Festkomma-Signalprozessor
fixed-point DSP

©Hanspeter Hochreutener
Zentrum für Signalverarbeitung und Nachrichtentechnik, ZHW

9. Juni 2007

Was ist bei „fixed-point DSPs“ speziell?
Die Overflow- und Quantisierungs-Problematik wird aufgezeigt.
Die „fractional“-Zahlenformate, die sättigende Arithmetik,
Guard-Bits und Compiler-Intrinsics werden vorgestellt.
Das Vorgehen für die Implementation von FIR- und IIR-Filtern und
das Skalieren von Signalen und Koeffizienten wird im Detail beschrieben.
Performance-Messungen und Optinmierungs-Tipps runden den Text ab.

Literatur

[von Grünigen] Daniel Ch. von Grünigen, „Digitale Signalverarbeitung“, 3. Auflage, Fachbuchverlag Leipzig, ISBN3-446-22861-6

[Kuo] Sen M. Kuo, Bob H. Lee; „Real-Time Digital Signal Processing“, John Wiley & sons, ISBN 0-470-84137-0

[Matlab] Matlab-Hilfe zu den Befehlen „tf2sos“ und „filternorm“

[TMS320C55x] „TMS320C55x Optimizing C/C++ Compiler User's Guide“, Texas Instruments, spru281f.pdf, http://www.ti.com/litv/pdf/spru281f

Inhaltsverzeichnis

1 Besonderheiten von Festkomma-DSPs

1.1 Fliesskomma- oder Festkomma-DSP?

Eine Festkommazahl hat den Dezimalpunkt immer an der gleichen Stelle. Üblich ist es den Dezimalpunkt an die vorderste Stelle zu setzen, womit sich ein Zahlenbereich von $-1 \leq zahl < +1$ ergibt.

Eine detaillierte Definition der Festkomma-Formate finden Sie im Abschnitt 1.3.1.

DSP-Eigenschaft	**Fliesskomma**	**Festkomma**
Wortbreite	32 Bit (8 Bit Exponent)	16-24 Bit (intern 32-40 Bit)
Numerische Stärke	grosser Dynamikumfang kein Überlauf	hohe Genauigkeit schnell
Algorithmen- Implementation	C, Assembler	Assembler, Compiler-Intrinsics, C
Chipfläche, Stromverbrauch und Hardware-Kosten	höher	niedriger
Entwicklungs-Aufwand	geringer	grösser
Schwierigkeit		richtige Skalierung

Tabelle 1: Vergleich von Fliesskomma- und Festkomma-DSP

Aus den wichtigsten Eigenschaften und Unterschieden in der Tabelle ist erkennbar, dass die Wahl Festkomma- oder Fliesskomma-DSP stark vom Einsatzgebiet und den Stückzahlen abhängt.

1.2 Quantisierung und Überlauf (overflow)

In jedem digitalen System werden Zahlen durch eine endliche Anzahl Bits dargestellt. Dadurch entstehen Rundungsfehler (= Quantisierungsfehler). Wenn der zulässige Zahlenbereich überschritten wird, liegt ein Überlauf (overflow) vor.

1.2.1 Quantisierung bei der Analog-Digital-Wandlung

Ein (analoger, zeitkontinuierlicher) Spannungsverlauf wird auf eine (digitale, zeitdiskrete) Zahlenfolge abgebildet. Die möglichen Zahlenwerte sind durch die Anzahl Bits des AD-Wandlers gegeben. Die analoge Spannung muss auf den nächsten Zahlenwert gerundet werden. Dieser Rundungsfehler ist zufällig und wirkt sich wie ein Rauschsignal aus (Quantisierungsrauschen).

Für ein Sinus-Signal das W Bit eines DA-Wandler ausnutzt berechnet sich das Signal-to-Noise-Ratio (SNR) zu:

$$SNR \approx W \cdot 6\text{dB} \tag{1}$$

Beispiel:
16-Bit-AD-Wandler mit Eingangs-Spannungsbereich: ± 2.5V
Sinus-Signal-Amplitude: ± 0.1V
Ausgenutzt wird $\frac{0.1\text{V}}{2.5\text{V}} = 0.04 = \frac{1}{25} \approx \frac{1}{2^5} \Rightarrow -5$Bit
Die 5 höchstwertigen Bits des AD-Wandlers werden nicht genutzt, da er nur zu ungefähr $\frac{1}{25}$ ausgesteuert wird.
$SNR \approx (16 - 5) \cdot 6\text{dB} = 66\text{dB}$

Der AD-Wandler soll möglichst gut ausgesteuert werden.
Übersteuern des AD-Wandlers muss aber unbedingt vermieden werden.

Die Aussteuerung des Eingangssignals lässt sich über die Referenzspannung des AD-Wandlers oder über einen vorgeschalteten analogen Verstärker oder Spannungsteiler anpassen.
Die Aussteuerung des Ausgangssignals lässt sich über die Referenzspannung des DA-Wandlers oder über einen nachgeschalteten analogen Verstärker oder Spannungsteiler anpassen.

1.2.2 Quantisierung der Filterkoeffizienten

Auch die Filterkoeffizienten werden in einem DSP durch eine endliche Anzahl Bits dargestellt. Dadurch werden die Nullstellen und Pole leicht verschoben.

Insbesondere Pole und Nullstellen in der Nähe von z = 1 (tiefe Frequenzen) und bei z = -1 (hohe Frequenzen in der Nähe von fs/2) sind stark betroffen [von Grünigen, Kap. 6.4.3].

Für ein stabiles System müssen alle Pole innerhalb des z-Einheitskreises liegen. Falls Pole in der Nähe des Einheitskreises liegen, besteht die Gefahr, dass sie durch Quantisierung ausserhalb zu liegen kommen und zu einem instabilen System führen.

Weil nur IIR-Filter Pole besitzen, betrifft das Quantisierungsproblem die FIR-Filter in einem viel geringeren Ausmass.

Massnahmen bei IIR-Filtern:

- **Möglichst tiefe Abtastfrequenz fs wählen, damit die Pole möglichst weit auseinanderrücken.**
- **Biquad-Kaskaden (second order sections) verwenden, damit sich ein Quantisierungsfehler nur auf ein einziges Polpaar auswirkt.**

1.2.3 Quantisierung und Überlauf beim Rechnen

Bei der Addition von Festkomma-Zahlen kann der Betrag des Resultats > 1 werden (overflow). Bei normaler Arithmetik hat das Resultat in diesem Fall ein falsches Vorzeichen, was gravierende Auswirkungen auf das Signal hat. Festkomma-DSPs entschärfen das Problem mit sättigender Arithmetik.

Bei der Multiplikation von Festkomma-Zahlen kann kein Überlauf auftreten, aber das Resultat kann sehr klein werden. Diese Quantisierung verschlechtert das SNR.

Signale und Koeffizienten müssen so skaliert werden, dass der Zahlenbereich (ohne overflow) möglichst voll ausgenutzt wird (siehe Abschnitt 1.4).
Um Rundungsfehler bei Zwischenresultaten zu vermeiden, können diese mit mehr Bits gespeichert werden (Datentyp „long“ statt „int“ oder Guard-Bits verwenden).

1.3 Festkomma-Zahlenformat und -Arithmetik

1.3.1 Die „fractional“-Zahlenformate

Die Erklärungen gelten für 16-Bit-Zweierkomplement-Zahlen, die übelicherweise in fixed-Point-DSPs verwendet werden. Das erste Bit ist das Vorzeichenbit.

Im Speicher ist für eine Zahl ein Bitmuster aus 16 Bit abgelegt. Je nach Datentyp wird dieses Bit-Muster unterschiedlich interpretiert. Dank der Typisierung bei der Variablen-Deklaration kann der Compiler sicherstellen, dass eine Variable immer gleich interpretiert wird. Wenn ein Bitmuster einmal z.B. als „int“ und einmal als „float“ interpretiert würde, wäre das Resultat unbrauchbar.

Bit-Muster	**Dezimal-Wert**
$b_0\ b_1\ b_2\ b_3\ b_4\ ...\ b_{14}\ b_{15}$	$-b_0 \cdot 2^{15} + b_1 \cdot 2^{14} + \ ... \ + b_{14} \cdot 2^1 + b_{15} \cdot 2^0$

Tabelle 2: int-Zahlenformat

Bei FestkommavZahlen gibt es eine Schwierigkeit. Wenn zwei 16-Bit-Zahlen miteinander multipliziert werden, entsteht eine 32-Bit-Zahl. Wie soll dieses Resultat nun in einer 16-Bit-Variablen gespeichert werden? Bei den MAC-Operationen (multiply and accumulate) im DSP gibt es sehr viele Multiplikationen. Um damit einfacher umgehen zu können, hat man das „fractional“-Zahlenformat mit einem Wertebereich von ± 1 definiert. Multipliziert man zwei solche fractional-Zahlen miteinander ist das Resultat ebenfalls im Wertebereich von ± 1 und kann problemlos weiterverwendet werden. DSPs haben spezielle Hardware-Multiplizierer, welche direkt mit fractional-Zahlen arbeiten.

