institute of telecommunications

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Digitale Signalverarbeitung (LVA 389.066, 389.067)

Musterprogramme für den Blackfin® ADSP-BF537

© G. Doblinger, März 2013

Gerhard.Doblinger@tuwien.ac.at

Inhaltsverzeichnis

1 Architektur des Blackfin® DSP

Wir verwenden im Seminar den BF537 DSP der Blackfin® Prozessorfamilie von Analog Devices, Inc. Die angegebenen Unterlagen im Literaturverzeichnis sind als PDF-Dokumente auf [1] zu finden. In diesem Abschnitt geben wir eine kurze Einführung in die Architektur der Blackfin® DSPs. Teile der folgenden Beschreibung sind meinem Buch über Signalprozessoren entnommen [2]. Eine ausführliche Darstellung der Funktionseinheiten des Blackfin® DSP ist im Hardware-Referenzmanual zu finden [3]. Die verwendete Hardware-Entwicklungsumgebung ist in [4] dokumentiert.

1.1 Kernarchitektur

Die Kernarchitektur der Blackfin® Prozessorfamilie ist sowohl für allgemeine Rechenaufgaben, als auch für DSP-Applikationen zugeschnitten. Das Architekturkonzept unterstützt außerdem durch Memory Management und verschiedene Betriebsmodi universelle Betriebsystem-Kernel, wie z.B. Linux. Im Vergleich zu den CPUs von PCs weisen die Blackfin® Prozessoren jedoch einen wesentlich geringeren Leistungsverbrauch auf, bieten diverse Schnittstellen bereits am Chip und haben spezielle DSP-Adressrechner.

Die Blackfin® Kernarchitektur ist in Abb. 1 schematisiert dargestellt. Es ist eine

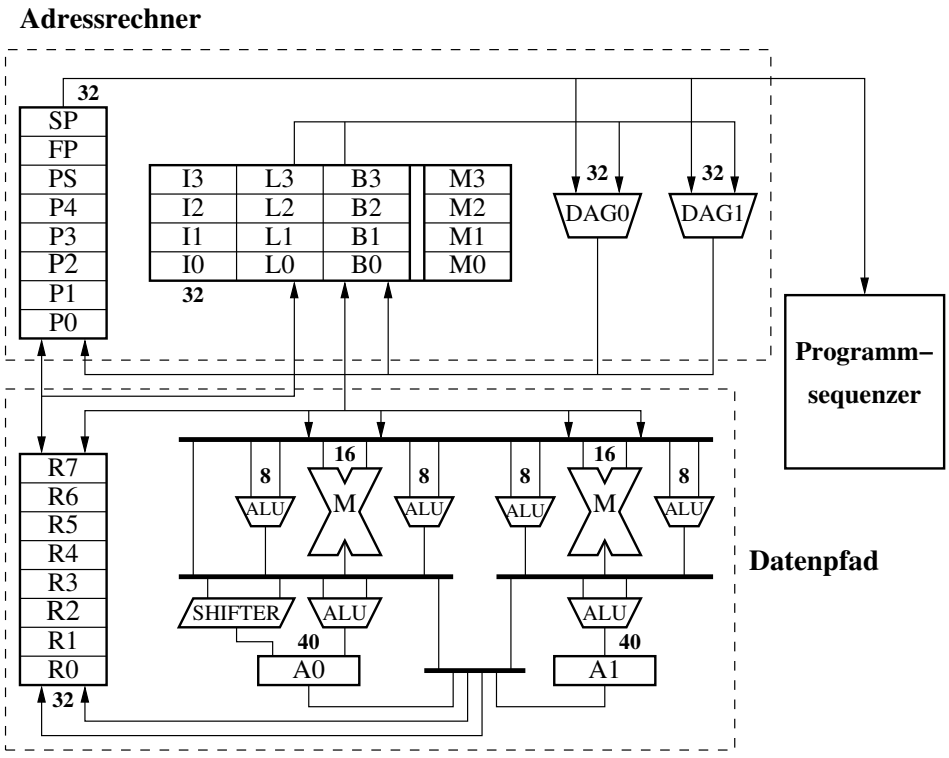

Abbildung 1: Kernarchitektur des Blackfin® DSP

registerbasierte Architektur, bei der der Dual-Core-Datenpfad, der Adressrechner und der Programmsequenzer simultan in einer 8-stufigen Pipeline betrieben werden. Die beiden

Recheneinheiten des Datenpfades arbeiten bei bestimmten Befehlen gekoppelt (SIMD-Betrieb). Der Blackfin® DSP ist kein Superskalarrechner, wie z.B. der SHARC™ DSP, da keine Mehrfachbefehle vorhanden sind, mit denen unterschiedliche Arithmetikoperationen pro Taktzyklus möglich sind.

