REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS
ECE 387 – LECTURE 13

PROF. DAVID ZARETSKY

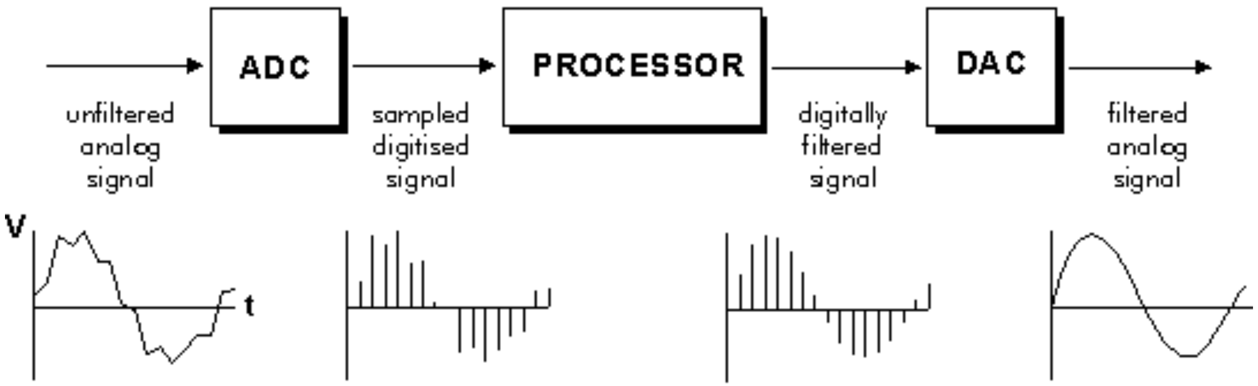
DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

- Digital Signal Processing
- FM Radio

DIGITAL SIGNAL PROCESSING BASICS

- A basic DSP system is composed of:
  - An ADC providing digital samples of an analog input
  - A Digital Processing system (μP/ASIC/FPGA)
  - A DAC converting processed samples to analog output
  - Real-time signal processing: All processing operation must be complete between two consecutive samples



TIME AND FREQUENCY DOMAINS

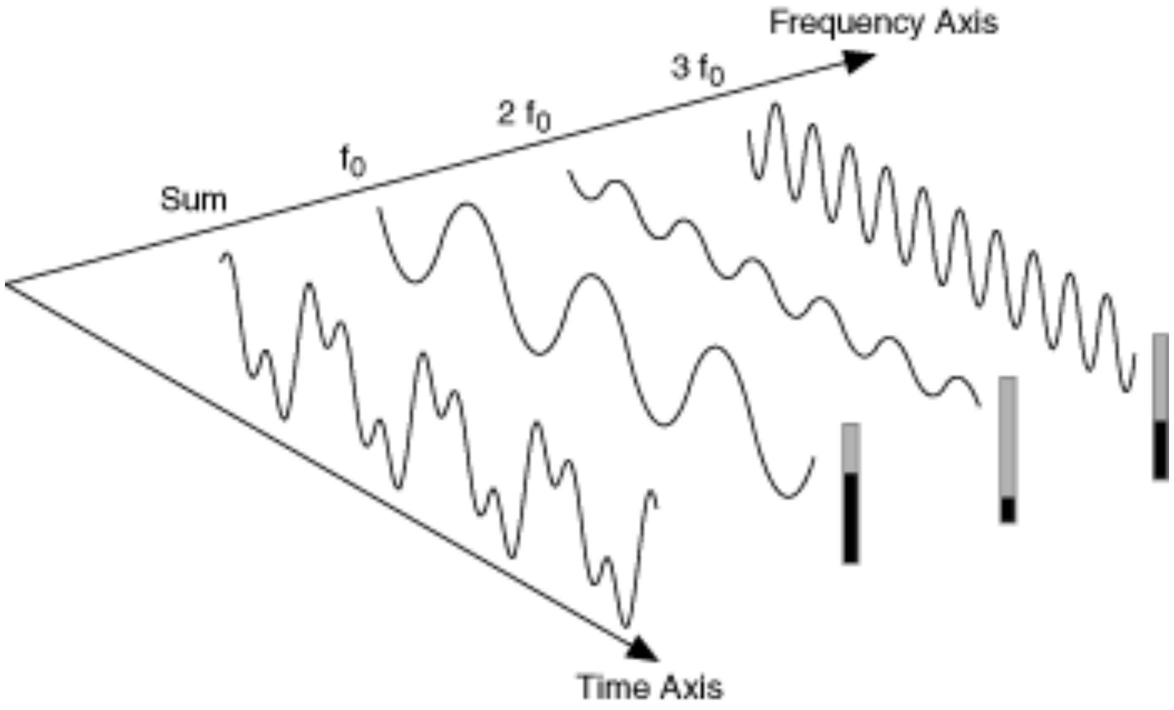
- The time-domain representation gives the amplitudes of signals at the instants of time during which it was sampled.
- Fourier's theorem states that any waveform in the time domain can be represented by the weighted sum of sines and cosines.
- The same waveform then can be represented in the frequency domain as a pair of amplitude and phase values at each component frequency.
- You can generate any waveform by adding sine waves, each with a particular amplitude and phase.

THE FREQUENCY DOMAIN

- The frequency domain does not carry any information that is not in the time domain.
- The power in the frequency domain is that it is simply another way of looking at signal information.
- Any operation or inspection done in one domain is equally applicable to the other domain, except that usually one domain makes a particular operation or inspection much easier than in the other domain.
- Frequency domain information is extremely important and useful in signal processing.

FREQUENCY VS TIME DOMAIN

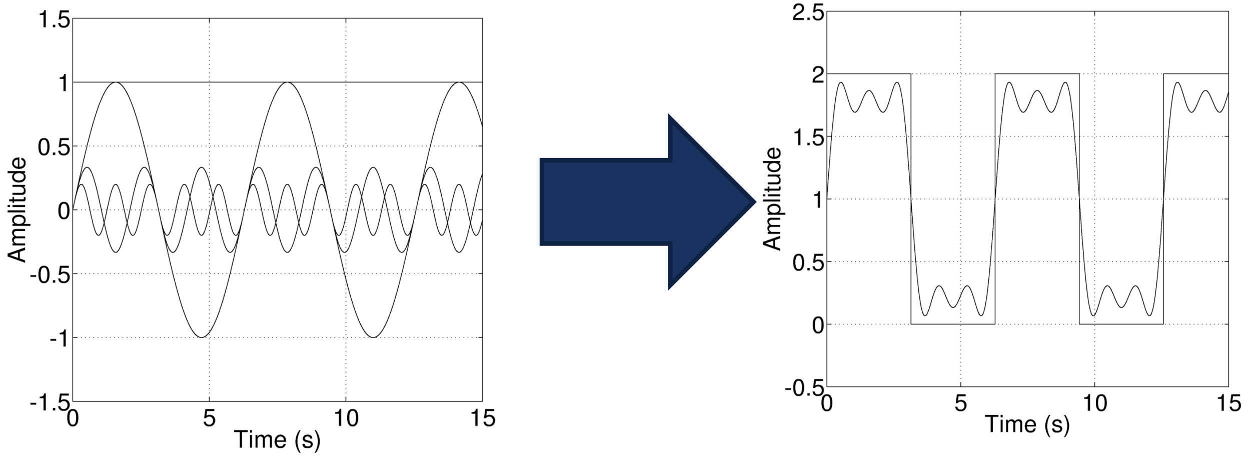
- The figure shows single frequency components spread out in the time domain, as distinct impulses in the frequency domain.
- The amplitude of each frequency line is the amplitude of the time waveform for that frequency component.



THE FOURIER SERIES

- C_k is frequency domain amplitude and phase representation
- For the given value $x_p(t)$ (a square value), the sum of the first four terms of trigonometric Fourier series are:
  - $x_p(t) \approx 1.0 + \sin(t) + C_2\sin(3t) + \ C_3\sin(5t)$

Periodic signal expressed as infinite sum of sinusoids.



$$x_p(t) = \sum_{k=-\infty}^{\infty} c_k e^{jk\omega_0 t}, \quad \text{where}$$

$$c_k = \frac{1}{T_p} \int_{T_p} x_p(t) e^{-jk\omega_0 t}\, dt$$

Complex Numbers !