Bit-Muster	**Dezimal-Wert**
$b_0\ b_1\ b_2\ b_3\ b_4\ ...\ b_{14}\ b_{15}$	$-b_0 \cdot 2^0 + b_1 \cdot 2^{-1} + \ ... \ + b_{14} \cdot 2^{-14} + b_{15} \cdot 2^{-15}$

Tabelle 3: Q15-fractional-Zahlenformat

Ein Vergleich der Tabellen 2 und 3 zeigt, dass es im gespeicherten Bitmuster keinen Unterschied gibt. Letztlich ist die Stelle des Dezimalpunktes eine reine Konvention, welche der Programmierer beachten muss. Ein Bitmuster kann also je nach Bedarf als Integer- oder als Q15-Wert interpretiert werden.

Die Position des Dezimalpunktes wird in den Variablen nicht gespeichert. Der Programmierer ist selbst dafür verantwortlich, dass bei Festkomma-Zahlen das „Bit=MMuster“ im Speicher richtig verwendet wird.

Die Tabelle 4 zeigt erläuternde Beispiele:

Das Q0.15-Format bedeutet kein Vorkommabit und 15 Nachkommabits. Oft spricht man abgekürzt vom Q15-Format. Der Wertebereich umfasst ± 1.
Im Texas-Instruments-DSP TMS320C55x entspricht der zu „int“ kompatible Datentyp „DATA“ dem Q15-Format.

Das Q1.14-Format (kurz Q14) hat 1 Vorkomma- und 14 Nachkommabits. Der Wertebereich ist ± 2.

Das Q15.0-Format hat 15 Vorkomma- und keine Nachkommabits; das ist das normale int-Format mit einem Wertebereich von $-32'768..32'767$

Das Q0.31-Format (kurz Q31) hat kein Vorkommabit und 31 Nachkommabits. Der Quantisierungsfehler ist also 2^{16} mal kleiner als beim Q0.15-Format. Der Wertebereich umfasst ± 1.

Binär	Hexadezimal	Integer-Wert	Q15-Wert
0000000000000000	0000	0	0
0000000000000001	0001	1	0.0000305...
1111111111111111	FFFF	-1	-0.0000305...
0111111111111111	7FFF	32'767	0.9999695...
1000000000000000	80000	-32'768	-1
0100000000000000	4000	16'384	0.5
0110000000000000	6000	24'576	0.75
1110000000000000	C000	-8'192	-0.25

Tabelle 4: Festkomma-Zahlenformate

Im Texas-Instruments-DSP TMS320C55x entspricht der zu „long“ kompatible Datentyp „LDATA“ dem Q31-Format.

Umwandlung	Operation	Bemerkung
Q0.15 ⇒ int	intZahl = Q0.15Zahl·2^{15}	
int ⇒ Q0.15	Q0.15Zahl = intZahl/2^{15}	
Q0.15 ⇒ Q1.14	Q1.14Zahl = Q0.15Zahl » 1	um 1 Position nach rechts schieben
Q1.14 ⇒ Q0.15	Q0.15Zahl = Q1.14Zahl « 1	**Overflow, falls Betrag ≥ 1**
Q0.15 ⇒ Q0.31	Q0.31Zahl = Q0.15Zahl « 16	um 16 Position nach links schieben
Q0.31 ⇒ Q0.15	Q0.15Zahl = Q0.31Zahl » 16	**letzte 16 Bits abschneiden**

Tabelle 5: Zahlenformate umwandeln/umrechnen/uminterpretieren

1.3.2 Sättigende Arithmetik

Mit den Q15- und Q31-Zahlenformaten ist ein Überlauf bei der Multiplikation ausgeschlossen. Bei Addition und Subtraktion sind Überläufe hingegen weiterhin möglich. Das vorderste Bit des Resultats hat im Speicher keinen Platz, was einem massiven Fehler gleichkommt.

Die Tabelle 6 illustriert den Sachverhalt beispielhaft:

Operation (Q15)	mit Überlauf	mit Sättigung
0100000000000000 + 0100000000000000 0.5 + 0.5	1000000000000000 -1.0 Vorzeichenbit!	0111111111111111 0.9999695...
0110000000000000 + 0110000000000000 0.75 + 0.75	1100000000000000 -0.5	0111111111111111 0.9999695...
1100000000000000 + 1100000000000000 -0.5 + (-0.5)	1000000000000000 -1.0 noch im Zahlenbereich	1000000000000000 -1.0
1010000000000000 + 1010000000000000 -0.75 + (-0.75)	0100000000000000 0.5	1000000000000000 -1.0

Tabelle 6: Vergleich Überlauf und sättigende Arithmetik

DSPs haben ein „overflow“-Bit, das gesetzt wird, wenn der Zahlenbereich überschritten wird. Wenn das „saturate“-bit gesetzt ist, erscheint bei overflow der maximal darstellbare Wert mit dem richtigen Vorzeichen. Aus der Tabelle 6 ist erkennbar, dass der Fehler beim gesättigten Wert viel kleiner ist und auf jeden Fall das Vorzeichen stimmt!

Bei Filteralgorithmen mit vielen MAC-Operationen muss unbedingt von der sättigenden Arithmetik Gebrauch gemacht werden. Überläufe von Zwischenresultaten erzeugen so lediglich ein etwas schlechteres SNR.

Sättigende Arithmetik gibt es in C leider nicht! Mit DSPs gibt es zwei Möglichkeiten von der sättigenden Arithmetik Gebrauch zu machen:

- **Programmieren in Assembler**
- **Benutzen von Compiler-Intrinsics**

1.3.3 Guard-Bits

Bei richtig skalierten Signalen und Filterkoeffizienten können Zwischenresultate ausserhalb ±1 vorkommen, auch wenn das Schlussresultat innerhalb ±1 liegt. Bei Festkomma-DSPs sind die internen Register deshalb meist 8 Bit länger. Es können also mindestens 256 Additionen/Subtraktionen ausgeführt werden, bevor das Register überläuft.

Die Guard-Bits sind Bestandteil der DSP-internen Register und können deshalb nur in Assembler-Programmen genutzt werden.

1.3.4 Compiler-Intrinsics

Spezielle Fähigkeiten der DSPs, wie sättigende Arithmetik und MAC-Operationen, können in Standard-C nicht abgebildet werden. Und programmieren in Assembler ist aber sehr aufwändig.
Die Compiler-Intrinsics stellen eine einfache Möglichkeit dar, um einen Teil der DSP-Spezialitäten aus einem C-Programm nutzen zu können. Wenn Assembler-Code oder Compiler-Intrinsics verwendet werden, ist der Code mit anderen DSPs nicht mehr kompatibel.

In der Tabelle 7 sind die meist gebrauchten Compiler-Intrinsics für den DSP TMS320C55x aufgeführt. Bemerkung: Der Q15-Datentyp „int“ ist kompatibel zu „DATA“ und „long“ zu „LDATA“.

Compiler-Intrinsic	**Beschreibung**
int _sadd(int src1, int src2); long _lsadd(long src1, long src2);	gesättigte Q15-Summe der Operanden dito für Q31
int _ssub(int src1, int src2); long _lssub(long src1, long src2);	gesättigte Differenz src1 - src2 dito für Q31
int _smpy(int src1, int src2); long _lsmpy(int src1, int src2);	gesättigtes Q15-fractional-Produkt dito für Q31
long _smac(long src1, int src2, int src3);	gesättigte Q31-Summe von src1 + Produkt src2*src3
long _smas(long src1, int src2, int src3);	gesättigte Q31-Differenz von src1 - Produkt src2*src3
int _sshl(int src1, int src2); long _lsshl(long src1, int src2);	gesättigtes Q15-Resultat von (src1«src2) dito für Q31
int _shrs(int src1, int src2); long _lshrs(long src1, int src2);	gesättigtes Q15-Resultat von (src1»src2) dito für Q31

Tabelle 7: Oft verwendete TMS320C55x-Compiler-Intrinsics

Eine vollständige Liste für diesen DSP findet sich in [TMS320C55x, Kapitel 6.5.4].

