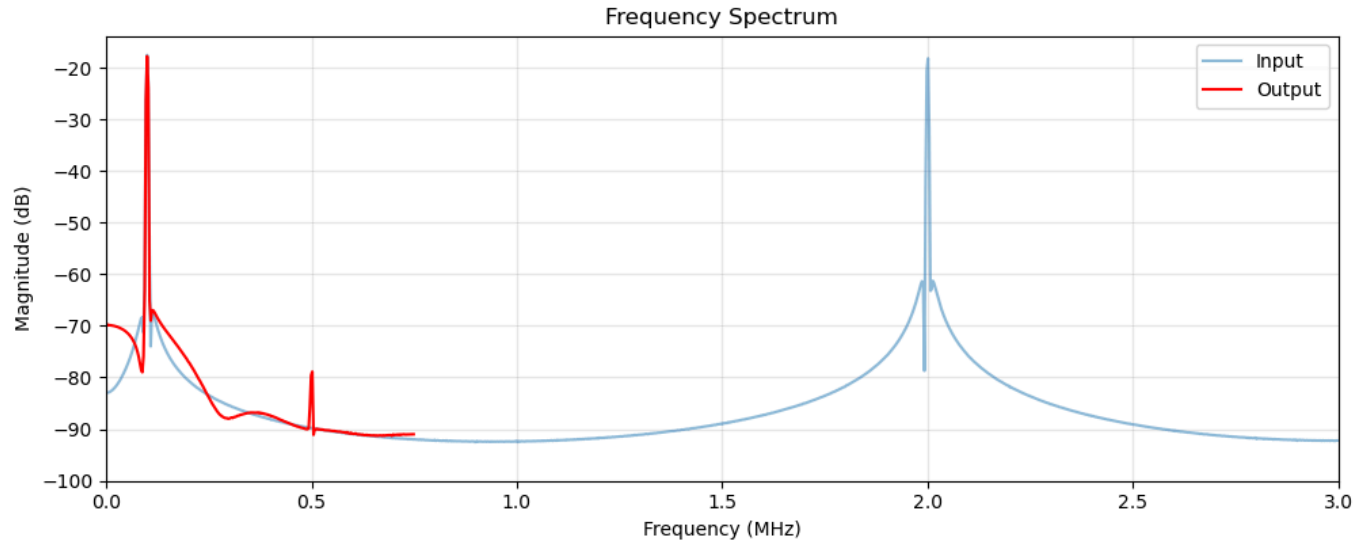
```
SCRIPT = plot_cic.py

.PHONY: all compile run view plot clean help

# Default Target
all: plot view

help:
    @echo "------------------------------------------------------------------"
    @echo "Available Commands:"
    @echo "  make compile   -> Compile SystemVerilog files"
    @echo "  make run       -> Run Simulation (generates .vcd and .txt)"
    @echo "  make view      -> Open Waveform Viewer"
    @echo "  make plot      -> Run Python Analysis Script"
    @echo "  make clean     -> Remove generated files"
    @echo "------------------------------------------------------------------"

# 1. Compile
$(OUT): $(DUT) $(TB)
    @echo "--- Compiling CIC Filter ---"
    $(CC) $(FLAGS) -o $(OUT) $(DUT) $(TB)

compile: $(OUT)

# 2. Run
run: $(OUT)
    @echo "--- Running Simulation ---"
    $(SIM) $(OUT)

# 3. View
view: run
    @echo "--- Opening Waveform Viewer ---"
    @if [ -f $(VCD) ]; then \
        $(VIEWER) $(VCD) & \
    else \
        echo "Error: $(VCD) not found."; \
    fi

# 4. Plot
plot: run
    @echo "--- Running Python Analysis ---"
    $(PYTHON) $(SCRIPT); \

# Cleanup
clean:
    @echo "--- Cleaning ---"
```

```
	rm -f $(OUT) $(VCD) $(TXT) *.png
```

Results


Surfer
File View Settings Help
Scopes
tb_cic_filter
Time
rate [4:0] 04
clk 0
valid_i 1
data_i [15:0] c2fe
rst_n 1
valid_o 0
data_o [15:0] 39c8
e3eb fccf 1619 2bb8 39c8 3db2 3719 26f9 0ff5 f655 de61
Variables
FS_IN
clk
data_i [15:0]
data_o [15:0]
fd [31:0]
i [31:0]
rate [4:0]
rst_n
t
val_pass
val_stop
cic_filter.vcd
Undo: Drag item
13231862 ps


Time Domain Response
Amplitude
Input Sample Index
Input (6.0 MHz)
Output (1.5 MHz)


Frequency Spectrum
Magnitude (dB)
Frequency (MHz)
Input
Output

DEF TOP FILE

RTL

```
`timescale 1ns / 1ps

module dfe_top #(
    parameter int DATA_WIDTH = 16
)(
    input  logic                        clk,
    input  logic                        rst_n,

    // Configuration
    input  logic [4:0]                  cic_rate_i,

    // Input Stream
    input  logic                        valid_i,
    input  logic signed [DATA_WIDTH-1:0] data_i,

    // Output Stream
    output logic                        valid_o,
    output logic signed [DATA_WIDTH-1:0] data_o
);

    //
================================================================================
    // 1. Input Pipeline Registers (CRITICAL FOR TIMING)
    //    Isolates IO Pins from the heavy fanout of the filters.
    //
================================================================================
    logic signed [DATA_WIDTH-1:0] r_data_i;
    logic                         r_valid_i;
    logic [4:0]                   r_cic_rate;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            r_data_i   <= 0;
            r_valid_i  <= 0;
            r_cic_rate <= 1;
        end else begin
            r_data_i   <= data_i;
            r_valid_i  <= valid_i;
            r_cic_rate <= cic_rate_i;
        end
    end
```

```
    //
================================================================================
    // Interconnect Signals
    //
================================================================================
    logic signed [DATA_WIDTH-1:0] w_resamp_data;
    logic                         w_resamp_valid;
    logic signed [DATA_WIDTH-1:0] w_notch_data;
    logic                         w_notch_valid;

    // Internal Output Signals
    logic signed [DATA_WIDTH-1:0] w_final_data;
    logic                         w_final_valid;

    //
================================================================================
    // Stage 1: Polyphase Resampler
    //
================================================================================
    polyphase_resampler u_stage1_resampler (
        .clk        (clk),
        .rst_n      (rst_n),
        .i_valid    (r_valid_i),  // Connect to Pipelined Register
        .i_data     (r_data_i),   // Connect to Pipelined Register
        .o_valid    (w_resamp_valid),
        .o_data     (w_resamp_data)
    );

    //
================================================================================
    // Stage 2: Notch Filter
    //
================================================================================
    notch_filter #(
        .DATA_WIDTH (DATA_WIDTH)
    ) u_stage2_notch (
        .clk        (clk),
        .rst_n      (rst_n),
        .valid_i    (w_resamp_valid),
        .data_i     (w_resamp_data),
        .valid_o    (w_notch_valid),
        .data_o     (w_notch_data)
    );

    //
```

```
// ============================================================================
    // Stage 3: CIC Filter
    //
// ============================================================================
    cic_filter #(
        .DATA_WIDTH (DATA_WIDTH)
    ) u_stage3_cic (
        .clk        (clk),
        .rst_n      (rst_n),
        .rate_i     (r_cic_rate), // Connect to Pipelined Register
        .valid_i    (w_notch_valid),
        .data_i     (w_notch_data),
        .valid_o    (w_final_valid),
        .data_o     (w_final_data)
    );

    //
// ============================================================================
    // 2. Output Pipeline Registers
    //    Improves timing for signals leaving the chip to PMODs/LEDs
    //
// ============================================================================
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            valid_o <= 0;
            data_o  <= 0;
        end else begin
            valid_o <= w_final_valid;
            data_o  <= w_final_data;
        end
    end

endmodule
```

