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

PROF. DAVID ZARETSKY

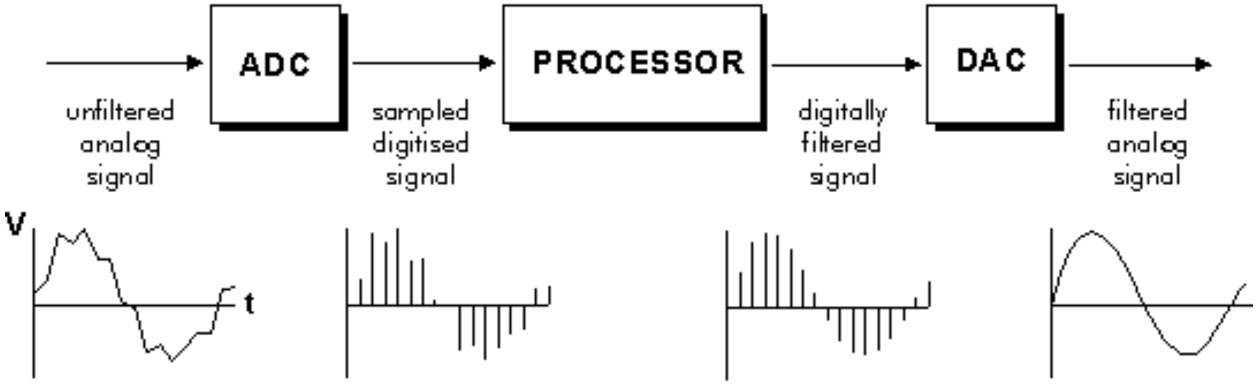
DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

- Digital Signal Processing

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

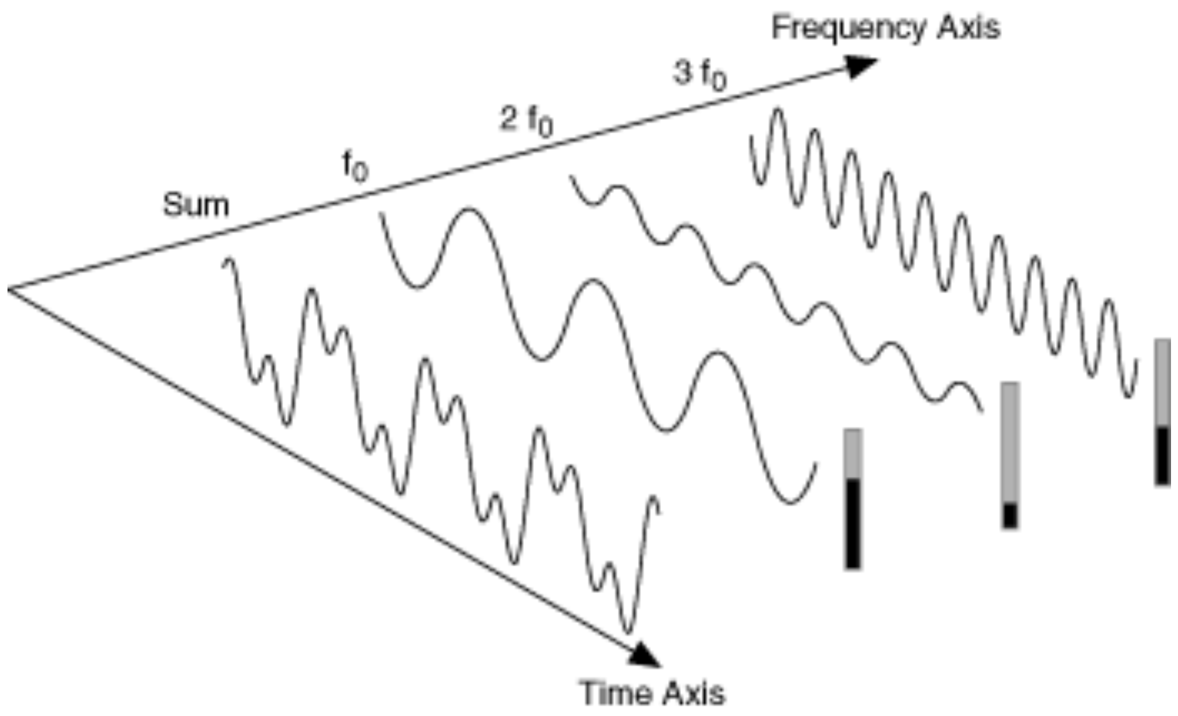
- The time-domain representation gives the amplitudes of signals at the instants of time during which it was sampled.
- Fourier's theorem states that any waveform in the time domain can be represented by the weighted sum of sines and cosines.
- The same waveform then can be represented in the frequency 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