1.4 Skalierung von Signalen und Filterkoeffizienten

Die Eingangs-Signale vom AD-Wandler müssen so skaliert werden, dass sie in den Datentyp passen, der für die Speicherung und Rechnung verwendet wird.

Die Filter-Koeffizienten müssen so skaliert werden, dass die (Zwischen-) Resultate ein möglichst gutes SNR aufweisen ohne überzulaufen. Ev. müssen Eingangssignale und Zwischenresultate skaliert werden, bevor der nächste Verarbeitungs-Schritt erfolgt. Sporadisches Überlaufen kann akzeptiert werden, wenn mit sättigender Arithmetik gearbeitet wird.

Die Ausgangs-Signale zum DA-Wandler müssen so skaliert werden, dass sie den DA-Wandler möglichst gut aussteuern. Überlaufen muss unbedingt verhindert werden, da durch das Modulo-Verhalten der Ausgang von einem „Anschlag“ zum anderen springen würde.

Die genaue Vorgehensweise beim Skalieren hängt von der Filterart und der verwendeten Struktur ab. In den nächsten Abschnitten wird es exemplarisch aufgezeigt für FIR-Filter in Direktform und für IIR-Filter mit Biquad-Kaskaden in Direktform II.

1.4.1 Filter-Normen l_2, l_∞ und l_1

Für die weiteren Betrachtungen wird, wo nichts anderes angegeben wurde, das fractional-Zahlenformat mit $-1 \leq zahl < +1$ angenommen.

Das Ausgangssignal eines Digitalfilters ist die Faltung des Eingangssignals mit der Impulsantwort.

$$y[n] = h[n] * x[n] = \sum_{k=0}^{N} h[k] \cdot x[n-k] \tag{2}$$

Da das Eingangssignal $|x[n]| < 1$ ist, ist $|y[n]| \leq$ als obige Summe.

$$l_1 = \sum_{k=0}^{N} |h[k]| \tag{3}$$

Wenn die Eingangsfolge x[n] durch die l_1-Norm des Filters dividiert wird, ist ein Zahlenüberlauf beim Ausgangssignal y[n] für jede beliebige Eingangsfolge ausgeschlossen.

Falls $l_1 < 1$ ist, wird durch die Skalierung des Eingangssignals mit $s = 1/l_1$ das SNR verschlechtert. Da die l_1-Norm den worst-case-Fall abdeckt, der praktisch kaum vorkommt, wurden zwei weitere Normen definiert.

$$l\infty = max_{(f)} \left| H(e^{j \cdot 2\pi \cdot f \cdot Ts}) \right| \tag{4}$$

Wenn die Eingangsfolge x[n] durch die $l\infty$-Norm (= Tschebyscheff-Norm) des Filters dividiert wird, ist ein Zahlenüberlauf beim Ausgangssignal y[n] für jedes stationäre Sinussignal ausgeschlossen.

$$l_2 = \sqrt{\sum_{k=0}^{N} h^2[k]} \tag{5}$$

Wenn die Eingangsfolge x[n] durch die l_2-Norm des Filters dividiert wird, erreicht man das beste SNR.

$$l_2 \leq l\infty \leq l_1 \tag{6}$$

Die Wahl der Filternorm, resp. des Skalierungsfaktors für die Eingangsfolge ist immer ein Kompromiss. Kleinere Filternorm verbessert das SNR und erhöht die Wahrscheinlichkeit eines Überlaufs.

Die l_2- und die $l\infty$-Filternormen können mit dem Matlab-Befehl „filternorm“ berechnet werden. Die l_1-Norm lässt sich im Matlab so berechnen: sum(abs(h))

2 Implementation von FIR-Filtern

wird exemplarisch aufgezeigt für FIR-Filter in der Direktform.

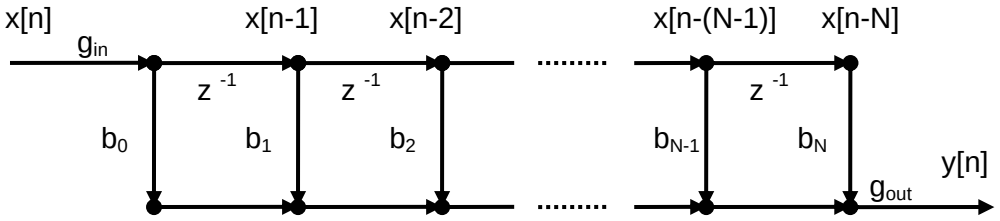


Abbildung 1: FIR-Filter in Direktform

Auf die Bestimmung der b-Koeffizienten der Übertragungs-Funktion H(z) wird hier nicht eingegangen. Es wird angenommen dass die Filterkoeffizienten bereits bestimmt wurden. Die Ausgangsfolge berechnet sich mit Hilfe der folgenden Differenzengleichung:

$$y[n] = b[n] * x[n] = \sum_{k=0}^{N} b[k] \cdot x[n-k] \tag{7}$$

2.1 Skalierung des Eingangs- und des Ausgangssignals

Zuerst die gewünschte Filternorm gemäss Abschnitt 1.4 berechnet. Falls die Norm > 1 ist, muss das Eingangssignal mit $g_{in} = 1/Norm$ skaliert werden, damit das Ausgangssignal nicht zu stark überläuft/sättigt.

Falls AD- resp. DA-Wandler einen anderen Zahlenbereich aufweisen als bei der Rechnung verwendet wird, kann das mit g_{in} resp. g_{out} angepasst werden.

2.2 Q15-Skalierung der Filterkoeffizienten

Der Q-15-Zahlenbereich ist $-1 \leq zahl < 1$. Alle Filterkoeffizienten müssen diese Bedingung erfüllen. Anderenfalls müssen die Filter-Koeffizienten skaliert werden mit $s = 1/max(|b_k|)$. Diese Skalierung kann rückgängig gemacht werden, indem der Ausgang mit $g_{out} = s$ multipliziert wird.

Oft wird s auf die nächste 2er-Potenz gerundet, da die Rückskalierung so durch eine einfache Schiebeoperation statt einer Multiplikation mit g_{out} erledigt werden kann.

2.3 Überlauf von Zwischenresultaten vermeiden

Einhalten der Filternorm sagt nur etwas aus über das fertig berechnete Ausgangssignal. Es verhindert nicht, dass Zwischenresultate überlaufen können. Z.B. können in der Gleichung 7 die ersten Produkte eine positive Zahl ergeben und die letzten eine negative. Das Resultat kann im zulässigen Wertebereich liegen, auch wenn Zwischensummen ausserhalb liegen.

Es gibt drei Massnahmen, die einen Überlauf bei den Zwischensummen verhindern:

- Anwenden der l_1-Norm verhindert prinzipiell einen Überlauf, verschlechtert aber das SNR

Listing 1: FIR-Code-Ausschnitt für den TMS320C55x

```
DATA x, y;                                  // x[n] und y[n], DATA = Q15
LDATA temp = 0;                             // Zwischenresultat, LDATA = Q31
DATA b[...];                                // b-Koeffizienten, DATA = Q15
DATA buffer[...];                           // verzögerte x-Werte
...                                         // for-Schleife Differenzen-Gleichung
temp = _smac(temp,b[k],buffer[n-k]);        // temp = temp+b[k]*x[n-k], sättigend
...                                         // Ende for-Schleife Differenzen-Gl.
y = temp >> 16;                             // Q31 => Q15
```

- Aufsummieren in den DSP-internen Registern mit Guard-Bits. Üblich sind 8 Guard-Bits, was einen Überlauf bis zu 256 Koeffizienten theoretisch und praktisch auch für Filter höherer Ordnung ausschliesst. Dazu muss man den Algorithmus in Assembler programmieren.
- Zwischenresultate speichern im Q8.23-Format. Die 8 Vorkommabits wirken wie die Guard-Bits und die (gegenüber Q15) zusätzlichen 8 Nachkommabits verhindern spürbare Quantisierungsfehler.

Listing 2: FIR-Code-Ausschnitt für den TMS320C55x mit Überlauf-Schutz

```
DATA x, y;                                  // x[n] und y[n], DATA = Q15
LDATA zwischen = 0, temp = 0;               // Zwischenresultate, LDATA = Q31
DATA b[...];                                // b-Koeffizienten, DATA = Q15
DATA buffer[...];                           // verzögerte x-Werte
...                                         // for-Schleife Differenzen-Gleichung
temp = _lsmpy(b[k], buffer[n-k];            // temp = b[k] * x[n-k], sättigend
temp = temp >> 8;                           // Q31 => Q8.23
zwischen = _lsadd(zwischen, temp);          // zwischen = zwischen+temp, sättigend
...                                         // Ende for-Schleife Differenzen-Gl.
zwischen = _lshrs(zwischen, 8);             // Q8.23 => Q31, sättigend
y = zwischen >> 16;                         // Q31 => Q15
```

Das ist weniger effizient als Assembler-Programmierung, aber viel effizienter als Rechnen mit „float“ auf einem Festkomma-DSP.