TestBench

```
`timescale 1ns / 1ps

module tb_dfe_top;

    // Parameters
    localparam int DATA_WIDTH = 16;
```

```
    localparam real FS_IN = 9.0e6;
    localparam int NUM_SAMPLES = 4096;

    // Signals
    logic                     clk;
    logic                     rst_n;
    logic [4:0]               cic_rate;
    logic                     valid_i;
    logic signed [DATA_WIDTH-1:0] data_i;
    logic                     valid_o;
    logic signed [DATA_WIDTH-1:0] data_o;

    int fd;
    real t;
    real val_0_2m, val_1_0m, val_2_4m, val_4_0m, val_total;
    integer i;
    int temp_val;

    // DUT
    dfe_top #(
        .DATA_WIDTH(DATA_WIDTH)
    ) u_dut (
        .clk        (clk),
        .rst_n      (rst_n),
        .cic_rate_i (cic_rate),
        .valid_i    (valid_i),
        .data_i     (data_i),
        .valid_o    (valid_o),
        .data_o     (data_o)
    );

    // Clock (100 MHz)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // VCD
    initial begin
        $dumpfile("dfe_top.vcd");
        $dumpvars(0, tb_dfe_top);
    end

    // Stimulus
    initial begin
        rst_n    = 0;
```

```
        valid_i  = 0;
        data_i   = 0;
        cic_rate = 4;

        fd = $fopen("dfe_io.txt", "w");
        if (fd == 0) begin
            $display("Error: Could not open output file.");
            $finish;
        end

        repeat(10) @(posedge clk);
        rst_n = 1;
        repeat(10) @(posedge clk);

        $display("Starting DFE Simulation...");
        $display("Input Fs: 9 MHz (Simulated Rate)");

        // ------------------------------------------------------------
        // DATA DRIVER LOOP
        // ------------------------------------------------------------
        for (i = 0; i < NUM_SAMPLES; i++) begin
            @(posedge clk);

            // 1. Calculate Signal Value
            t = real'(i) / FS_IN;
            val_0_2m = $sin(2.0 * 3.14159 * 0.2e6 * t);
            val_1_0m = $sin(2.0 * 3.14159 * 1.0e6 * t);
            val_2_4m = $sin(2.0 * 3.14159 * 2.4e6 * t);
            val_4_0m = $sin(2.0 * 3.14159 * 4.0e6 * t) * 0.5;

            val_total = val_0_2m + val_1_0m + val_2_4m + val_4_0m;
            temp_val = $rtoi(val_total * 0.25 * 32767.0); // Scale to Q1.15

            if (temp_val > 32767) temp_val = 32767;
            if (temp_val < -32768) temp_val = -32768;

            // 2. Drive Valid Pulse
            data_i  <= temp_val[15:0];
            valid_i <= 1'b1;

            // 3. Wait / Throttle
            // The Interpolator (L=2) needs 2 clock cycles to process 1
input.
            // If we drive every cycle, it resets and fails to interpolate.
            // We assert valid for 1 cycle, then deassert.
            @(posedge clk);
```

```
            valid_i <= 1'b0;

            // Wait an extra cycle to ensure the 2-phase filter finishes
            // (Total period = 2 clocks, Rate = 50 MHz vs 100 MHz clk)
            @(posedge clk);
        end

        @(posedge clk);
        valid_i = 0;
        data_i  = 0;

        repeat(200) @(posedge clk);
        $fclose(fd);
        $display("Simulation Finished.");
        $finish;
    end

    // File Write
    always @(posedge clk) begin
        if (rst_n) begin
            $fdisplay(fd, "%d, %d, %d, %d", valid_i, data_i, valid_o,
data_o);
        end
    end

endmodule
```

FPGA constraints

constraints1.xdc

output given to led J input from PMOD JA & JB

```
## This file is a specific .xdc for the DFE Top Level on Nexys4 DDR
## ------------------------------------------------------------------------
---------

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk}];
```

```
## Reset (Active Low - CPU_RESETN)
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
rst_n }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

## ------------------------------------------------------------------------
---------
## Configuration & Controls (Switches)
## ------------------------------------------------------------------------
---------

## CIC Decimation Rate (5 bits) -> Switches [4:0]
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]

## Input Valid Signal -> Switch [15]
## Toggle this high to enable data processing
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
valid_i }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## ------------------------------------------------------------------------
---------
## Output Data Visualization (LEDs)
## ------------------------------------------------------------------------
---------

## Output Data (16 bits) -> LEDs [15:0]
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {
data_o[0] }];  #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {
data_o[1] }];  #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {
data_o[2] }];  #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {
data_o[3] }];  #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {
data_o[4] }];  #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {
data_o[5] }];  #IO_L18N_T2_A11_D27_14 Sch=led[5]
```

```
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {
data_o[6] }];  #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {
data_o[7] }];  #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {
data_o[8] }];  #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {
data_o[9] }];  #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {
data_o[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports {
data_o[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {
data_o[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {
data_o[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports {
data_o[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports {
data_o[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

## Output Valid -> RGB LED 16 (Green Channel)
## Will light up green when valid output data is present
set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports {
valid_o }]; #IO_L10P_T1_D14_14 Sch=led16_g

## ----------------------------------------------------------------------------
---------
## Input Data Mapping (PMOD Headers)
## ----------------------------------------------------------------------------
---------
## Since data_i is 16 bits, we map it to PMOD JA (8 bits) and PMOD JB (8
bits).
## This is where you would connect your external signal source (ADC).

## PMOD JA -> data_i[7:0]
set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[0] }]; #IO_L20N_T3_A19_15 Sch=ja[1]
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[1] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[2] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[3] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[4] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
```

```
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[5] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[6] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[7] }]; #IO_L22P_T3_A17_15 Sch=ja[10]

## PMOD JB -> data_i[15:8]
set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports {
data_i[8] }];  #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[9] }];  #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[10] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports {
data_i[11] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[12] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports {
data_i[13] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports {
data_i[14] }]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[15] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]

## ------------------------------------------------------------------------
---------
## Configuration Properties
## ------------------------------------------------------------------------
---------
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

constraints2.xdc

output given to PMOD JC & JC input from PMOD JA & JB

