1 Zahlenformate

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

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

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

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)

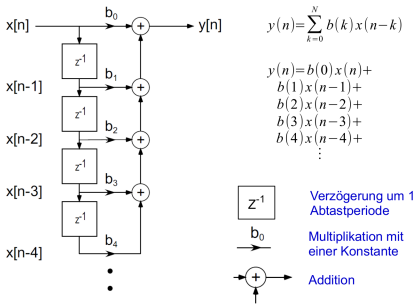
Bsp. Q3.3 $(100.001)_2$

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

2 Filter in C

2.1 FIR

2.1.1 Blockschaltbild und math. Zusammenhänge:



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

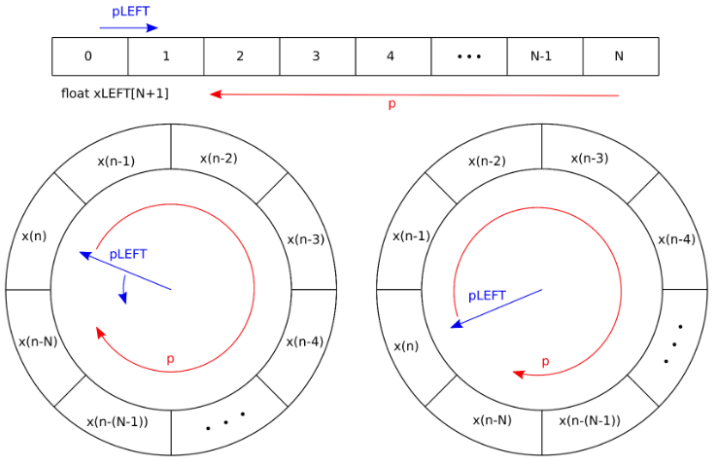
2.1.2 Realisierung des FIR in C (Brute-Force):

```
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

2.1.3 FIR Ringbuffer

Vorteil: Kein Kopieren nötig



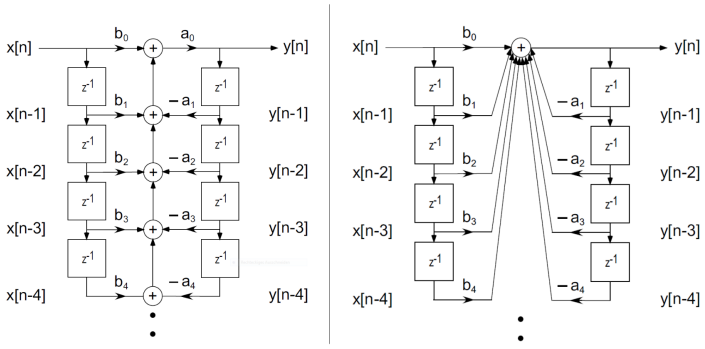
Realisierung des FIR-Ringpuffers in C:

```
void firRing()
{
    float xLeft[N+1];
    float pLeft = xLeft;
    *pLeft = CodecDataIn.Channel[LEFT]; // store input value
    output = 0 ;                        // set up for channel
    p = pLeft ;                         // save current sample pointer
    if(++pLeft > &xLeft[N])             // update pointer
    pLeft = xLeft;                      // and store
    for(i = 0 ; i <= N ; i++) {         // do LEFT channel FIR
        output+=(*p--) * B[i];          // multiply and accumulate
        if(p < &xLeft[0])               // check for pointer wrap around
        p = &xLeft[N];
    }
    CodecDataOut.Channel[LEFT]=output; // store filtered value
}
```

2.2 IIR

2.2.1 Blockschaltbild und math. Zusammenhänge:

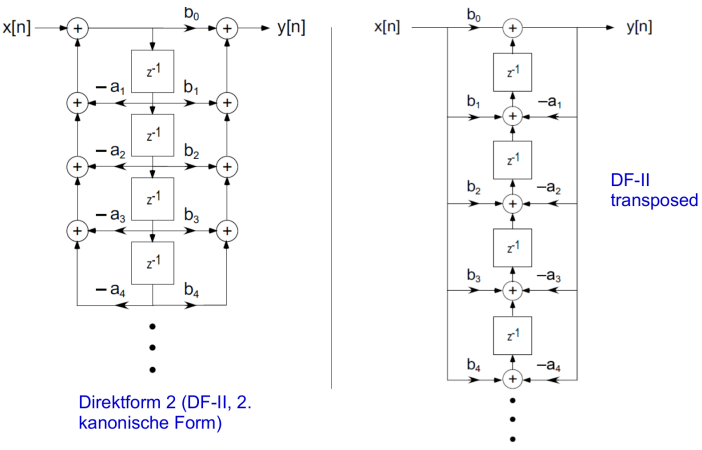
Direktform I:



Direktform 1 (DF-I, erste kanonische Form)

Direkt <=> a, b erscheinen in der Struktur unverändert

Direktform II:

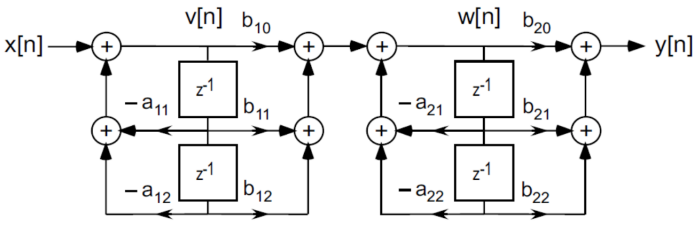


Direktform 2 (DF-II, 2. kanonische Form)

DF-II transposed

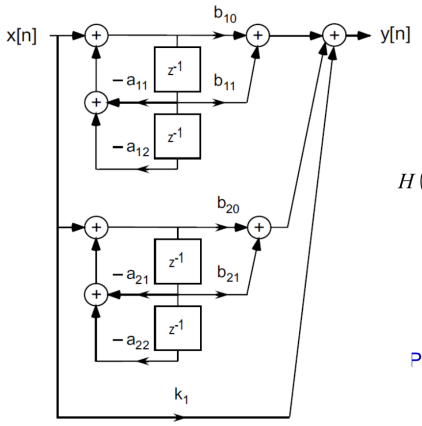
3. Kanonische Form:

$$H(z)=\frac{Y(z)}{X(z)}=\prod_{i=1}^{p}\frac{b_{i0}+b_{i1}z^{-1}+b_{i2}z^{-2}}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}$$



Dritte kanonische Form, serial second order sections, SOS (Biquad)

4. Kanonische Form:



$$H(z)=k_1+\sum_{i=1}^{p}\frac{b_{i0}+b_{i1}z^{-1}+b_{i2}z^{-2}}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}$$

Parallele Implementierung mit SOS, 4. kanonische Form

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

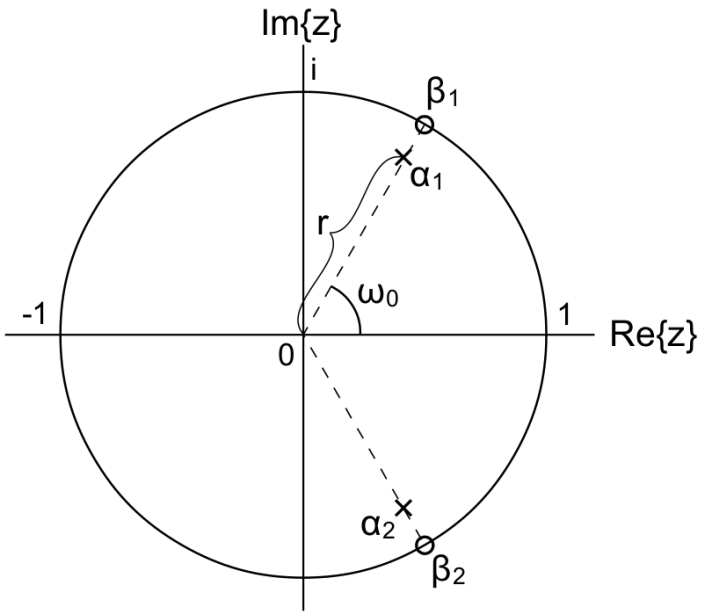
2.2.2 Realisierung des IIR in C:

