

# Zahlenformate

## 1.1 Zweierkomplement

Umwandlung: Bsp. 8-Bit  $(-4)_{10}$  (Funktioniert in beide Richtungen)

- Vorzeichen Ignorieren  $(4)_{10} = (00000100)_2$
- Bits Invertieren  $(0000\ 0100)_2 \rightarrow (1111\ 1011)_2$
- Eins Addieren  $(1111\ 1011)_2 + (0000\ 0001)_2 = (1111\ 1100)_2$

## 1.2 Fixed Point (unsigned)

Qk.l mit k = Vorkomma und l = Nachkomma

$$x_{(10)} = \sum_{i=0}^{k-1} b_i \cdot 2^i + \sum_{j=-l}^{-1} b_j \cdot 2^j \quad (1)$$

Bsp Q4.5  $a = (01010110)_2$

$$0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = 5.375$$

## 1.3 Fixed Point (signed)

Bsp. Q3.3  $(100.001)_2$

- Vorzeichen Merken  $(100.001) \rightarrow -1$
- Bits Invertieren  $(100\ 001)_2 \rightarrow (011\ 110)_2$
- $1 \cdot 2^{-k}$  Addieren  $(011\ 110)_2 + (000\ 001)_2 = (011\ 111)_2 = -3.875$

Bits	Format	Numeric Range	Precision	Dynamic Range
8	Unsigned integer	$0 \rightarrow +255$	1	$\approx 48\text{ dB}$
8	Signed integer	$-128 \rightarrow +127$	1	$\approx 48\text{ dB}$
16	Unsigned integer	$0 \rightarrow +65,536$	1	$\approx 96\text{ dB}$
16	Signed integer	$-32,768 \rightarrow +32,767$	1	$\approx 96\text{ dB}$
16	Fixed-point (Q12)	$-8.0 \rightarrow \approx +7.999756$	$\approx 0.000244$	$\approx 96\text{ dB}$
16	Fixed-point (Q15)	$-1.0 \rightarrow \approx +0.9999695$	$\approx 0.0000305$	$\approx 96\text{ dB}$
32	Unsigned integer	$0 \rightarrow +4,294,967,296$	1	$\approx 193\text{ dB}$
32	Signed integer	$-2,147,483,648 \rightarrow +2,147,483,647$	1	$\approx 193\text{ dB}$
32	Single-precision	$\approx \pm 3.402823 \times 10^{38}$	$\approx 1.19 \times 10^{-7}$	$\approx 138\text{ dB}$
64	Double-precision	$\approx \pm 1.797693 \times 10^{308}$	$\approx 2.22 \times 10^{-16}$	$\approx 314\text{ dB}$

# 2 Filter in c

## 2.1 FIR

$$A(0)y(k) = \sum_{i=0}^N B(i)x(k-i) \quad (2)$$

### 2.1.1 FIR Brute-Force

```
void fir()
{
    CodecDataIn.UINT = ReadCodecData(); // get input data samples
    int i;
    xLeft[0] = CodecDataIn.Channel[LEFT]; // current input value
    yLeft = 0; // initialize the output
    for (i = 0; i <= N; i++) { // x is length N+1
        yLeft += xLeft[i] * B[i]; // perform the dot-product
    }
    for (i = N; i > 0; i--) {
        xLeft[i] = xLeft[i-1]; // shift for the next input
    }
    CodecDataOut.Channel[LEFT] = yLeft; // output the value
}
```

- nach jedem dot-product ineffizientes Shiften von xLeft
- Problem kann mit Ringbuffer behoben werden

