Beuth Hochschule für Technik Berlin
Fachbereich VII
Labor für Digitaltechnik und Digitale Signalverarbeitung
Prof. Dr.-Ing. Marcus Purat

Vorbereitende Aufgaben zum DSP-Labor (Studiengang BEL-EK/ES)

In der Laborübung zur Digitalen Signalverarbeitung (LV Methoden der Signalverarbeitung im BEL-EK und LV Signalverarbeitung III im BEL-ES) sollen Sie Kenntnisse der digitalen Signalverarbeitung aus dem seminaristischen Unterricht durch praktische Übungen vertiefen sowie den Aufbau und die Programmierung eines modernen digitalen Signalprozessors (DSP) kennen lernen. Hierzu werden Sie in Gruppen mit maximal drei Studierenden an acht (BEL-EK) bzw. zwölf (BEL-ES) Terminen drei bis fünf Versuche zu den Themen

Analog-Digital-Umsetzung und digitale Signale
FIR-Filter
IIR-Filter
DTMF-Detektor mit FFT (optional für BEL-EK, Pflicht für BEL-ES)
DTMF-Detektor mit Goertzel-Algorithmus für FFT (Pflicht für BEL-ES)

bearbeiten.

Um diese Übungen in der im Labor verfügbaren Zeit absolvieren zu können, müssen die Übungen gut vorbereitet werden und insbesondere die folgenden Aufgaben vor den jeweiligen Terminen bearbeitet werden. Zu Beginn jeder neuen Laborübung werden diese vorbereitenden Aufgaben im Labor besprochen. Es wird erwartet, dass alle Laborteilnehmer in der Lage sind, die Lösungen oder zumindest Lösungsansätze zu den Aufgaben zu präsentieren.

Die zur Vorbereitung zu studierenden Manuals finden Sie auf der Webseite public.beuth-hochschule.de/~purat unter Lehre → DSP-Labor oder in Moodle.

1. Versuch (Analog-Digital-Umsetzung und digitale Signale)

Der erste Versuch soll das Verständnis für die Umsetzung zwischen analogen und digitalen Signalen vertiefen. Er soll deutlich machen, wie die analogen Signale auf dem DSP digital repräsentiert werden. Darüber hinaus gibt Ihnen dieser Versuch die Gelegenheit, Programmiererfahrungen mit dem DSP zu erwerben.

Aufbau des DSP und wesentliche Eigenschaften:

Im DSP-Labor werden Evaluation-Boards (EVB) mit dem Blackfin BF-561 von Analog Devices eingesetzt. Im Prinzip lässt sich dieser Prozessor in einer Hochsprache wie C oder C++ programmieren, jedoch ist für die rechenintensiven Aufgaben der digitalen Signalverarbeitung eine hardwarenahe Programmierung in Assembler unumgänglich. Darüber hinaus werden die Schnittstellen des DSPs zur Kommunikation mit dem Audio-Codec und anderen Hardwareelementen des EVB benötigt. Damit Sie die Abläufe besser verstehen können, werden Sie sich im Folgenden grundlegende Kenntnisse der Hardware des DSPs erarbeiten

1.1 Aufgabe: Studieren Sie den Aufbau des DSPs und des DSP-Kerns (Core) in den Kapiteln 1 „Introduction“ und 2 „Computational Units“ im Manual „ADSP-BF561 Blackfin® Processor Hardware Reference“.

Lesen Sie sich dazu die folgenden Abschnitte durch:

- Peripherals, Core Architecture, Memory Architecture (1-1 bis 1-9)
- SPI, SPORT (1-15 bis 1-17)
- Programmable Flags (1-20 und 1-21)
- Instruction Set Description, Development Tools (1-26 bis 1-30)
- Data Formats, Register Files (2-1 bis 2-7)
- ALU (2-25 bis 2-29)
- Barrel Shifter (2-50 bis 2-54)

und beantworten Sie die folgenden Fragen:

a) Wieviele DSP-Cores besitzt der Blackfin BF-561 (im Folgenden kurz DSP)?
b) Welche Schnittstellen besitzt der DSP?
c) Welche Recheneinheiten (computational units) besitzt jeder DSP-Core?
d) Welche Rechnerarchitektur besitzt der DSP? Wie unterscheidet sie sich von einer klassischen von-Neumann- oder Harvard-Architektur?
e) Welche Befehlswortlängen besitzt der DSP?
f) Wie groß ist der Addressraum des DSPs?
g) Über welche internen Speicherressourcen verfügt der DSP?
h) Wieviele programmable Flags (PF) besitzt der DSP? Nennen Sie 4 Register, die zur Konfiguration der PFs dienen.
i) In welchem Datenformat werden vorzeichenbehaftete Zahlen vom DSP verarbeitet?
j) Welche Datenregister (Data Registers) besitzt der DSP zur Verarbeitung von Daten in der DAU (Data Arithmetic Unit)? Welche Länge haben sie?
k) Welche Zeigerregister (Pointer Registers) besitzt der DSP zur Adressierung von Daten in der AAU (Address Arithmetic Unit)? Welche Länge haben sie?
l) Welche Operationen lassen sich mit der ALU (Arithmetic Logic Unit) ausführen? Welche Wortbreite können die Operanden dabei haben?
m) Welche Operationen lassen sich mit dem Barrel Shifter ausführen? Welche Wortbreite können die Operanden dabei haben?

Benutzung der Taster und LEDs auf dem EVB:

Das EVB besitzt 16 LEDs und 4 Taster, die zu Kontrollzwecken und zur Programmablaufsteuerung eingesetzt werden können. Die LEDs und Taster sind mit den programmable Flags (PF) des DSP verbunden und können somit durch Software an- und ausgeschaltet bzw. abgefragt werden.

1.2 Aufgabe: Studieren Sie die Kontrollregister der programmable Flags des DSP im Kapitel 14 „Programmable Flags“ im Manual „ADSP-BF561 Blackfin® Processor Hardware Reference“.

Beantworten Sie die folgende Frage:

a) Wozu dienen die Register FI0n_DIR, FI0n_INEN und FI0n_FLAG_D und wie müssen Sie gesetzt werden, um die PFs zu konfigurieren und zu benutzen?

Kommunikation mit dem Audio-Codec:

Das EVB ist primär für die Verarbeitung von Audiosignalen ausgelegt. Hierzu befindet sich auf dem EVB ein Audio-Codec (AD1836 von Analog Devices), mit dem bis zu vier analoge Audioeingänge (zwei Stereokanäle) digitalisiert werden können, und der die analoge Ausgabe von bis zu sechs Audioausgängen (drei Stereokanäle) unterstützt.

1.3 Aufgabe: Studieren Sie das Datenblatt des Audio-Codecs „AD1836 Data Sheet“ und finden Sie die folgenden Kennwerte des Codecs heraus:

a) Auflösung (Bit)
b) Abtastfrequenz (kHz)
c) Analoge Aussteuerung der Eingänge und Ausgänge (Volt)

Die einmalige Konfiguration des Codecs durch den DSP erfolgt über die SPI-Schnittstelle. Ist der Codec konfiguriert, so werden permanent Audiodaten zwischen DSP und Codec übermittelt. Der gesamte Signallaufweg zwischen den analogen Ein- und Ausgängen und dem DSP ist in Abb. 1 dargestellt. Auf dem EVB dienen zunächst analoge Operationsverstärkerschaltungen (OPV) zur Vorfilterung und Pegelanpassung der analogen Audiokanäle. Im Codec werden die analogen Signale dann mittels $\Sigma\Delta$-Umsetzung mit sehr hoher Überabtastung digitalisiert. Eine nachfolgende Dezimationsstufe im Codec verringert die Abtastfrequenz und filtert gleichzeitig das durch die Digitalisierung entstandene Quantisierungsrauschen. Die Audiodaten der vier digitalisierten Eingangskanäle (ADC-Data) werden dann als 32-Bit-Werte über einen als I²S-Schnittstelle konfigurierten SPORT0-Kanal vom DSP eingelesen. Dabei werden, wie in Abbildung 2 zu sehen, nur vier von acht möglichen Worten des Datenrahmens für die Audiodaten genutzt. Vom SPORT0 werden die Daten via DMA in einen DMA-Lesepuffer (iDMARxBuffer) im DSP-Speicher geschrieben. Ist ein vollständiger Rahmen mit acht Datenworten eingelesen, wird vom SPORT0 ein Interrupt ausgelöst. Da pro Abtasttakt acht Datenworte übertragen werden, erfolgt der Interrupt genau einmal pro Abtasttakt. Sollen die Audiodaten durch den DSP verarbeitet werden, muss eine Interrupt-Service-Routine (ISR) dafür sorgen, dass die Daten in einen anderen Speicherbereich kopiert und nicht durch neue Daten überschrieben werden.