```c
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

2.3.1 Allgemeines:

SOS-IIR-Notch-Filter (Beispiel):



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

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

$$k = \frac{1-2r\cos(\Omega_0)+r^2}{1-2\cos(\Omega_0)+1} \quad (5)$$

$$\Omega_0 = 2\pi\frac{f_0}{f_s} \quad (6)$$

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

2.3.2 Anwendungsbeispiel:

1. Ermitteln des spektralen Maximum in durch die FFT generierten Daten
2. Ermitteln der entsprechenden Störfrequenzen
3. Errechnen der Koeffizienten des Notch-Filters mit der Störfrequenz als Kerb-Frequenz
4. Filtern des diskretisierten Signales mit errechnetem Filter
5. Ausgabe der bearbeiteten Audio-Sequenz

3 Spektralschätzung

3.1 Fensterfunktionen

Name	Breite Hauptzipfel	Nebenzipfel-Pegel (dB)	Transitions-bandbreite (dB)	Ripple im Passband (dB)	Dämpfung im Stoppband (dB)
Rectangular	$4\pi/N$	-13.5	$1.8\pi/N$	0.75	21
Bartlett	$8\pi/N$	-27	$6.1\pi/N$	0.45	25
von Hann	$8\pi/N$	-32	$6.2\pi/N$	0.055	44
Hamming	$8\pi/N$	-43	$6.6\pi/N$	0.019	53
Blackman	$12\pi/N$	-57	$11\pi/N$	0.0017	74
Kaiser, $\alpha = 4$	$6.8\pi/N$	-30	$5.2\pi/N$	0.049	45
Kaiser, $\alpha = 8$	$10.8\pi/N$	-58	$10.2\pi/N$	0.00077	81
Kaiser, $\alpha = 12$	$16\pi/N$	-90	$15.4\pi/N$	0.000011	118

4 Oszillatoren / Signalgeneratoren

4.1 IIR-Oszillator (Digital Resonator)

4.1.1 Allgemeines:

- Oszillator auf Basis einer z-Transformation
- Anregung des Systems mit Impuls bei $k = 0$
- System mit Polen auf Einheitskreis
  → Grenzstabiles System
  → System schwingt mit konst. Frequenz

Vorgehen:

1. z-Transformation des kontinuierlichen Systems
2. Ausmultiplizieren von $H(z)$ mit $Y(z)$ bzw. $X(z)$
3. Anwenden von
   $Y(z)\cdot z^{-1} = y[n-1]$ bzw. $x(z)\cdot z^{-1} = x[k-1]$
4. Auflösen nach y(k)
5. Impuls als Eingangssignal $x[k] = [1,0,0,...]$

Vorteil:

- Minimaler Verbrauch von Speicher und Rechenleistung
- Anpassbarkeit an beliebige Funktionen
- Hohe Auflösung und Flexibilität

Nachteil:

- Frequenz muss vor Start festgelegt werden
- durch Quantisierung kann Pol aus dem Einheitskreis rutschen und instabil werden
- Oszillator muss einschwingen

$$sin(\Omega_0 k) \leftrightarrow \frac{sin(\Omega_0)z^{-1}}{1-2\cos(\Omega_0)z^{-1}+z^{-2}} \quad (8)$$

$$y(n) = sin(\Omega_0)x(n-1) + 2cos(\Omega_0)y(n-1) - y(n-2) \quad (9)$$

4.1.2 Realisierung des IIR-Oszillators in C:

```
enum lrtype {LEFT, RIGHT};
volatile union {unsigned UINT; short Channel[2];}
CodecDataIn, CodecDataOut;
float fDesired = 1000; // your desired signal frequency
float A = 32000; // your desired signal amplitude
float pi = 3.1415927; // value of pi
float theta;// digital frequency (omega0 in textbook)
float y[3] = {0, 1, 0}; // the last 3 output values.
unsigned fs = 48000; // sample frequency

void isr_resonator(){
  CodecDataIn.UINT = ReadCodecData(); // get input data samples
  theta = 2 * pi * fDesired / fs; // calc. the digital frequency
  y[0] = 2 * cosf(theta) * y[1] - y[2]; // calculate the output
  y[2] = y[1]; // prepare for the next ISR
  y[1] = y[0]; // prepare for the next ISR
  CodecDataOut.Channel[ LEFT] = A * sinf(theta) * y[0]; // just scale
  CodecDataOut.Channel[RIGHT] = CodecDataOut.Channel[LEFT];
}

WriteCodecData(CodecDataOut.UINT);
```