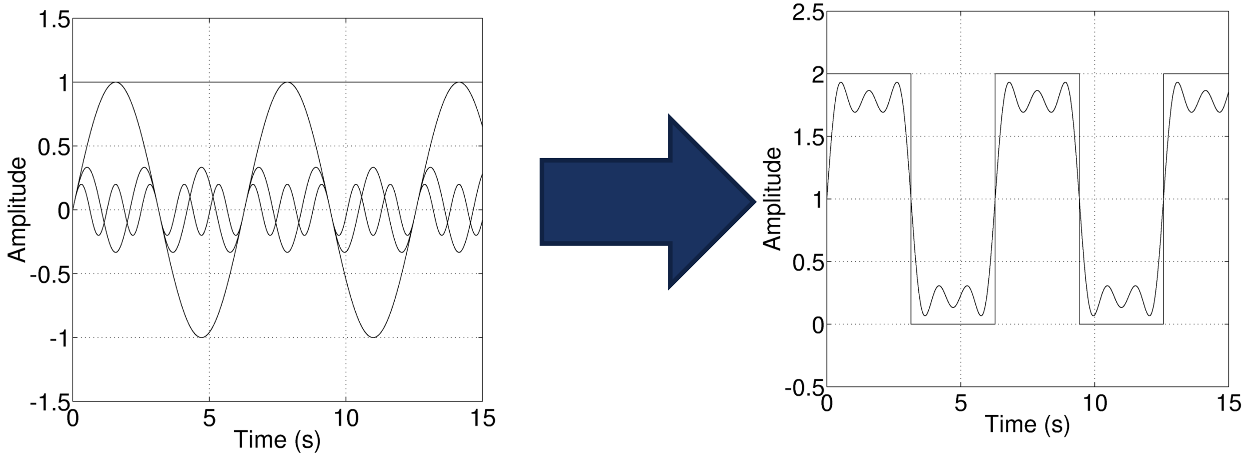
- The figure shows single frequency components spread out in the time domain, as distinct impulses in the frequency domain.
- The amplitude of each frequency line is the amplitude of the time waveform for that frequency component.



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

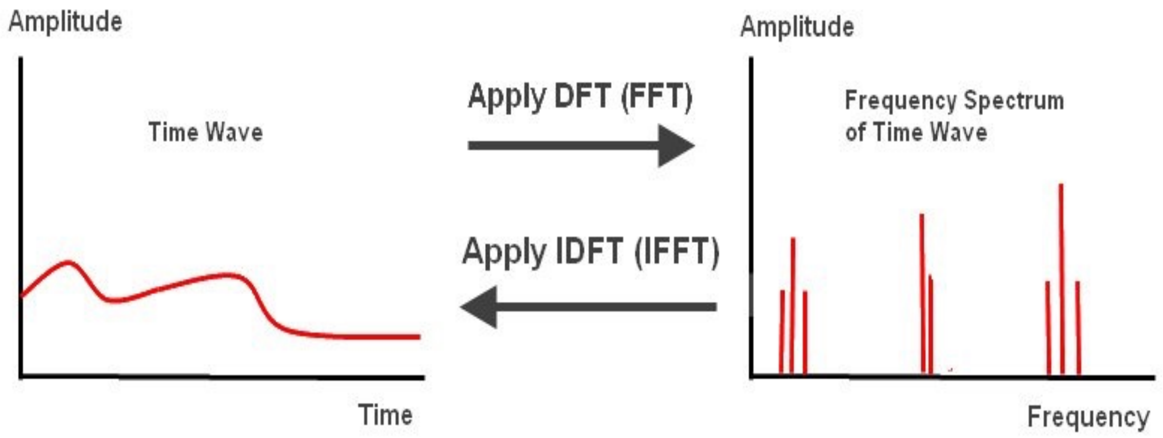
- Filters
  - Remove unwanted parts of the signal, such as random noise
  - Extract useful parts of the signal, such as the components lying within a certain frequency range
- Analog Filters
  - Input: electrical voltage or current which is the direct analogue of a physical quantity (sensor output)
  - 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

FFT

- Fourier Transform converts a time-domain wave to its frequency components (sine waves of varying frequencies and amplitudes).
- DFT converts discrete time samples into a frequency spectrum.
- FFT is a faster algorithm to compute the DFT efficiently.
  - DFT requires $O(N^2)$ operations ($1024^2 = 1{,}048{,}576$)
  - FFT requires $O(N \log N)$ operations ($1024 \times 10 = 10{,}240$ => 100x speedup!)