Gleichzeitig können vom DSP verarbeitete Daten von der ISR in einen DMA-Schreibpuffer (iDMATxBuffer) im DSP-Speicher kopiert werden. Diese werden dann per DMA zu einem zweiten SPORT0-Kanal übertragen, der ebenfalls als I²S-Schnittstelle konfiguriert ist. Über diese Schnittstelle werden die Audiodaten (DAC-Data) von sechs Audiokanälen zum Audio-Codec geschickt. Im Codec erfolgt zunächst eine Interpolation auf eine höhere Abtastfrequenz, bevor dann die Audiodaten per $\Sigma\Delta$-Umsetzung in analoge Signale umgesetzt werden und wiederum über Operationsverstärkerschaltungen zu den Ausgangsbuchsen des EVB gelangen.

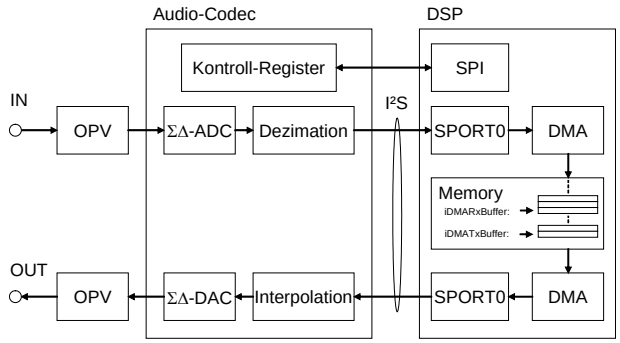

Abb. 1 – Kommunikation zwischen DSP und Codec und Signallaufweg

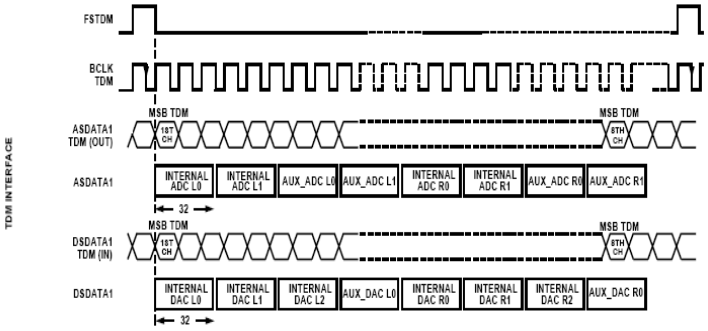

Abb. 2 - Serielle Datenübertragung zwischen DSP und Audio-Codec

1.4 Aufgabe: Erstellen Sie in der Programmiersprache C eine Funktion *copyData*, welche beim Aufruf aus der o.g. ISR (d.h. mit jedem Abtasttakt) die Audiodaten des Kanals „Internal ADC R0“ (rechter Kanal des ADC 0) aus dem DMA-Lesepuffer in einen Speicherbereich *iInput* der Größe *N* schreibt, so dass in diesem Speicherbereich immer die letzten *N* eingelesenen Werte x(n) … x(n−N+1) des Audiosignals verfügbar sind, s. Abbildung 3. Die Adresse **pWrite,* unter der der aktuelle Signalwert abgelegt werden soll, muss von der Funktion *copyData* jeweils inkrementiert werden und beim Erreichen der Speichergrenze wieder auf die Basisadresse *iInput* zurückgesetzt werden. Der Speicher *iInput* wird dadurch zirkular adressiert.

Der Funktionsprototyp der Funktion *copydata* laute:

void copyData(int iInput, int** pWrite, int N);*

Die Variable *iDMARxBuffer*, die auf den DMA-Lesepuffer vom Typ integer zeigt, kann als global definiert angenommen werden.

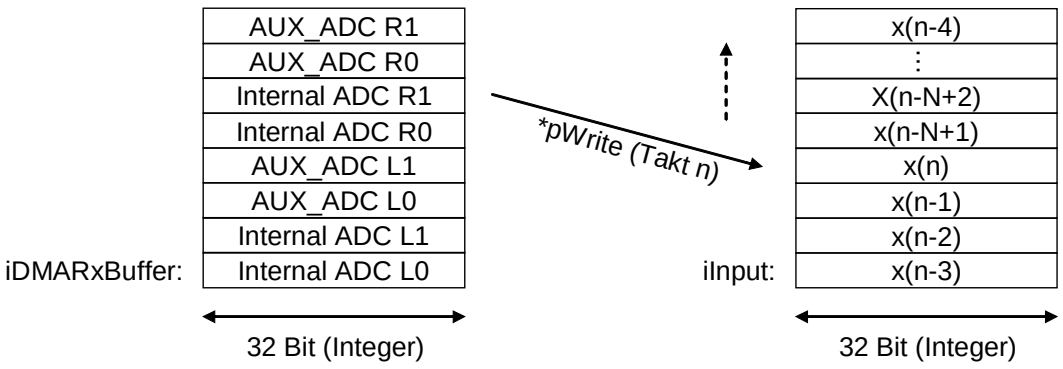

Abb. 3 – Kopieren der Audiodaten aus dem DMA-Lesepuffer in einen zirkularen Speicher

1.5 Aufgabe: Erstellen Sie in der Programmiersprache C eine Funktion *genSinus*, welche beim Aufruf aus der o.g. ISR (d.h. mit jedem Abtasttakt) jeweils den Wert einer Sinusfunktion liefert, so dass bei sukzessivem Aufruf dieser Funktion im Takt der Abtastfrequenz und Ausgabe des Wertes

über den Audio-Codec am Ausgang des EVB ein Sinussignal generiert wird. Die generierte Sinusfunktion soll dabei eine normierte Amplitude *A* im Bereich 0 bis 1 besitzen. Die Frequenz soll über einen Parameter *F200* in 200-Hertz-Schritten mit *F200* = 0 … 50 einstellbar sein. Beispielsweise soll für *F200* = 10 ein Sinussignal mit der Frequenz 2 kHz erzeugt werden.

Der Funktionsprototyp der Funktion *genSinus* laute:

float genSinus(float A, short Freq200);

Hinweis: Die fortschreitende Zeit kann auf dem DSP durch eine bei jedem Aufruf (d.h. mit jedem Takt) inkrementierte Variable gebildet werden. Da ein (theoretisch) unendlich langes Sinus-Signal erzeugt werden soll, würde dies irgendwann unweigerlich zu einem Überlauf des Wertebereichs der Variable führen. Nutzen Sie daher die in Abb. 4. gezeigte Periodizität der Sinusfunktion $\sin(\pi x)$ aus. Die Variable x kann auf den Wertebereich -1 … 1 eingeschränkt werden[1]. Anstelle der Werte $\sin(\pi\cdot(x_0+2))$, $\sin(\pi\cdot(x_0+4))$, etc., bei denen das Argument fortlaufend größer wird und irgendwann überläuft, kann dann auch immer der Wert $\sin(\pi\cdot x_0)$ berechnet werden.

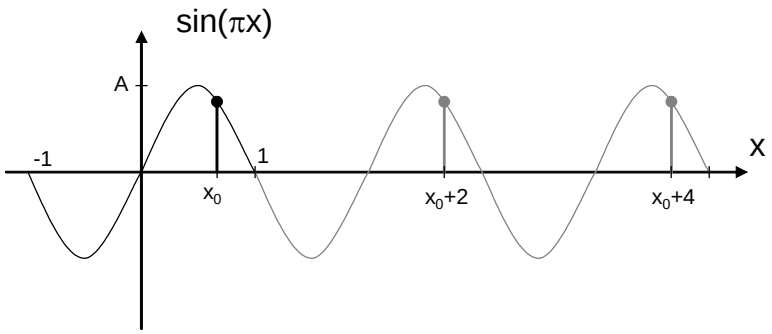

Abb. 4 – sinus-Funktion

Die Sinusfunktion der Standard-C-Bibliothek kann vorausgesetzt werden.

Programmierung in Assembler:

Obwohl der DSP direkt in einer Hochsprache wie C programmiert werden kann, sind viele Funktionen leichter und vor allem effizienter realisierbar, wenn sie direkt in der Assemblersprache programmiert werden. Daher ist es unerlässlich, sich auch mit der Assemblersprache des DSPs auseinanderzusetzen. Die Assemblerbefehle für die DSPs der Blackfin-Familie können dem Manual „ADSP-BF53x-BF56x Blackfin® Processor Programming Reference“ entnommen werden.