4.2 DDS-Oszillator

4.2.1 Allgemeines:

- Direct Digital Synthesizer
- Errechnen des Funktionsverlaufs
- Verwendung von Accumulator und Modulo-Operator
- sin() kann berechnet oder mittels Lookup-Table realisiert werden

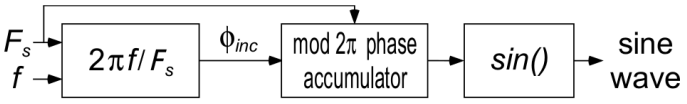
Vorteil:

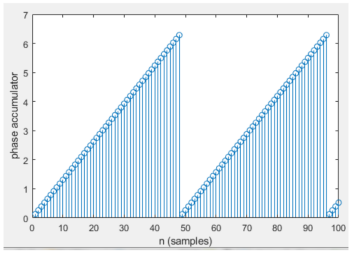
- Kann einfach bei FPGA verwendet werden

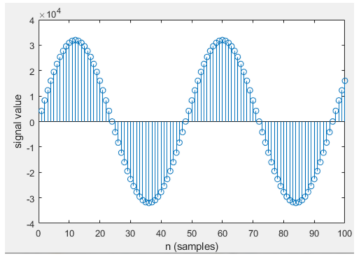
- Robuster Phasen- oder Frequenzwechsel mit sofortiger Wirkung
- kontinuierliche Signalform
- keine Einschwingzeit

Nachteil:

- Höherer Speicher- und Rechenleistungs-Bedarf
- Frequenzauflösung abhängig im Wesentlichen von Auflösung der Lookup ab







z.B NCO

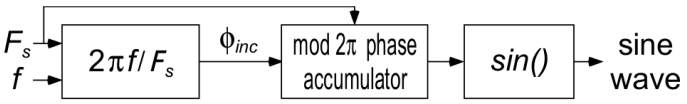
- Direct Digital Synthesizer
- Errechnen des Funktionsverlaufs
- Verwendung von Accumulator und Modulo-Operator
- sin() kann berechnet oder mittels Lookup-Table realisiert werden
- Frequenzauflösung abhängig von Wortbreite des Phasenakkumulators $\frac{f_{clk}}{2^{N_{Bits}}}$

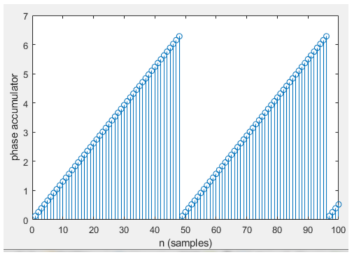
Vorteil:

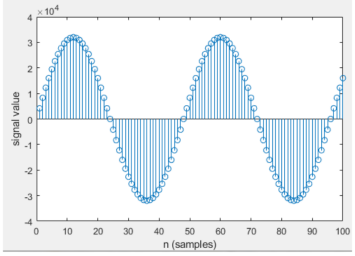
- Kann einfach bei FPGA verwendet werden
- Robuster Phasen- oder Frequenzwechsel mit sofortiger Wirkung
- kontinuierliche Signalform
- keine Einschwingzeit

Nachteil:

- Höherer Speicher- und Rechenleistungs-Bedarf
- Frequenzauflösung abhängig im Wesentlichen von Auflösung der Lookup ab







$$\varphi_{inc} = 2\pi \frac{f_0}{f_s} \tag{10}$$

$$\varphi = \varphi + \varphi_{inc} \tag{11}$$

$$\varphi_{inc} < \pi(sonstAliasing) \tag{12}$$

$$x(n) = Asin(n\varphi) \tag{13}$$

4.2.2 Realisierung des DDR-Oszillators in C:

```
float A = 32000; //signal's amplitude
float fDesired = 1000; // signal's frequency
float phase = 0; // signal's initial phase
float pi = 3.1415927; // value of pi
float phaseIncrement; // incremental phase
int fs = 48000; // sample frequency

void sineGen_ISR(){
CodecDataIn.UINT = ReadCodecData();
// algorithm begins here
phaseIncrement = 2*pi*fDesired/fs;
phase += phaseIncrement;
if (phase >= 2*pi) phase -= 2*pi;
// get input data samples
// calculate the phase increment
// calculate the next phase
// modulus 2*pi operation
CodecDataOut.Channel[ LEFT] = A*sinf(phase); // scaled L output
CodecDataOut.Channel[RIGHT] = A*cosf(phase); // scaled R output
// algorithm ends here
}

WriteCodecData(CodecDataOut.UINT);
```

4.2.3 Spezialfälle:

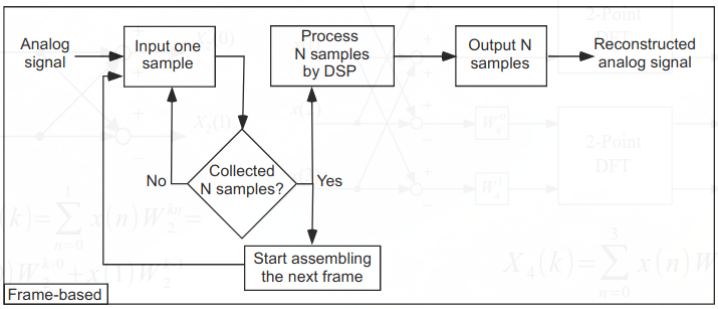
1. **Niquist-Frequenz:** $f = \frac{f_s}{2} \rightarrow \varphi_{inc} = \pi$
   $\rightarrow sin(n \cdot \pi) = 0$ bzw. $cos(n \cdot \pi) = [1, -1, 1, -1, ...]$
2. $f = \frac{f_s}{4} \rightarrow \varphi_{inc} = \frac{\pi}{2}$
   $\rightarrow sin(n \cdot \pi) = [0, -1, 0, -1, ...]$
   $\rightarrow cos(n \cdot \pi) = [1, 0, -1, 0, ...]$
3. $f_s = N \cdot f$ **(Ganzzahliges Vielfaches):**
   $\rightarrow$ nur N Werte müssen berechnet werden
   $\rightarrow cos(\varphi_{inc} \cdot n)$ für n = 0, 1, ..., N