Bei der Implementation des (langen) Verzögerungsbuffers in einem FIR-Filter muss unbedingt ein Ringbuffer verwendet werden. Statt so viele Variabeln zu kopieren, wie das Filter lang ist, muss bloss ein Pointer inkrementiert und eine Modulo-Operation druchgeführt werden.

3 Implementation von IIR-Filtern

wird exemplarisch aufgezeigt für IIR-Filter mit Biquad-Kaskaden in Direktform II.

Auf die Bestimmung der b- und a-Koeffizienten der Übertragungs-Funktion H(z) wird hier nicht eingegangen. Es wird angenommen dass die Filterkoeffizienten bereits bestimmt wurden.

Die Ausgangsfolge berechnet sich mit Hilfe der folgenden Differenzengleichung:

$$y[n] = \sum_{k=0}^{N} b[k] \cdot x[n-k] - \sum_{k=1}^{M} a[k] \cdot y[n-k] \tag{8}$$

Die Skalierung der Signale und Filterkoeffizienten beeinflussen sich gegenseitig, weil vergangene Ausgangssignale in die Berechnung einfliessen (IIR-Filter sind rekursiv).
Quantisierungsfehler wirken sich weniger stark aus, wenn IIR-Filter als Biquad-Kaskade (second order sections) implementiert werden, weil sich ein Quantisierungsfehler so nur auf jeweils ein Polpaar auswirken kann.
Die Biquads können in Direktstruktur I, in Direktstruktur II oder in transponierter Direktstruktur II realisiert werden. Im Folgenden wird die Direktstruktur II verwendet, da sie durch Matlab-Tools am Besten unterstützt wird.
Aus den b- und a-Koeffizienten der Übertragungs-Funktion H(z) können mit dem Matlab-Befehl „tf2sos“ die Filterkoeffizienten der Biquads berechnet werden.

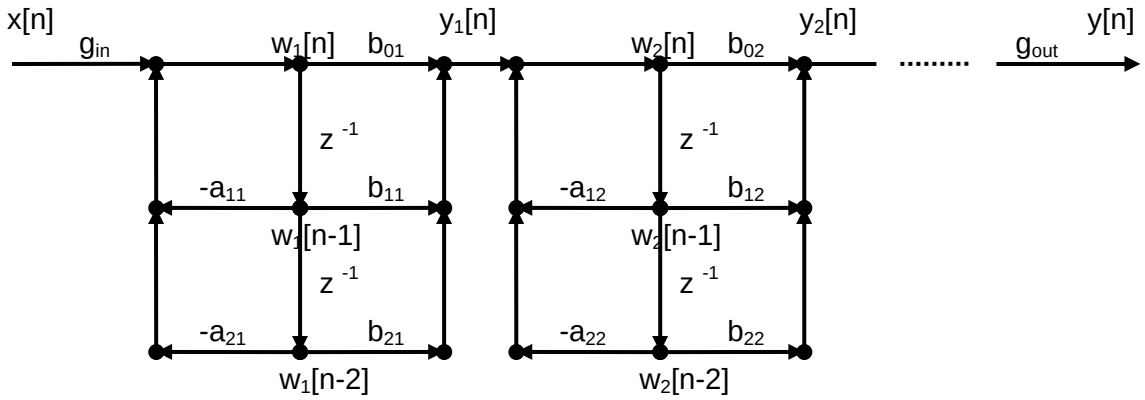


Abbildung 2: IIR-Filter mit Biquad-Kaskaden in Direktform II

Die Berechnung eines einzelnen Biquads erfolgt mit diesen Formeln:

$$\begin{aligned} w[n] = &\quad x[n] - a_1 \cdot w[n-1] - a_2 \cdot w[n-2] \\ y[n] = &\quad b_0 \cdot w[n] + b_1 \cdot w[n-1] + b_2 \cdot w[n-2] \end{aligned} \tag{9}$$

3.1 Skalierung eines einzelnen Biquads

Sowohl bei der Berechnung von w[n] als auch von y[n] kann ein Überlauf auftreten.
Da a_0 immer 1 ist, können a_1 und a_2 nicht skaliert werden, ohne die Lage der Pole zu verändern. Ein Überlauf bei der Berechnung von w[n] kann deshalb nur vermieden werden, wenn das Eingangssignal richtig skaliert wird.
Diese Einschränkung gibt es bei den b-Koeffizienten nicht. Sie können so skaliert werden, dass der Ausgang y[n] und das Zwischenresultat w[n] des nächsten Biquads nicht überlaufen.

Aus dem Nenner einer Biquad-Übertragungsfunktion können die Pole berechnet werden. Für einen stabilen Biquad mit reellen Koeffizienten müssen die konjugiert komplexen Pole im z-Einheitskreis liegen und es gilt:

$$\begin{aligned} 1 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2} &= (1 - p \cdot z^{-1}) \cdot (1 - p^* \cdot z^{-1}) \quad p = Pol \\ a_0 &= 1 \\ a_1 &= -(p + p^*) = -2 \cdot \Re(p) \qquad |a_1| < 1 + a_2 < 2 \\ a_2 &= p \cdot p^* = |p|^2 \qquad |a_2| < 1 \end{aligned} \tag{10}$$

Das heisst, das a_1 für übliche Filter nicht im Q15-Format gespeichert werden kann!

Lösung: Alle Koeffizienten werden halbiert und im Q15-Format abgespeichert und der Algorithmus entsprechend modifiziert.
Da die b-Filterkoeffizienten vor dem Abspeichern halbiert werden, muss $|b_k| < 2$ erfüllt sein.
Das Eingangssignal x[n] wird mit 1/4 skaliert, um einen Überlauf bei den Zwischenresultaten und bei $w[n]$ zu vermeiden. Beim Ausgangssignal $y[n]$ wird die Skalierung rückgängig gemacht.
Wenn man für das Zwischenresultats das Format Q31 verwendet, gehen keine Bits verloren und das SNR wird nicht verschlechtert.
Die Gleichung 9 wird modifiziert zu:

$$\begin{aligned} w[n] = &\quad 2 \cdot (x[n]/4 - a_1/2 \cdot w[n-1] - a_2/2 \cdot w[n-2]) \\ y[n] = &\quad 4 \cdot (b_0/2 \cdot w[n] + b_1/2 \cdot w[n-1] + b_2/2 \cdot w[n-2]) \end{aligned} \tag{11}$$

Das Vorgehen beim Halbieren aller Filterkoeffizienten kann aus einem anderen Blickwinkel auch so beschrieben werden (das Resultat ist genau das selbe):

- alle Koeffizienten als Q1.14 abspeichern (entspricht den Halbieren)
- Eingangssignal $x[n]$ durch 4 dividieren `temp = (long)x[n]<<14` (Q0.15 zu Q2.29 wandeln)
- Zwischenresultat *temp* für die Berechnung von $w[n]$ in Q2.29 = Resultat der Multiplikation von zwei Q1.14-Zahlen
- Verzögerungskette $w[n]$ mit 2 multiplizieren `w[n] = temp>>15` (Q2.29 zu Q1.14 wandeln)
- Zwischenresultat *temp* für die Berechnung von $y[n]$ in Q2.29 = Resultat der Multiplikation von zwei Q1.14-Zahlen
- Ausgangssignal $y[n]$ mit 4 multiplizieren `y[n] = temp>>14` (Q2.29 zu Q0.15 wandeln)

3.2 Skalierung einer Kaskade von Biquads: schrittweises Vorgehen

Für jeden Biquad muss die Skalierung einzel durchgerechnet werden, damit nirgends ein untolerierbarer Überlauf entsteht.
Für Biquad-Kaskaden in Direktform II kann man dazu den Matlab-Befehl „tf2sos“ verwenden (aber nicht für andere Realisierungsstrukturen).

Das schrittweise Vorgehen wird Anhand des Matlab-Listings erläutert.
Es folgen das Listing, die Ausgabe des Skript und anschliessend die Erklärungen.