1.6 Aufgabe: Studieren Sie die Load-, Move- und Store-Befehle des DSP in den Kapiteln 8 und 9 des o.g. Manuals.

Beantworten Sie folgende Fragen:

a) Was ist der Unterschied zwischen einem unmittelbaren (immediate) und einem indirekten (indirect) Ladebefehl (Load)? Wie wird syntaktisch dazwischen unterschieden?
b) Erläutern Sie den Unterschied zwischen den folgenden indirekten Speicherbefehlen (Store):

```
[P0] = R1; W[P0] = R1; B[P0] = R1;
```

c) Was bewirkt die folgende Befehlssequenz?

```
P1.H = 0xFFFF; P1.L = 0x0000; R2 = [P1];
P2.H = 0xFFFF; P2.L = 0x0010; W[P2] = R2.L;
```

[1] Die Bedeutung dieses Wertebereiches wird in Übung 2 noch deutlicher.

d) Erläutern Sie den Unterschied zwischen den folgenden indirekten Ladebefehlen:

```
R4 = W[P0] (Z); R4 = W[P0] (X); R4.H = W[P0]; R4.L = W[P0];
```

e) Was bewirkt die folgende Befehlssequenz?

```
P1.H = 0xFFFF; P1.L = 0x0010; R7.H = W[P1];
P1.H = 0xFFFF; P1.L = 0x0000; [P1] = R7;
```

f) Erläutern Sie den Unterschied zwischen den folgenden indirekten Ladebefehlen:

```
R5 = [P0]; R5 = [P0++]; R5 = [P0 + 2];
```

1.7 Aufgabe: Studieren Sie die Shifter-Befehle des DSP im Kapitel 14 des o.g. Manuals.

Beantworten Sie folgende Fragen:

a) Was ist der Unterschied zwischen einem arithmetischen und einem logischen Shift? Wie wird syntaktisch dazwischen unterschieden?
b) Im 16-Bit-Register R0.L stehe der Wert 20dez = 0014hex = 0x00142. Durch welche Shifter-Befehle lässt sich dieser Wert verdoppeln bzw. halbieren?
c) Im 16-Bit-Register R0.L stehe der Wert -20 = 0xFFEC. Durch welche Shifter-Befehle lässt sich dieser Wert verdoppeln bzw. halbieren?
d) Im 16-Bit-Register R1.L stehe der Wert 10000 = 0x2710. Welcher Wert (dezimal bzw. hexadezimal) steht jeweils in dem Register R2.L nach den Befehlen

```
R2.L = R1.L << 2; bzw. R2.L = R1.L << 2 (S);
```

e) Im 16-Bit-Register R1.L stehe der Wert -10000 = 0xD8F0. Welcher Wert (dezimal bzw. hexadezimal) steht jeweils in dem Register R2.L nach den Befehlen

```
R2.L = R1.L << 2; bzw. R2.L = R1.L << 2 (S);
```

[2] Im Folgenden wird für die hexadezimale Schreibweise immer das Präfix 0x verwendet. Ohne dieses Präfix handelt es sich um eine dezimale Schreibweise.

2. Versuch (FIR-Filter)

In der zweiten Laborübung werden FIR-Filter programmiert. Die für die Programmierung der entsprechenden Differenzengleichungen notwendigen Multiplikationen bzw. Multiplikationen und Additionen (MAC, Multiply-Accumulate) erfordern eine Beschäftigung mit der Multipliziereinheit und den Zahlenformaten des DSP. Darüber hinaus sollen die Programmiererfahrungen mit dem DSP erweitert werden.

Festkommazahlen:

Der Blackfin-DSP gehört zur Klasse der Festkommaprozessoren, besitzt also keine Gleitkomma-Arithmetikeinheit. Daher können Gleitkommamultiplikationen vom DSP nur emuliert werden, was mit einer großen Anzahl von Prozessorzyklen verbunden ist. Deutlich schneller werden die Programme, wenn nur ganze Zahlen bzw. Festkommazahlen verarbeitet werden. Die Arithmetikeinheiten des Blackfin-DSP sind primär für die Verarbeitung von 16-Bit-Festkommazahlen ausgelegt. Hierfür erzielt er eine optimale Performance.

Die Darstellung von Zahlen in einem polyadischen Zahlensystem basiert auf Ziffern x_s und Regeln, wie der Zahlenwert X aus diesen Ziffern zu bestimmen ist. Bei der binären Zahlendarstellung sind die Ziffern die Bits x_s. Bei dualen Festkommazahlen ist jedem Bit x_s eine feste Zweierpotenz als Stellenwert zugeordnet. Für die Darstellung vorzeichenloser ganzer Zahlen (*unsigned Integer*) mit M Bit hat das LSB (least significant bit) den Stellenwert 2^0 und das MSB (most significant bit) den Stellenwert 2^{M-1}:

x_{M-1} x_{M-2} … x_3 x_2 x_0

Der Zahlenwert und der darstellbare Zahlenbereich ergibt sich dann in dieser Darstellung zu:

$$X_{IU} = \sum_{s=0}^{M-1} x_s \cdot 2^s, \quad X_{IU} \in \left[0, 2^M - 1\right]$$

Bei vorzeichenbehafteten ganzen Zahlen (*signed Integer*) wird in der auch vom Blackfin DSP benutzten Zweierkomplementdarstellung dagegen der höchste Stellenwert negativ gebildet, wodurch sich folgende Zahlendarstellung und folgender Zahlenbereich ergeben[3]:

$$X_{IS} = -x_{M-1} \cdot 2^{M-1} + \sum_{s=0}^{M-2} x_s \cdot 2^s, \quad X_{IS} \in \left[-2^{M-1}, 2^{M-1} - 1\right]$$

Für den Zusammenhang zwischen dem *signed Integer* und *unsigned Integer* Format lässt sich folgender Zusammenhang herleiten, der es ermöglicht, die Zahlenwerte der verschiedenen Formate leicht ineinander zu überführen:

$$X_{IS} = -x_{M-1} \cdot (2^M - 2^{M-1}) + \sum_{s=0}^{M-2} x_s \cdot 2^s = -x_{M-1} \cdot 2^M + \sum_{s=0}^{M-1} x_s \cdot 2^s = X_{IU} - x_{M-1} \cdot 2^M$$

Um auch Nachkommastellen darstellen zu können, werden die Stellenwerte um eine beliebige feste Zweierpotenz reduziert. Für K (duale) Nachkommastellen beträgt dieser Skalierungsfaktor 2^{-K}; das Zahlenformat nennt sich *scaled Integer* und ist sowohl für vorzeichenlose als auch vorzeichenbehaftete Zahlen anzuwenden. Da für die Vorkommastellen nach der Skalierung nur noch N=M-K Stellen der Dualzahl übrig bleiben, wird für dieses Zahlenformat auch die Bezeichnung N.K-Format benutzt. Die ganzzahlige Zahlendarstellung heißt in dieser Nomenklatur auch N.0-Format. Zur Trennung der Vor- und Nachkommastellen wird der Punkt benutzt:

x_{N-1} x_{N-2} … x_2 x_0 . x_{-1} x_{-2} … x_{-K+1} x_{-K}

.

[3] Das Zweierkomplement kann praktisch einfach durch Invertieren aller Bits (Einerkomplement) und Addition eines LSB gebildet werden.

Die Zahlenwerte und Zahlenbereiche im unsigned bzw. signed N.K-Format ergeben sich aus den Integerdarstellungen wie folgt:

$$\text{Unsigned}: X_{N.K} = X_{IU} \cdot 2^{-K} \quad X_{N.K} \in \left[0, 2^{M-K} - 2^{-K}\right] = \left[0, 2^{N} - 2^{-K}\right]$$
$$\text{Signed}: X_{N.K} = X_{IS} \cdot 2^{-K} \quad X_{N.K} \in \left[-2^{M-K-1}, 2^{M-K-1} - 2^{-K}\right] = \left[-2^{N-1}, 2^{N-1} - 2^{-K}\right]$$

Unter Vernachlässigung des (kleinen) Wertes 2^{-K} ist der Wertebereich im N.K-Festkommaformat also nach wie vor nur durch die Anzahl N der Vorkommastellen gegeben. Die betragsmäßig kleinste darstellbare Zahl im N.K-Format beträgt 2^{-K}, ist also nur durch die Anzahl der Nachkommastellen gegeben. Alle Zahlen im N.K-Festkommaformat sind gemäß der o.a. Darstellung ganzzahlige Vielfache dieser kleinsten darstellbaren Zahl, d.h. die Auflösung in diesem Zahlenformat beträgt 2^{-K}.