DIGITAL FILTERING

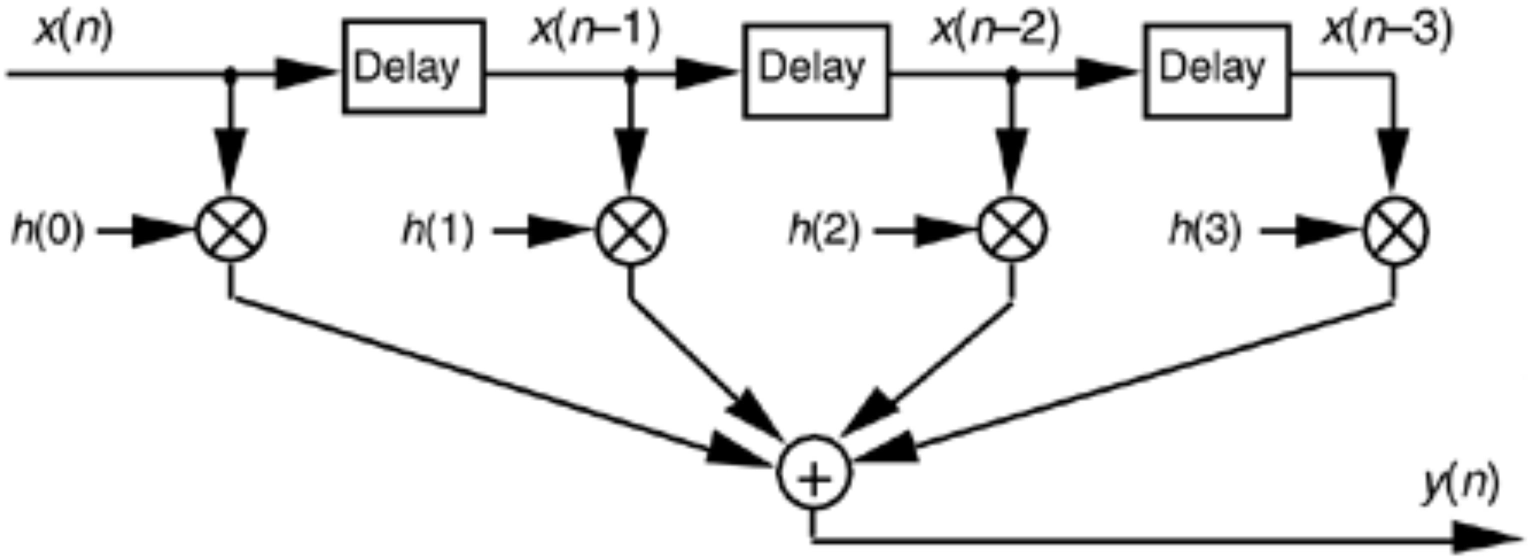
- Filters
  - Remove unwanted parts of the signal, such as random noise
  - Extract useful parts of the signal, such as the components lying within a certain frequency range
- Analog Filters
  - Input: electrical voltage or current which is the direct analogue of a physical quantity (sensor output)
  - Components: resistors, capacitors and op amps
  - Output: Filtered electrical voltage or current
  - Applications: noise reduction, video signal enhancement, graphic equalisers
- Digital Filters
  - Input: Digitized samples of analog input (requires ADC)
  - Components: Digital processor (PC/DSP/ASIC/FPGA)
  - Output: Filtered samples (requires DAC)
  - Applications: noise reduction, video signal enhancement, graphic equalisers

FAST FOURIER TRANSFORM (FFT)

- FFT is a digital implementation of the Fourier transform.
- FFT resolves a time waveform into its sinusoidal components.
- Converts time-domain data into the frequency spectrum of the data.
- FFT returns a discrete spectrum, in which the frequency content of the waveform is resolved into a finite number of frequency lines, or bins.
- FFT is a faster version of the Discrete Fourier Transform (DFT)
  - It utilizes some clever algorithms to do the same thing as the DTF, but in much less time.
  - Without a discrete-time to discrete-frequency transform we would not be able to compute the Fourier transform with a microprocessor or FPGA
- Use cases for Fourier Transform:
  - Analyze the frequency spectrum of audio data
  - Find the frequency components of a signal buried in noise

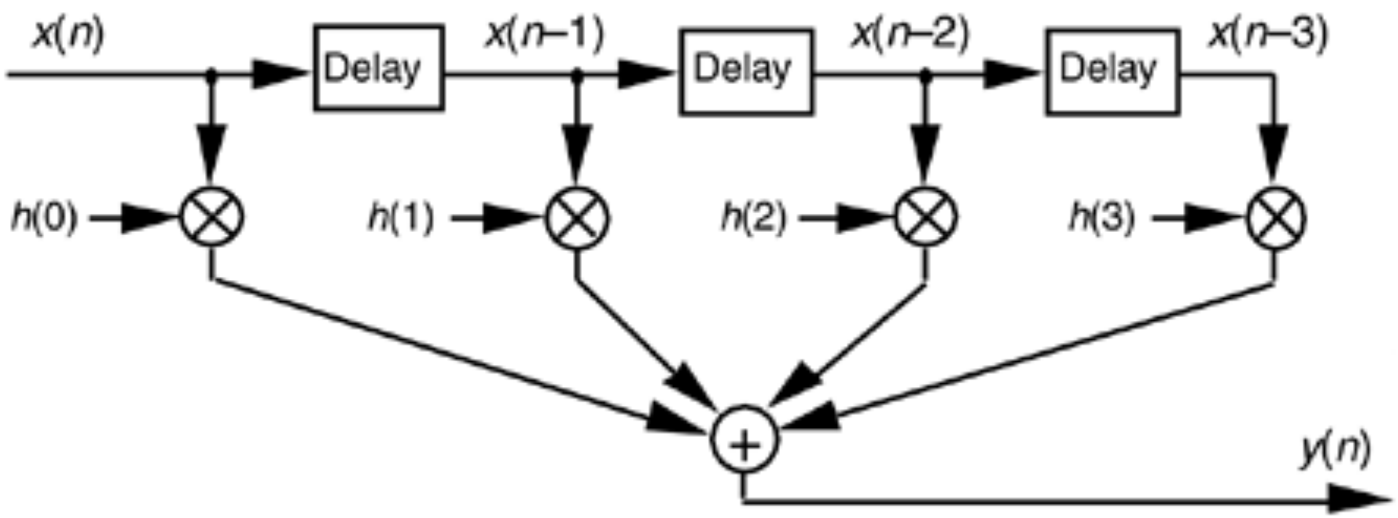
FINITE IMPULSE RESPONSE (FIR) FILTERS

- FIR filters use past inputs to calculate new output
- y(n) = h(0)*x(n) + h(1)*x(n-1) + h(2)*x(n-2) + h(3)*x(n-3)



FIR SOFTWARE IMPLEMENTATION

```
int yn=0;          //filter output initialization
short xdly[N+1];        //input delay samples array

void fir()
 {
      short i;
      yn=0;
      short h[N] = { //coefficients };
      xdly[0] = input_sample();
      for (i=0; i<N; i++)
         yn += (h[i]*xdly[i]);
      for (i=N-1; i>0; i--)
         xdly[i] = xdly[i-1];
       output_sample(yn >> 15);
}
```

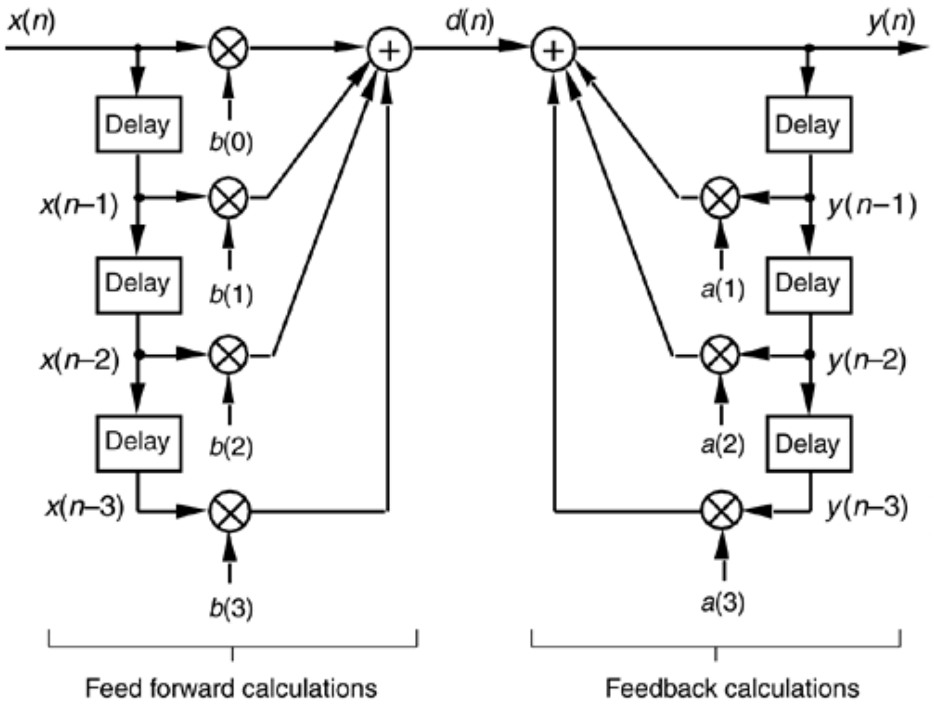


FIR HARDWARE IMPLEMENTATION IN VHDL

```
entity my_fir is
port (clk, rst: in std_logic;
   sample_in: in std_logic_vector(length-1 downto 0);
   sample_out: out std_logic_vector(length-1 downto 0)
   );
end entity my_fir;

architecture rtl of my_fir is
   type taps is array 0 to 3 of std_logic_vector(length-1 downto 0);
   constant h : taps := (…);  -- coeffients
     signal x : taps;  --past samples
   signal y: std_logic_vector(2*length-1 downto 0);
begin

   fir_process : process(x)
        variable y_tmp := std_logic_vector(2*length-1 downto 0);
    begin
         y_tmp := (others => ‘0’);
         for i in 0 to length-1 loop
             y_tmp := std_logic_vector(signed(h(i)) * signed(x(i)));
         end loop;
         y <= y_tmp;
    end process;

   clock_process : process (clk, rst)
   begin
      if rst=‘1’ then
         x <= (others => (others => ‘0’));
      elsif rising_edge(clk) then
         for i in length-1 downto 1 loop
             x(i) <= x(i-1);  -- shift
         end loop;
         x(0) <= sample_in;  -- new sample
         sample_out <= y(2*length-1 downto length);
      end if;
   end process;

end architecture rtl;
```