DFT:

$$F(n) = \sum_{k=0}^{N-1} x(k) e^{\frac{-j2\pi kn}{N}}$$

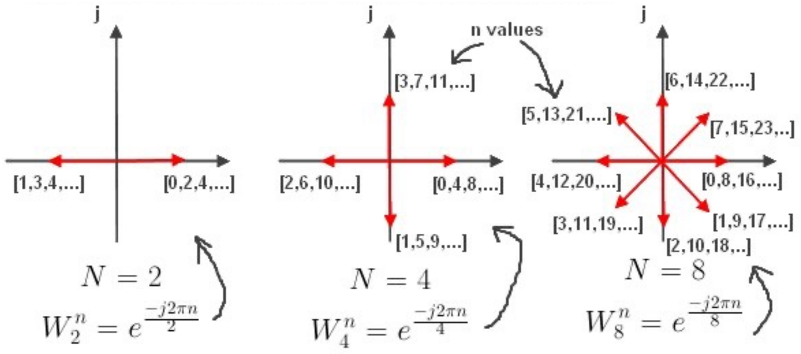
for $n = 0 \ldots N - 1$

Where F(n) is the amplitude at the frequency, n, and N is the number of discrete samples taken.

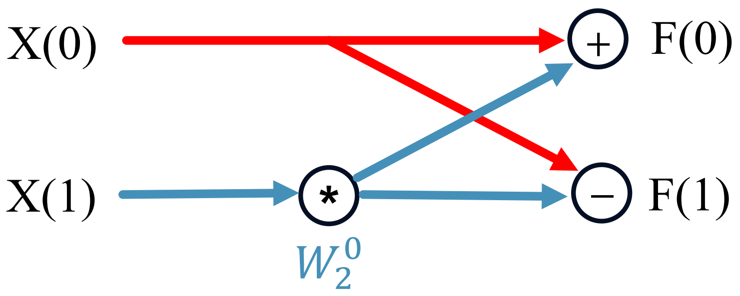
FFT BUTTERFLY

- The butterfly is based on the Danielson-Lanczos Lemma
- Basic butterfly unit consists of 2 inputs and 2 outputs
- Each input follows a unique path to the output, with values along the path multiplied by the input twiddle factor and added at the output.
- The twiddle factor, W, describes a "rotating vector", which rotates in increments according to the number of samples, N.

Twiddle Factor



Vectors have redundancy and symmetry



$$F(0) = x(0) + W_2^0 * x(1)$$
$$F(1) = x(0) - W_2^0 * x(1)$$

DANIELSON-LANCZOS (D-L) LEMMA

- DFT of size N can be broken into two smaller DFTs
  - Each stage is N/2 with 1 even-indexed + 1 odd-indexed inputs
  - Combined with a set of multiplication factors (twiddle factors)
- Note the order of bits (reversed) when D-L Lemma is expanded
  - 4-input ordering: [0,2,1,3]
  - 8-input ordering: [0,4,2,6,1,5,3,7]
  - 16-input ordering: [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]