Ein besonders wichtiges scaled Integerformat ist das gebrochene Format (*fractional*), in dem der Zahlenbereich auf den Bereich $[0,1-2^{-K}]$ (unsigned) bzw. $[-1,1-2^{-K}]$ (signed) eingeschränkt ist (normierter Wertebereich). Für das unsigned fractional Format beträgt die Anzahl der Vorkommastellen N = 0 (0.K-Format), d.h. alle M Ziffern werden als Nachkommastellen verwendet (K = M). Im signed fractional Format beträgt die Anzahl der Vorkommastellen N=1, wobei die Vorkommastelle (also das MSB) wie im Zweierkomplement üblich nur das Vorzeichen angibt. Hier werden also M-1 Ziffern als Nachkommastellen verwendet (K = M-1).

$$X_{FU} = X_{IU} \cdot 2^{-M} \quad X_{FU} \in \left[0, 2^{M-M} - 2^{-M}\right] = \left[0, 1 - 2^{-M}\right]$$
$$X_{FS} = X_{IS} \cdot 2^{-(M-1)} \quad X_{FS} \in \left[-2^{M-1-(M-1)}, 2^{M-1-(M-1)} - 2^{-(M-1)}\right] = \left[-1, 1 - 2^{-M+1}\right]$$

Beispiel:

Ein Bitstring mit 16 Bit lautet (abgekürzt als Hexadezimalzahl) $ABCD_{hex}$ = 0xABCD. Je nach Zahlenformat können diese 16 Bit unterschiedlich als Zahl interpretiert werden:

unsigned Integer:	X = 43981
signed Integer:	X = -21555
unsigned 8.8-Format:	X = 171.80078125
signed 8.8-Format:	X = -84.19921875
unsigned Fractional:	X = 0.671096801
signed Fractional:	X = -0.657806396

Erkenntnis: Ein Bitstring repräsentiert also unterschiedliche Zahlen, je nachdem in welchem Zahlenformat er interpretiert wird.

2.1 Aufgabe: Überprüfen Sie die o.a. Beispiele!

2.2 Aufgabe: Stellen Sie die folgenden Zahlenwerte in den angegebenen Formaten dar, geben Sie also den jeweiligen Bitstring an, abgekürzt als Hexadezimalzahl (16 Bit):

unsigned Integer:	X = 30000
signed Integer:	X = 30000 und -30000
unsigned 8.8-Format:	X = 11.46875
signed 8.8-Format:	X = 10.390625 und -10.390625
unsigned Fractional:	X = 0.40234375
signed Fractional:	X = 0.40234375 und -0.40234375

Festkommaaddition

Ist für beide Operanden das Zahlenformat identisch, besitzen also alle Ziffern der Operanden die gleichen Stellenwerte, muss bei einer Addition (oder Subtraktion) im Festkommaformat das Zahlenformat nicht berücksichtigt werden. Dieselbe Additionseinheit kann also Integer und fractional Formate verarbeiten.

Festkommamultiplikation

Bei der Multiplikation ist dies anders. Werden zwei Integer-Werte mit N Stellen miteinander multipliziert, so addieren sich die Exponenten der Stellenwerte, d.h. das Ergebnis besitzt deutlich höhere Stellenwerte als die Operanden. Bei vorzeichenlosen Zahlen ergibt sich für das Ergebnis ein Wertebereich mit N' = 2N Stellen, d.h. das Ergebnis kann exakt nur als Integer mit 2N Stellen dargestellt werden. Die gleiche Darstellung kann auch für das Ergebnis einer Multiplikation von zwei vorzeichenbehafteten Zahlen verwendet werden, obwohl hier der Zahlenbereich kleiner ist und eigentlich ein Bit redundant ist.

Beispiele:

1. Operanden 16 bit unsigned Integer (16.0), Ergebnis 32 bit unsigned Integer (32.0):

   $X_1 = X_2 = 2^{14} = 0x4000$; $Y = X_1 \cdot X_2 = 2^{14} \cdot 2^{14} = 2^{28} = 0x10000000$

2. Operanden 16 bit signed Integer (16.0), Ergebnis 32 bit signed Integer (32.0):

   $X_1 = 2^{14} = 0x4000$; $X_2 = -2^{14} = 0xC000$; $Y = X_1 \cdot X_2 = -2^{14} \cdot 2^{14} = -2^{28} = 0xF0000000$

Wird nach der Multiplikation von ganzen Zahlen für eine nachfolgende Verarbeitung das Ergebnis wieder auf N Stellen begrenzt, so können Überläufe entstehen, wenn die Operanden zu groß waren. Werden dagegen zwei Fractional-Werte mit N Stellen miteinander multipliziert, so liegt aufgrund des normierten Wertebereichs das Ergebnis wieder im normierten Wertebereich vor. Allerdings entstehen durch die Multiplikation der kleinsten Stellenwerte noch kleinere Stellenwerte, so dass sich die Anzahl der Nachkommastellen verdoppelt, K' = 2K.

Gemäß der scaled Integer Darstellung lässt sich die fractional Multiplikation auf eine Ganzzahlmultiplikation zurückführen. Das Ergebnis muss jedoch sowohl für vorzeichenbehaftete als auch vorzeichenlose Multiplikation mit 2K Nachkommastellen interpretiert werden:

$$Y_{FU} = X_{1,FU} \cdot X_{2,FU} = X_{1,IU} \cdot 2^{-K} \cdot X_{2,IU} \cdot 2^{-K} = \left(X_{1,IU} \cdot X_{2,IU}\right) \cdot 2^{-2K}$$
$$Y_{FS} = X_{1,FS} \cdot X_{2,FS} = X_{1,IS} \cdot 2^{-K} \cdot X_{2,IS} \cdot 2^{-K} = \left(X_{1,IS} \cdot X_{2,IS}\right) \cdot 2^{-2K}$$

Beispiele:

3. Operanden 16 bit unsigned Fractional (0.16), Ergebnis 32 bit unsigned Fractional (0.32):

   $X_1 = X_2 = 2^{-2} = 0x4000$; $Y = X_1 \cdot X_2 = 2^{-2} \cdot 2^{-2} = 2^{-4} = 0x10000000$

4. Operanden 16 bit signed Fractional (1.15), Ergebnis 32 bit signed scaled Integer (2.30):

   $X_1 = 2^{-1} = 0x4000$; $X_2 = -2^{-1} = 0xC000$; $Y = X_1 \cdot X_2 = -2^{-1} \cdot 2^{-1} = -2^{-2} = 0xF0000000$

Bei signed fractional Operanden liegt das Ergebnis also mit zwei Vorkommastellen vor, also nicht mehr als fractional Format. Da von den Vorkommastellen wegen des normierten Bereichs jedoch das MSB redundant ist (redundantes Vorzeichenbit), kann man das Ergebnis stattdessen auch im (1.2K+1)-Format, also wieder im fractional Format angeben. Dazu ist das Ergebnis der Ganzzahlmultiplikation jedoch noch einmal um eine Stelle nach links zu shiften („left-shift correction"), da sich das Festkomma ja genau um diese eine Stelle gegenüber dem 2.2K-Format verschiebt.

Hinweis: Bei der Multiplikation zweier vorzeichenbehafteter Zahlen entsteht immer ein zusätzliches redundantes Vorzeichenbit. Dieses redundante Vorzeichenbit kann durch einen Linksshift entfernt werden. Der ADSP-BF 561 entfernt bei der Multiplikation zweier vorzeichenbehafteter Zahlen im fractional-Format immer das redundante Vorzeichenbit durch eine „left-shift correction", so dass durch die Multiplikation zweier Operanden im 1.15-Format ein Ergebnis im 1.31-Format entsteht. Bei der Multiplikation zweier vorzeichenbehafteter Integer-Zahlen führt der DSP keine „left-shift correction" durch. Das redundante Vorzeichenbit wird hier also nicht entfernt, so dass durch die Multiplikation zweier Operanden im 16.0-Format ein Ergebnis im 32.0-Format entsteht. Einzelheiten hierzu gibt die Tabelle 15-2 auf Seite 15-46 des Manuals „ADSP-BF53x-BF56x Blackfin® Processor Programming Reference".

Beispiel:

5. Operanden 16 bit signed Fractional (1.15), Ergebnis 32 bit signed Fractional (1.31):

$X_1 = 2^{-1} = 0x4000$; $X_2 = -2^{-1} = 0xC000$; $Y = X_1 \cdot X_2 = -2^{-1} \cdot 2^{-1} = -2^{-2} = 0xE0000000$

2.3 Aufgabe: Überprüfen Sie die o.a. Beispiele 1. – 5.!

2.4 Aufgabe: Führen Sie folgende Festkommamultiplikationen durch und geben Sie die Ergebnisse im angegebenen Format an:

Operanden 16 bit unsigned Int (16.0), Ergebnis 32 bit unsigned Int (32.0): X_1= 3000, X_2=4000
Operanden 16 bit signed Int (16.0), Ergebnis 32 bit signed Int (32.0): X_1= 2000, X_2= -4000
Operanden 16 bit unsigned Fract (0.16), Ergebnis 32 bit unsigned Fract (0.32):
X_1= 0.40625, X_2=0.71875
Operanden 16 bit signed Fract (1.15), Ergebnis 32 bit signed Fract (1.31):
X_1= 0.3125, X_2= -0.71875

Programmierung in Assembler:

Für das zu realisierende FIR-Filter soll die (rechenintensive) Berechnung der Differenzengleichung effizient in Assembler programmiert werden. Hierzu sind Kenntnisse der Multiplizierereinheit, der zyklischen Adressierung und der Schleifenprogrammierung notwendig. Die Assemblerbefehle für die DSPs der Blackfin-Familie können dem Manual „ADSP-BF53x-BF56x Blackfin® Processor Programming Reference“ entnommen werden.