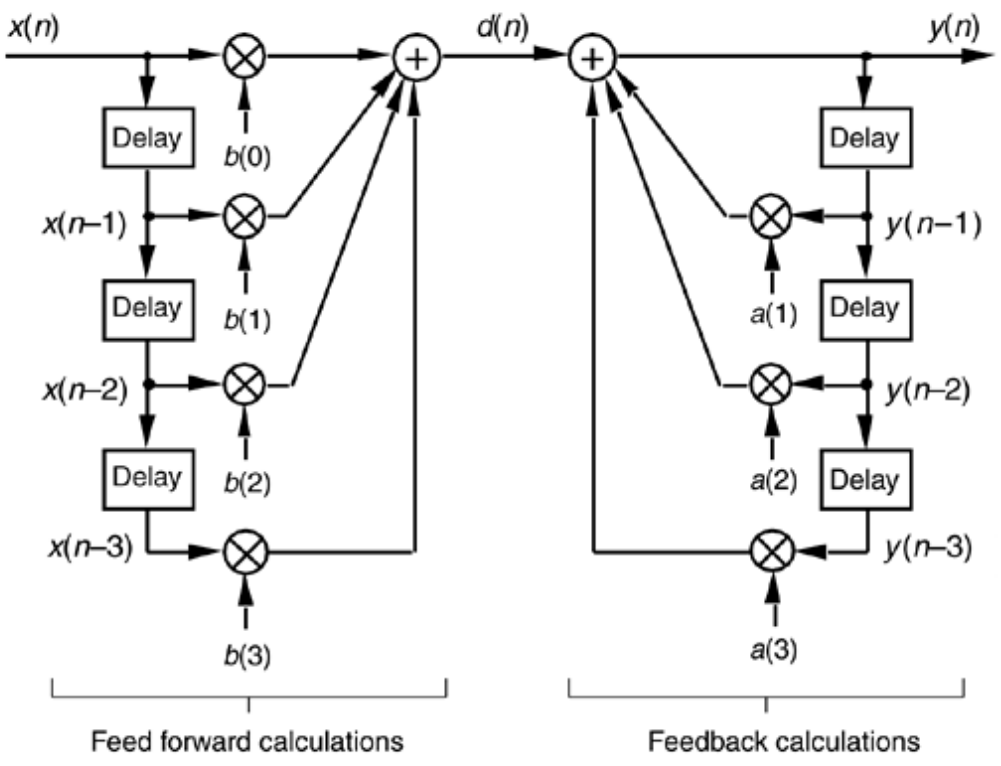
INFINITE IMPULSE RESPONSE (IIR) FILTERS

- IIR filters use past inputs and outputs to calculate new output
- y(n) = b(0)*x(n) + b(1)*x(n-1) + b(2)*x(n-2) + b(3)*x(n-3)
  + a(0)*y(n) + a(1)*y(n-1) + a(2)*y(n-2) + a(3)*y(n-3)



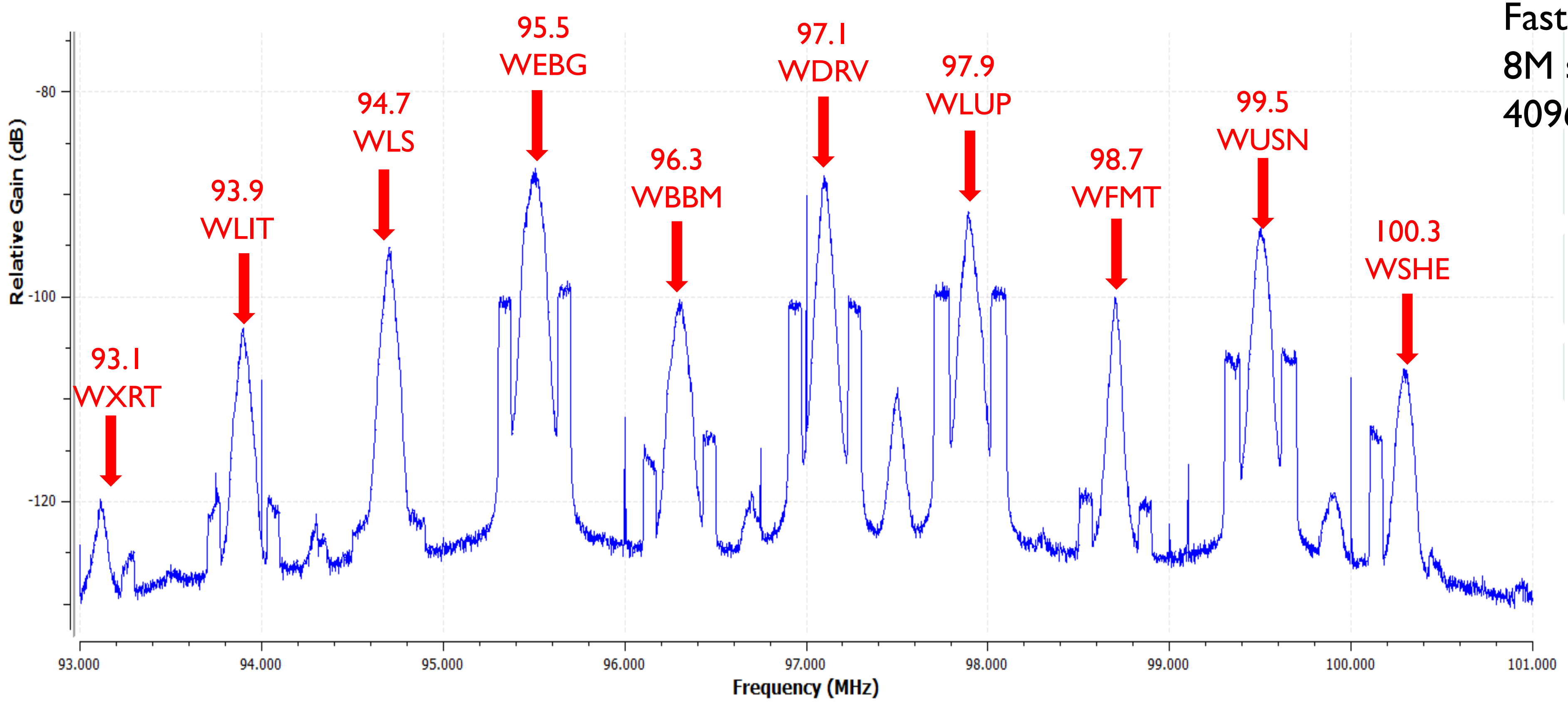
IIR SOFTWARE IMPLEMENTATION

```
int yn=0;          //filter output initialization
short xdly[N+1];  //input delay samples array
short ydly[M]; //output delay array

void iir()
{
     short i;
     yn=0;
     short a[N] = { //coefficients };
     short b[M] = { //coefficients };
     xdly[0]=input_sample();
     for (i=0; i<N; i++)
       yn += (b[i]*xdly[i]);
     for (i=0; i<M; i++)
        yn += (a[i]*ydly[i]);
     for (i=N-1; i>0; i--)
       xdly[i] = xdly[i-1];
         ydly[0] = yn >> 15;
     for (i=M-1; i>0; i--)
       ydly[i] = ydly[i-1];
      output_sample(yn >> 15);
}
```