5 Blocksignalverarbeitung:

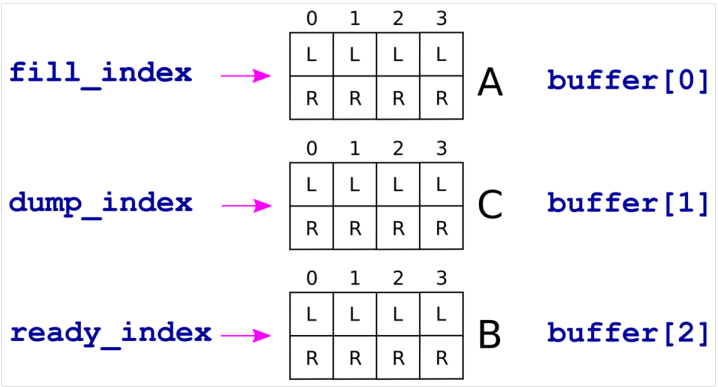
5.1 Dreifach-Puffer:

Allgemein:

- Kein Kopieren notwendig
- 3 unabhängige Buffer nötig
- 3 Pointer zeigen welcher Buffer befüllt, verarbeit bzw. entleert wird



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



Block

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

5.1.1 Realisierung eines Dreifach-Puffers in C:

```
#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 to fill
    static Uint8 dump_index = INITIAL_DUMP_INDEX; // index 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) // sover_run-Flag
            over_run = 1;
        buffer_ready = 1;
    }
        WriteCodecData(CodecDataOut.UINT);// send output data to  port
}
```

5.1.2 Blocksignalverarbeitung mit DMA

Vorteil: Prozessor muss sich nicht mit Befüllen beschäftigen sondern kann verarbeiten
→ Geschwindigkeitsvorteil

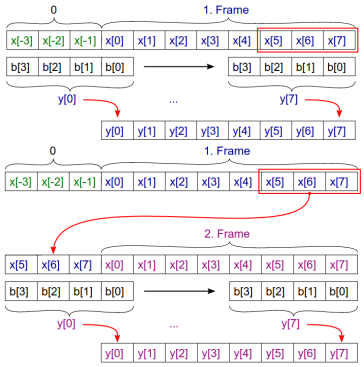
- DMA kopiert Sample von ADC nach Eingangsbuffer
- DMA kopiert Processed von Ausgangsbuffer nach DAC
- DMA generiert Interrupt, wenn N Samples transfert wurden → Buffer-Swap

```
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
}
```

5.2 FIR mit Blocksignalverarbeitung

5.2.1 Allgemeines:

Bsp. Ordnung Filter N = 4, Framesize = 8



Problem: Die Start und Endzustände müssen jeweils berücksichtigt werden, um den FIR korrekt zu implementieren.
→ Bei Frame-Übergängen müssen die N-1 letzten Werte des letzten Frames berücksichtigt werden.
→ Audiotechnisch würde dies ein “Knacken-“ und “Klicken“ hervorrufen
»»»»> christoph **Lösung:** Der Puffer muss groß genug sein, um sowohl die Frame-Werte als auch die nötigen Randwerte speichern zu können (Framesize += N).
Latenz: $\frac{2N}{f_c lk}$ 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]