Listing 3: Biquad-Kaskaden in Direktform II skalieren: iirsos.m

```
1 % Biquad-Kaskade entwerfen
2 % und Filterkoeffizienten für Festkomma-DSP berechnen
3 % (c) Hanspeter Hochreutener, ZHW, 22.4.2007
4 %
```

```
5  % Abgespeichert werden:
6  %
7  % Die Filter-Koeffizienten der Biquads
8  % a11, a21, b01, b11, b21, a12, a22, b02, b12, b22, ...
9  % im Q1.14-Format in dieser Reihenfolge
10 % in der Datei coef.h
11 %
12 % Die Skalierung des Eingangs- und des Ausgangssignals
13 % gin, gout
14 % als Anzahl Bit-Shifts (positiv = links, negativ = rechts schieben)
15 % in der Datei gain.h
16
17 clc; clear; close all;
18 format compact; format short;
19
20 Nbiq = 4;                          % Anzahl Biquad-Kaskaden
21 [b,a] = ellip(2*Nbiq,1,40,0.2)     % Filter-Koeffizienten berechnen
22 figure(1); freqz(b,a);             % Frequenzgang anzeigen zur Kontrolle
23 title('Frequenzgang mit genauen Filter-Koeffizienten');
24
25 % Auswählen der Filternorm: Tschebyscheff oder L2
26 % und umwandeln in ins Biquad-Kaskaden (second order sections)
27 %[sos,g] = tf2sos(b,a,'up','inf')    % Tschebyscheff: Minimiert Overflow-Wahrs.
28 [sos,g] = tf2sos(b,a,'down','two')  % L2: Minimiert Signal-to-Noise-Ratio SNR
29
30 % Skalierung des Eingangssignals vor der Kaskade
31 gin = 0;                           % Initialisieren: Gain am Eingang = 2^0 = 1
32 while (g < 1)                      % Eingangssignal skalieren
33     g = g*2;                       % mit einer 2er-Potenz
34     gin = gin-1;                   % Eingangssignal um Faktor 2 abschwächen
35 end
36
37 % Skalierung der b-Filter-koeffizienten der Biquads
38 for i = 1:Nbiq                     % Alle Biquads skalieren
39     sos(i,1:3) = sos(i,1:3)*g;     % b-Koeffizienten des Biquads anpassen
40     g = max(abs(sos(i,1:3)));      % grössten b-Koeffizienten bestimmen
41     if (g > 1.9)                   % falls b-Koeff. ausserhalb Q1.14-Format
42         g = g/1.9;                 % diese Verstärkung g kompensiert
43         sos(i,1:3) = sos(i,1:3)/g; % in diesem Biquad (b-Koeff. anpassen) und
44                                    % im folg. Biquad (nächster for-Durchgang)
45     else                           % sonst
46         g = 1;                     % keine Anpassung der Verstärkung nötig
47     end
48 end
49
50 % Skalierung des Ausgangssignals nach der Kaskade
51 gout = 0;                          % Initialisieren: Gain am Ausgang = 2^0 = 1
52 while (g > 1)                      % Ausgangssignal skalieren
53     g = g/2;                       % mit einer 2er-Potenz
54     gout = gout+1;                 % Ausgangssignal um Faktor 2 verstärken
55 end
56
57 sos(Nbiq,1:3) = sos(Nbiq,1:3)*g;   % b-Koeff. des letzten Biquads anpassen
58
59 gin, gout, sos                     % Zum Vergleich ausgeben
60
```

```
61 sos_dsp = round(sos*2^14)              % Koeffizienten umrechnen ins Q1.14-Format
62
63 [b_dsp,a_dsp] = sos2tf(sos_dsp/2^14, 2^(gin+gout))  % Zurückrechnen auf die ÜTF
64 figure(2); freqz(b_dsp,a_dsp);          % Frequenzgang anzeigen zur Kontrolle
65 title('Frequenzgang mit skalierten Filter-Koeffizienten');
66
67 dlmwrite('gain.h', [gin, gout]);        % Skalierung Ein- und Ausgang speichern
68 % sos_dsp = [sos_dsp(:,5:6), sos_dsp(:,1:3)]; % Umsortieren der Koeffizienten
69 % dlmwrite('coef.h', sos_dsp);          % Koeffizienten in Header-Datei speichern
70 % dlmwrite: Zeilenumbrüche statt , am Zeilenende => geht nicht für C-Import
71 fid = fopen('coef.h', 'w');             % Also von "Hand" ausgeben }-(
72 for i = 1:Nbiq-1
73     fprintf(fid, '%i, %i, %i, %i, %i, ', ...
74         sos_dsp(i,5), sos_dsp(i,6), sos_dsp(i,1), sos_dsp(i,2), sos_dsp(i,3));
75 end;
76 i = Nbiq;                               % Am Schluss Newline statt Komma
77 fprintf(fid, '%i, %i, %i, %i, %i \n', ...
78     sos_dsp(i,5), sos_dsp(i,6), sos_dsp(i,1), sos_dsp(i,2), sos_dsp(i,3));
79 fclose(fid);
```

Nach dem Listing mit der Ausgabe des Skripts folgt dessen Beschreibung.

Listing 4: Ausgabe des Matlab-Skripts 3

```
1  % Resultate des Skripts iirsos.m
2
3  % Für folgende Befehlszeilen/Parameter:
4
5  Nbiq = 4;                            % Anzahl Biquad-Kaskaden
6  [b,a] = ellip(2*Nbiq,1,40,0.2)       % Filter-Koeffizienten berechnen
7  [sos,g] = tf2sos(b,a,'down','two') % L2: Minimiert Signal-to-Noise-Ratio SNR
8
9  % erscheint diese Ausgabe:
10
11 b =
12     0.0154  -0.0733   0.1811  -0.2917   0.3397  -0.2917   0.1811  -0.0733   0.0154
13 a =
14     1.0000  -6.4154  18.9395  -33.3652  38.2328  -29.1270  14.3989  -4.2251   0.5646
15 sos =
16     1.1828   -1.8934    1.1828    1.0000   -1.6129    0.9937
17     1.3677   -2.1528    1.3677    1.0000   -1.6091    0.9656
18     0.5825   -0.8319    0.5825    1.0000   -1.6027    0.8695
19     0.2476   -0.0372    0.2476    1.0000   -1.5907    0.6767
20 g =
21     0.0661
22 gin =
23     -4
24 gout =
25      0
26 sos =
27     1.1869   -1.9000    1.1869    1.0000   -1.6129    0.9937
28     1.2071   -1.9000    1.2071    1.0000   -1.6091    0.9656
29     0.6955   -0.9932    0.6955    1.0000   -1.6027    0.8695
30     0.2476   -0.0372    0.2476    1.0000   -1.5907    0.6767
31 sos_dsp =
32        19447       -31130        19447        16384       -26427        16280
33        19778       -31130        19778        16384       -26363        15821
```

```
34          11394        -16272         11394         16384        -26258         14246
35           4057          -609          4057         16384        -26063         11087
36  b_dsp =
37      0.0154   -0.0733    0.1811   -0.2917    0.3397   -0.2917    0.1811   -0.0733    0.0154
38  a_dsp =
39      1.0000   -6.4155   18.9397  -33.3655   38.2330  -29.1270   14.3988   -4.2250    0.5646
```

Das schrittweise Vorgehen wird Anhand des Matlab-Listings 3 und der Ausgabe für ein konkretes Beispiel erläutert.

1. Für die gewünschte Übertragungsfunktion H(z) werden die b- und a-Filterkoeffizienten für die Differenzengleichung 8 bestimmt.

   Im Beispiel wurde in der Zeile 20 des Matlab-Skripts festgelegt, dass 4 Biquads eingesetzt werden und in Zeile 21 die Koeffizienten für ein elliptisches Tiefpass-Filter mit 1dB Rippel im Durchlass- und 40dB Dämpfung im Sperrbereich mit einer Grenzfrequenz von 0.2*fs/2 berechnet.
   Die Filterkoeffizienten sind in der Ausgabe in den Zeilen 11-14 aufgelistet.

2. Nun muss man sich für eine Filternorm entscheiden.
   Und aus den b- und a-Koeffizienten der Übertragungsfunktion werden die Koeffizienten sos (b01 b11 b21, 1, a11, a21; b02 b12 b22, 1, a12, a22; ...) der Biquads berechnet.
   Mit dem Faktor g muss das Eingangssignal multipliziert werden, um einen Overflow im ersten Biquad zu verhindern.

   Im Skript kann man wählen zwischen Zeile 27 für die Tschebyscheff-Norm l_∞ oder Zeile 28 für die l_2-Norm. Gewählt wurde die l_2-Norm, welche das SNR optimiert.
   Die berechneten sos und g finden sich in der Ausgabe in den Zeilen 15-21.