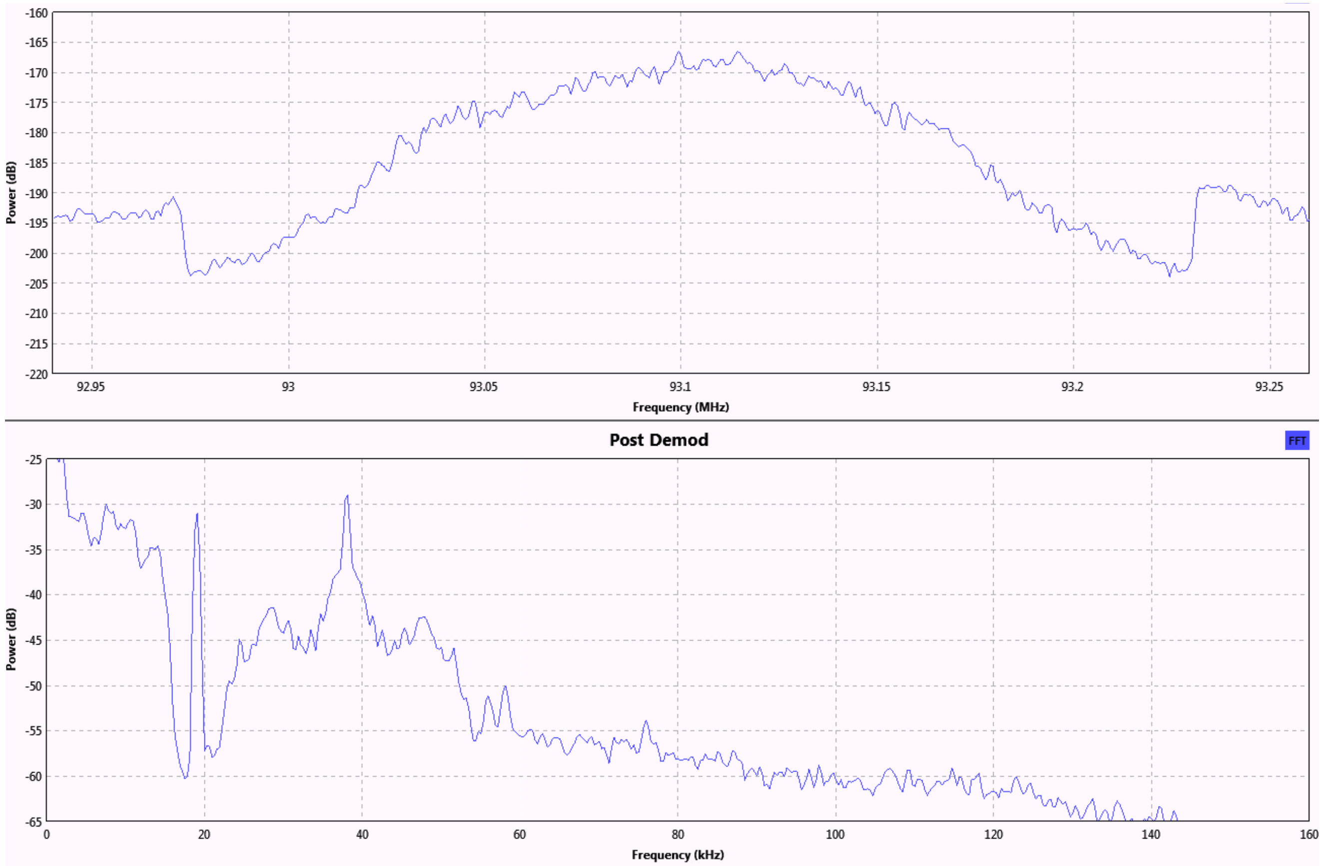
FM STEREO RADIO

FM FREQUENCY SPECTRUM



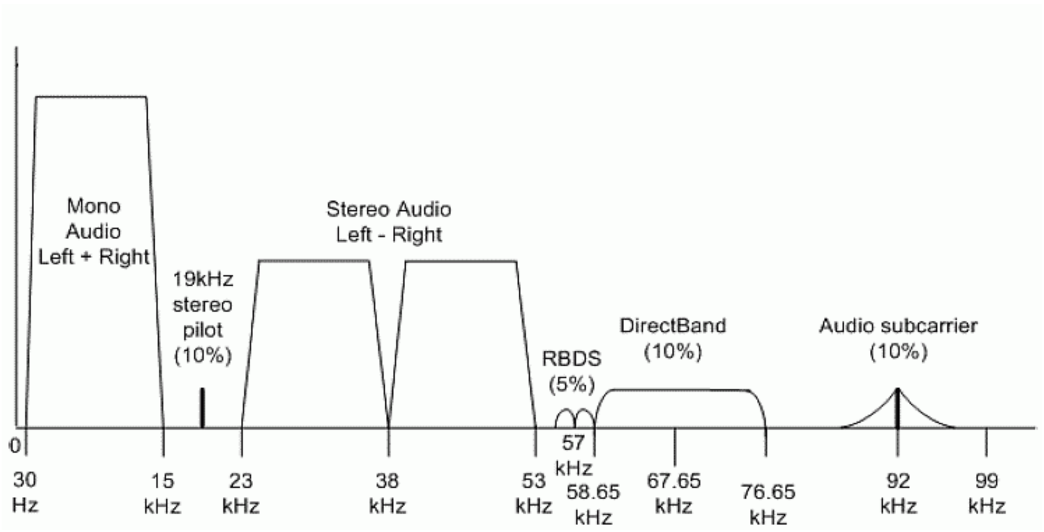
Fast Fourier Transform (FFT)
8M samples / sec
4096 Bins

DEMODULATING SIGNALS



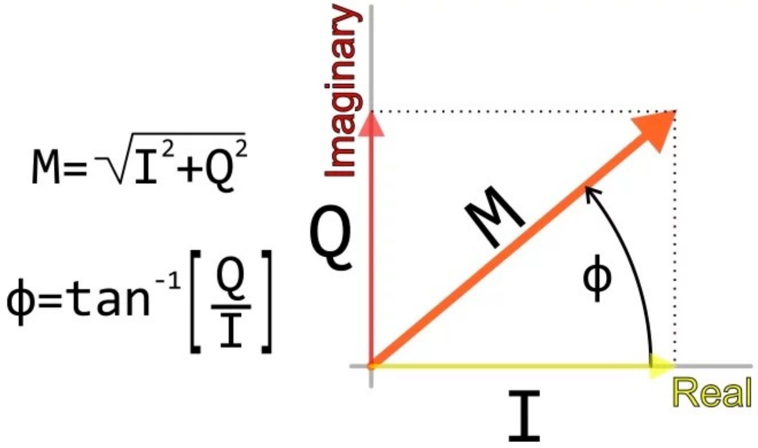
INTRODUCTION TO FM RECEIVERS

- $f(t) = k * m(t) + f_c$
  - m(t): the input signal
  - k: constant that controls the frequency sensitivity
  - fc : the frequency of the carrier
- To recover m(t), two steps are required:
  - Remove the carrier fc
  - Compute the instantaneous frequency of the baseband signal
- Left & Right audio channels encoded as (L+R) and (L-R)
  - L+R Mono Channel at f_c
  - L-R Channel at f_c + 38 kHz
  - Stereo Pilot tone at f_c + 19 kHz
  - Total 100 kHz spread

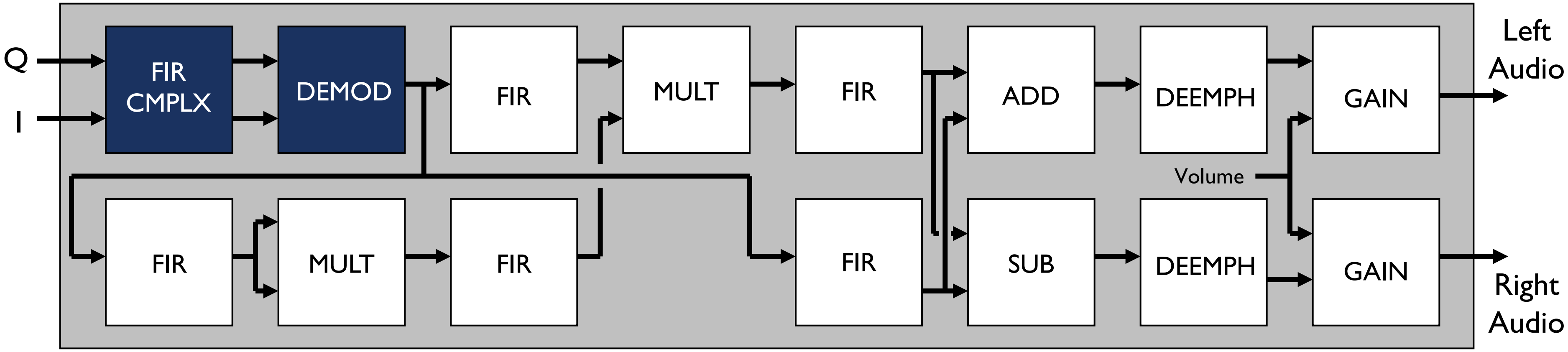


FM DEMODULATION

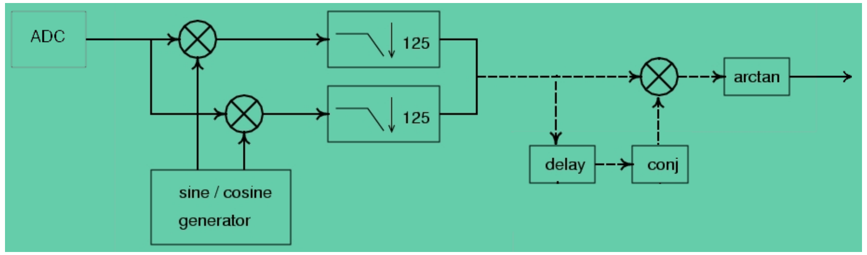
- Quadrature demodulation produces 2 baseband waveforms that convey the information that was encoded into the carrier of the received signal.
- I and Q waveforms are equivalent to the real and imaginary parts of a complex number, and are 90-degrees out of phase
- Separating I and Q in this way allows you to measure the relative phase of the components of the signal.
- The baseband waveform contained in the modulated signal corresponds to a magnitude+phase representation of I and Q signals.
    - The magnitude is $M = \sqrt{(I^2 + Q^2)}$
    - The angle of the I/Q data is $\varphi = \arctan(Q/I)$