5.2.2 Realisierung eines FIR mit Blocksignalverar. in C:

»»»»> christoph

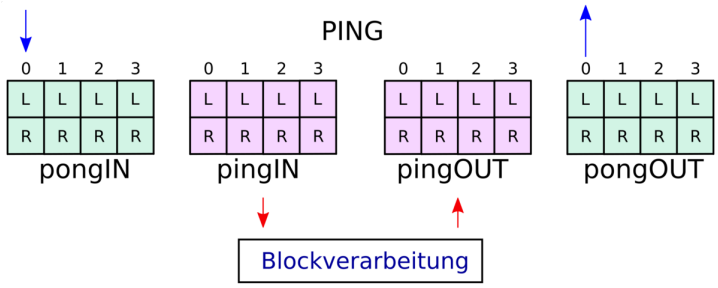
```
void ProcessBuffer()
{
    short *pBuf = buffer[ready_index];
    // extra buffer room for convolution "edge effects"
    // N is filter order from coeff.h
    static float Left[BUFFER_COUNT+N]={0}, Right[BUFFER_COUNT+N]={0};
    float *pL = Left, *pR = Right;
    float yLeft, yRight;
    int i,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

5.2.3 Allgemein:

Vorteile:

- Kein Kopieren zu Float-Arrays
- Robuste Buffer-Identifizierung, die Breakpoints unterstützt

Nachteile: Aufwendiges Schieben in dem Filterspeicher



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

6 FFT

6.1 Allgemein:

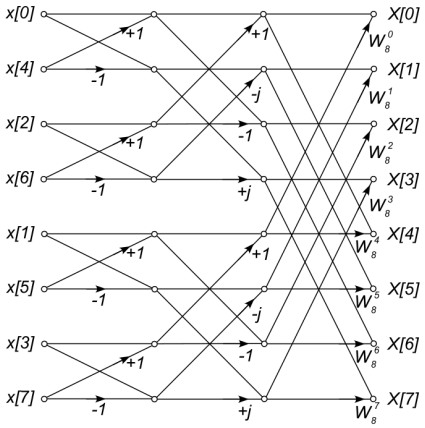
- basiert auf dem **Devide-and-Conquer** Prinzip,
- Zwischenergebnisse werden wiederverwendet
- wesentliche Beschleunigung im Vergleich zur DFT
- **Ordnung DFT:** $N^2 \rightarrow$ **Ordnung FFT:** $N \cdot log_2(N)$

Mögliche Realisierungsformen:

- Decimation in Frequency
- Decimation in Time

6.2 Decimation in Time (DIT)

Butterfly-Diagramm DIT-FFT radix-2 (N=8):



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

Twiddle-Faktor: $w_N^{nk} = e^{-j\frac{2\pi nk}{N}}$ (14)

$$w_N^{2nk} = e^{-j\frac{4\pi nk}{N}} = w_{\frac{N}{2}}^{nk} \quad (15)$$

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

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

$$Y(n) = \sum_{k=0}^{N/2-1} y(2k) w_{\frac{N}{2}}^{nk} + w_N^n \sum_{k=0}^{N/2-1} y(2k+1) w_{\frac{N}{2}}^{nk} \quad (18)$$

(19)

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

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

$$Y(n+\frac{N}{2}) = \sum_{k=0}^{N/2-1} y(2k) w_{\frac{N}{2}}^{(n+\frac{N}{2})k} + w_N^{n+\frac{N}{2}} \sum_{k=0}^{N/2-1} y(2k+1) w_{\frac{N}{2}}^{(n+\frac{N}{2})k} \quad (21)$$

mit

$$\textbf{Twiddle-Faktor: } w_{\frac{N}{2}}^{(n+\frac{N}{2})k} = w_{\frac{N}{2}}^{nk} \cdot \underbrace{w_{\frac{N}{2}}^{k\frac{N}{2}}}_{=1} = w_{\frac{N}{2}}^{nk} \quad (22)$$

$$w_N^{n+\frac{N}{2}} = w_N^n \cdot \underbrace{w_N^{\frac{N}{2}}}_{=-1} = -w_N^n \quad (23)$$

folgt

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

$$Y(n+\frac{N}{2}) = \sum_{k=0}^{N/2-1} y(2k) w_{\frac{N}{2}}^{nk} - w_{\frac{N}{2}}^n \sum_{k=0}^{N/2-1} y(2k+1) w_{\frac{N}{2}}^{kn} \quad (25)$$

$$Y_{left}(n) = Y_{even}(n) + w_{\frac{N}{2}}^n Y_{odd}(n) \quad (26)$$

$$Y_{right}(n) = Y_{even}(n) - w_{\frac{N}{2}}^n Y_{odd}(n) \quad (27)$$

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

7 Multirate:

7.1 Änderung der Abtastfrequenz:

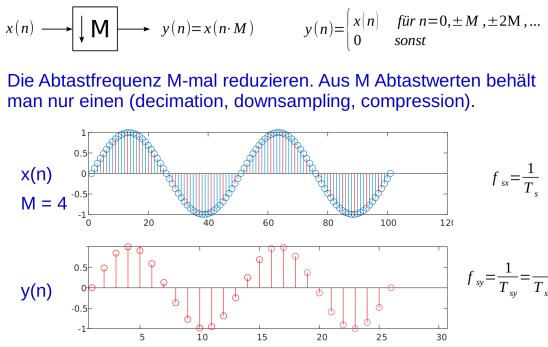
Wunsch: Annpassung der Sample-Rate an das abzutastende Signal → Variable Abtastfrequenz mit gegebenen Samples