2.5 Aufgabe: Studieren Sie die Multiplizier-Befehle des DSP im Kapitel 15 (Seite 15-43 bis 15-72) des o.g. Manuals.

Beantworten Sie folgende Fragen:

a) In den 16-Bit-Registern R1.L und R0.L stehen die Werte 1000_{hex} = 0x1000 bzw. 0800_{hex} = 0x0800. Geben Sie den Inhalt der Ergebnisregister jeweils nach den folgenden Befehlen an:

```
R2 = R1.L * R0.L (IS);          // Ergebnisregister R2
R2 = R1.L * R0.L;    // Ergebnisregister R2
R2.L = R1.L * R0.L (IS);        // Ergebnisregister R2.L
R2.H = R1.L * R0.L; // Ergebnisregister R2.H
```

b) Welcher Unterschied besteht zwischen den beiden letzten Befehlen hinsichtlich der Durchführung der Multiplikation auf dem DSP.

c) In den 16-Bit-Registern R1.L und R0.L stehen wiederum die Werte 1000_{hex} = 0x1000 bzw. 0800_{hex} = 0x0800. Geben Sie den Inhalt des Akkumulators A0 nach der folgenden Befehlsfolge an:

```
A0 = R1.L * R0.L; A0 += R1.L * R0.L;
```

d) Welcher Unterschied besteht zwischen den beiden folgenden Befehlen hinsichtlich der Durchführung der Multiplikation auf dem DSP:

```
R0.L = (A0 += R1.L * R0.L); und R0.H = (A1 += R1.L * R0.L);
```

2.6 Aufgabe: Studieren Sie die zyklische Adressierung im DSP im Kapitel 5 (Seiten 5-12 bis 5-15) des o.g. Manuals.

Beantworten Sie folgende Fragen:

a) Welche Aufgaben haben die B-, M-, L- und I-Register für die zyklische Adressierung.

b) Der Speicher des DSP sei wie folgt initialisiert:

Adresse	0xFF800CA4	0xFF800CA5	0xFF800CA6	0xFF800CA7	0xFF800CA8	0xFF800CA9
Datum	0x11	0x22	0x33	0x44	0x55	0x66

Bestimmen Sie den Inhalt der Register R1.L und I0 (Hex-Werte) nach folgenden Operationen:

```
B0.L=0x0CA4; B0.H=0xFF80;
I0.L=0x0CA4; I0.H=0xFF80;
L0=6;
R1.L=W[I0++];
R1.L=W[I0++];
R1.L=W[I0++];
```

2.7 Aufgabe: Studieren Sie die Schleifenprogrammierung des DSP im Kapitel 4 (Seiten 4-21 bis 4-24) des o.g. Manuals.

Beantworten Sie folgende Frage:

a) Bestimmen Sie den Inhalt des Indexregisters I0 (Hex-Wert) nach folgenden Operationen:

```
I0.L=0x0CA0; I0.H=0xFF80; L0=0;
P1=2;
M3=4;
LSETUP(_LOOP_START,_LOOP_END) LC0 = P1;
_LOOP_START:
    R1=[I0++M3];
_LOOP_END:
    R1=[I0++M3];
```

SIMD- und Parallel-Befehle:

Aufgrund seiner Architektur ist der DSP in der Lage, bestimmte Befehle bzw. Befehlsfolgen sehr effizient zu bearbeiten. Unter SIMD-Befehlen (Single Instruction Multiple Data, auch Vektorbefehl genannt) versteht man solche Befehle, bei denen der Prozessor den Inhalt eines Datenregisters (z.B. eines 32-Bit-Registers) als mehrere individuelle Datenworte (z.B. zwei 16-Bit-Worte) verarbeitet. Unter Parallelbefehlen versteht man Befehlsfolgen, die vom DSP gleichzeitig bearbeitet werden, typischerweise eine arithmetisch/logische Operation und ein oder mehrere Lade- bzw. Speicher-Befehle.

2.8 Aufgabe: Studieren Sie die SIMD-Befehle (*Vector Operations*) zum Shiften und Multiplizieren im Kapitel 19 (Seiten 19-23 bis 19-31 bzw. 19-38 bis 19-45) des o.g. Manuals.

Beantworten Sie folgende Fragen:

a) Im 32-Bit-Register R4 stehe der Wert 0x000F0400. Im 16-Bit-Register R3.L stehe der Wert 0xFFFF. Geben Sie den Inhalt des Registers R0 nach dem folgenden Befehl an:

```
R0 = ASHIFT R4 BY R3.L (V);
```

b) Im den 32-Bit-Registern R2 und R3 stehen der Wert 0x10000800 bzw. 0x08001000. Geben Sie die Inhalte der Akkumulatoren nach dem folgenden Befehl an:

```
A0 = R2.L * R3.L, A1 = R2.H * R3.H;
```

2.9 Aufgabe: Studieren Sie die Parallel-Befehle des DSP im Kapitel 20 des o.g. Manuals.

Beantworten Sie folgende Frage:

a) Der Speicher des DSP sei wie folgt initialisiert:

Adresse	0xFF800CA4	0xFF800CA5	0xFF800CA6	0xFF800CA7
Datum	0x00	0x10	0x00	0x08

Bestimmen Sie den Inhalt des Akkumulators A0 und der Register R0.L und R1.L nach folgenden Operationen:

```
I0.L=0x0CA4; I0.H=0xFF80;
I1=I0;
A0 = 0 || R0.L = W[I0++] || R1.L = W[I1++];
A0 += R0.L * R1.L || R0.L = W[I0] || R1.L = W[I1];
```

Mixed-C-Assembler-Programmierung:

Obwohl die Programmierung in Assembler effizienter ist, ermöglicht eine Hochsprache wie C doch eine übersichtlichere, strukturellere Programmierung und wird für weniger rechenintensive Programmteile eingesetzt. In Assembler geschriebene Routinen können dabei direkt aus einem C-Programm aufgerufen werden. Im Prinzip wird der Aufruf einer C-Funktion *function(…)* vom Compiler lediglich in einen Sprungbefehl zu einer Programmadresse *_function* umgesetzt, unter der wiederum eine Assemblerroutine definiert sein kann. Das Compilermanual „Blackfin® Compiler Manual“ erläutert, wie die Übergabe der aktuellen Funktionsparameter vom Compiler umgesetzt wird.

2.10 Aufgabe: Studieren Sie die Umsetzung der Parameterübergabe des Compilers im Kapitel 1 des o.g. Manuals (Seiten 1-308 bis 1-311).

Beantworten Sie folgende Frage:

a) In einem C-Programm finden Variablendefinitionen und ein Funktionsaufruf wie folgt statt.

```
struct a {short x;short y;} = {100,200};
int b=300, c;
c=fct(&a,b);
```

Die zugehörige Assemblerfunktion sei wie folgt definiert:

```
_fct:
        I0 =R0;
        R2.L = W[I0++];
        R2.H = W[I0];
        R3 = R2.L*R2.H (IS);
        R0 =R3 + R1;
        RTS;
```

Bestimmen Sie den Inhalt der Variablen c nach dem Aufruf der Assembler-Funktion.

Systemtheoretische Vorbetrachtungen:

Als erstes FIR-Filter mit der Differenzengleichung

$$y(n) = \sum_{k=0}^{N} b_k \cdot x(n-k)$$

soll ein einfaches Mittelwertfilter 4. Ordnung (N=4) auf dem DSP implementiert werden. Die Koeffizienten eines solchen Filters lauten b_k = 1/(N+1) für k = 0, …,N.

2.11 Aufgabe: Bestimmen Sie für dieses Filter die z-Übertragungsfunktion (Systemfunktion), die Pole und Nullstellen, den Frequenzgang, die Impulsantwort und die Sprungantwort.