$$F(n) = \sum_{k=0}^{N-1} x(k) e^{\frac{-j2\pi kn}{N}}$$

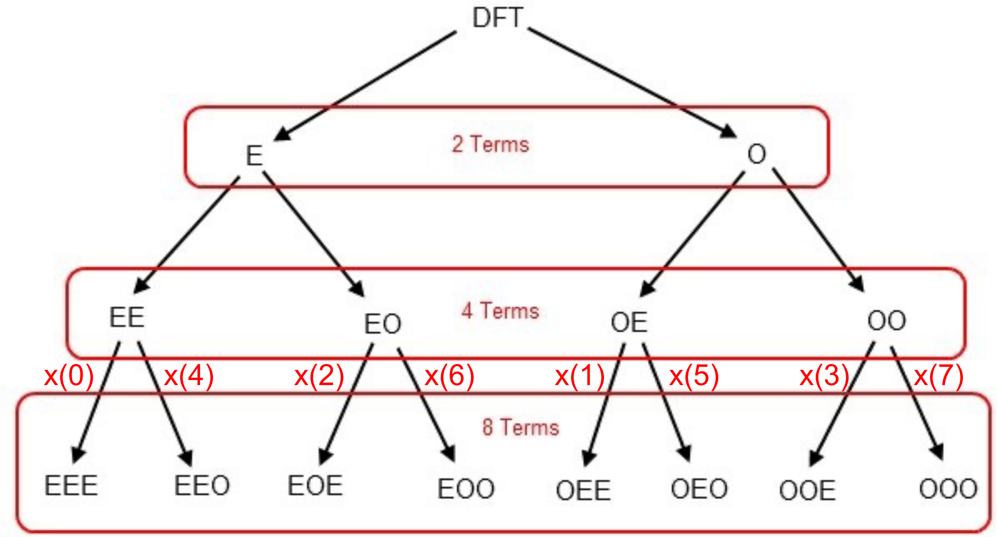
$$= \sum_{k=0}^{\frac{N}{2}-1} x(2k) e^{\frac{-j2\pi kn}{\left(\frac{N}{2}\right)}} + W_N^n \sum_{k=0}^{\frac{N}{2}-1} x(2k+1) e^{\frac{-j2\pi kn}{\left(\frac{N}{2}\right)}}$$

E = Even Term

O = Odd Term

$$W_N^n = e^{\frac{-j2\pi n}{N}}$$

Twiddle factor

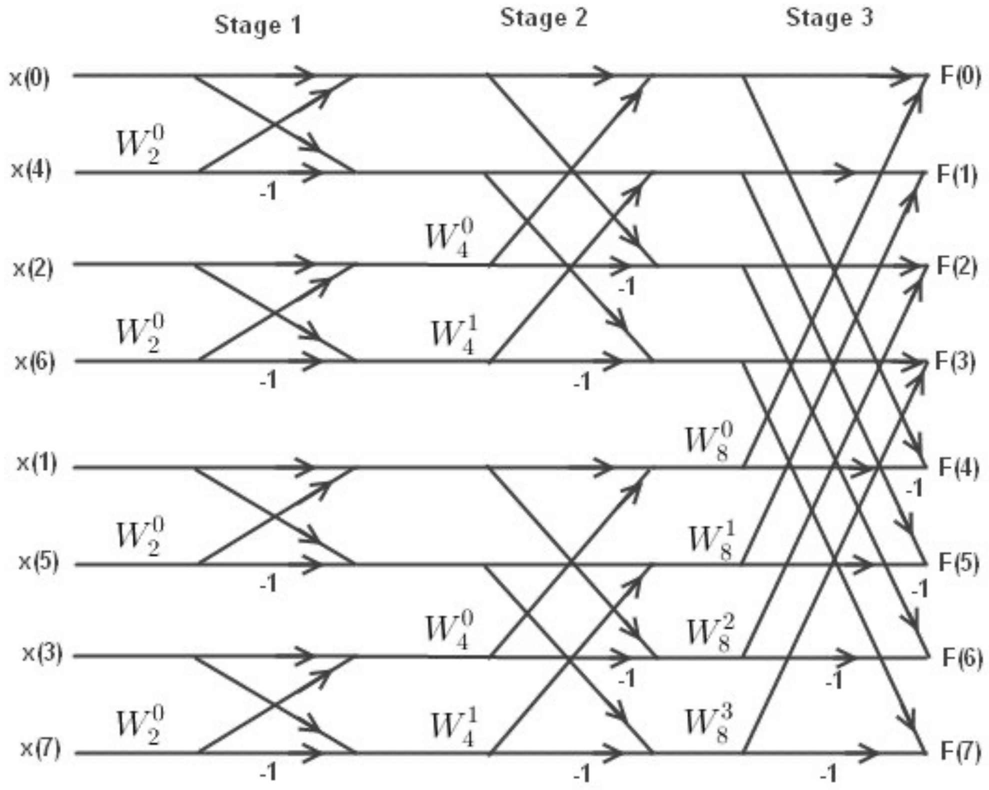


The Danielson-Lanczos for 8 input values

$$F(n) = x(0) + W_2^n x(4) + W_4^n x(2) + W_4^n W_2^n x(6) + W_8^n x(1) + W_8^n W_2^n x(5) + W_8^n W_4^n x(3) + W_8^n W_4^n W_2^n x(7)$$

N-STAGE FFT

- N-input FFT results in $\log_2(N)$ stages
- For example, an 8-input FFT would have:
  - (8/2) butterflies x 3 stages = 12 total butterflies
  - 12 butterflies x 2 multiplies = 24 multiplies
  - In hardware, these multiplies contain constant W which can be optimized!