```
## This file is a specific .xdc for the DFE Top Level on Nexys4 DDR
## ------------------------------------------------------------------------
---------

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
```

```
clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk}];

## Reset (Active Low - CPU_RESETN)
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
rst_n }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

## ------------------------------------------------------------------------
---------
## Configuration & Controls (Switches)
## ------------------------------------------------------------------------
---------

## CIC Decimation Rate (5 bits) -> Switches [4:0]
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
cic_rate_i[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]

## Input Valid Signal -> Switch [15]
## Toggle this high to enable data processing
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
valid_i }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## ------------------------------------------------------------------------
---------
## Input Data Mapping (PMOD Headers JA & JB)
## ------------------------------------------------------------------------
---------
## PMOD JA -> data_i[7:0] (Low Byte)
set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[0] }]; #IO_L20N_T3_A19_15 Sch=ja[1]
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[1] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[2] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[3] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports {
```

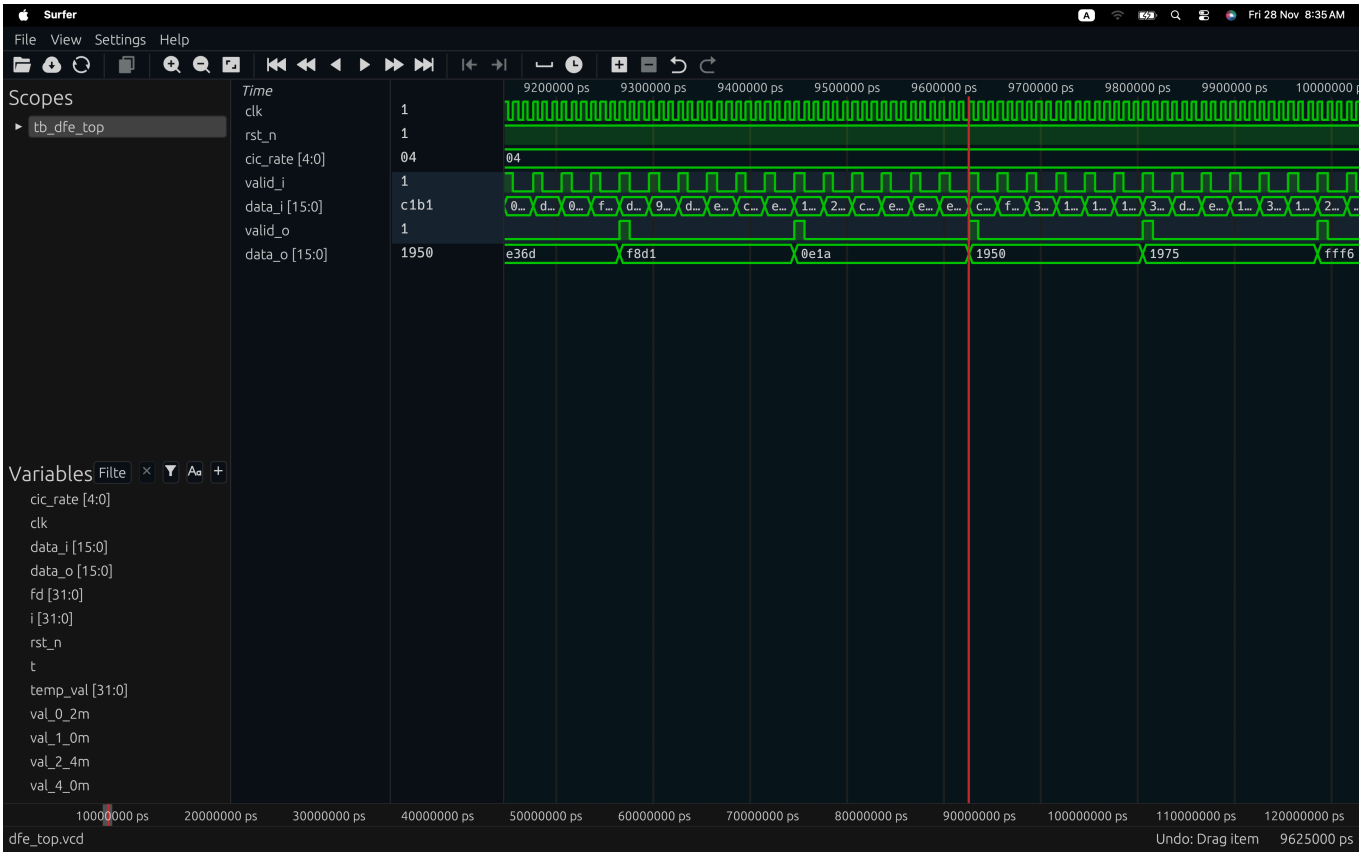
```
data_i[4] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports {
data_i[5] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[6] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {
data_i[7] }]; #IO_L22P_T3_A17_15 Sch=ja[10]

## PMOD JB -> data_i[15:8] (High Byte)
set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports {
data_i[8] }];  #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[9] }];  #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[10] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports {
data_i[11] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[12] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports {
data_i[13] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports {
data_i[14] }]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports {
data_i[15] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]

## ------------------------------------------------------------------------
---------
## Output Data Mapping (PMOD Headers JC & JD)
## ------------------------------------------------------------------------
---------
## NOTE: Output moved from LEDs to PMODs for Logic Analyzer connection.