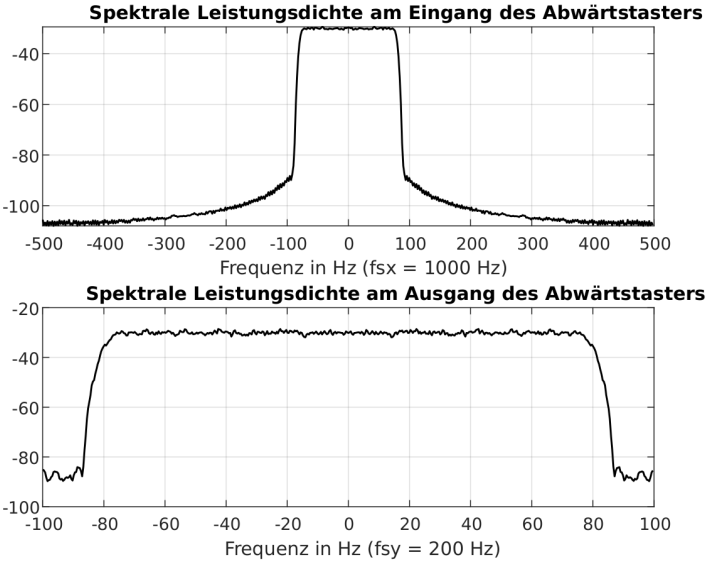
7.1.1 Dezimieren:

Vorteile einer Reduzierten Taktzahl:

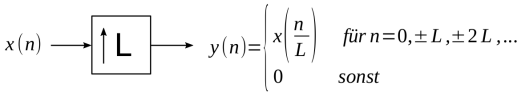
- geringerer Energieverbrauch
- Reduzieren des Frequenzbandes (“nur was interessiert“)

ACHTUNG: Anti-Aliasing-Filter sollte vorgeschalten werden

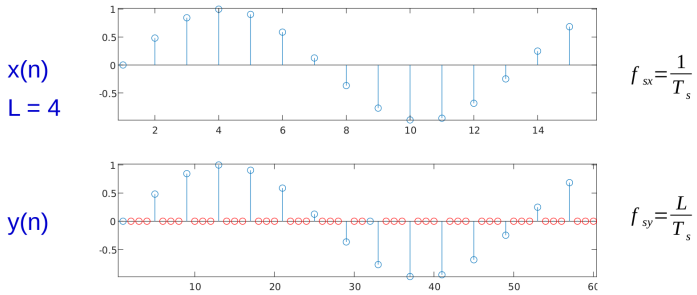


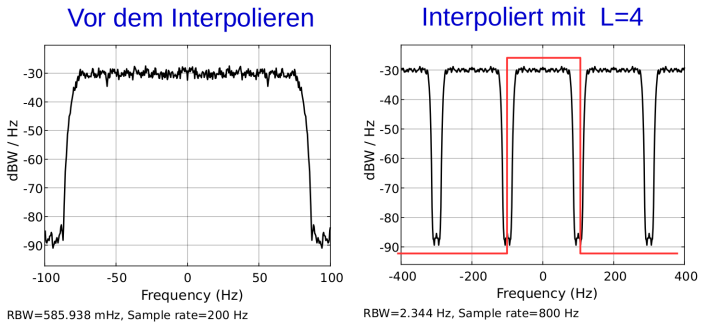


7.1.2 Interpolation:

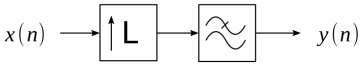


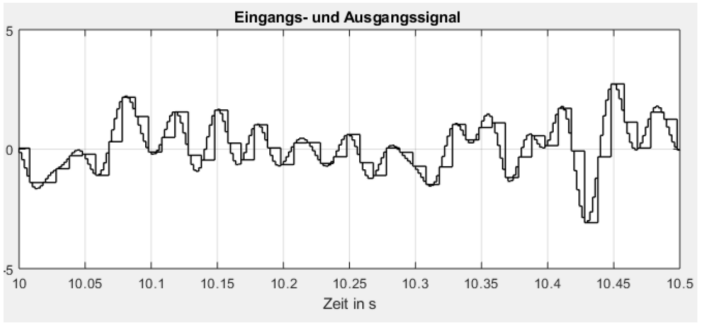
Die Abtastfrequenz L-mal erhöhen. Nach jedem Abtastwert L-1 Nullen einfügen (interpolation, expansion).

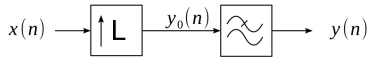


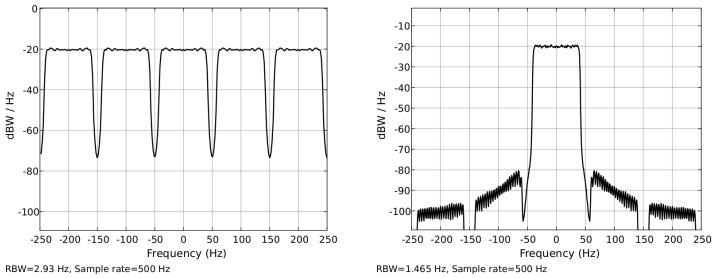


Filterung nach Interpolation:









7.1.3 Problem bei Interpolation oder Dezimierung:

Problem:

- eingesetzten Filter haben harte Anforderungen
- Phasenlinearität verlangt nach komplexen FIR-Filtern

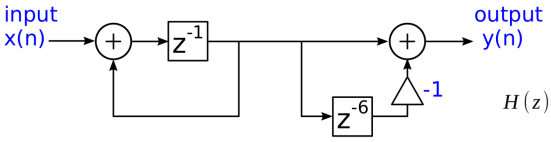
Lösung:

- Dezimieren / Interpolieren in mehreren Stufen (einfachere Filter)
- Verwendung von Alternativlösungen für Filterung → z.B.: CIC-Filter

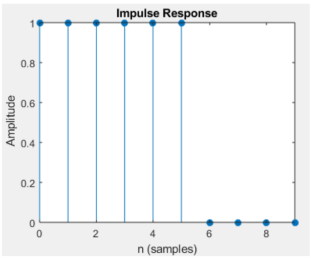
7.2 Cascaded Integrator Comb (CIC):

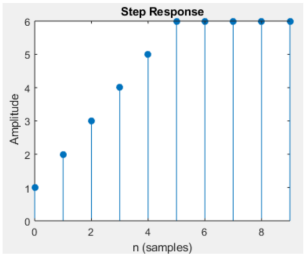
7.2.1 Allgemeines :

- effiziente Architektur zum Filtern bei hohen Dezimations- / Interpolations-Raten
- **Verhalten:** Moving-Average-Filter
- benötigt keine Koeffizienten-Speicher
- vergleichbarer nichtrekursiver FIR-Filter würde 5 ($z^{-6} \rightarrow D-1=6-1=5$) Addierer benötigen



$$H(z)=\frac{Y(z)}{X(z)}=\frac{1-z^{-6}}{1-z^{-1}}$$

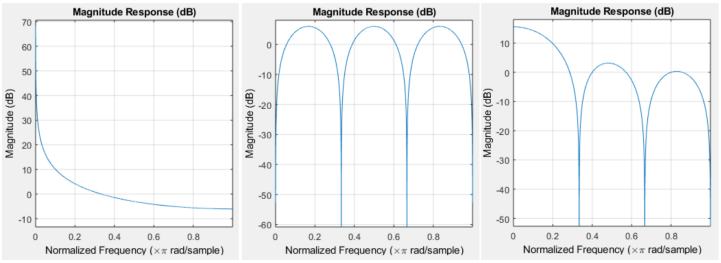




$$H(z)=\frac{Y(z)}{X(z)}=\frac{1-z^{-6}}{1-z^{-1}}=1+z^{-1}+z^{-2}+z^{-3}+z^{-4}+z^{-5}$$

„moving average“

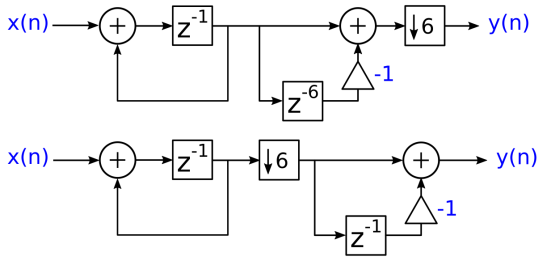
Integrator Comb CIC



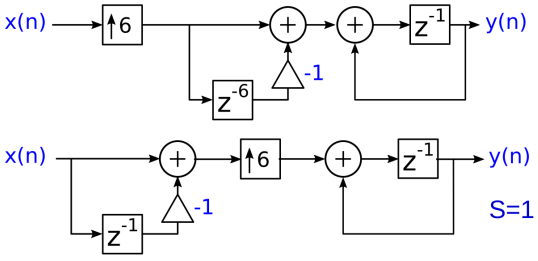
7.2.2 Verbesserungen:

- Kaskadierung von CIC-Filtern (Verbesserung der Filterwirkung)
- Anwendung der Noble-Äquivalenzen (Reduziert Speicherzellen-Anzahl)

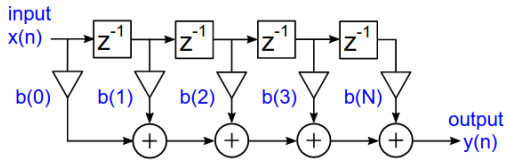
CIC-Dezimator (Noble):



CIC-Interpolator (Noble):



8 FIR in FPGA

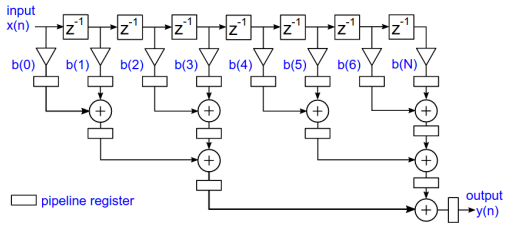


Pro Takt müssen 1 Multiplizierer und N Addierer durchlaufen werden.
Die max. Taktfrequenz f_{clk} ist

$$f_{clk} = \frac{1}{T_{mul} + NT_{add}} \tag{28}$$

8.1 FIR - Pipelined

Jede Operationsstufe wird mit einem Pipelineregister gebuffert.
Vorteil: Massive Erhöhung der Taktfrequenz
Nachteil: zusätzliche Register, Latenz (Zeitl. Versatz (hier 4))

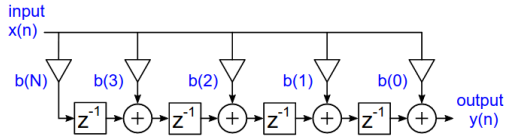


Die max. Taktfrequenz f_{clk} annähernd durch T_{mul} definiert:

$$f_{clk} = \frac{1}{max(T_{mul}, T_{add})} = \frac{1}{T_{mul}} \tag{29}$$

8.2 FIR - transposed (DF1)

Bevorzugte Implementierungsvariante



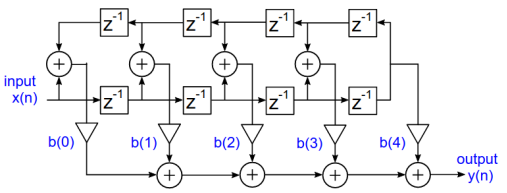