3. g ist der Faktor, mit dem das Eingangssignal skaliert werden muss, um einen Überlauf im ersten Biquad zu vermeiden.
   Für die Implementation ist es günstiger die Skalierung mit der nächstgelegenen 2er-Potenz zu machen, da statt der Multiplikation nur eine Schiebeoperation durchgeführt werden muss. Die veränderte Skalierung muss bei den b-Koeffizienten des ersten Biquads berücksichtigt werden.

   Im Skript wird der Faktor für die Eingangsskalierung in den Zeilen 31-35 bestimmt.
   Aus den Zeilen 20-23 der Ausgabe sieht man, dass statt $0.0661\ 2^{-4} = 0.0625$ verwendet wird.
   Die abgeänderte Skalierung wird in der Zeile 39 des Skripts im ersten Biquad einberechnet.

4. Der Betrag der a-Koeffizienten ist für einen stabilen Biquad immer < 2.
   Das trifft für die b-Koeffizienten nicht unbedingt zu. Ist in einem Biquad ein b-Koeffizient zu gross, werden die b-Koeffizienten entsprechend skaliert und die b-Koeffizienten des nachfolgenden Biquads entsprechend angepasst.

   In der Ausgabe auf Zeile 17 sieht man, dass der Koeffizient b11 -2.1528 zu gross ist.
   Das Skalieren der b-Koeffizienten für alle Biquads geschieht in den Zeilen 38-48. In den Zeilen 41-42 wird der Betrag der Koeffizienten auf 1.9 begrenzt; damit entstehen beim Runden keine Probleme.
   Die Zeilen 26-30 der Ausgabe zeigen, dass nun alle Koeffizienten im erlaubten Zahlenbereich liegen.

5. Das verbleibende g ist der Faktor mit dem das Ausgangssignal skaliert werden muss.
   Wie bei der Eingangsskalierung wird wieder die nächstgelegene 2er-Potenz gesucht. Die veränderte Skalierung muss bei den b-Koeffizienten des letzten Biquads berücksichtigt werden.

Im Skript wird der Faktor für die Ausgangsskalierung in den Zeilen 51-57 bestimmt.
Die Zeilen 22-30 der Ausgabe enthalten nun die skalierten Koeffizienten.

6. Für die Realisierung auf einen Festkomma-DSP müssen die Koeffizienten ins Q1.14-Format umgerechnet werden (Q1.14 entspricht Q0.15 mit halbierten Koeffizienten).

   In der Skriptzeile 61 werden die Koeffizienten mit 2^{14} multipliziert und gerundet.
   Ausgabe der quantisierten Koeffizienten in der Zeile 31-35.

7. Man muss unbedingt kontrollieren, ob die quantisierten Koeffizienten den „richtigen“ Frequenzgang ergeben.

   Im Skript werden die Frequenzgänge in den Zeilen 22-23 und 63-65 gezeichnet, was eine schnelle visuelle Kontrolle erlaubt.
   In der Ausgabe können die Zeilen 36-39 mit 11-14 verglichen werden.

8. Zum Schluss werden die Koeffizienten für den DSP umsortiert abgespeichert. Sie lassen sich so in den C-Code des DSPs einbinden.

   Siehe Skriptzeilen 67-79.

3.3 Beispiel: IIR-Implementation mit Compiler-Intrinsics auf dem TMS320C55x

Zuerst eine Implementaion in C mit float-Variabeln.

Listing 5: Ausschnitt: DSP-C-Code mit float-Variabeln

```
/* Implementation der IIR-Biquad-Kaskade in Direktform II*/
/* mit float-Arithmetik als Referenz-Implementaion */
/* float muss auf einem Fixed-Point-DSP SW-mässig emuliert wird */
/* und ist deshalb extrem ineffizient*/
DSK5510_LED_on(1);                  // Start Zeitmessung Floating-Point
q15tofl(&sample_in_r, &ftemp, 1);         //  Q15 => float
/* C-Code für die Vergleichs-Implementation mit float in C */
ftemp = ftemp * fgain[0];                   //  Eingang skalieren mit gin
for (biq = 0; biq < Nbiq; biq++) {          //  Alle Biquads abarbeiten
    biq3 = biq*3;    biq5 = biq*5;          //  Für Array-Indexierung
    /* Werte um einen Takt verzögern; wäre besser mit Ringbuffer */
    fw[biq3+2] = fw[biq3+1];                //  w(n-2) = w(n-1)
    fw[biq3+1] = fw[biq3];                  //  w(n-1) = w(n)
    /* w[n] = x[n] -a1*w[n-1] -a2*w[n-2] */
    // temp = temp - a1*w[n-1]
    ftemp = ftemp - fcoef[biq5] * fw[biq3+1];
    // w[0] = temp - a2*w[n-2]
    fw[biq3] = ftemp - fcoef[biq5+1] * fw[biq3+2];
    /* y[n] = 4*(b0*w[n] +b1*w[n-1] + b2*w[n-1]) */
    // temp = b0*w[n]
    ftemp = fcoef[biq5+2] * fw[biq3];
    // temp = temp + b1*w[n-1]
    ftemp = ftemp + fcoef[biq5+3] * fw[biq3+1];
    // temp = temp + b2*w[n-2]
    ftemp = ftemp + fcoef[biq5+4] * fw[biq3+2];
    /* temp = Ausgang dieses Biquads = Eingang des nächsten Biquads */
}
ftemp = ftemp * fgain[1];                   //  Ausgang skalieren mit gout
fltoq15(&ftemp, &sample_out_r, 1);          //  float => Q15
DSK5510_LED_off(1);                 // Stop Zeitmessung Floating-Point
```

Die float-Implementation ist mit 13 Zeilen Code sehr übersichtlich und kompakt. Aber auf einem Fixed-Point-DSP muss die float-Arithmetik software-mässig emuliert werden und das Filter ist etwa 100 mal langsamer als handoptimierter Assembler-Code! C-Code ist also nur geeignet als erster Approach oder wenn man Rechenpower zum Versauen hat.

Es folgt der Ausschnitt aus dem DSP-Code implementiert mit Compiler-Intrinsics. Das vollständige C-File befindet sich im Anhang.

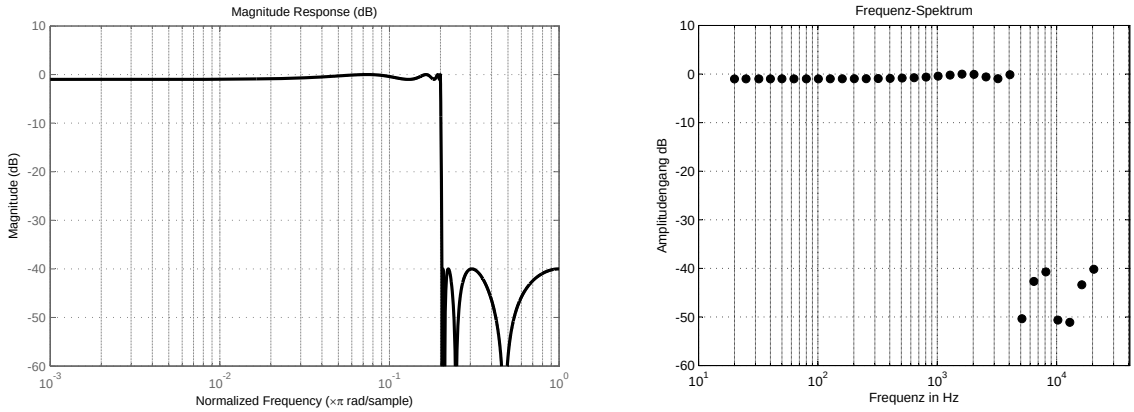
Listing 6: Ausschnitt: DSP-C-Code für das IIR-Biquad-Filter in Direktform II

```
/* Implementation der IIR-Biquad-Kaskade in Direktform II*/
/* mit sättigender Arithmetik und Compiler-Intrinsics */
DSK5510_LED_on(0);                  // Start Zeitmessung Fixed-Point
/* Formale Beschreibung                          Q-Format-Sichtweise */
// Eingang mit "gin" skalieren                   x = Q0.15
// temp = 2^gin * x[n]/4                         skalieren + Q0.15 zu Q2.29
temp = (long)sample_in_l<<(gain[0]+14); //  Q2.29
for (biq = 0; biq < Nbiq; biq++) {      //  Alle Biquads abarbeiten
    biq3 = biq*3;    biq5 = biq*5;      //  Für Array-Indexierung
    /* Werte um einen Takt verzögern; wäre besser mit Ringbuffer */
    w[biq3+2] = w[biq3+1];              //  w(n-2) = w(n-1)
    w[biq3+1] = w[biq3];                //  w(n-1) = w(n)
    /* w[n] = 2*(x[n]/4 -a1*w[n-1] -a2*w[n-2]) */
    // temp = temp - a1*w[n-1]          Q2.29 = Q2.29 - Q1.14*Q1.14
    temp = _smas(temp, coef[biq5], w[biq3+1]);
    // temp = temp - a2*w[n-2]          Q2.29 = Q2.29 - Q1.14*Q1.14
    temp = _smas(temp, coef[biq5+1], w[biq3+2]);
    // temp = temp*2                    Q2.29 zu Q1.30 wandeln
    temp = _lsshl(temp,1);
    // w[0] = temp                      Q1.30 zu Q1.14 wandeln
    w[biq3] = temp >>16;
    /* y[n] = 4*(b0*w[n] +b1*w[n-1] + b2*w[n-2]) */
    // temp = b0*w[n]                   Q2.29 = Q1.14*Q1.14
    temp = _lsmpy(coef[biq5+2], w[biq3]);
    // temp = temp + b1*w[n-1]          Q2.29 = Q2.29 + Q1.14*Q1.14
    temp = _smac(temp, coef[biq5+3], w[biq3+1]);
    // temp = temp + b2*w[n-2]          Q2.29 = Q2.29 + Q1.14*Q1.14
    temp = _smac(temp, coef[biq5+4], w[biq3+2]);
    /* temp = Ausgang dieses Biquads = Eingang des nächsten Biquads */
}
// Ausgang mit "gout" skalieren
// temp = 2^gout * temp*4                       skalieren + Q2.29 zu Q0.31
temp = _lsshl(temp,gain[1]+2);
// y[n] = temp                                  Q0.31 zu Q0.15 wandeln
sample_out_l = temp >>16;
DSK5510_LED_off(0);                 // Stop Zeitmessung Fixed-Point
```