Die Datenregister R0-R7 können 32 Bit Daten oder jeweils 2 16 Bit Daten pro Register speichern. Damit können maximal 4 16 Bit (oder 2 32 Bit) Operanden bzw. Ergebnisse transferiert werden. Die Multiplikationsergebnisse können in 2 40 Bit Registern A0 und A1 akkumuliert werden (MAC-Operationen).

Die Register des Adressrechners werden als Pointer (SP-, FP-, PS- und P-Register) bzw. zur Adressierung zyklischer Puffer (I-, L-, B-Register) verwendet. Die M-Register dienen der Adressmodifikation. Der Programmsequenzer unterstützt u.a. bedingte Befehlsausführung, Programmverzweigungen und die Verwaltung von Schleifen.

Die Recheneinheiten sind für verschiedene Festkommaformate ausgelegt und zwar Integer, Fraction und Block Floating Point Format. Gleitkommaoperationen sind als C-Funktionen in der Sotwareentwicklungsumgebung vorhanden. Der Blackfin® DSP kann nur bedingt als Gleitkomma-DSP eingesetzt werden, da die Rechenzeiten für Gleitkommaoperationen durch die Emulation mit 32 Bit Festkommaarithmetik zu lang sind. Die höchste Durchsatzrate erhält man für 16 Bit Daten (abgesehen von speziellen Videoverarbeitungsbefehlen für 8 Bit Daten). Operationen mit 32 Bit Festkommazahlen werden durch die beiden 32/40 Bit ALUs ebenfalls effizient ausgeführt. Abgesehen von einer speziellen 32 Bit $\times$ 32 Bit Integer-Multiplikation, muss die Multiplikation von 32 Bit Operanden in 16 Bit Teilmultiplikationen aufgespalten werden. Die Multiplikation mit doppelter Genauigkeit erfolgt also nicht in einem Taktzyklus.

Der Blackfin® DSP ist ein Prozessor mit variabler Befehlswortlänge (16, 32, und 64 Bit). Einzelbefehle haben entweder 16 Bit oder 32 Bit Wortlänge. Mehrfachbefehle (64 Bit) können maximal einen 32 Bit ALU/MAC Befehl enthalten mit Parallelausführung zweier 16 Bit Befehle der jeweils anderen Funktionseinheiten der Kernarchitektur von Abb. 1. Dabei gibt es Einschränkungen betreffend die Verwendung der Adressregister. So kann z.B. nur ein Transfer eines Registerinhalts in den Speicher parallel zu Registerladevorgängen erfolgen. Es sind jedoch 2 Ladevorgänge von Registern mit Speicherinhalten möglich oder 2 Modifikationen von I-Registern. Im Vergleich zu den SHARC™ DSPs können zu Programmverzweigungen (JUMP, CALL) keine Paralleloperationen ausgeführt werden. Auch gibt es bei Schleifen stärkere Restriktionen als bei anderen DSPs. Hier wirkt sich die mehrstufige Pipeline durch eine Erschwernis der Assemblerprogrammierung aus. Da jedoch durch die Pipeline wesentlich höhere Taktfrequenzen möglich sind, wird man trotz der Einschränkungen bei den Mehrfachbefehlen eine erheblich größere Datendurchsatzrate erzielen.

Befehlsworte unterschiedlicher Wortlänge werden lückenlos in den Programmspeicher gepackt. Damit wird eine vergleichsweise hohe Programmcodedichte erreicht. Bei der Befehlsausführung sorgt eine Alignment Unit dafür, dass pro Taktzyklus das gesamte Befehlswort zur Verfügung steht.

1.2 Betriebszustände des Blackfin® Prozessors

Im Gegensatz zu Standard-DSPs sind beim Blackfin® DSP verschiedene Betriebsmodi möglich, in denen verschiedene Ressourcen zur Verfügung stehen. Damit kann der Prozessor z.B. in einem geschützten Modus betrieben werden und ist somit für den Einsatz von Operating System Kernels (z.B. Embedded Linux) wesentlich besser geeignet, als ein DSP mit uneingeschränktem Zugang zu allen Prozessoreinheiten. Die drei Betriebszustände sind

- User Mode (niedrigste Priorität),
- Supervisor Mode,
- Emulator/Debug Mode (höchste Priorität).