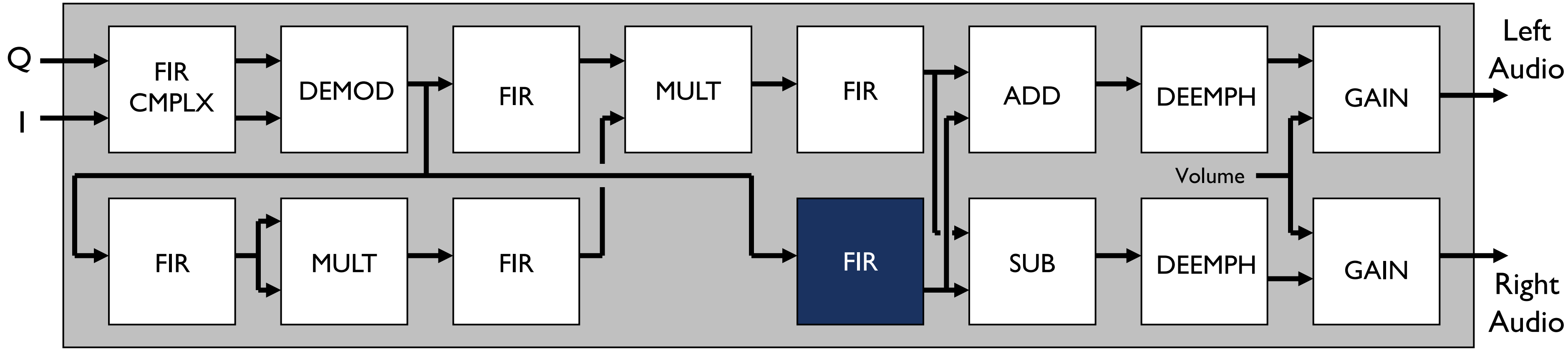
FM RADIO: DECONSTRUCTED



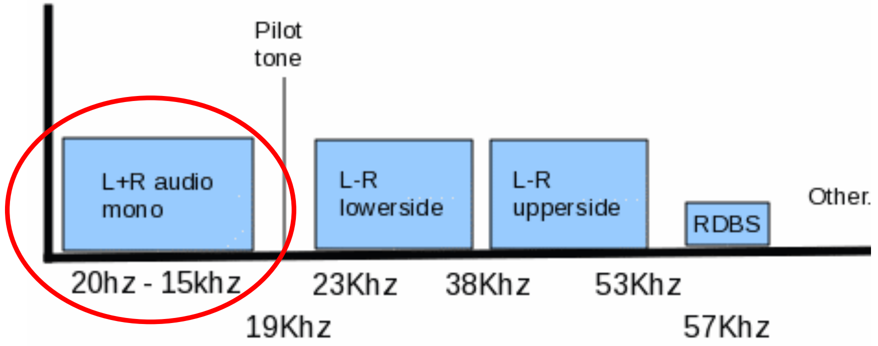
- Channel Filter
  - 20-tap FIR Complex Filter
  - Cuts off all frequencies above 80 kHz
- Demodulator
  - Differentiates freq by finding diff in angle of phase between consecutive I/Q samples
  - demod = k * atan(IQ_1 * conj(IQ_0))
  - k is the demodulator gain



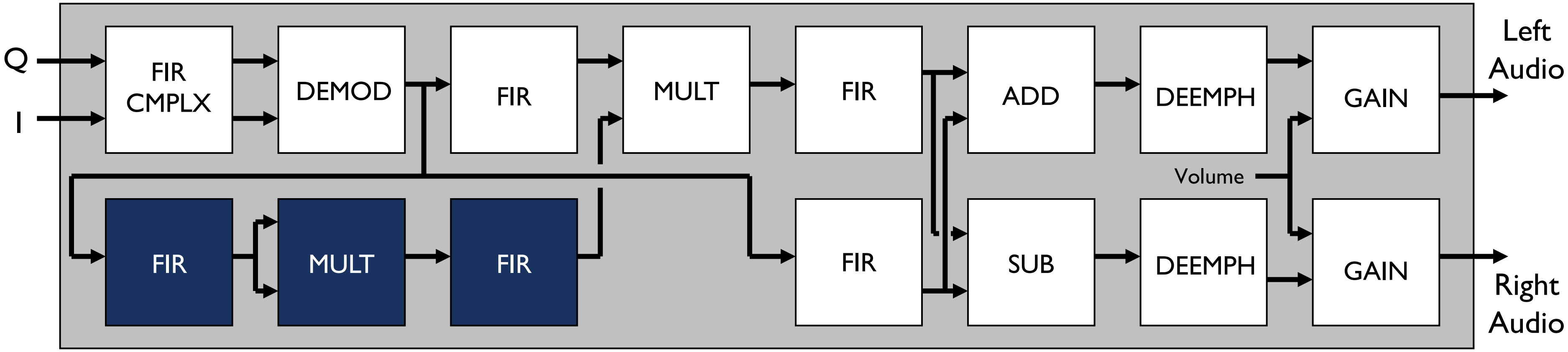
FM RADIO: DECONSTRUCTED



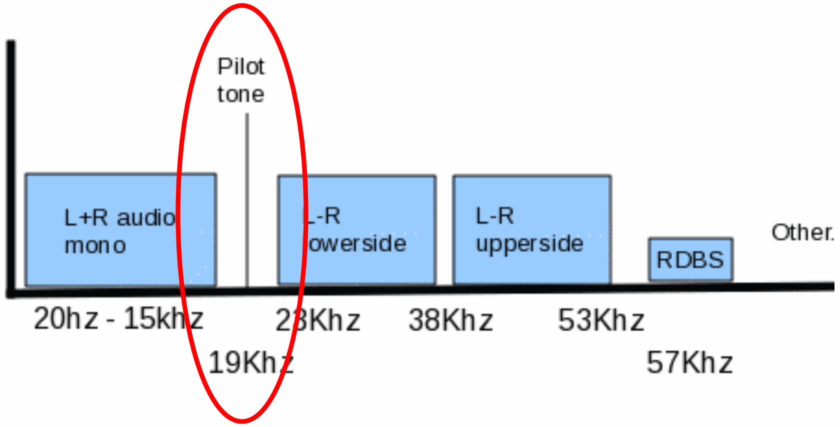
- L+R Channel Filter
  - Low-Pass 32-tap decimation FIR filter (decimation = 10)
  - Reduces sampling rate from 320 kHz to 32 kHz
  - Filters frequencies above 16 kHz



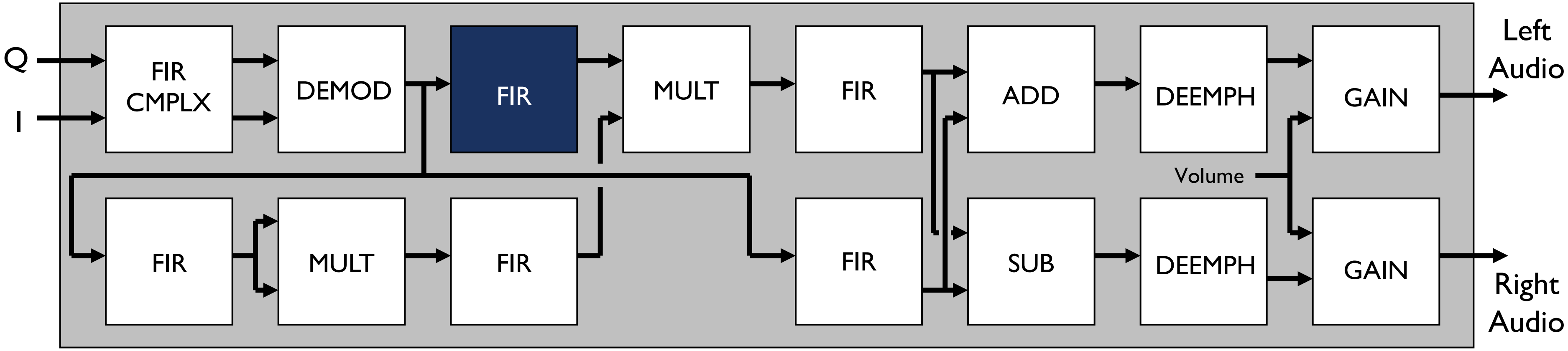
FM RADIO: DECONSTRUCTED



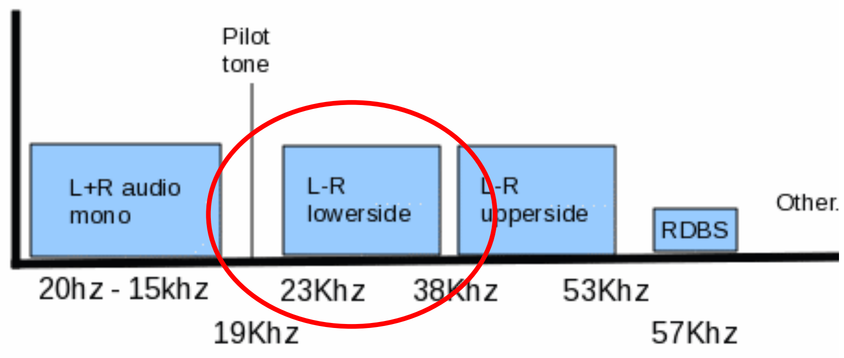
- Stereo Pilot Tone
  - Identifies whether a stereo signal exists
  - Band pass 32-tap FIR filter extracts the 19kHz pilot tone
  - The signal is squared to obtain a 38 kHz cosine
  - A high pass filter removes the tone at 0Hz created after the pilot tone is squared