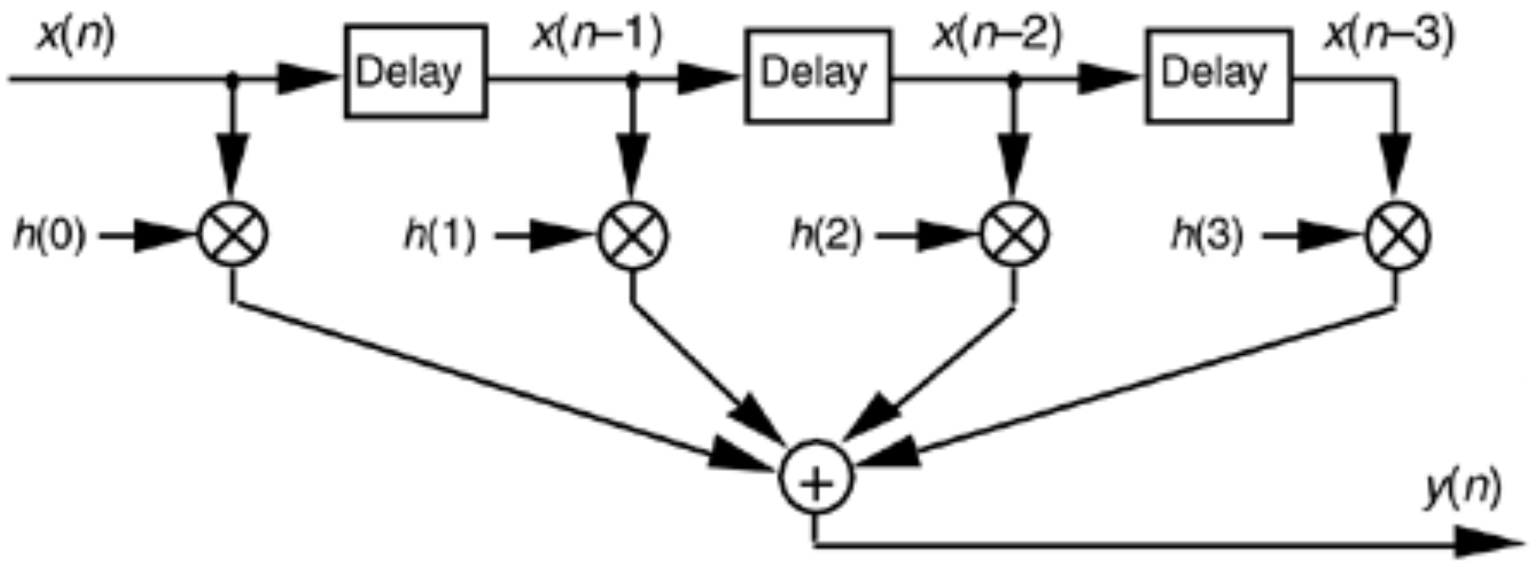
FFT IN SOFTWARE

```c
typedef struct {int real; int imag; } Complex;

void bit_reversal(Complex *in, Complex *out, int N)
{
    int bit_reversal_table[N];                          // Pre-computed table
    for (int i = 0; i < N; i++) {
        int j = 0;
        for (int bit = 0; bit < log2(N); bit++) {
            if (i & (1 << bit)) {
                j |= (1 << ((int)log2(N) - bit - 1));
            }
        }
        bit_reversal_table[i] = j;
    }
    for (int i = 0; i < N; i++) {                       // Concurrent assignments
        out[ bit_reversal_table[i] ] = in[i];
    }
}

void butterfly(Complex *in1, Complex *in2, Complex *out1, Complex *out2, Complex w)
{
    Complex v = { DEQUANTIZE_I(w.real * in2->real) - DEQUANTIZE_I(w.imag * in2->imag),
                  DEQUANTIZE_I(w.real * in2->imag) + DEQUANTIZE_I(w.imag * in2->real) };
    out1->real = in1->real + v.real;
    out1->imag = in1->imag + v.imag;
    out2->real = in1->real - v.real;
    out2->imag = in1->imag - v.imag;
}
```

```c
// FFT function with feed-forward memory allocation
void fft(Complex *in, Complex *out, int N)
{
    const int NUM_STAGES = log2(N);                     // Independent butterfly paths
    const int TOTAL_SIZE = N * (NUM_STAGES + 1);
    Complex x[TOTAL_SIZE];

    // Bit-reversed input
    bit_reversal(in, x, N);

    // FFT computation across stages                    // Generate-Loop
    for (int stage = 0; stage < NUM_STAGES; stage++) {
        int step = 1 << (stage + 1);
        for (int i = 0; i < N; i += step) {
            for (int j = 0; j < step / 2; j++) {
                int read_offset = stage * N;            // Constants
                int write_offset = (stage + 1) * N;
                int in1_idx = read_offset + i + j;
                int in2_idx = read_offset + i + j + step / 2;
                int out1_idx = write_offset + i + j;
                int out2_idx = write_offset + i + j + step / 2;

                float angle = j * (-PI / (step / 2));
                Complex w = {QUANTIZE_F(cos(angle)), QUANTIZE_F(sin(angle))};

                butterfly(&x[in1_idx], &x[in2_idx], &x[out1_idx], &x[out2_idx], w);
            }
        }
    }

    for (int i = 0; i < N; i++) {                       // Concurrent assignments
        out[i] = x[NUM_STAGES * N + i];
    }
}
```