Daneben gibt es noch Idle und Reset State in denen der Prozessorkern jedoch nicht arbeitet. Der User Mode wird für Anwenderprogramme verwendet, die keinen Zugriff auf Systemregister und geschützte Speicherbereiche benötigen. Der Prozessor wechselt vom User Mode in den Supervisor Mode durch einen Interrupt, ein Execption oder Emulator Event oder durch Reset. Nach einem Reset befindet sich der DSP automatisch im Supervisor Mode. Normale DSP-Programme, die keinen Betriebssystem-Kernel verwenden, wird man normalerweise im Supervisor Mode ausführen, Diese Programme weisen im Normalfall eine einfache Struktur auf: Reset, Initialisierung, Warten auf Interrupts (z.B. von ADC/DAC-Schnittstellen), Ausführen von Interrupt-Serviceroutinen, Rückkehr in die Warteschleife.

1.3 Programmsequenzer

Neben den üblichen Aufgaben der Programmsteuerung (Programmsprünge, Unterprogrammaufrufe, Schleifen, Interrupts) verwaltet der Programmsequenzer die 8-stufige Befehls-Pipeline, die mit dem Prozessortakt (mindestens 300 MHz Taktfrequenz) betrieben wird. Die einzelnen Stufen der Pipeline sind in Tabelle 1 angegeben. Bei der Steuerung der Pipeline wird sichergestellt, dass bei der Abarbeitung der Befehle Datenzugriffskonflikte automatisch behandelt werden (Interlocked Pipeline). Im Fall von Konflikten oder ungültigen Befehlen (zufolge von Programmsprüngen) fügt der Progammsequenzer “Leerbefehle” (Stalls) ein, um die richtige Programmausführung sicherzustellen. Mit dem Pipeline Viewer der Programmierumgebung werden Stalls angezeigt und können evt. durch eine Umordnung des Programms eliminiert werden. Zum Unterschied zu anderen DSPs erfolgt die Einfügung notwendiger Stalls während der Laufzeit und nicht beim Compilieren des Programms.

Bedingte Sprungbefehle werden durch ein Condition Code Flag gesteuert, mit dessen Hilfe auch eine Branch Prediction vorgenommen wird. Bei richtiger Vorhersage einer Programmverzweigung reduziert sich die Verzögerungszeit von 6 auf 3 Taktzyklen. Ist kein Sprung notwendig, so ergibt sich bei richtiger Vorhersage auch keine Verzögerung, da der Zustand der Pipeline nicht verändert werden muss.

Stufe	Beschreibung
Instruction Fetch 1	Beginn des Zugriffs auf Befehlsspeicher
Instruction Fetch 2	Beenden des Speicherzugriffs
Instruction Decode	Beginn der Adressdecodierung, Zugriff auf Pointer Register
Address Calculation	Adressberechnung für Daten und für Programmsprünge
Execute 1	Beginn des Zugriffs auf Datenspeicher, Register lesen
Execute 2	Beenden des Speicherzugriffs, Beginn der Ausführung von Dual Cycle Instructions
Execute 3	Ausführen von Single Cycle Instructions, Transfer der Ergebnisse in die Akkumulatoren A0, A1
Write Back	Speichern der Ergebnisse in die Register Files

Tabelle 1: Stufen der Befehls-Pipeline

Für die Ausführung von Schleifen ist im Programmsequenzer Hardware vorgesehen, so dass für diese Schleifen keine zusätzlichen Register bzw. Prüfbefehle notwendig sind. Um das Instruction Fetch während der Schleifenverarbeitung zu beschleunigen, gibt es einen Instruction Loop Buffer mit 4 Speicherplätzen. Damit werden z.B. bei Schleifen mit maximal 4 Befehlen die Befehle lokal in diesem Buffer gespeichert.

Der Programmsequenzer des Blackfin® DSP hat einen sehr leistungsfähigen Event Controller, der die Verwaltung von Events vornimmt. Zu den Events zählen Ereignisse, wie z.B. Interrupts oder Ausnahmezustände (Exceptions), die den Programmfluss softwaregesteuert oder hardwaregesteuert durch periphere Einheiten unterbrechen. Die Struktur des Event Controller ist zweistufig: Der System Interrupt Controller ordnet die verschiedenen Interrupts der Peripherie den allgemeinen Interrupts des Prozessorkerns zu. Die Zuordnung der Interrupts ist programmierbar und kann damit an verschiedene Hardware-Konfigurationen angepasst werden.