### 2.1.2 FIR Ringbuffer

## 2.2 IIR

$$A(0)y(k) = \sum_{i=0}^N B(i)x(k-i) - \sum_{i=0}^N A(i)y(k-i) \quad (3)$$

```
void iir()
{
    xLeft[0] = CodecDataIn.Channel[LEFT] + CodecDataIn.Channel[RIGHT];
    yLeft[0] = 0; // initialize the output value

    for (i = 0; i <= N; i++){
        yLeft[0] += B[i] * xLeft[i];
    }
    for (i = 1; i <= N; i++){
        yLeft[0] -= A[i] * yLeft[i];
    }
    for (i = N; i > 0; i--)
    {
        xLeft[i] = xLeft[i-1];
        yLeft[i] = yLeft[i-1];
    }
    CodecDataOut.Channel[LEFT] = yLeft[0]; // output the filtered value
}
```

## 2.3 Notch-IIR-Filter

$$H(z) = k \frac{(z - \beta_1)(z - \beta_2)}{(z - \alpha_1)(z - \alpha_2)} = k \frac{1 - 2 \cos(\omega_0)z^{-1} + z^{-2}}{1 - 2r \cos(\omega_0)z^{-1} + r^2 z^{-2}} \quad (4)$$

$f_0$  = Kerbfrequenz,  $f_s$  = Abtastfrequenz  $B_{3dB}$  = Kerbbreite

$$k = \frac{1 - 2r \cos(\omega_0) + r^2}{1 - 2 \cos(\omega_0) + 1} \quad (5)$$

$$\omega_0 = 2\pi \frac{f_0}{f_s} \quad (6)$$

$$r = 1 - \left(\frac{B_{3dB}}{f_s}\right)\pi \quad (7)$$

## 2.4 IIR-Oszillator

$$\sin(\omega_0 k) \leftrightarrow \frac{\sin(\omega_0)z^{-1}}{1 - 2 \cos(\omega_0)z^{-1} + z^{-2}} \quad (8)$$

$$y(n) = \sin(\omega_0)x(n-1) + 2\cos(\omega_0)y(n-1) - y(n-2) \quad (9)$$

## 2.5 DDS-Oszillator

z.B NCO

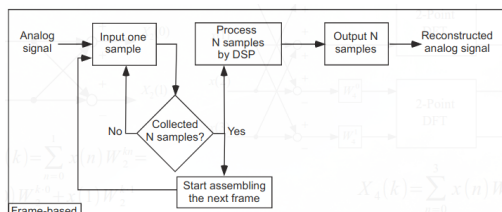
- Frequenzauflösung abhängig von Wortbreite des Phasenakkumulators  $\frac{f_{clk}}{2^{N_{Bits}}}$
- keine Einschwingzeit  $\rightarrow$  stabil
- hoher Speicherbedarf (LUT)
- robuster Phasen/Frequenzwechsel mit sofortiger Wirkung

$$\varphi_{inc} = 2\pi \frac{f_0}{f_s} \quad (10)$$

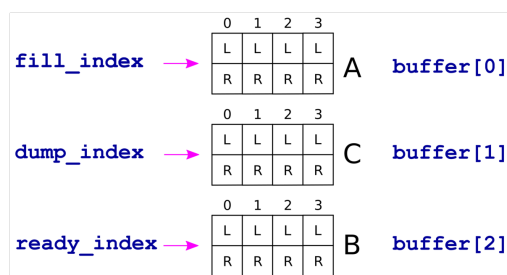
$$\varphi = \varphi + \varphi_{inc} \quad (11)$$

$$x(n) = A \sin(n\varphi) \quad (12)$$

### 3 Blocksignalverarbeitung



ISR schreibt N samples nach `buffer[fill_index]` und setzt `ready_index = fill_index` (`buffer[fill_index]` ist jetzt dran mit `ProcessBuffer`). Jeder Sample generiert einen Interrupt