## PMOD JC -> data_o[7:0] (Low Byte)
set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports {
data_o[0] }]; #IO_L23N_T3_35 Sch=jc[1]
set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 } [get_ports {
data_o[1] }]; #IO_L19N_T3_VREF_35 Sch=jc[2]
set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports {
data_o[2] }]; #IO_L22N_T3_35 Sch=jc[3]
set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports {
data_o[3] }]; #IO_L19P_T3_35 Sch=jc[4]
set_property -dict { PACKAGE_PIN E7    IOSTANDARD LVCMOS33 } [get_ports {
data_o[4] }]; #IO_L6P_T0_35 Sch=jc[7]
set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports {
data_o[5] }]; #IO_L22P_T3_35 Sch=jc[8]
```

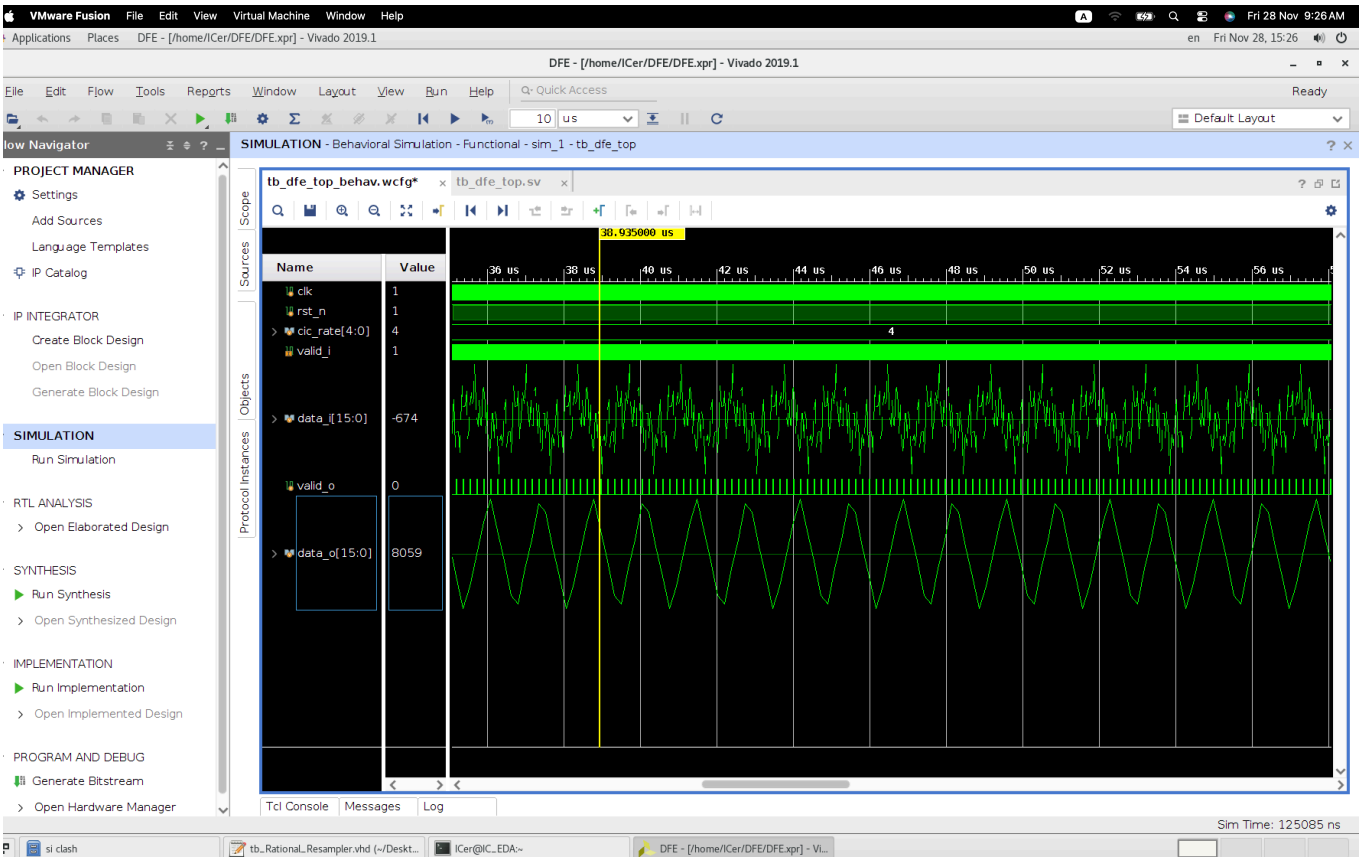
```
set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 } [get_ports {
data_o[6] }]; #IO_L21P_T3_DQS_35 Sch=jc[9]
set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 } [get_ports {
data_o[7] }]; #IO_L5P_T0_AD13P_35 Sch=jc[10]

## PMOD JD -> data_o[15:8] (High Byte)
set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports {
data_o[8] }];  #IO_L21N_T3_DQS_35 Sch=jd[1]
set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports {
data_o[9] }];  #IO_L17P_T2_35 Sch=jd[2]
set_property -dict { PACKAGE_PIN G1    IOSTANDARD LVCMOS33 } [get_ports {
data_o[10] }]; #IO_L17N_T2_35 Sch=jd[3]
set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports {
data_o[11] }]; #IO_L20N_T3_35 Sch=jd[4]
set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports {
data_o[12] }]; #IO_L15P_T2_DQS_35 Sch=jd[7]
set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports {
data_o[13] }]; #IO_L20P_T3_35 Sch=jd[8]
set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 } [get_ports {
data_o[14] }]; #IO_L15N_T2_DQS_35 Sch=jd[9]
set_property -dict { PACKAGE_PIN F3    IOSTANDARD LVCMOS33 } [get_ports {
data_o[15] }]; #IO_L13N_T2_MRCC_35 Sch=jd[10]

## ------------------------------------------------------------------------
---------
## Output Valid Indicator
## ------------------------------------------------------------------------
---------
## Output Valid -> RGB LED 16 (Green Channel)
set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports {
valid_o }]; #IO_L10P_T1_D14_14 Sch=led16_g

## ------------------------------------------------------------------------
---------
## Configuration Properties
## ------------------------------------------------------------------------
---------
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