1.4 Speicher- und Busarchitektur des Blackfin® DSP

Im Prinzip besitzt der Blackfin® DSP eine modifizierte Harvard-Architektur. Die internen Speicher sind jedoch hierarchisch in Level 1 (L1) Memory und Level 2 (L2) Memory gegliedert. Der L1 Speicher kann neben der Funktion als normales SDRAM auch zum Teil als Cache konfiguriert werden. Die Speicherarchitektur ist in Abb. 2 schematisiert dargestellt. Der L1 Speicherblock hat eine modifizierte Harvard-Architektur und ermöglicht bis zu drei Speicherzugriffe (1 64 Bit Befehl, 2 32 Bit Daten) des Prozessors pro Taktzyklus. Bei direktem Speicherzugriff (DMA) können sowohl der Prozessor als auch der DMA Controller simultan auf den L1 Speicher zugreifen. Damit wird der Pro-

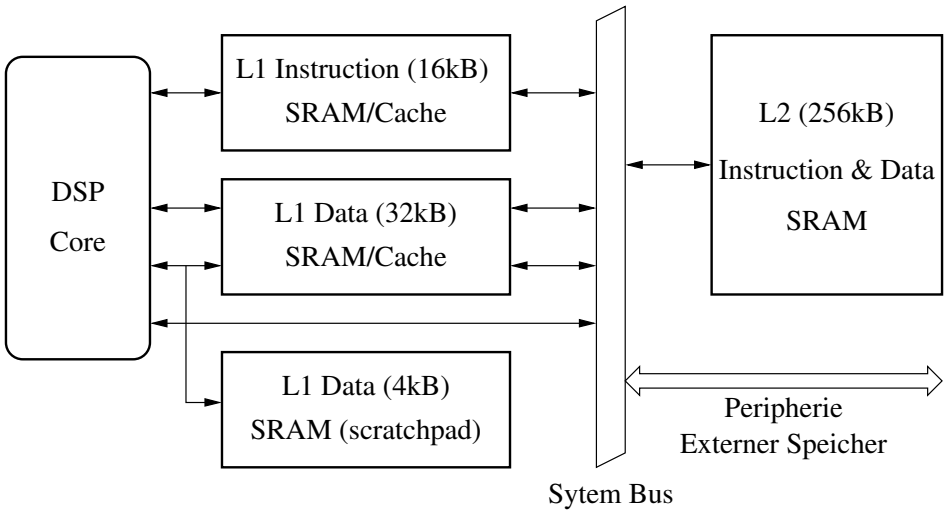

Abbildung 2: Speicherarchitektur des Blackfin® DSP

zessor durch das Transferieren von externen Daten nicht unterbrochen. Für zeitkritische DSP-Programme wird man den L1 Speicher normalerweise als SRAM verwenden und auf die Cache-Funktion verzichten. Eine Besonderheit ist das Scratchpad Memory des L1 Speichers, das z.B. als schneller Stack-Speicher eingesetzt werden kann. Dadurch kann die Verarbeitung von C-Funktionen erheblich beschleunigt werden.

Der 256 KB Speicherumfang des L2 Speichers ist in 8 Speicherbänke zu 32 kB organisiert. Unterschiedliche Speicherbänke können simultan von Prozessor und Peripherie adressiert werden. Die Zugriffszeit hängt von der Konfiguration des L1 Speichers ab. Ist dieser als Cache konfiguriert, dann werden jeweils 4×64 Bit Befehle aus dem L2 Speicher gelesen und in der Pipeline des Prozessors verarbeitet. Nach 7 Taktzyklen kann der erste Befehl ausgeführt werden, danach folgen die anderen Befehle im Prozessortakt, da das Nachladen des Cache und die Befehlsausführung parallel erfolgt. Ohne Cache-Funktion des L1 Speichers benötigt die Befehlsausführung aus dem L2 Speicher 7 Taktzyklen für jeden Befehl.

Der 4 GB Adressraum des Blackfin® DSP wird in Speicherseiten unterteilt. Als Seitengrößen können 1 kB, 4 kB, 1 MB und 4 MB verwendet werden. Jede Seite kann unterschiedliche Attribute haben wie z.B. Wait States oder Zugriffsrechte. Zum Schutz von Speicherbereichen wird eine Memory Management Unit (MMU) verwendet, mit der die einzelnen Speicherseiten oder Cache-Bereiche für einzelne Prozessor Tasks geschützt werden können. Der Betrieb der MMU kostet keine Extrataktzyklen. Die MMU kommt vorzugsweise dann zum Einsatz, wenn ein Betriebssystem auf dem Blackfin® DSP läuft und bei einem Task Switch der neue Processor Task nicht den gesamten Speicherbereich des alten Task übernehmen darf.