Vorteil: Pipeline-Register für Addierer werden eingspart, keine Latenz

$$f_{clk} = \frac{1}{T_{mult} + T_{add}} \tag{30}$$

8.3 FIR - lineare Phase

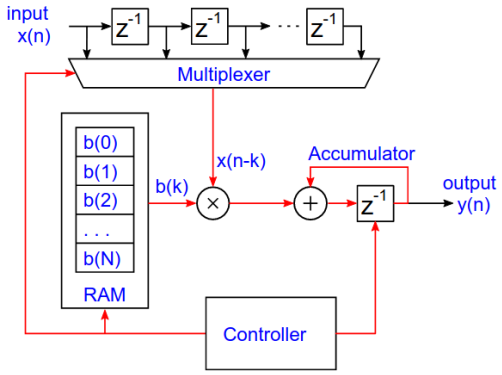
Voraussetzung: NST "gespiegelt" am Einheitskreis

Vorteil: Koeffizienten sind symmetrisch
→ Anzahl an Multiplizierer kann halbiert werden



8.4 FIR - seriell

Addierer und Multiplizierer sind teuere Ressourcen im FPGA. Ein Controller Steuert einen MUX(Inputwerte) und RAM(Koeffs) und AKKU an.

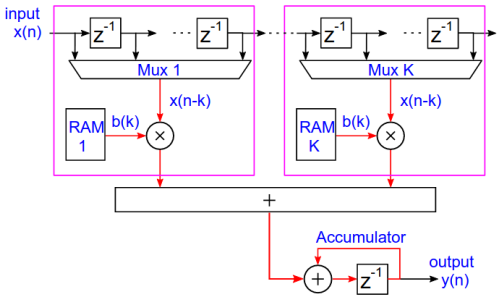


Die Maximale Taktfrequenz des FPGA f_{clk} hängt von der Filterordnung N und der Abtastfrequenz f_A ab und muss min.

$$f_{clk} = M \cdot f_A = (N-1) \cdot f_A \tag{31}$$

8.5 FIR - semi-parallel

Bei der Seriellen Architektur ist die max. Filterordnung stark begrenzt. ($f_{max,FPGA}$ im 3st. MHz-Bereich) Für höhere Filterordnung kann die semi-parallele Architektur verwendet werden.

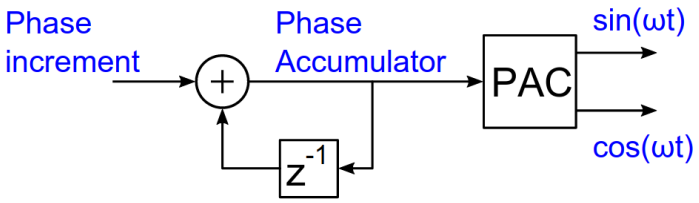


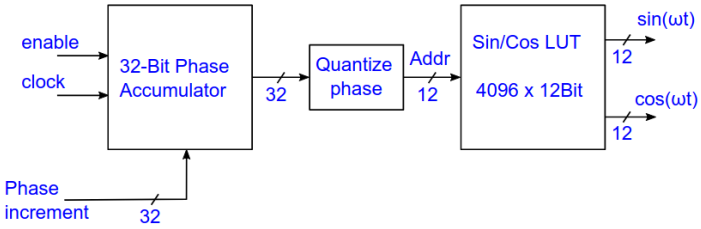
K: Sektionen **M:** Multiplizierer
Die max. Taktfrequenz des FPGA muss jetzt nur noch ein K-tel so hoch sein:

$$f_{clk} = \frac{M}{K} \cdot f_A = \frac{N-1}{K} \cdot f_A \tag{32}$$

9 NCO

- NCO = "Numerically Controlled Oscillator"
- Phasenakkumolator, der Jeden Takt um ein Phaseninkrement μ erhöht wird
- Ausgang des Counters wird mit Look-Up-Table (LUT) in Signalform (sin,cos,sägezahn) umgewandelt (**PAC:** Phase Amplitude Converter)
- LUT ist mit $N = 2^n$ 12-bit breiten Werten gefüllt





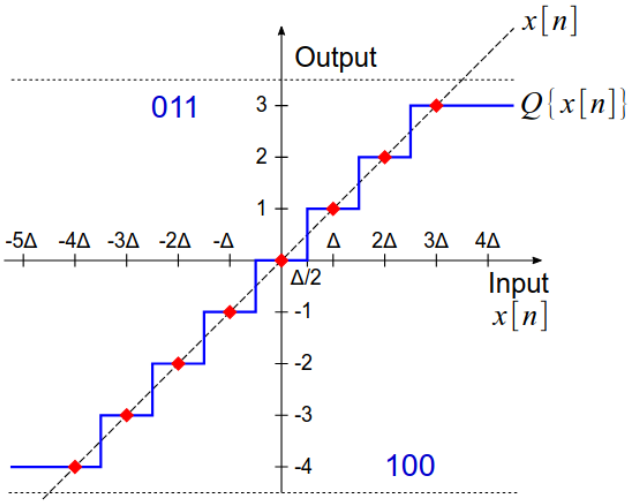
$$\mu = N\frac{f_d}{f_s} \tag{33}$$

10 Quantisierung

Ein ADC gibt Ganzzahlwerte von 0 bis N zurück. Wir müssen diese Ganzzahl auf die Referenzspannung U_{Ref} beziehen. Der exakte Spannungswert $U(k) = U_{Ref} \cdot \frac{k}{N}$.
Dieses Ergebnis wird in gewisser Weise durch eines der Verfahren quantisiert:

- Runden (Bevorzugte Variante)
- Abschneiden (Wert fällt auf nächstiefere)
- Betragsabschneiden (Wert fällt Richtung 0)

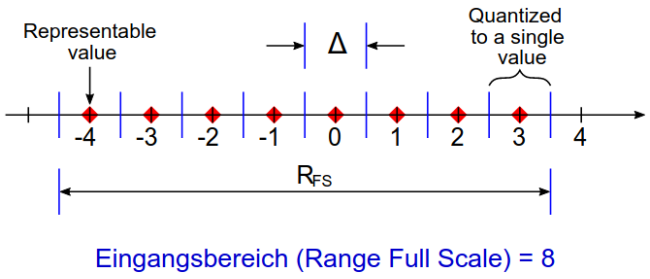
10.1 Quantisierung mit Runden



Es ensteht ein Fehler beim Quantisieren $e(n)$

$$e(n) = Q\{x(n)\} - x(n) \tag{34}$$

$$-\frac{\Delta}{2} < e(n) \leq \frac{\Delta}{2} \tag{35}$$



Eingangsbereich (Range Full Scale) = 8

$$\Delta = \frac{R_{FS}}{2^B} \qquad B = 3$$

Der Volle Eingangsbereich R_{FS} teilt sich auf $2^B - 1$ Teile der Länge Δ auf.

$$R_{FS} = \Delta \cdot 2^B \tag{36}$$

10.2 Modelierung des Quantisierungsfehlers

Versuch den Quantisierungsfehlers zahlenmäßig zu erfassen. $e(n)$ kann durch unkorreliertes, mittelwertfreies, gleichverteiltes, weißes Rauschen modeliert werden. Dieses Modell ist zulässig wenn

- Quantisierungsstufe Δ klein in Vergleich zur Signalamplitude
- Das Signal soll einige Q-Stufen zwischen zwei Abtastwerten überqueren