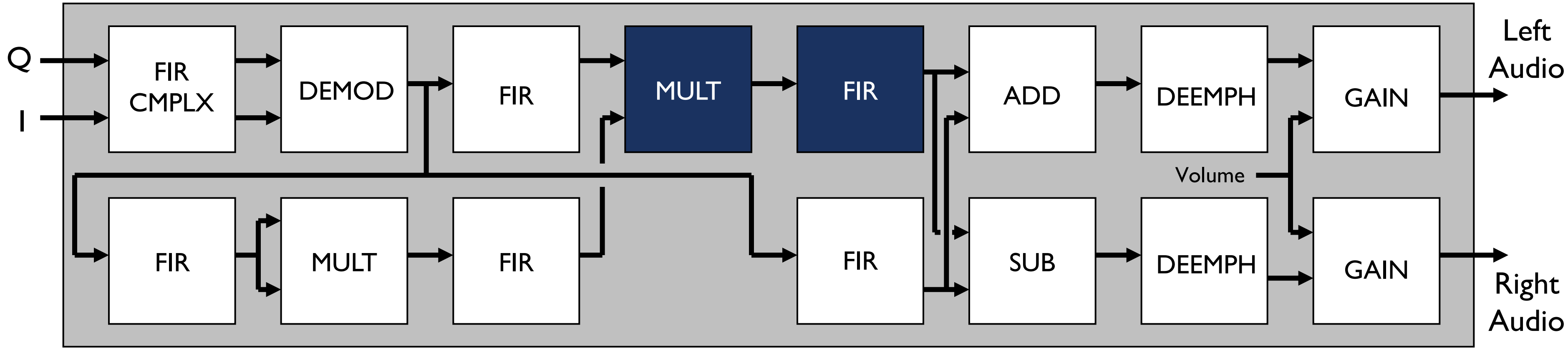
FM RADIO: DECONSTRUCTED



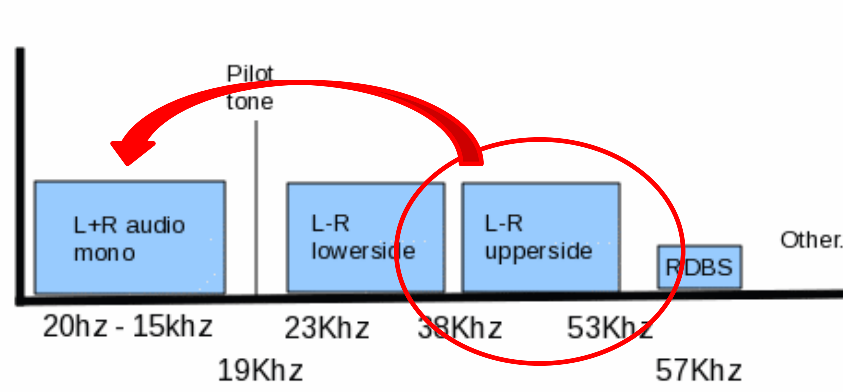
- L-R Channel Filter
  - Band-pass 32-tap FIR filter
  - Extracts the L-R (23-53 kHz) sub-carrier frequencies



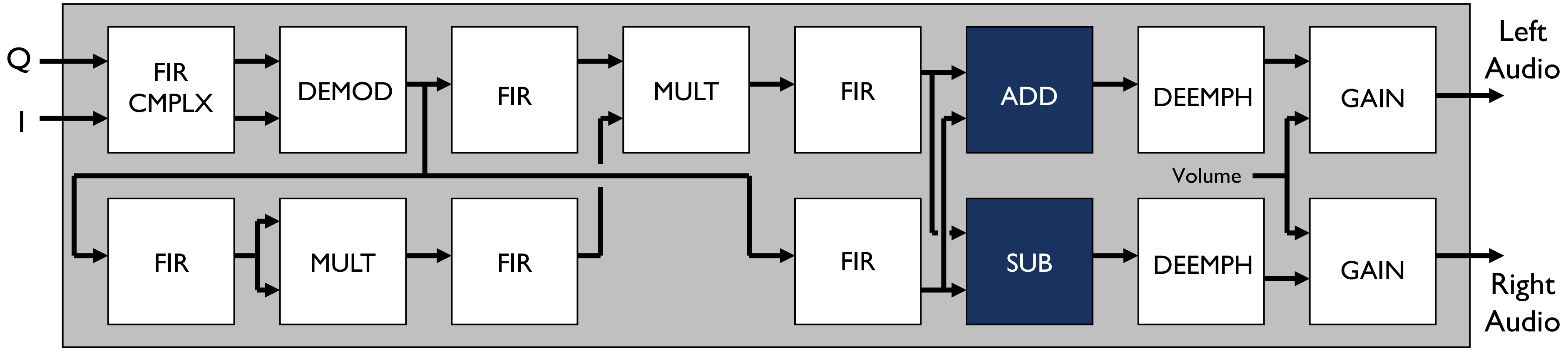
FM RADIO: DECONSTRUCTED



- L-R Channel Filter
  - L-R channel is multiplied by the squared pilot signal
  - L-R channel is demodulate from 38kHz to baseband
  - Low-Pass decimation FIR reduces sampling rate (decimation = 10)

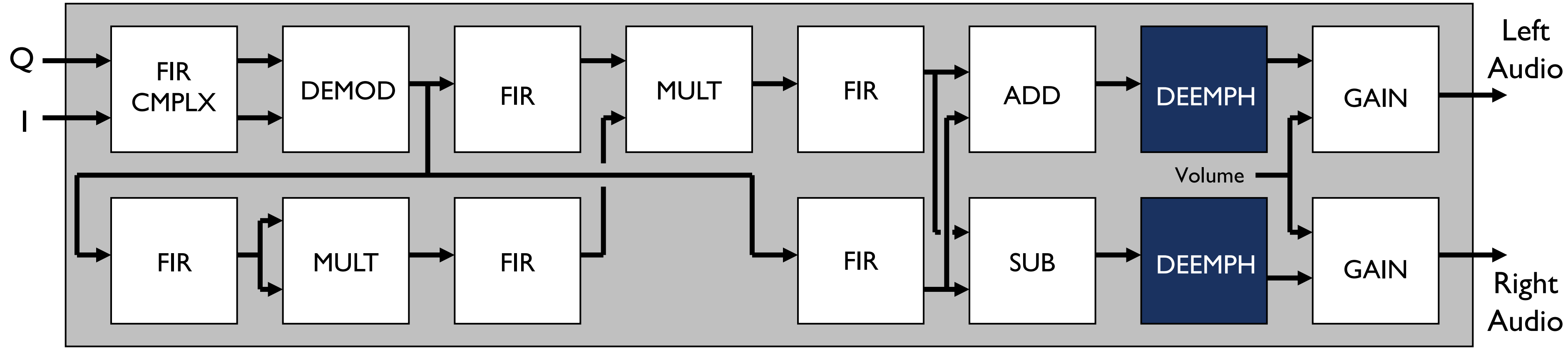


FM RADIO: DECONSTRUCTED



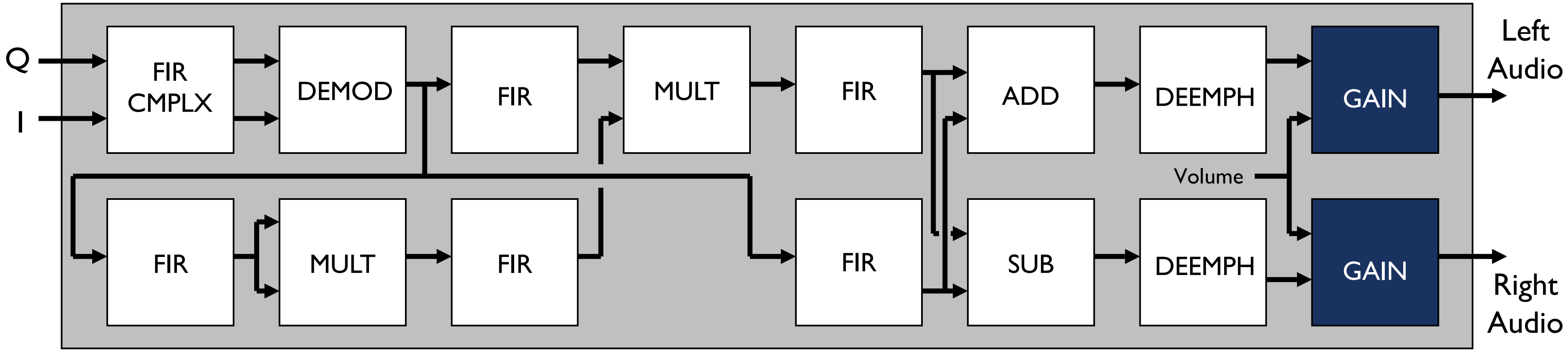
- Left & Right Channel Reconstruction
  - Left Channel: (L+R) + (L-R) = 2L
  - Right Channel: (L+R) – (L-R) = 2R