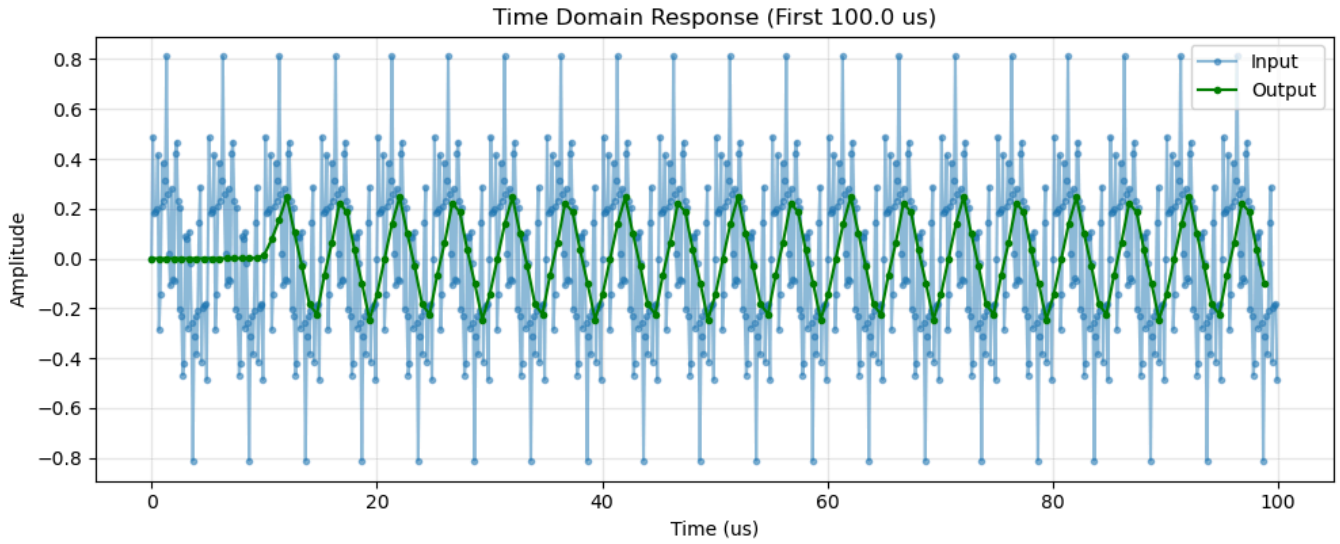
Results

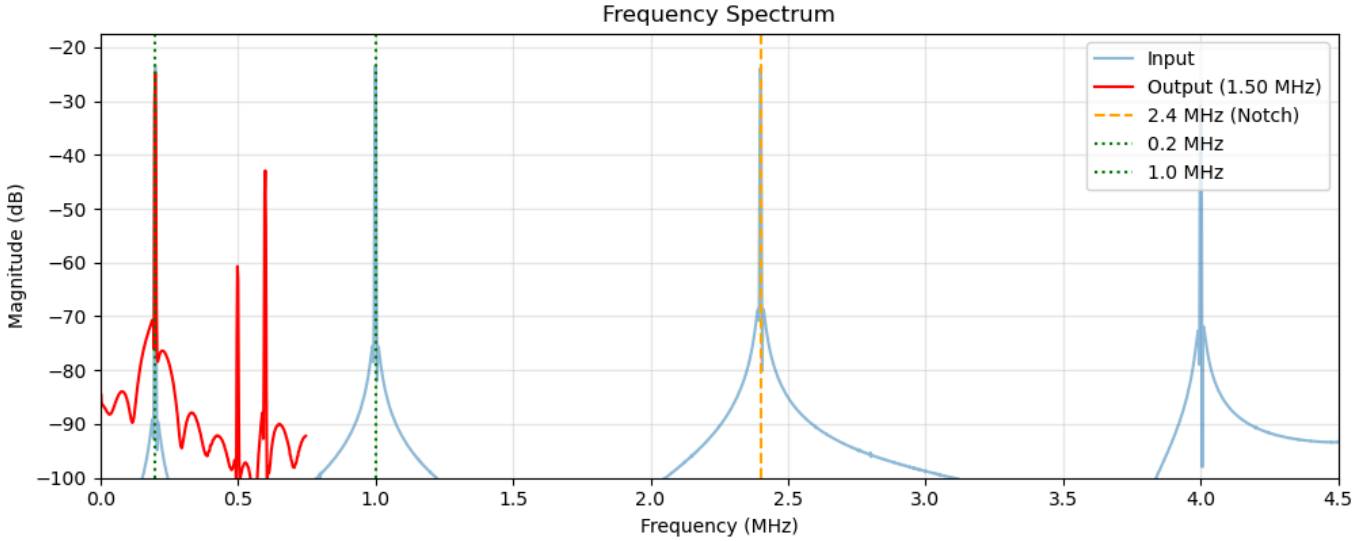
```
hazemysr@MacPro ~/Desktop/projects/DFE/DFE_top_level % make view_dfe
--- Running DFE Simulation ---
```

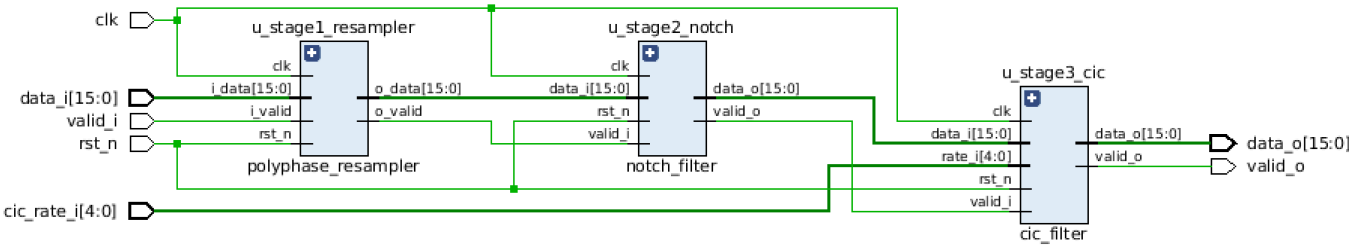
```
vvp dfe.vvp
VCD info: dumpfile dfe_top.vcd opened for output.
Starting DFE Simulation...
Input Fs: 9 MHz (Simulated Rate)
Simulation Finished.
tb_dfe_top.sv:112: $finish called at 125085000 (1ps)
--- Opening Waveform ---
```