Die Implementation mit Compiler-Intrinsics hat etwa gleich viele Code-Zeilen wie die float-Implementation, läuft aber 17 mal schneller! Man muss allerdings sehr genau auf die Skalierung der Signale und Filter-Koeffizienten und die verwendeten „fractional“-Daten-Formate achten.

Der vollständige C-Code mit Compiler-Intrinsics befindet sich im Anhang.

Nach der Implementation muss man sich vergewissern, dass das DSP-Filter den Anforderungen entspricht. Das wurde mit einem Multiträger-Signal (bestehend aus 31 logarithmisch verteilten Sinussignale) verifiziert. Ein Vergleich der nachfolgenden Bilder bestätigt die Korrektheit der Implementation.



(a) Matlab-Entwurf (b) Messung am DSP-Board

Abbildung 3: Entworfener und gemessener Frequenzgang des IIR-Filters

3.3.1 Performance-Messung und mögliche Verbesserungen

Die Zahlenangaben beziehen Sich auf die im Beispiel realisierte Kaskade mit 4 Biquads und sampleweiser Berechnung.

Die Implementierung mit Compiler-Intrinsics und ohne Ringbuffer benötigt ca. 540 Zyklen ($2.7\mu s$).

Zum Vergleich:
Die optimierte DSP-Library Assembler-Funktion „iircas5.asm“ benötigt 85 Zyklen ($0.4\mu s$). Achtung: Diese „iircas“-Funktionen der DSP-Library können mit Koeffizienten $|a_1| > 1$ nicht umgehen, da alle Koeffizienten im Q15-Format erwartet werden. Es gibt eine „inoffizielle“ Funktion „iircas52.asm“ welche das Problem mit Q1.14-Koeffizienten löst.

Zum Vergleich:
Die C-Implementation mit float-Variablen benötigt ca. 8500 Zyklen ($42\mu s$), da die float-Arithmetik auf einem Festkomma-DSP SW-mässig emuliert werden muss. Mit dieser Implementation ist eine Samplingrate von 48kHz also bei Weitem nicht erreichbar.

Verbesserungsmöglichkeiten gegenüber der Variante „Compiler-Intrinsics ohne Ringbuffer“:

- Compiler-Optimierung maximieren im Menu „Project -> Build Options... -> Compiler -> Basic -> Opt.Level -> File (-o3)“ spart 1/3 ein.
  Der gleiche Source-Code läuft nun in 360 Zyklen ($1.8\mu s$).
- Ringbuffer für den Verzögerungsbuffer. (Einsparung unbekannt, da nicht getestet.)
- Ringbuffer-Länge auf die nächste Zweier-Potenz aufrunden. Die Modulo-Operation für den Pointer kann dann durch eine sehr viel effizienteres Bitweises-And ersetzt werden. Es folgt eine massive Einsparung an Rechenzeit, dafür wird etwas mehr Speicherplatz benötigt.
- Anstelle der for-Schleife den Code pro Biquad einmal kopieren und die Array-Indices vorausberechnen. Der Compiler setzt die feste Adresse in den Code ein; deshalb bringt ein Ringbuffer hier keinen Vorteil. Es werden ca. 200 Zyklen ($1.0\mu s$) benötigt.
- Daten und Filter-Koeffizienten in verschiedenen Speicherbereichen ablegen, damit der DSP diese parallel statt sequenziell einlesen kann.
- Bei symmetrischen und assymmetrischen FIR-Filtern wird jede Multiplikation zweimal ausgeführt. Wenn an Stelle der Direktform I die Linear-Phasen-Struktur verwendet wird, kann

die Rechenzeit unter Umständen halbiert werden (siehe Compiler-Intirnsic-Befehle _firs und _firsn).

- Durch Inline-Assembler in der for-Schleife und ausnutzen der zweiten MAC-Einheit des DSPs könnte die Rechenzeit (theoretisch) auf ca. 40 Zyklen ($0.2\mu s$) gesenkt werden!