FM RADIO: DECONSTRUCTED



- De-emphasis
  - First-Order 2-Tap IIR Filter improves the signal-to-noise ratio (SNR)
  - Uses the transfer function H(s) = 1 / (1+s) for RC time circuit and bilinear z-transform to obtain coefficients (t = 75 us)

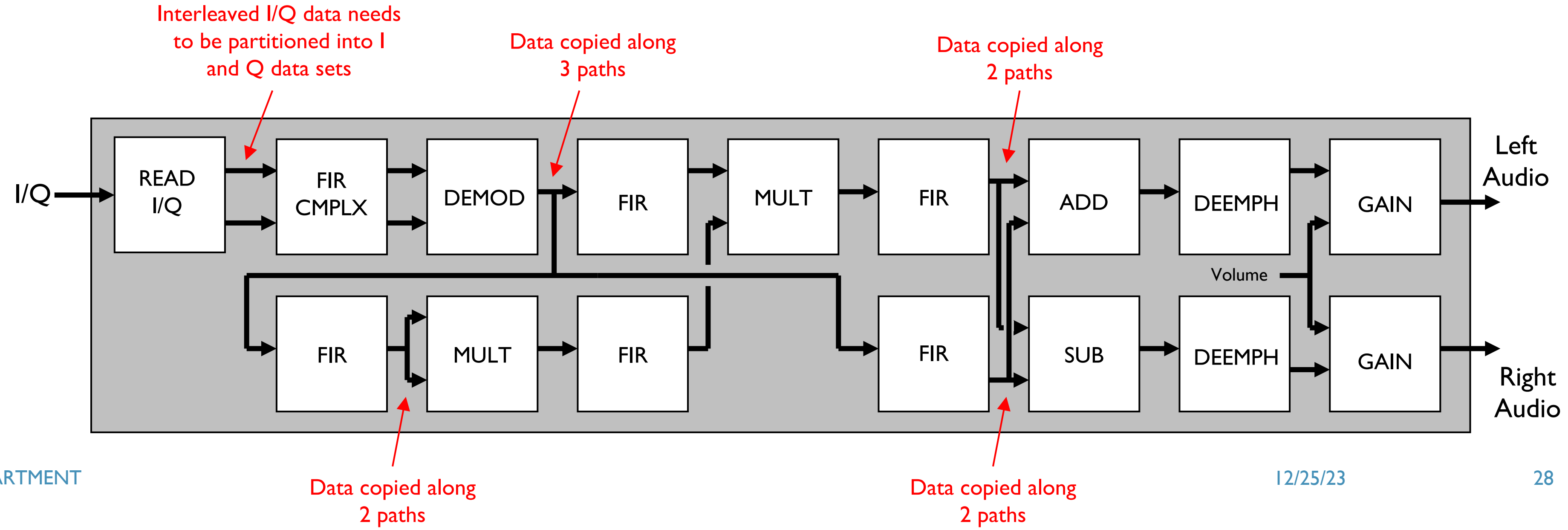
FM RADIO: DECONSTRUCTED



- Volume / Gain
  - Multiply the signal by volume control to increase signal strength
  - Left/Right channels maintain the same volume control

STREAMING DESIGN CONSIDERATIONS

- Complex streaming architecture
- Has multiple delay paths
- How do you remove bottlenecks to ensure data pipeline flows smoothly?
- Data quantization to eliminate round-off errors



FM STEREO RADIO IN SOFTWARE

```
int main(int argc, char **argv)
{
    static unsigned char IQ[SAMPLES*4];
    static int left_audio[AUDIO_SAMPLES];
    static int right_audio[AUDIO_SAMPLES];

    if ( argc < 2 )
    {
        printf("Missing input file.\n");
        return -1;
    }

    // initialize the audio output
    int audio_fd = audio_init( AUDIO_RATE );
    if ( audio_fd < 0 )
    {
        printf("Failed to initialize audio!\n");
        return -1;
    }

    FILE * usrp_file = fopen(argv[1], "rb");
    if ( usrp_file == NULL )  {
        printf("Unable to open file.\n");
        return -1;
    }
```

```
    // run the FM receiver
    while( !feof(usrp_file) )
    {
        fread( IQ, sizeof(char), SAMPLES*4, usrp_file );
        fm_radio_stereo( IQ, left_audio, right_audio );
        audio_tx( audio_fd, AUDIO_RATE, left_audio,
                  right_audio, AUDIO_SAMPLES );
    }

    fclose( usrp_file );
    close( audio_fd );
    return 0;
}
```

Move to Hardware

FM STEREO RADIO IN SOFTWARE

```
void fm_radio_stereo(unsigned char *IQ, int
    *left_audio, int *right_audio)
{
    static int I[SAMPLES];
    static int Q[SAMPLES];
    static int I_fir[SAMPLES];
    static int Q_fir[SAMPLES];
    static int demod[SAMPLES];
    static int bp_pilot_filter[SAMPLES];
    static int bp_lmr_filter[SAMPLES];
    static int hp_pilot_filter[SAMPLES];
    static int audio_lpr_filter[AUDIO_SAMPLES];
    static int audio_lmr_filter[AUDIO_SAMPLES];
    static int square[SAMPLES];
    static int multiply[SAMPLES];
    static int left[AUDIO_SAMPLES];
    static int right[AUDIO_SAMPLES];
    static int left_deemph[AUDIO_SAMPLES];
    static int right_deemph[AUDIO_SAMPLES];
    static int fir_cmplx_x_real[MAX_TAPS];
    static int fir_cmplx_x_imag[MAX_TAPS];
    static int demod_real[] = {0};
    static int demod_imag[] = {0};
    static int fir_lpr_x[MAX_TAPS];
    static int fir_lmr_x[MAX_TAPS];
    static int fir_bp_x[MAX_TAPS];
    static int fir_pilot_x[MAX_TAPS];
    static int fir_hp_x[MAX_TAPS];
    static int deemph_l_x[MAX_TAPS];
    static int deemph_l_y[MAX_TAPS];
    static int deemph_r_x[MAX_TAPS];
    static int deemph_r_y[MAX_TAPS];

    // read the I/Q data from the buffer
    read_IQ( IQ, I, Q, SAMPLES );

    // Channel low-pass filter cuts off all frequnties above 80 Khz
    fir_cmplx_n( I, Q, SAMPLES, CHANNEL_COEFFS_REAL,
     CHANNEL_COEFFS_IMAG, fir_cmplx_x_real, fir_cmplx_x_imag,
     CHANNEL_COEFF_TAPS, 1, I_fir, Q_fir );

    // demodulate
    demodulate_n( I_fir, Q_fir, demod_real, demod_imag, SAMPLES,
     FM_DEMOD_GAIN, demod );

    // L+R low-pass FIR - reduce sampling rate from 256 KHz to 32
     KHz
    fir_n( demod, SAMPLES, AUDIO_LPR_COEFFS, fir_lpr_x,
       AUDIO_LPR_COEFF_TAPS, AUDIO_DECIM, audio_lpr_filter );

    // L-R band-pass extracts the L-R channel from 23kHz to 53kHz
    fir_n( demod, SAMPLES, BP_LMR_COEFFS, fir_bp_x,
     BP_LMR_COEFF_TAPS, 1, bp_lmr_filter );

    // Pilot band-pass filter extracts the 19kHz pilot tone
    fir_n( demod, SAMPLES, BP_PILOT_COEFFS, fir_pilot_x,
     BP_PILOT_COEFF_TAPS, 1, bp_pilot_filter );

    // square the pilot tone to get 38kHz
    multiply_n( bp_pilot_filter, bp_pilot_filter, SAMPLES, square );

    // high-pass removes the tone at 0Hz after pilot tone is squared}
    fir_n( square, SAMPLES, HP_COEFFS, fir_hp_x, HP_COEFF_TAPS, 1,
     hp_pilot_filter );

    // demodulate the L-R channel from 38kHz to baseband
    multiply_n( hp_pilot_filter, bp_lmr_filter, SAMPLES, multiply );

    // L-R low-pass FIR - reduce sampling rate from 256 KHz to 32
     KHz
    fir_n( multiply, SAMPLES, AUDIO_LMR_COEFFS, fir_lmr_x,
     AUDIO_LMR_COEFF_TAPS, AUDIO_DECIM, audio_lmr_filter );

    // Left audio channel - (L+R) + (L-R) = 2L
    add_n( audio_lpr_filter, audio_lmr_filter, AUDIO_SAMPLES, left
     );

    // Right audio channel - (L+R) - (L-R) = 2R
    sub_n( audio_lpr_filter, audio_lmr_filter, AUDIO_SAMPLES, right
     );

    // Left channel deemphasis
    deemphasis_n( left, deemph_l_x, deemph_l_y, AUDIO_SAMPLES,
     left_deemph );

    // Right channel deemphasis
    deemphasis_n( right, deemph_r_x, deemph_r_y, AUDIO_SAMPLES,
     right_deemph);

    // Left volume control
    gain_n( left_deemph, AUDIO_SAMPLES, VOLUME_LEVEL, left_audio );

    // Right volume control
    gain_n( right_deemph, AUDIO_SAMPLES, VOLUME_LEVEL, right_audio
     );
```