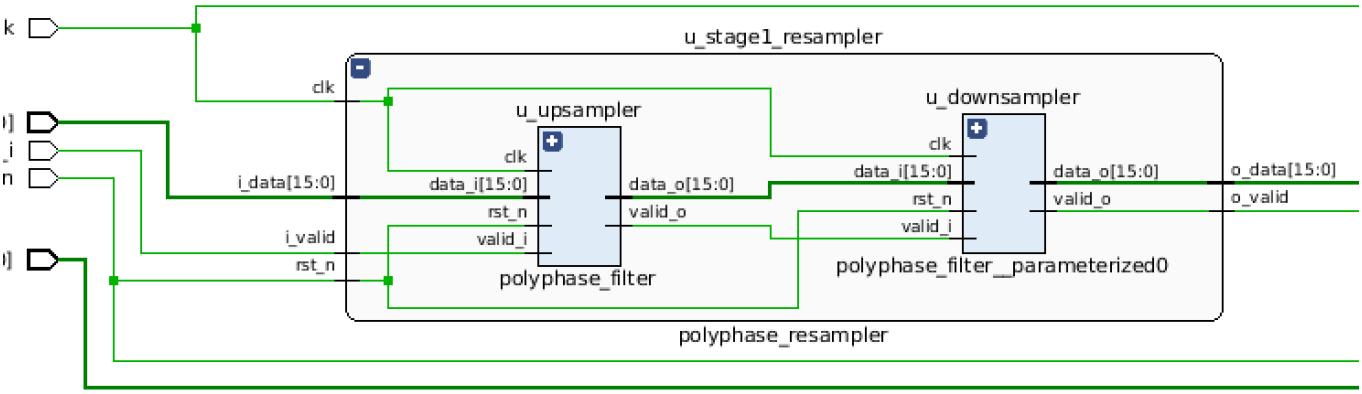


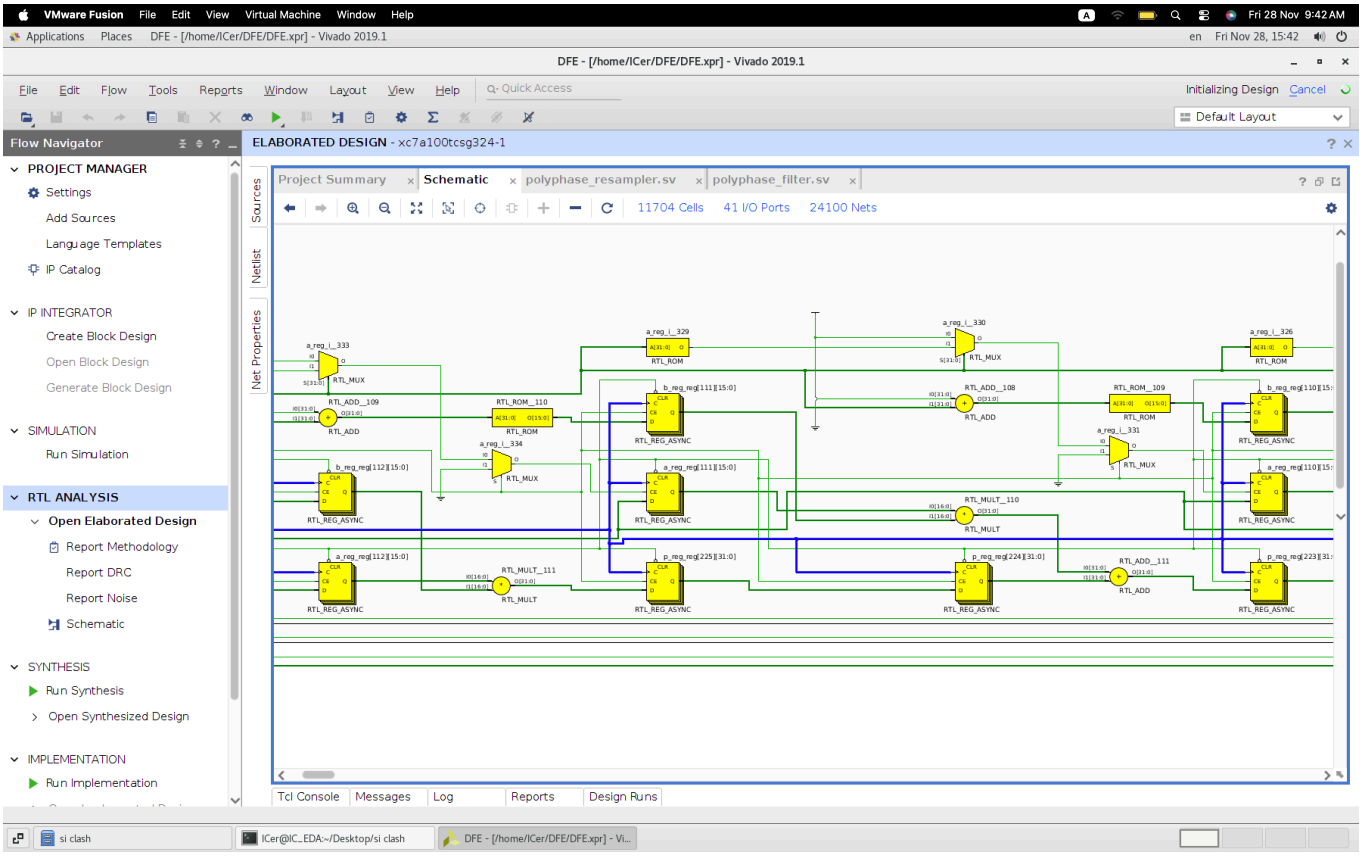

VMware Fusion File Edit View Virtual Machine Window Help
Fri 28 Nov 9:26 AM
Applications Places DFE - [/home/ICer/DFE/DFE.xpr] - Vivado 2019.1
en Fri Nov 28, 15:26
DFE - [/home/ICer/DFE/DFE.xpr] - Vivado 2019.1
File Edit Flow Tools Reports Window Layout View Run Help
Quick Access
Ready
Default Layout
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
SIMULATION - Behavioral Simulation - Functional - sim_1 - tb_dfe_top
tb_dfe_top_behav.wcfg*
tb_dfe_top.sv
Scope
Sources
Objects
Protocol Instances
38.935000 us
Name
Value
clk
1
rst_n
1
cic_rate[4:0]
4
valid_i
1
data_i[15:0]
-674
valid_o
0
data_o[15:0]
8059
36 us
38 us
40 us
42 us
44 us
46 us
48 us
50 us
52 us
54 us
56 us
Tcl Console
Messages
Log
Sim Time: 125085 ns
si clash
tb_Rational_Resampler.vhd (~/Deskt...
ICer@IC_EDA:~
DFE - [/home/ICer/DFE/DFE.xpr] - Vi...