A DSP-C-Code für das IIR-Biquad-Filter in Direktform II

Listing 7: DSP-C-Code für das IIR-Biquad-Filter in Direktform II

```
1  /*
2   *  Digitale Signal-Verarbeitung
3   *  Beispiel-Implementation einer IIR-Biquad-Kaskade
4   *  mit Compiler-Intrinsics und sättigender Arithmetik
5   *  Hanspeter Hochreutener, ZHW, April 2007
6   */
7
8  /*
9   *  Gleichungen für transponierte Direktstruktur 2:
10  *  w[n] = x[n] -a1*w[n-1] -a2*w[n-2];
11  *  y[n] = b0*w[n] +b1*w[n-1] + b2*w[n-1];
12  *  Da abs[a1] < 2, ist a1 oft ausserhalb des Q0.15-Zahlenbereichs.
13  *  Alle Filter-Koeffizienten müssen deshalb halbiert (= Q1.14) vorliegen:
14  *  A1 = a1/2, A2 = a2/2, B0 = b0/2, B1 = b1/2, B2 = b2/2
15  *  Das ergibt folgende Gleichungen:
16  *  w[n] = 2*(x[n] -A1*w[n-1] -A2*w[n-2]);  // Faktor 2 der Koeff. kompensieren
17  *  y[n] = B0*w[n] +B1*w[n-1] + B2*w[n-1];
18  *  Nun entsteht ein Overflow-Problem bei 2*x[n].
19  *  x[n] wird deshalb mit 1/4 skaliert und y[n] anschliessend rückskaliert.
20  *  Das ergibt für die Implementation schlussendlich diese Gleichungen:
21  *  w[n] = 2*(x[n]/4 -a1*w[n-1] -a2*w[n-2]);
22  *  y[n] = 4*(b0*w[n] +b1*w[n-1] + b2*w[n-1]);
23  */
24
25 /*
26  *  Die System-Konfiguration ist in der Datei tone.cdb gespeichert.
27  *  Zum Editieren/Ansehen Projektansicht öffen mit Menupukt View -> Project
28  *  Die Datei tone.cdb liegt im Projektordner "DSP/BIOS Config"
29  *  Aus dieser Konfig.-Datei werden beim Compilieren versch. Dateien erzeugt.
30  */
31 #include "tonecfg.h"
32
33 /*
34  *  std.h und log.h werden benötigt für Logging
35  */
36 #include "std.h"
37 #include "log.h"
38
39 /*
40  *  Libraries für die Hardware des Evaluation-Boards
41  */
42 #include "dsk5510.h"          // Library für das Evaluation-Board
43 #include "dsk5510_aic23.h"    // Library zum Ansprechen des Codecs, AD/DA-Wandler
44 #include "dsk5510_led.h"      // Library zum Ansteuern der LEDs
45 #include "dsk5510_dip.h"      // Library zum Auslesen der DIP-Schalter
46
47 /*
48  *  Libraries für Berechnungen und Signal-Verarbeitung
49  */
50 #include "math.h"             // Mathematische Funktionen
51 #include "tms320.h"           // Definit. für den Q15-Datentyp der DSP-Library
52 #include "dsplib.h"           // DSP-Library mit optimierten Filtern, etc.
53 #include "dsplib_c.h"         // DSP-Library mit Filtern für komplexe Signale
```

```
54
55
56 /*
57  *  Globale Variabeln
58  */
59
60 DSK5510_AIC23_Config config = {          // Codec Konfiguration
61     0x0017, // 0 DSK5510_AIC23_LEFTINVOL  Left line input channel volume
62     0x0017, // 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume
63     0x00d8, // 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone volume
64     0x00d8, // 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume
65     0x0011, // 4 DSK5510_AIC23_ANAPATH    Analog audio path control
66     0x0000, // 5 DSK5510_AIC23_DIGPATH    Digital audio path control
67     0x0000, // 6 DSK5510_AIC23_POWERDOWN  Power down control
68     0x0043, // 7 DSK5510_AIC23_DIGIF      Digital audio interface format
69     0x0081, // 8 DSK5510_AIC23_SAMPLERATE Sample rate control
70     0x0001  // 9 DSK5510_AIC23_DIGACT     Digital interface activation
71 };
72 DSK5510_AIC23_CodecHandle hCodec;        // Codec-Adresse
73
74 Int16 dip[4];                            // Zustand der DIP-Schalter
75 Int16 led[4] = {0, 0, 0, 0};             // Zustand der LEDs
76
77 DATA sample_in_l, sample_in_r;           // Eingelesene Samples vom AD-Wandler
78 DATA sample_out_l, sample_out_r;         // Auszugebende Samples zum DA-Wandler
79
80 int nr;                                  // Hilfsvariable, Schleifenzähler
81 DATA sample;                             // Hilfsvariable, Samples
82
83 /*
84  *  Variabeln für das IIR-Filter
85  */
86 #define Nbiq 4                           // Anzahl der Biquads in der Kaskade
87
88 // Filter-Koeffizienten aus Datei einlesen, Q1.14-Format
89 // Reihenfolge: a11, a21, b01, b11, b21, a12, a22, b02, b12, b22, ...
90 DATA coef[5*Nbiq] = {
91     #include "../matlab/coef.h"
92 };
93
94 // Skalierungsfaktoren für Eingangs- und Ausgangssignal einlesen
95 // Anzahl Bit-Shifts (positiv = links, negativ = rechts schieben)
96 DATA gain[2] = {
97     #include "../matlab/gain.h"
98 };
99
100 DATA w[3*Nbiq];              // Buffer für die Verzögerungskette w[n]
101
102 LDATA temp;      // Hilfsvar. mit doppelter Anzahl Bits für Zwischenresultate
103
104 int biq;         // Laufvariable, enthält die aktuelle Biquad-Nummer
105 int biq3, biq5; // Hilfsvariabeln, damit Index  nicht immer neu berechnet wird
106
107 /*
108  *  Hauptprogramm
109  */
```

```
110
111 void main()
112 {
113     // Lokale Variablen deklarieren und initialisiseren
114
115     DSK5510_init();                         // Board-Treiber initialisieren
116     DSK5510_LED_init();                     // LED-Treiber initialisieren
117     DSK5510_DIP_init();                     // Treiber für DIP-Schalter initialis.
118
119     hCodec = DSK5510_AIC23_openCodec(0, &config);   // Codec starten
120     while (DSK5510_AIC23_read16(hCodec, &sample));  // AD-Wandler-Buffer leeren
121
122     for (nr = 0; nr < 3*Nbiq; nr++) {
123         w[nr] = 0;                          // Verzögerungskette löschen
124     }
125
126     /*
127      *  Hauptschleife, wird mit der Samplingfrequenz abgearbeitet
128      *  Es wird je ein Sample vom linken und rechten AD-Wandler-Kanal gelesen,
129      *  verarbeitet,
130      *  und zum linken und rechten DA-Wandler-Kanal ausgegeben
131      */
132     while (1) {                             // Endlos-Schleife
133
134         // Daten vom AD-Wandler (Codec) einlesen
135         while (!DSK5510_AIC23_read16(hCodec, &sample_in_l)); // linker Kanal
136         while (!DSK5510_AIC23_read16(hCodec, &sample_in_r)); // rechter Kanal
137
138         // Beispiel für Logging
139         LOG_printf(&trace, "links: %d  rechts: %d", sample_in_l, sample_in_r);
140
141
142         /* Implementation der IIR-Biquad-Kaskade in Direktform II*/
143         /* mit sättigender Arithmetik und Compiler-Intrinsics */
144         DSK5510_LED_on(0);                  // Start Zeitmessung Fixed-Point
145         /* Formale Beschreibung                    Q-Format-Sichtweise */
146         // Eingang mit "gin" skalieren              x = Q0.15
147         // temp = 2^gin * x[n]/4                    skalieren + Q0.15 zu Q2.29
148         temp = (long)sample_in_l<<(gain[0]+14); //  Q2.29
149         for (biq = 0; biq < Nbiq; biq++) {      //  Alle Biquads abarbeiten
150             biq3 = biq*3;    biq5 = biq*5;      //  Für Array-Indexierung
151             /* Werte um einen Takt verzögern; wäre besser mit Ringbuffer */
152             w[biq3+2] = w[biq3+1];              //  w(n-2) = w(n-1)
153             w[biq3+1] = w[biq3];                //  w(n-1) = w(n)
154             /* w[n] = 2*(x[n]/4 -a1*w[n-1] -a2*w[n-2]) */
155             // temp = temp - a1*w[n-1]          Q2.29 = Q2.29 - Q1.14*Q1.14
156             temp = _smas(temp, coef[biq5], w[biq3+1]);
157             // temp = temp - a2*w[n-2]          Q2.29 = Q2.29 - Q1.14*Q1.14
158             temp = _smas(temp, coef[biq5+1], w[biq3+2]);
159             // temp = temp*2                    Q2.29 zu Q1.30 wandeln
160             temp = _lsshl(temp,1);
161             // w[0] = temp                      Q1.30 zu Q1.14 wandeln
162             w[biq3] = temp >>16;
163             /* y[n] = 4*(b0*w[n] +b1*w[n-1] + b2*w[n-2]) */
164             // temp = b0*w[n]                   Q2.29 = Q1.14*Q1.14
165             temp = _lsmpy(coef[biq5+2], w[biq3]);
```

```
166             // temp = temp + b1*w[n-1]                  Q2.29 = Q2.29 + Q1.14*Q1.14
167             temp = _smac(temp, coef[biq5+3], w[biq3+1]);
168             // temp = temp + b2*w[n-2]                  Q2.29 = Q2.29 + Q1.14*Q1.14
169             temp = _smac(temp, coef[biq5+4], w[biq3+2]);
170             /* temp = Ausgang dieses Biquads = Eingang des nächsten Biquads */
171         }
172         // Ausgang mit "gout" skalieren
173         // temp = 2^gout * temp*4                        skalieren + Q2.29 zu Q0.31
174         temp = _lsshl(temp,gain[1]+2);
175         // y[n] = temp                                   Q0.31 zu Q0.15 wandeln
176         sample_out_l = temp >>16;
177         DSK5510_LED_off(0);                 // Stop Zeitmessung Fixed-Point
178
179
180         sample_out_r = sample_out_l;
181
182         if (!DSK5510_DIP_get(0)) {          // Filter überbrücken mit Schalter 0
183             sample_out_l = sample_in_l;
184             sample_out_r = sample_in_r;
185         }
186
187         // Daten zum Da-Wandler (Codec) ausgeben
188         while (!DSK5510_AIC23_write16(hCodec, sample_out_l)); // linker Kanal
189         while (!DSK5510_AIC23_write16(hCodec, sample_out_r)); // rechter Kanal
190
191     }                                       // Endlos-Schleife: Ende
192
193     /* Close the codec */
194     DSK5510_AIC23_closeCodec(hCodec);       // Codec schliessen
195                                             // wird nie erreicht, wegen while(1)
196 }
197
198
199 /*
200  *  Funktionen
201  */
```

B Verzeichnisse

Tabellenverzeichnis

subsection

Abbildungsverzeichnis

Listings