SIMPLE FM RADIO FUNCTIONS

Read I/Q

```
void read_IQ( unsigned char *IQ, int *I, int *Q, int samples )
{
   for ( int i = 0; i < samples; i++ )
   {
      I[i] = QUANTIZE_I((short)(IQ[i*4+1] << 8) | (short)IQ[i*4+0]);
      Q[i] = QUANTIZE_I((short)(IQ[i*4+3] << 8) | (short)IQ[i*4+2]);
   }
}
```

Multiplication

```
void multiply_n( int *x_in, int *y_in, const int n_samples, int *output )
{
   for ( int i = 0; i < n_samples; i++ )
   {
      output[i] = DEQUANTIZE( x_in[i] * y_in[i] );
   }
}
```

Addition / Subtraction

```
void add_n( int *x_in, int *y_in, const int n_samples, int *output )
{
   for ( int i = 0; i < n_samples; i++ )
   {
      output[i] = x_in[i] + y_in[i];
   }
}
```

Volume / Gain

```
void gain_n( int *input, const int n_samples, int gain, int *output )
{
   for ( int i = 0; i < n_samples; i++ )
   {
      output[i] = DEQUANTIZE(input[i] * gain) << (14-BITS);
   }
}
```

DEMODULATION

```
#define QUANT_VAL       (1 << 10)
#define QUANTIZE_F(f)   (int)(((float)(f) * (float)QUANT_VAL))
#define QUANTIZE_I(i)   (int)((int)(i) * (int)QUANT_VAL)
#define DEQUANTIZE(i)   (int)((int)(i) / (int)QUANT_VAL)

void demod_n( int *real, int *imag, int *real_prev, int *imag_prev,
         const int n_samples,  const int gain, int *demod_out )
{
    for ( int i = 0; i < n_samples; i++ )    {
        demodulate( real[i], imag[i], real_prev, imag_prev,
               gain, &demod_out[i] );
    }
}
```

Streaming input & output => FIFO

```
void demodulate( int real, int imag, int *real_prev, int *imag_prev,
               const int gain,  int *demod_out )
{
    // k * atan(c1 * conj(c0))
    int r = DEQUANTIZE(*real_prev * real) -
         DEQUANTIZE(-*imag_prev * imag);
    int i = DEQUANTIZE(*real_prev * imag) +
         DEQUANTIZE(-*imag_prev * real);

    *demod_out = DEQUANTIZE(gain * qarctan(i, r));

    *real_prev = real;
    *imag_prev = imag;
}
```

```
int qarctan(int y, int x)
{
    const int quad1 = QUANTIZE_F(PI / 4.0);
    const int quad3 = QUANTIZE_F(3.0 * PI / 4.0);

    int abs_y = abs(y) + 1;
    int angle = 0;
    int r = 0;

    if ( x >= 0 )
    {
        r = QUANTIZE_I(x - abs_y) / (x + abs_y);
        angle = quad1 - DEQUANTIZE(quad1 * r);
    }
    else
    {
        r = QUANTIZE_I(x + abs_y) / (abs_y - x);
        angle = quad3 - DEQUANTIZE(quad1 * r);
    }

    // negate if in quad III or IV
    return ((y < 0) ? -angle : angle);
}
```

Division by a variable

FIR DECIMATION FILTER

```
#define BITS            10
#define QUANT_VAL       (1 << BITS)
#define QUANTIZE_F(f)   (int)(((float)(f) * (float)QUANT_VAL))
#define QUANTIZE_I(i)   (int)((int)(i) * (int)QUANT_VAL)
#define DEQUANTIZE(i)   (int)((int)(i) / (int)QUANT_VAL)


void fir_n( int *x_in, const int n_samples, const int *coeff,
         int *x, const int taps, const int decimation,
         int *y_out )
{
    int i = 0;
    int j = 0;

    int n_elements = n_samples / decimation;
    for ( i = 0; i < n_elements; i++, j+=decimation )
    {
        fir( &x_in[j], coeff, x, taps, decimation, &y_out[i] );
    }
}
```

Streaming input & output => FIFO

```
void fir( int *x_in, const int *coeff, int *x, const int taps,
      const int decimation, int *y_out )
{
    int i = 0, j = 0, y = 0;

    for ( j = taps-1; j > decimation-1; j-- )
    {
        x[j] = x[j-decimation];
    }

    for ( i = 0; i < decimation; i++ )
    {
        x[decimation-i-1] = x_in[i];
    }

    for ( j = 0; j < taps; j++ )
    {
        y += DEQUANTIZE( coeff[taps-j-1] * x[j] );
    }

    *y_out = y;
}
```

FIR COMPLEX FILTER

```
#define BITS            10
#define QUANT_VAL       (1 << BITS)
#define QUANTIZE_F(f)   (int)(((float)(f) * (float)QUANT_VAL))
#define QUANTIZE_I(i)   (int)((int)(i) * (int)QUANT_VAL)
#define DEQUANTIZE(i)   (int)((int)(i) / (int)QUANT_VAL)


void fir_cmplx_n( int *x_real_in, int *x_imag_in,
      const int n_samples, const int *h_real, const int *h_imag,
         int *x_real, int *x_imag, const int taps, const int decimation,
      int *y_real_out, int *y_imag_out )
{
    int i = 0, j = 0;
    int n_elements = n_samples / decimation;
    for ( ; i < n_elements; i++, j+=decimation )
    {
        fir_cmplx( &x_real_in[j], &x_imag_in[j], h_real, h_imag, x_real,
                   x_imag, taps, decimation, &y_real_out[i],
                   &y_imag_out[i] );
    }
}
```

Streaming input & output => FIFO

```
void fir_cmplx( int *x_real_in, int *x_imag_in, const int *h_real,
         const int *h_imag, int *x_real, int *x_imag,  const int taps,
         const int decimation, int *y_real_out, int *y_imag_out )
{
    int i = 0, j = 0, y_real = 0, y_imag = 0;

    for ( j = taps-1; j > decimation-1; j-- )     {
        x_real[j] = x_real[j-decimation];
        x_imag[j] = x_imag[j-decimation];
    }

    for ( i = 0; i < decimation; i++ )    {
        x_real[decimation-i-1] = x_real_in[i];
        x_imag[decimation-i-1] = x_imag_in[i];
    }

    for ( i = 0; i < taps; i++ )     {
        y_real += DEQUANTIZE((h_real[i] * x_real[i])
                        - (h_imag[i] * x_imag[i]));
        y_imag += DEQUANTIZE((h_real[i] * x_imag[i])
                        - (h_imag[i] * x_real[i]));
    }

    *y_real_out = y_real;
    *y_imag_out = y_imag;
}
```

DEEMPHASIS / IIR

```
#define BITS            10
#define QUANT_VAL       (1 << BITS)
#define QUANTIZE_F(f)   (int)(((float)(f) * (float)QUANT_VAL))
#define QUANTIZE_I(i)   (int)((int)(i) * (int)QUANT_VAL)
#define DEQUANTIZE(i)   (int)((int)(i) / (int)QUANT_VAL)


void iir_n( int *x_in, const int n_samples, const int *x_coeffs,
         const int *y_coeffs, int *x, int *y, const int taps,
         int dec, int *y_out )
{
    int i = 0,  j = 0;
    int n_elements = n_samples / decimation;
    for ( ; i < n_elements; i++, j+=decimation )
    {
        iir(&x_in[j], x_coeffs, y_coeffs, x, y, taps, dec, &y_out[i] );
    }
}
```

Streaming input & output => FIFO

```
void iir( int *x_in, const int *x_coeffs, const int *y_coeffs, int *x,
      int *y, const int taps, const int decimation, int *y_out )
{
    int y1 = 0, y2 = 0, i = 0, j = 0;

    for ( j = taps-1; j > decimation-1; j-- )     {
        x[j] = x[j-decimation];
    }

    for ( i = 0; i < decimation; i++ )    {
        x[decimation-i-1] = x_in[i];
    }

    for ( j = taps-1; j > 0; j--)    {
        y[j] = y[j-1];
    }

    for ( i = 0; i < taps; i++ )    {
        y1 += DEQUANTIZE( x_coeffs[i] * x[i] );
        y2 += DEQUANTIZE( y_coeffs[i] * y[i] );
    }

    y[0] = y1 + y2;

    *y_out = y[taps-1];
}
```

NEXT…

- Final Project: FM Radio