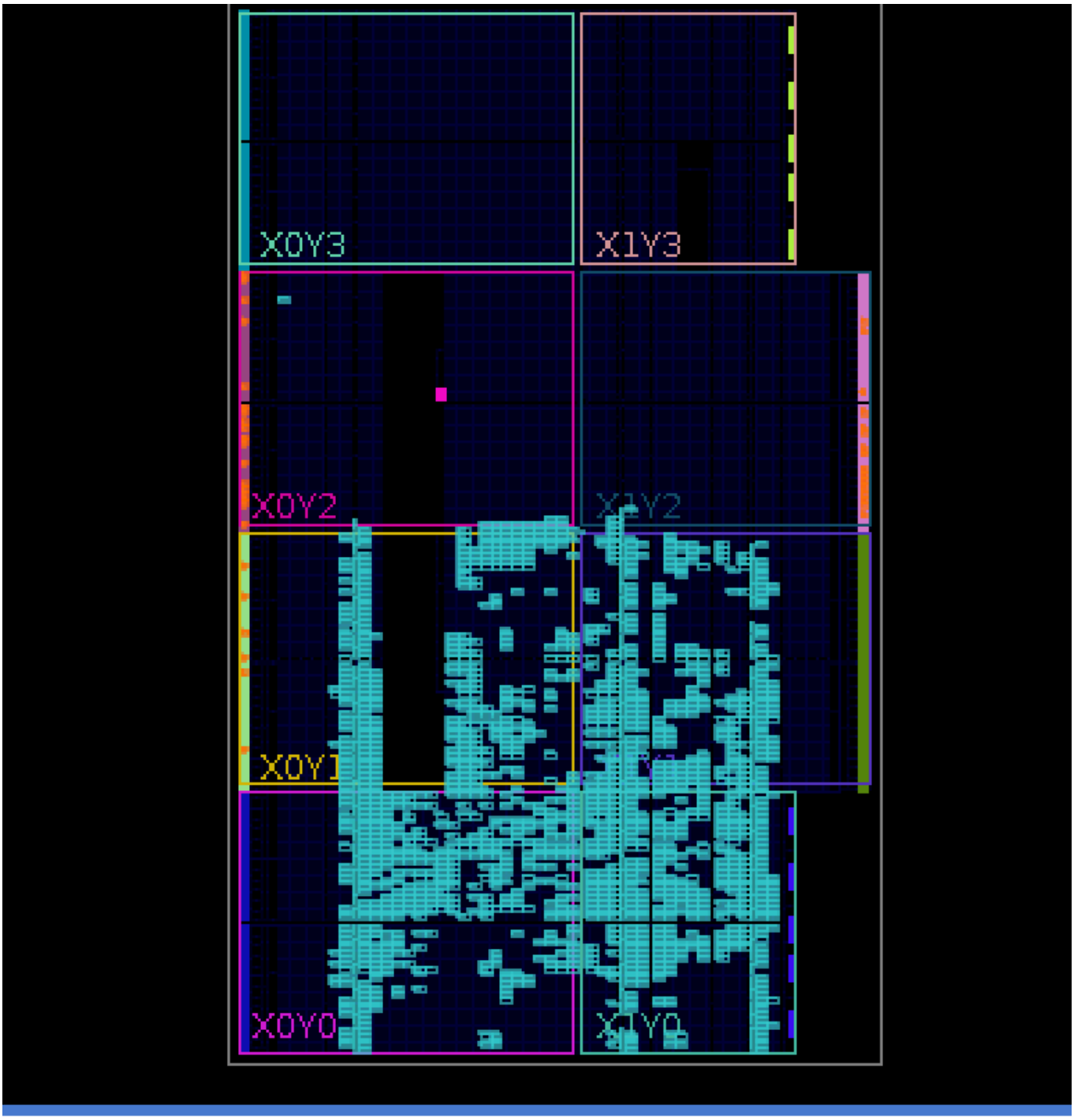

Time Domain Response (First 100.0 us)
Input
Output
Amplitude
Time (us)


Frequency Spectrum
Input
Output (1.50 MHz)
2.4 MHz (Notch)
0.2 MHz
1.0 MHz
Magnitude (dB)
Frequency (MHz)

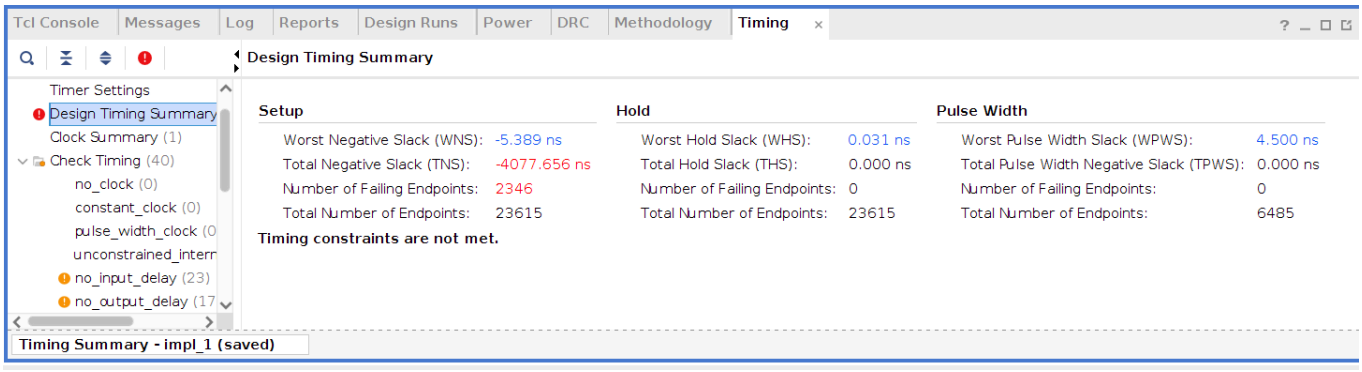

clk
data_i[15:0]
valid_i
rst_n
cic_rate_i[4:0]
u_stage1_resampler
i_data[15:0]
i_valid
o_data[15:0]
o_valid
polyphase_resampler
u_stage2_notch
data_i[15:0]
rst_n
valid_i
data_o[15:0]
valid_o
notch_filter
u_stage3_cic
data_i[15:0]
rate_i[4:0]
rst_n
valid_i
data_o[15:0]
valid_o
cic_filter
data_o[15:0]
valid_o


u_stage1_resampler
clk
u_upsampler
clk
i_data[15:0]
data_i[15:0]
data_o[15:0]
rst_n
valid_o
i_valid
valid_i
rst_n
polyphase_filter
u_downsampler
clk
data_i[15:0]
data_o[15:0]
o_data[15:0]
rst_n
valid_o
o_valid
valid_i
polyphase_filter__parameterized0
polyphase_resampler