3. Versuch (IIR-Filter)

In der dritten Laborübung werden IIR-Filter programmiert. Aus programmiertechnischer Sicht sind hierzu keine weiteren Kenntnisse notwendig. Die besondere Problematik bei der Implementierung der IIR-Filter liegt vielmehr in der korrekten Umsetzung der rekursiven Differenzengleichung, so dass Rundungsfehler möglichst gering gehalten und Überläufe vermieden werden.

Eine prinzipiell mögliche Struktur für die Implementierung eines IIR-Filters beliebig hoher Ordnung ist durch eine Kaskadenschaltung von *M* IIR-Systemen zweiter Ordnung (*second order sections* (SOS), *Biquad*s) wie in Abb. 5 gegeben.

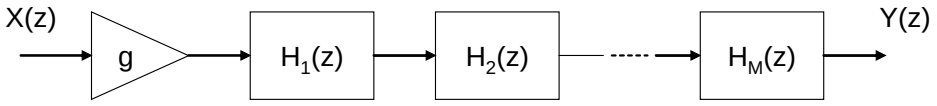

Abb. 5 – Reihenschaltung von Teilsystemen zweiter Ordnung

Jedes der *M* Teilsysteme ist durch zwei Nullstellen $z_{1,i}$ und $z_{2,i}$ und zwei Polstellen $p_{1,i}$ und $p_{2,i}$ gekennzeichnet. Zusätzlich besitzt das gesamte System einen Verstärkungsfaktor *g*, so dass das gesamte System eine gewünschte Verstärkung im Durchlassbereich (typischerweise 0dB) aufweist. Die Übertragungsfunktion des gesamten Systems lautet somit:

$$H(z) = g \cdot \prod_{i=1}^{M} H_i(z) \text{ mit } H_i(z) = \frac{(z - z_{1,i}) \cdot (z - z_{2,i})}{(z - p_{1,i}) \cdot (z - p_{2,i})} = \frac{z^2 - (z_{1,i} + z_{2,i}) \cdot z + z_{1,i} \cdot z_{2,i}}{z^2 - (p_{1,i} + p_{2,i}) \cdot z + p_{1,i} \cdot p_{2,i}} \tag{3.1}$$

Besitzt das System eine gerade Ordnung *N* = *2M*, so liegen die Pole und Nullstellen aller Teilsysteme außerhalb des Ursprungs. Ist die Filterordnung dagegen ungerade, *N* = *2M*-1, so besitzt ein Teilsystem einen Pol und eine Nullstelle im Ursprung, weist also letztlich nur die Ordnung eins auf. Sind bei den Teilsystemen zweiter Ordnung die Pole bzw. Nullstellen jeweils konjugiert komplex zueinander, d.h. $z_{1,i} = z_{2,i}^*$ und $p_{1,i} = p_{2,i}^*$, so folgt für die Übertragungsfunktion $H_i(z)$ des *i*-ten Teilsystems:

$$H_i(z) = \frac{z^2 - 2\,\mathrm{Re}\{z_{1,i}\} \cdot z + |z_{1,i}|^2}{z^2 - 2\,\mathrm{Re}\{p_{1,i}\} \cdot z + |p_{1,i}|^2} \tag{3.2}$$

Im Prinzip können so durch entsprechend viele Teilsysteme Filter beliebig hoher Ordnung realisiert werden. Allerdings ist die Realisierung nach Abb. 5 für eine Implementierung nicht besonders geeignet, da der Verstärkungsfaktor g unmittelbar auf das Eingangssignal wirkt. Da jedes einzelne Teilsystem typischerweise eine sehr hohe Verstärkung besitzt, ist der Verstärkungsfaktor g entsprechend sehr klein, so dass das Signal stark gedämpft wird und alle nachfolgenden durch Rundung entstehenden Rauschsignale im Verhältnis zum Eingangssignal sehr groß sind. Im Extremfall würde das Eingangssignal bereits unter die Auflösungsgrenze der digitalen Verarbeitung gedämpft (g < 1 LSB) und damit eliminiert. Würde hingegen der Verstärkungsfaktor erst nach der Reihenschaltung der Teilsysteme wirksam, würden alle einzelnen Teilsysteme wegen ihrer großen Verstärkung übersteuern, d.h. die Signalwerte würden den darstellbaren Zahlenbereich verlassen (Überlauf). Daher muss die Gesamtverstärkung g geeignet auf die einzelnen Teilsysteme verteilt werden und jedes Teilsystem erhält die Übertragungsfunktion

$$H_i'(z) = g_i \cdot \frac{z^2 - (z_{1,i} + z_{2,i}) \cdot z + z_{1,i} \cdot z_{2,i}}{z^2 - (p_{1,i} + p_{2,i}) \cdot z + p_{1,i} \cdot p_{2,i}} = g_i \cdot \frac{z^2 - 2\,\mathrm{Re}\{z_{1,i}\} \cdot z + |z_{1,i}|^2}{z^2 - 2\,\mathrm{Re}\{p_{1,i}\} \cdot z + |p_{1,i}|^2} \text{ mit } g = \prod_{i=1}^{M} g_i \tag{3.3}$$

Die Wahl der einzelnen Verstärkungen (*scales*) und die Reihenfolge (*order*) der Teilsysteme beeinflusst die Qualität des realisierten IIR-Filters. Die Wahl ist dabei im Wesentlichen von der Polgüte der Teilsysteme abhängig d.h. von dem Abstand der Pole zum Einheitskreis. Nach [Oppenheim,

Schafer: Zeitdiskrete Signalverarbeitung, Pearson, 2004] ergeben sich meist immer gute Ergebnisse, wenn folgende Regeln für die Reihenfolge der Teilsysteme beachtet werden:

1. Das dem Einheitskreis am nächsten liegende Polstellenpaar sollte mit dem Nullstellenpaar kombiniert werden, das ihm in der z-Ebene am nächsten liegt.
2. Regel 1 wird solange wiederholt, bis alle Pole und Nullstellen paarweise geordnet sind.
3. Die resultierenden Teilsysteme zweiter Ordnung sollten nach ihrer zunehmenden Nähe zum Einheitskreis bzw. ihrer zunehmenden Entfernung vom Einheitskreis geordnet werden. Wir werden in diesem Versuch mit dem System mit den kleinsten Polradien beginnen, also mit den Polen, die am weitesten entfernt vom Einheitskreis liegen.

Nach Anwendung dieser Regeln liegen die Pol- und Nullstellen für die Teilsysteme $i=1, \ldots, M$ fest. Der letzte Schritt besteht dann in der Bestimmung geeigneter Verstärkungen g_i für die Teilsysteme, so dass keine Überläufe entstehen.

Systemtheoretische Vorbetrachtungen:

Ein Tiefpassfilter-Entwurf liefere für ein IIR-Filter 8. Ordnung folgende Pole und Nullstellen:

Nullstellen	Pole
-0.2008384 ± j · 0.9796244	0.8298131 ± j · 0.4503930
0.8150114 ± j · 0.5794450	0.8486339 ± j · 0.5000172
0.6380986 ± j · 0.7699547	0.8073076 ± j · 0.3360904
0.7807243 ± j · 0.6248756	0.7857682 ± j · 0.1292345

3.1 Aufgabe: Sortieren Sie Pole und Nullstellen nach den obigen Regeln und legen Sie somit die Übertragungsfunktionen $H_i(z)$ der Teilsysteme fest.

3.2 Aufgabe: Bestimmen Sie die Durchlassverstärkung des Gesamtsystems (ohne Verstärkungsfaktor g) für die Frequenz f=0. Wie groß muss der Verstärkungsfaktor g gewählt werden, damit die Durchlassverstärkung (mit g) zu Eins (0 dB) wird? Wie groß muss der Verstärkungsfaktor g_i für jedes Teilsystem gewählt werden, wenn alle Teilsysteme den gleichen Verstärkungsfaktor besitzen?

Auf dem DSP werden die Teilsysteme gemäß folgender Struktur realisiert.

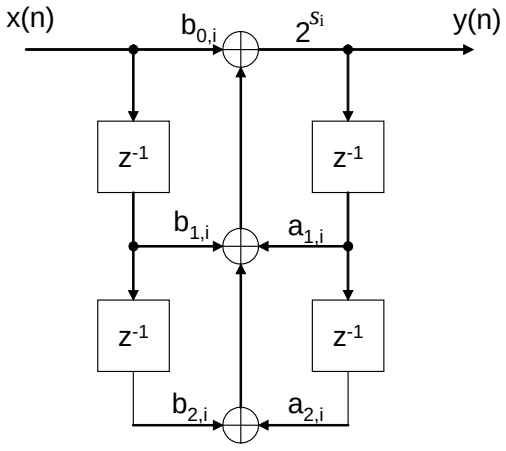

Abb. 6 – Realisierung eines Teilsystems zweiter Ordnung auf dem DSP

3.3 Aufgabe: Bestimmen Sie die Differenzengleichung für diese Struktur und vergleichen Sie sie mit der Differenzengleichung, die sich für jedes Teilsystem nach Gl. 3.1 bzw. 3.2 ergibt. Ermitteln Sie für das Tiefpassfilter 8. Ordnung aus den gegebenen Polen und Nullstellen die Transversalkoeffizienten $b_{k,i}$, k=0,1,2 und Rekursivkoeffizienten $a_{l,i}$, l=1,2 für alle Teilsysteme, i=1,..,4. Wählen Sie die Skalierungsfaktoren s_i ($s_i \in$ IN), so dass alle Koeffizienten vom Betrage

her kleiner als Eins werden und damit im fractional Format dargestellt werden können. Geben Sie die Koeffizienten im fractional (1.15) Format an.