- `fill_index` wird von ADC gefüllt
- `dump_index` wird an DAC geschrieben
- `ready_index` Buffer für Blocksignalverarbeitung

```
#define BUFFER_LENGTH      1024      // buffer length in samples
#define NUM_BUFFERS      3
volatile float buffer[NUM_BUFFERS][2][BUFFER_LENGTH];

void ProcessBuffer()
// Processes the data in buffer[ready_index]
{
    volatile float *pL = buffer[ready_index][LEFT];
    volatile float *pR = buffer[ready_index][RIGHT];
    // Do the Process
    // ...
    buffer_ready = 0; // means were done here
}

interrupt void Codec_ISR()
{
    static Uint8 fill_index = INITIAL_FILL_INDEX; // index of buffer to fill
    static Uint8 dump_index = INITIAL_DUMP_INDEX; // index of buffer to dump
    static Uint32 sample_count = 0; // current sample count in buffer

    // get input data samples
    CodecDataIn.UINT = ReadCodecData();
    // IN
    buffer[fill_index][LEFT][sample_count] = LEFT + RIGHT; // cropped
    buffer[fill_index][RIGHT][sample_count] = RIGHT + LEFT; // cropped
    // OUT
    CodecDataOut.channel[LEFT] = buffer[dump_index][LEFT][sample_count];
    CodecDataOut.channel[RIGHT] = buffer[dump_index][RIGHT][sample_count];

    // update sample count and swap buffers when filled
    if(++sample_count >= BUFFER_LENGTH) {
        sample_count = 0;
        ready_index = fill_index;
        if(++fill_index >= NUM_BUFFERS)
            fill_index = 0;
        if(++dump_index >= NUM_BUFFERS)
            dump_index = 0;
        if(buffer_ready == 1) // set a flag if buffer isn't processed in time
            over_run = 1;
        buffer_ready = 1;
    }

    WriteCodecData(CodecDataOut.UINT); // send output data to port
}
```

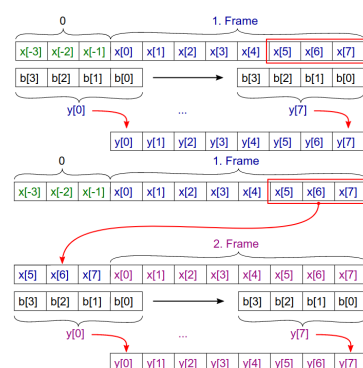
### 3.1 Blocksignalverarbeitung mit DMA

- DMA kopiert Sample von ADC nach Eingangsbuffer
- DMA kopiert Processed von Ausgangsbuffer nach DAC
- DMA generiert Interrupt, wenn N Samples transfert wurden → Buffer-Swap

```
interrupt void EDMA_ISR()
{
    if(++ready_index >= NUM_BUFFERS)
        ready_index = 0;
    if(buffer_ready == 1) //buffer isnt processed in time
        over_run = 1;
    buffer_ready = 1; //buffer is now ready for processing
}
```