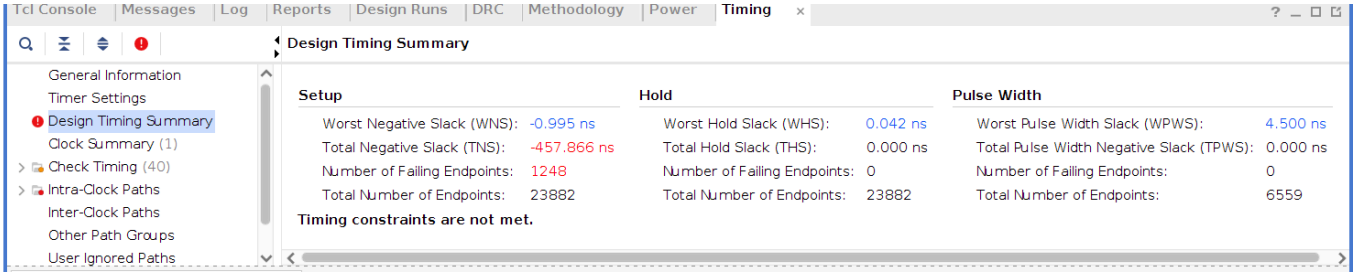

VMware Fusion File Edit View Virtual Machine Window Help
Fri 28 Nov 9:42 AM
Applications Places DFE - [/home/ICer/DFE/DFE.xpr] - Vivado 2019.1
en Fri Nov 28, 15:42
DFE - [/home/ICer/DFE/DFE.xpr] - Vivado 2019.1
File Edit Flow Tools Reports Window Layout View Help
Quick Access
Initializing Design Cancel
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
Report Methodology
Report DRC
Report Noise
Schematic
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
ELABORATED DESIGN - xc7a100tcsg324-1
Project Summary
Schematic
polyphase_resampler.sv
polyphase_filter.sv
11704 Cells
41 I/O Ports
24100 Nets
Sources
Netlist
Net Properties
Tcl Console
Messages
Log
Reports
Design Runs
si clash
ICer@IC_EDA:~/Desktop/si clash
DFE - [/home/ICer/DFE/DFE.xpr] - Vi...

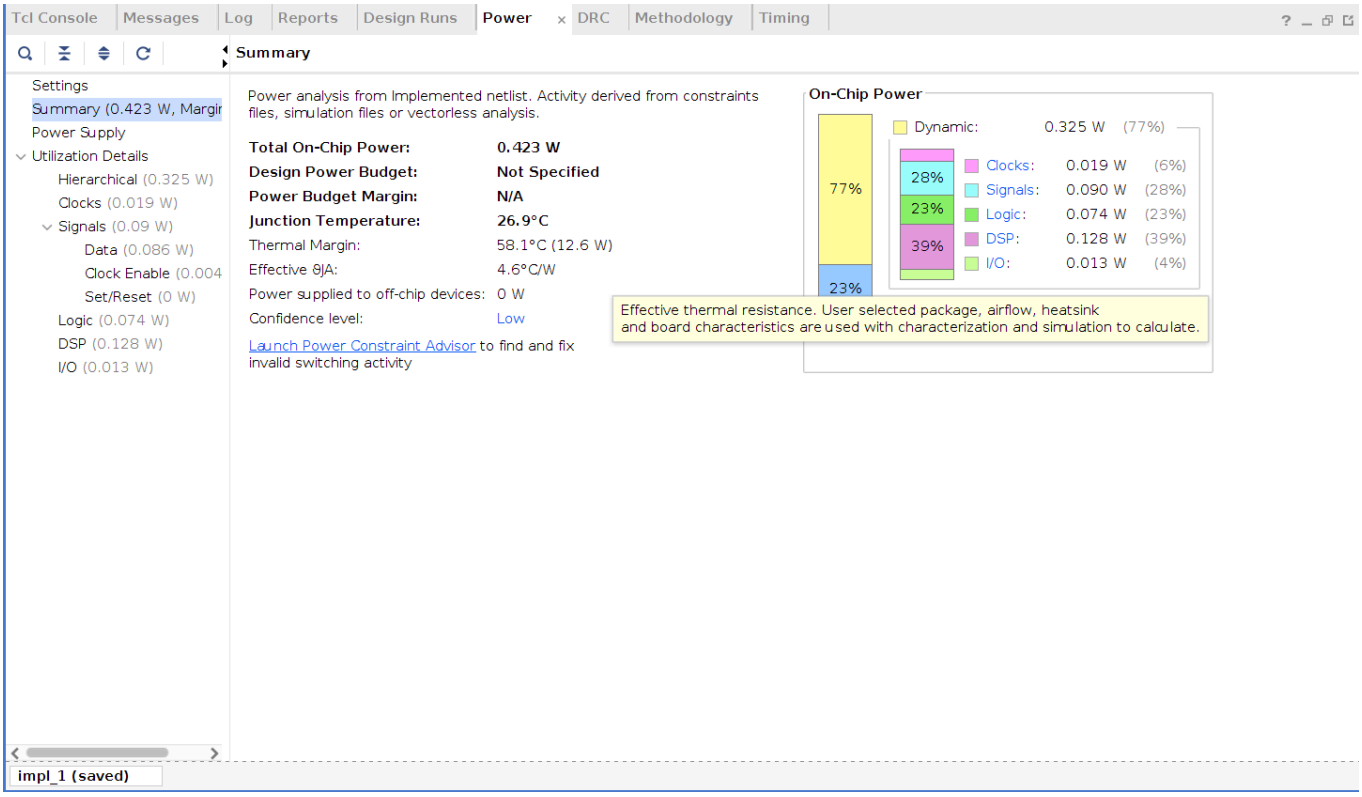


before NOTCH pipelining



after pipelining to further solve it you can run DFE at 50 MHz it won’t cause any issue neitjr speed nor functionality it will even have lower power





solution to timing violations

after creating an clk divider with 100MHz in 75MHz out

our timing violations disappeared