4. Übung (DTMF-Detektor mit FFT)

In der vierten Laborübung soll ein DTMF-Detektor (*Dual Tone Multiple Frequency*) auf Basis einer FFT entwickelt werden. DTMF Detektoren kommen beim sogenannten Mehrfrequenzwahlverfahren (MFV) in Telefonanlagen zum Einsatz. Bei der DTMF-Übertragung werden die Informationen der Telefontastatur mittels einer Kombination zweier Sinustöne im Sprachband codiert. Der Detektor hat die Aufgabe, die Töne herauszufinden und die beiden gefundenen Töne wieder in die Tasteninformation zu decodieren. Die Tasteninformation wird über die LEDs auf dem EVB angezeigt. Die prinzipielle Struktur des Detektors ist in der folgenden Abbildung dargestellt.

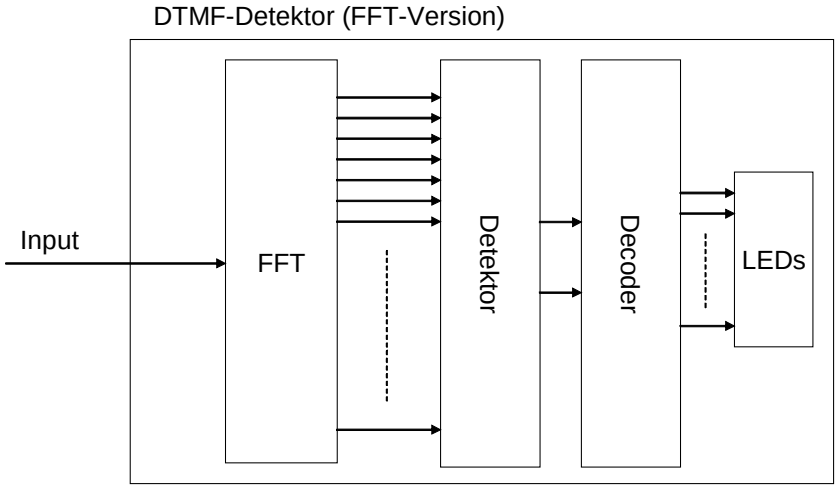

Abb. 7 – DTMF-Detektor mit FFT

4.1 Aufgabe: Geben Sie die Frequenzen in Hz und als normierte Kreisfrequenzen (Abtastfrequenz: 48 kHz) an, mit denen die DTMF-Übertragung arbeitet. Die Information können Sie in Büchern oder im Internet finden.

Fast-Fourier-Transformation und Segmentverarbeitung

Die FFT ist eine schnelle Implementierungsform einer diskreten Fourier-Transformation (DFT) mit Hilfe spezieller Strukturen, die hier nicht weiter besprochen werden sollen. Das DFT-Spektrum X(k) eines Signalsegmentes x(n), n=0, …, N-1 ergibt sich zu

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-jkn\frac{2\pi}{N}} = \sum_{n=0}^{N-1} x(n) \cdot w_N^{-kn}, \quad k = 0,...,N-1,$$

d.h. es werden spektrale Stützstellen des realen Spektrums im Abstand von

$$\Delta\Omega = \frac{2\pi}{N} = 2\pi\frac{\Delta f}{f_T}$$

berechnet.

4.2 Aufgabe: Bestimmen Sie die Ordnung N der FFT, so dass bei der Detektion der DTMF-Signale zwischen allen relevanten Stützstellen (d.h. FFT-Frequenzen, die den Frequenzen des DTMF-Signals möglichst genau entsprechen) mindestens drei (d.h. drei oder mehr) weitere Stützstellen berechnet werden.

Um die DFT berechnen zu können, sind immer *N* Signalwerte erforderlich, die in einem Buffer gespeichert werden müssen. Man spricht hier auch von „*frame processing*“ oder „*segment processing*“ anstelle von „*sample processing*“, so wie es in den vorherigen Übungen angewandt wurde. Um die Verarbeitung nach dem Einlesen von *N* Signalwerten in den Buffer nicht unterbrechen zu müssen und somit eine fortlaufende Verarbeitung zu gewährleisten, stellt man zwei Buffer zur Verfügung, in die abwechselnd eingelesen wird. Der Buffer, der gerade nicht zum Einlesen der Werte benötigt wird, wird

verarbeitet. Der gleiche Wechselbuffer-Mechanismus wird auch bei der Ausgabe der Werte verwendet, so dass sich schematisch eine Verarbeitung wie in Abb. 8 dargestellt ergibt.

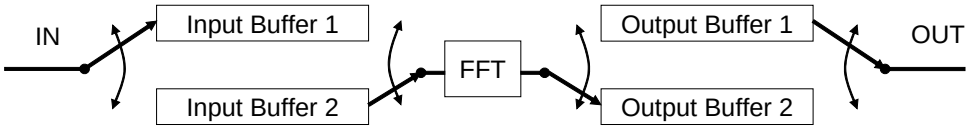

Abb. 8 – Segmentverarbeitung mit Wechselbuffer

4.3 Aufgabe: Erweitern Sie auf der Basis des ISR-Moduls aus der ersten Übung (Teil 1.3) das für die fünfte Übung im Netz vorhandene Template isr.c so, dass die Interrupt-Service-Routine eine segmentweise Verarbeitung der Daten ermöglicht. Die zu verarbeitenden Daten sollen unter dem Pointer pInFrame, die verarbeiteten Daten unter dem Pointer pOutFrame verfügbar sein. Alle Pointer sind dabei vom Datentyp fract16, der dem short Datentyp entspricht. Immer wenn ein neues Segment vollständig eingelesen wurde, soll ein Flag cNewFrame (vergleichbar mit dem Flag cNewSample bei der Sample-Verarbeitung) gesetzt werden. Das Signal wird via ADC2R eingelesen, das berechnete Spektrum wird über DAC1R ausgegeben. Zusätzlich soll an einem Signalausgang (hier DAC1L) ein kurzer Impuls erzeugt werden, der als stabiles Triggersignal für ein Oszilloskop verwendet werden kann.

Betragsquadratbildung

Zur Detektion der DTMF-Frequenzen wird das Betragsquadrat (Leistung) bei einer bestimmten Frequenz betrachtet. Hierzu müssen die Betragsquadrate der durch die FFT berechneten DFT-Werte gebildet werden. Die DFT-Werte selber liegen dabei als komplexe *Fractionals* mit dem Datentyp *complex_fract16* vor. Hinter diesem Datentyp verbirgt sich letztlich die folgende Struktur, in der die reellen Real- und Imaginärteile als *Fractionals* abgelegt sind:

```
typedef struct complex_fract16 { fract16 re, im;} complex_fract16;
```

4.4 Aufgabe: Erstellen Sie in dem für die vierte Übung im Netz vorhandenen Template tools.asm eine Funktion abs2_spec im Assemblercode zur Bestimmung des Betragsquadratsspektrums. Der Funktionsprototyp ist wie folgt vorgegeben:

```
void abs2_spec(fract16 *abs2_spectrum, complex_fract16 *spectrum, int ord);
```

Real- und Imaginärteil in der Struktur sind dabei immer nacheinander im Speicher abgelegt, d.h. der Pointer spectrum verweist auf den Realteil des DFT-Koeffizienten X(0), d.h. Re(X(0)); danach folgen im Speicher Im(X(0)), Re(X(1)), Im(X(1)), usw. Das Betragsquadratsspektrum soll unter der Adresse abs2_spectrum berechnet werden. Der Wert ord gibt die FFT-Ordnung an. In der Assembler-Funktion soll die Möglichkeit bestehen, die berechneten Werte um einen beliebigen Wert arithmetisch nach links zu shiften, um die Werte skalieren zu können.