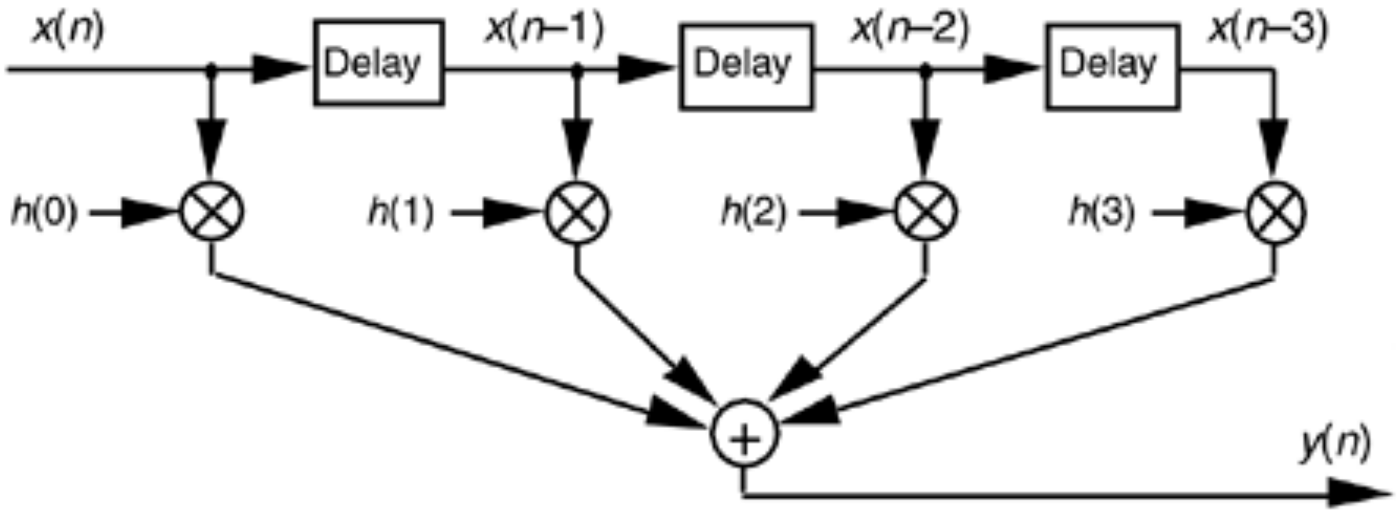
FINITE IMPULSE RESPONSE (FIR) FILTERS

- 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);
}
```

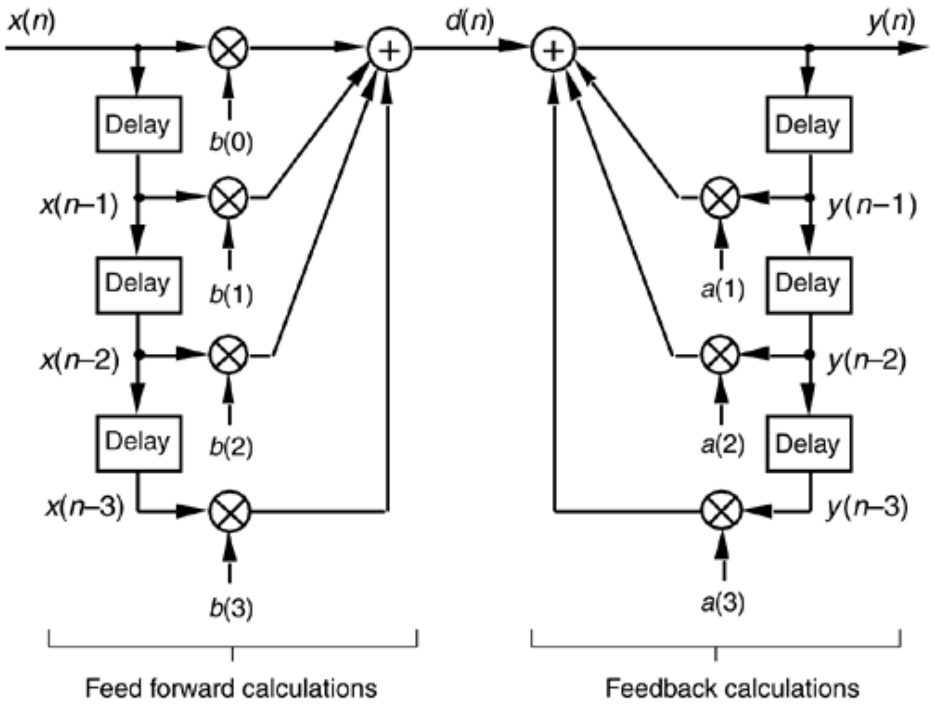


FIR HARDWARE IMPLEMENTATION IN VHDL

```