### 3.2 FIR mit Blocksignalverarbeitung

Bsp. Ordnung Filter N = 4, Framesize = 8



**Problem:** Bei Frame-Übergängen müssen die N-1 letzten Werte des letzten Frames berücksichtigt werden.

**Lösung:** `Framesize += N`

**Latenz:**  $\frac{2N}{f_{clk}}$  2N: Eingangs- und Verarbeitungsbuffer

- `Left[N:FRAMESIZE]`, `buffer[FRAMESIZE]`
- `Left[N:FRAMESIZE+N] = buffer[0:FRAMESIZE]`
- `buffer[0:FRAMESIZE] = Left * B` (B wird drüber FRAMESIZE-mal drüber geschoben s.o)
- `Left[0:N] = Left[FRAMESIZE:FRAMESIZE+N]`

```
void ProcessBuffer()
{
    short *pBuf = buffer[ready_index];
    // extra buffer room for convolution "edge effects"
    // N is filter order from coeff.h
    static float Left[BUFFER_COUNT+N]={0}, Right[BUFFER_COUNT+N]={0};
    float *pL = Left, *pR = Right;
    float yLeft, yRight;
    int i, j, k;
    // offset pointers to start filling after N elements

    pR += N;
    pL += N;

    // extract data to float buffers
    for(i = 0; i < BUFFER_COUNT; i++)
    {
        *pR++ = *pBuf++;
        *pL++ = *pBuf++;
    }
    // reinitialize pointer before FOR loop
    pBuf = buffer[ready_index];

    // Implement FIR filter
    for(i=0; i < BUFFER_COUNT; i++)
    {
        yLeft = 0; // initialize the LEFT output value
        yRight = 0; // initialize the RIGHT output value

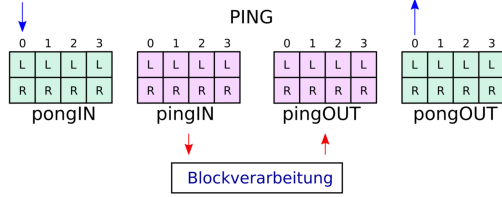
        for(j=0, k=i+N; j <= N; j++, k--)
        {
```

```

        yLeft += Left[k] * B[j]; // perform the LEFT dot-product
        yRight += Right[k] * B[j]; // perform the RIGHT dot-product
    }
    // pack into buffer after bounding (must be right then left)
    *pBuf++ = _spint(yRight * 65536) >> 16;
    *pBuf++ = _spint(yLeft * 65536) >> 16;
}

// save end values at end of buffer array for next pass
// by placing at beginning of buffer array
for(i=BUFFER_COUNT,j=0; i < BUFFER_COUNT+N; i++,j++)
{
    Left[j] = Left[i];
    Right[j] = Right[i];
}
buffer_ready = 0; // signal we are done
}
    
```

### 3.3 Ping-Pong



- gleiche Latenz (=Durchlaufzeit 2 Buffer)
- Ping-Pong einfacher zu verwalten mit DMA

## 4 FFT

Die FFT basiert auf dem **Devide-and-Conquer** Prinzip, so dass schon berechnete Zwischenergebnisse wiederverwendet werden. Mögliche Realisierungsformen:

- Decimation in Frequency
- Decimation in Time

$$w_N^{nk} = e^{-j \frac{2\pi nk}{N}} \quad (13)$$

$$w_N^{2nk} = e^{-j \frac{4\pi nk}{N}} = w_{\frac{N}{2}}^{nk} \quad (14)$$

### 4.1 Decimation in Time (DIT)

1. Aufteilung in  $Y(n) = Y_{even}(n) + Y_{odd}(n)$

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k)w_N^{2nk} + \sum_{k=0}^{N/2-1} y(2k+1)w_N^{(2k+1)n} \quad (15)$$

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k)w_N^{2nk} + w_N^n \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{2kn} \quad (16)$$

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k)w_{\frac{N}{2}}^{nk} + w_N^n \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{nk} \quad (17)$$

$$(18)$$

2. Aufteilung in  $Y(n) = Y_{left}(n) + Y_{right}(n)$

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k)w_{\frac{N}{2}}^{nk} + w_N^n \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{nk} \quad (19)$$

$$Y(n + \frac{N}{2}) = \sum_{k=0}^{N/2-1} y(2k)w_{\frac{N}{2}}^{(n+\frac{N}{2})k} + w_N^{n+\frac{N}{2}} \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{(n+\frac{N}{2})k} \quad (20)$$

mit

$$w_{\frac{N}{2}}^{(n+\frac{N}{2})k} = w_{\frac{N}{2}}^{nk} \cdot \underbrace{w_{\frac{N}{2}}^{k\frac{N}{2}}}_{=1} = w_{\frac{N}{2}}^{nk} \quad (21)$$

$$w_N^{n+\frac{N}{2}} = w_N^n \cdot \underbrace{w_{\frac{N}{2}}^{\frac{N}{2}}}_{=-1} = -w_N^n \quad (22)$$

folgt

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k)w_{\frac{N}{2}}^{nk} + w_N^n \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{kn} \quad (23)$$

$$Y(n + \frac{N}{2}) = \sum_{k=0}^{N/2-1} y(2k)w_{\frac{N}{2}}^{nk} - w_N^n \sum_{k=0}^{N/2-1} y(2k+1)w_{\frac{N}{2}}^{kn} \quad (24)$$