Fensterung

Zur Reduzierung des „Leckeffektes“ soll die Möglichkeit einer Fensterung des zu transformierenden Signalsegments in dem Programm implementiert werden. Die Fensterung entspricht einer Multiplikation der Signalwerte mit einer vorgegebenen Fensterfunktion. Die Werte der Fensterfunktion sind im Speicher unter der Adresse window abgelegt. Auch hierbei handelt es sich um *Fractionals*.

4.5 Aufgabe: Erstellen Sie in dem für die vierte Übung im Netz vorhandenen Template tools.asm eine Funktion winmul im Assemblercode zur Durchführung der Fensterung. Der Funktionsprototyp ist wie folgt vorgegeben:

```
void winmul(fract16 *pInFrame, fract16 *window, int ord);
```

Die Signalwerte im fractional Zahlenformat liegen unter der Adresse pInFrame. Der Wert ord gibt die FFT-Ordnung an. In der Assembler-Funktion soll die Möglichkeit bestehen, die berechneten

(d.h. mit der Fensterfunktion multiplizierten) Werte um einen beliebigen Wert arithmetisch nach links zu shiften, um die Werte skalieren zu können.

5. Übung (DTMF-Detektor mit Goertzel-Algorithmus)

Der DTMF-Detektor aus der vorherigen Übung soll in dieser Übung auf algorithmisch andere Weise umgesetzt werden. Hierzu wird der Goertzel-Algorithmus[4] eingesetzt, der die Stützstellen der DFT auf andere Art und Weise berechnet und immer dann sinnvoll eingesetzt werden kann, wenn nur wenige Werte der DFT benötigt werden. Dies ist zum Beispiel bei dem DTMF-Detektor der Fall, wo von den N (bei uns N=2048) Werten der DFT nur theoretisch acht den DTMF-Frequenzen entsprechende Stützstellen gesucht sind. Werden, wie in unserer Implementierung mehrere (M=3) Stützstellen ausgewertet, so vergrößert sich der Rechenaufwand natürlich entsprechend.

Die DFT-Berechnung nach Goertzel ist ein einfaches Beispiel dafür, wie durch geschickte Anwendung eines anderen Algorithmus der Rechenaufwand und damit auch die Leistungsaufnahme eines digitalen Systems reduziert werden kann. Ohne solche algorithmischen Betrachtungen wären moderne digitale Signalverarbeitungssysteme nicht denkbar.

Goertzel-Algorithmus

Das DFT-Spektrum X(k) eines Signalsegmentes x(n), n=0, …, N-1 ergibt sich wie oben bereits beschrieben zu

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-jkn\frac{2\pi}{N}} = \sum_{n=0}^{N-1} x(n) \cdot \cos\left(kn\frac{2\pi}{N}\right) - j \cdot \sum_{n=0}^{N-1} x(n) \cdot \sin\left(kn\frac{2\pi}{N}\right), \quad k = 0,...,N-1$$

Beim Goertzel-Algorithmus werden die Werte X(k) mit Hilfe eines rekursiven Filters zweiter Ordnung mit Impulsantwort $h_k(n)$ berechnet, d.h. als Ergebnis einer Faltungssumme, die sich allgemein bekanntermaßen als

$$y_k(m) = \sum_{n=-\infty}^{\infty} x(n) \cdot h_k(m-n)$$

schreiben lässt und – vereinfacht für den Fall eines rechtseitigen Signals $x(n) \equiv 0$ für $n<0$ und eines kausalen Systems $h_k(n) \equiv 0$ für $n<0$ – als:

$$y_k(m) = \sum_{n=0}^{m} x(n) \cdot h_k(m-n)$$

Berechnet man den Wert des Filterausgangsignals $y_k(m)$ nur zum Zeitpunkt m=N, so lässt sich aus der Bedingung:

$$y_k(N-1) = \sum_{n=0}^{N-1} x(n) \cdot h_k(N-1-n) \overset{!}{=} X(k)$$

für die kausalen Filter $h_k(n)$ unter Verwendung der Sprungfunktion $\sigma(n)$ folgende komplexe Impulsantwort herleiten:

$$h_k(n) = \sigma(n) \cdot \left(\cos\left(k(n+1)\frac{2\pi}{N} \right) + j \sin\left(k(n+1)\frac{2\pi}{N} \right) \right) = h_k^{RE}(n) + j h_k^{IM}(n)$$

Jede komplexe Spektrallinie X(k) ergibt sich also als Wert des Ausgangssignals eines komplexen Filters mit Impulsantwort $h_k(n)$ nach N Abtastwerten. Um die Rechtseitigkeit des Signals x(n) zu gewährleisten, muss das Filter zum Zeitpunkt n=0 zurückgesetzt werden; das heißt, alle Speicher in dem rekursiven System müssen auf Null gesetzt werden, so als wäre das System bis zum Zeitpunkt n=0 nicht durch ein Eingangssignal angeregt worden.

[4] Algorithmus von 1958, beschrieben durch Gerald Goertzel (*1919 †2002).

5.1 Aufgabe: Bestimmen Sie die Übertragungsfunktion $H_k(z)$ für das komplexe Filter mit Impulsantwort $h_k(n)$

5.2 Geben Sie die Filterstruktur für das komplexe Filter in kanonischer Direktform II an.

Durchführung des Goertzel-Algorithmus

Die rekursive Berechnungsvorschrift in der kanonischen Direktform II kann (in MATLAB-Syntax) durch folgende Befehle realisiert werden:

```
a1 = cos(2*pi*k/N);
z1=0;
z2=0;
for i=1:N;
    temp = x(i) + 2*a1*z1 - z2;
    z2 = z1;
    z1 = temp;
end
```

In dieser Form ist die Implementierung insofern nachteilig, als dass zwei Befehle nur für die Aktualisierung der inneren Delayline mit den beiden Zustandsvariablen z1 und z2 notwendig sind. Dies lässt sich durch das teilweise Entrollen der Schleife (*loop unrolling*) mit einem *unroll factor* von zwei verbessern[5].

Werden zwei aufeinander folgende Schleifendurchläufe explizit formuliert:

```
for i=1:2:N;
    temp = x(i) + 2*a1*z1 - z2;
    z2 = z1;
    z1 = temp;
    temp = x(i+1) + 2*a1*z1 - z2;
    z2 = z1;
    z1 = temp;
end
```

so lassen sich die beiden Durchläufe wie folgt zusammenfassen:

```
for i=1:2:N;
    z2 = x(i)   + a1*z1 - z2;
    z1 = x(i+1) + a1*z2 - z1;
end
```

und die Aktualisierung der Delayline mit den Zustandsvariablen z2 und z1 entfällt, da diese nun direkt in der Berechnungsvorschrift enthalten ist.

Die Realisierung des Görtzel-Algorithmus in dieser Form finden Sie im Netz in der Assembler-Datei goertzel.asm als Teil des Projekts für den 5. Versuch. Der Funktionsprototyp für die realisierte Funktion lautet:

```
void goertzel(fract16 *x, complex_fract32* fft_goertzel, GOERTZELState *sgoertz);
```

Diese Funktion berechnet für ein Signal x vom Typ fract16 Real- und Imaginärteil der DFT-Koeffizienten im fract32-Format, welche im Array fft_goertzel abgelegt werden. Die Parametrisierung der Berechnung erfolgt über eine Struktur sgoertz vom Typ GOERTZELState:

[5] *Loop Unrolling* stellt eine allgemeine Optimierungsmethode zur Reduzierung der Laufzeit eines Programms auf Kosten des Programmspeicheraufwandes. Hierbei werden die Befehle innerhalb der Schleife explizit teilweise oder vollständig formuliert. Das Verhältnis der Anzahl der Schleifendurchläufe vor und nach dem „Abwickeln“ bezeichnet man als *unroll factor* („Abwickelfaktor“). Ein Geschwindigkeitsvorteil ergibt sich generell durch die geringere Anzahl der Überprüfung der Schleifenbedingungen, weitere Vorteile können sich durch eine mögliche Vektorisierung, verbesserten Speicherzugriff oder andere Optimierungen ergeben.

```
struct {
        fract16* coeff;
        unsigned long n_fft;
        unsigned long n_goertzel;
} GOERTZELState;
```

Hierin zeigt der Pointer coeff auf die zur Berechnung notwendigen Koeffizienten a1 und b1 im fract16-Format, n_fft ist die DFT-Ordnung und n_goertzel ist die Anzahl der zu berechnenden DFT-Koeffizienten.

5.3 Aufgabe: Schreiben Sie den zur Definition der Struktur notwendigen C-Code, indem Sie sich anhand des Assemblercodes überlegen, wie die Funktion goertzel() auf die Elemente der Struktur zugreift. Bestimmen Sie hierzu insbesondere alle Filterkoeffizienten der komplexen Goertzel-Filter, die für die acht DTMF-Frequenzen notwendig sind.