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;
```

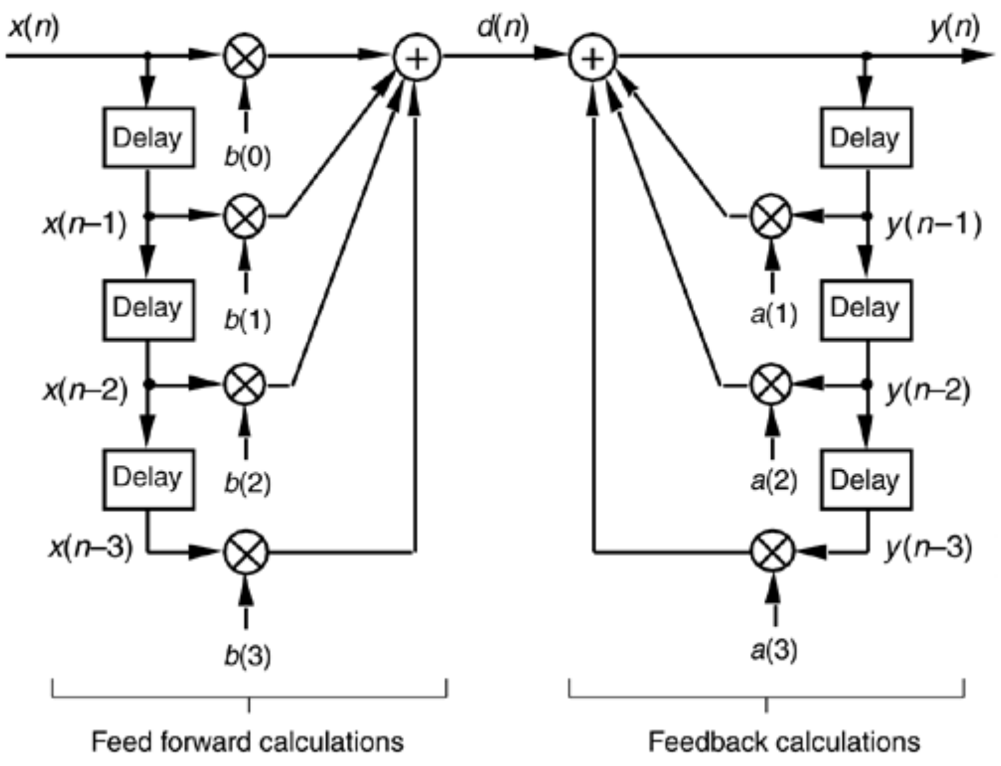
INFINITE IMPULSE RESPONSE (IIR) FILTERS

- 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);
}
```



NEXT…

- Final Project: FM Radio