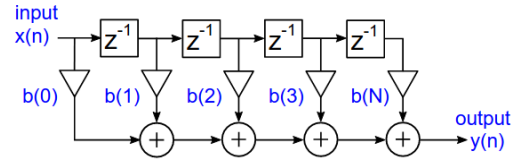
$$Y_{left}(n) = Y_{even}(n) + w_N^n Y_{odd}(n) \quad (25)$$

$$Y_{right}(n) = Y_{even}(n) - w_N^n Y_{odd}(n) \quad (26)$$

Die Komplexität ist  $O(N \log_2(N))$ :

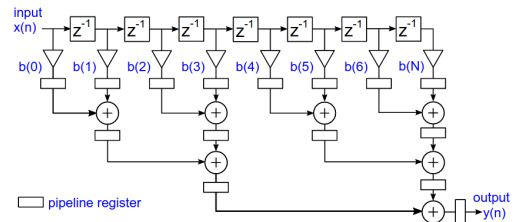
Es gibt 2  $\log_2(N)$  Splitting-Steps mit je  $O(n)$

## 5 FIR in FPGA

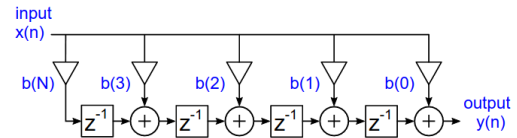


### 5.1 FIR - Pipelined

Massive Erhöhung der Taktfrequenz



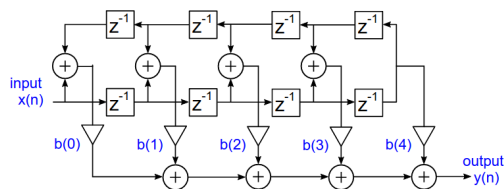
### 5.2 FIR - transposed (DF1)



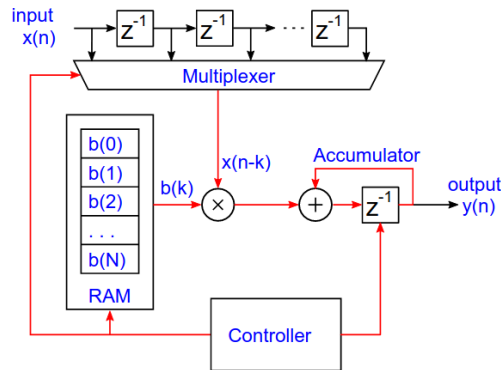
Max. Sampling Frequency  $f_{sampling}$

$$f_{sampling} = \frac{1}{T_{mult} + 4T_{add}} \quad (27)$$

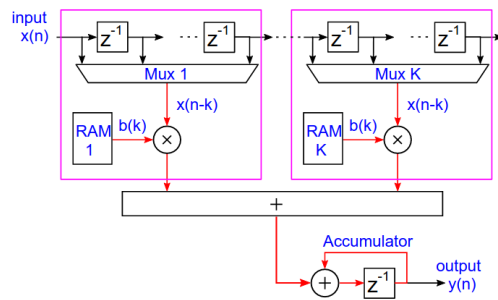
### 5.3 FIR - lineare Phase



### 5.4 FIR - seriell

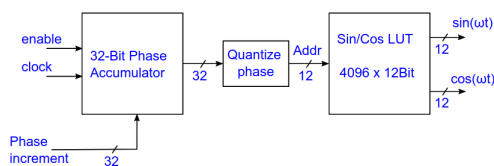


### 5.5 FIR - semi-parallel



## 6 NCO

NCO = Counter, der Jeden Takt um ein Phaseninkrement  $\mu$  (hier 32) erhöht wird. Der Ausgang des Counters wird mit LUT in Signalform (sin,cos,sägezahn) umgewandelt. Die LUT ist mit  $N = 2^n$  12-bit breiten Werten gefüllt



$$\mu = N \frac{f_d}{f_s} \quad (28)$$