- Kein Überlauf und keine Saturation
- x(n) nicht periodisch mit Vielfaches von f_A
- Rauschen muss vom Signal kommen

10.3 Quantisierungsrauschleistung

$$E_e = \sigma_e^2 = \frac{\Delta^2}{12} = \frac{R_{FS}^2}{12 \cdot 2^{2B}} \tag{37}$$

(38)

Bsp:

Werte für Q1.15: $R_{FS} = 2$, B=16

$$\sigma_e^2 = \frac{2^2}{12 \cdot 2^{2 \cdot 16}} = 77.61 \cdot 10^{-12}$$

$$\sigma_e^2[dB] = 10\log_{10}(77.61 \cdot 10^{-12}) = -101\,dB$$

10.4 Signal-to-Quantization Noise Ratio SQNR

$$S_{max} = \frac{A_{max}^2}{2} \approx \frac{R_{FS}}{8} = 2^{2B-3} \cdot \Delta^2 \tag{39}$$

mit Quantisierungsrauschleistung σ_e^2 ergiebt sich

$$SQNR_{max} = \frac{S_{max}}{\sigma_e^2} = \frac{2^{2B-3} \cdot \Delta^2}{\frac{\Delta^2}{12}} = (6,02B + 1,76)dB \tag{40}$$

Es ergiebt sich eine Verschlechterung von $\frac{6dB}{Bit}$

10.5 Addieren

Pro Addition verlängert sich das Ergebnis um 1 Bit

1. im FPGA die Wortbreite nach jedem Addierer erhöhen
2. auf dem DSP wählt man die Wortlänge (16 / 32 Bits)
3. Wenn zu lang, herunter skalieren (LSBs abschneiden)
4. Bedingt herunter skalieren und die Anzahl der Skalierungen merken („block floating point“)
5. Nichts tun, wenn die Signale klein genug sind

10.6 Multiplizieren

Beim Multiplizieren ist die Ergebnislänge die Summe der Längen der Faktoren.
Konsequenzen:

1. Nach jeder Multiplikation ist eine Skalierung wegen der schnell wachsender Länge denkbar
2. In FPGA ist es ratsam, die Längen der fest implementierten Multiplizierer auszunutzen (z.B. 18x18)
3. Wenn sinnvoll, Multiplikation mit Konstanten als CSD implementieren (dann nur Additionen)

10.7 Overflow

Wie viele Bits brauchen wir für den max. Ausgang $y(n)$

Pessimistisch:
Wenn der Betrag des Eingangssignals ≤ 1, dann kann schlimmsten Falls bei der Faltung die komplette Impulsantwort $h(k)$ unter dem Eingangssignal sein. Daraus folgt:

$$|x(n)| \leq 1 \rightarrow |y(n)| \leq \sum_{k=0}^{N} |h(k)| \tag{41}$$

Für ein Schmalbandiges Signal (Sinus) ist das Ausgangssignal höchstens der größte Spektralwert der Impulsantwort.

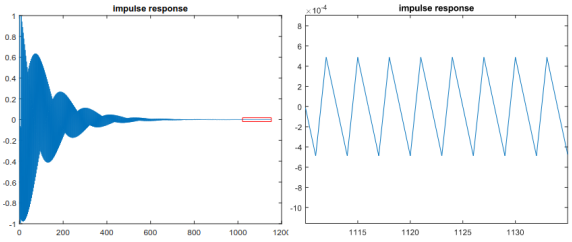
$$|x(n)| \leq 1 \rightarrow |y(n)| \leq max\{\left|H(e^{j\Omega})\right|\} \tag{42}$$

Durch die Quantisierung überlagert sich der Fehler mit der Übertragungsfunktion $H(e^{j\Omega})$

$$\left|\Delta H(e^{-j\Omega}\right| \leq \sum_{k=0}^{N} |\Delta h(k)|\left|e^{-jk\Omega}\right| \tag{43}$$

10.8 Grenzzyklus

Schwingungen die sich durch die Quantisierung einstellen.



11 Nützliche Code-Ausschnitte:

Maximalwert-Ermittlung z.B. für Sepktralauswertung:

```
max_value = 0 ; //Startwert des Vergleichs
max_idx = 2 ; //Startindex des Vergleichs
for(int i = 2; i<(N-1); i++){
  if( y[i] > max_value){
    max_value = y[i];
    max_idx = i;
  }
}
```