In Abb. 2 ist ersichtlich, dass der Blackfin® DSP einen zentralen Systembus aufweist, an dem DSP-Kernarchitektur, interne Speicher und diverse periphere Einheiten angeschlossen sind. Auf die vielfältige Peripherie kann hier nicht eingegangen werden. Sie umfasst Komponenten, wie Power Management, Timer, Real-time Clock, verschiedene se-

rielle Ports, USB- und PCI-Interface, Emulator und Test Port. Damit können Blackfin® DSPs mit geringem Hardwareaufwand z.B. in Audio/Videokarten für PCs eingesetzt werden. Weitere Einsatzgebiete in Massenprodukten sind Home Entertainment Equipment auf DVD-Basis und Digitalkameras.

2 Programmierung des Blackfin® DSP

In diesem Abschnitt fassen wir kurz die Syntaxmerkmale zusammen. Die ausführliche Beschreibung der Blackfin® Programmierung ist in [5] zu finden. Da wir den DSP bis auf wenige zeitkritische Funktionen in C programmieren, verwenden wir die Funktionen der C-Library und der DSP-Library in [6].

Die Assemblersprache des Blackfin® DSP verwendet eine algebraische Syntax, ähnlich jener der Programmiersprache C. Da der Blackfin® DSP kein Superskalarrechner ist, sind die meisten Anweisungen Einzelbefehle, die in der 8-stufigen Pipeline verarbeitet werden. Es ist jedoch möglich, bestimmte Befehle zu kombinieren, so dass pro Taktzyklus mehrere Resultate vorliegen. Diese "Mehrfachbefehle" (in der Syntax sind die Einzelbefehle durch `||`-Striche getrennt) werden jedoch wegen der Pipeline nicht in einem einzigen Taktzyklus verarbeitet. Die Ausnahme sind die SIMD-Erweiterungen (beim Blackfin® DSP etwas verwirrend auch Vektorbefehle genannt), die bestimmte Befehle auf verschiedene Datenregister anwenden. Für den Programmierer ist wichtig, dass bei den "Mehrfachbefehlen" keine Datenkonflikte auftreten, da diese Data Hazards wie bereits erwähnt vom Prozessor durch Einfügen von Stall Cycles automatisch verhindert werden.

Die Befehle sind unterteilt in Transferbefehle (Register Load/Store), Programmsteuerbefehle (JUMP, CALL), Logische Verknüpfungen (AND, OR, XOR, NOT), Bit-Operationen (z.B. Bit Clear, Bit Set), Shift Operations (arithmetisch und logisch) und Arithmetikoperationen ($+$, $-$, $*$). Zusätzlich gibt es Sonderbefehle z.B. für Event Management, Cache Control und Video Pixel Operations.

Alle Operationen erfolgen mit Ausnahme der beiden 40 Bit Akkumulatoren A0 und A1 über 32 Bit Register. Da der Rechner 16 Bit und 32 Bit Daten verarbeiten kann, muss dies bei der Befehlssyntax berücksichtigt werden. Bei 32 Bit Daten sieht z.B. die indirekte Adressierung folgendermaßen aus:

```
R0 = [I1++];        // load register R0
                    // [ ] stands for indirekt addressing
                    //  ++ means postincrement pointer I1
[I0] = R1;          // store register R1
```

Bei 16 Bit Daten gibt es mehrere Möglichkeiten:

```
R0 = W[I1++] (X);  // load 16 Bit word in lower half of R0
                   // extend sign in upper half of R0
R0.L = W[I0];      // load 16 Bit word in lower half of R0
R0.H = W[I1];      // load 16 Bit word in upper half of R0
W[I2--] = R1.L;    // store lower half of R1
W[I2] = R1.H;      // store upper half of R1
```

Als Beispiel für die Assemblersprache des Blackfin® DSP soll der Kern einer FIR-Filterroutine betrachtet werden, bei dem für einen festen Zeitindex n die Faltungssumme

$$y[n] = \sum_{k=0}^{N-1} h[k]\, x[n-k], \quad n \geq N-1$$

berechnet wird. Dabei wird vorausgesetzt, dass $h[k]$ und $x[n-k]$ in aufeinander folgenden 16 Bit Speicherplätzen (d.h. word aligned) gespeichert sind.

In der folgenden C-Funktion werden Fractions als Festkommavariable verwendet. Die Built-in Functions (auch Compiler Intrinsics genannt) ermöglichen die Verwendung von Hardwareeigenschaften des DSP, die in C normalerweise nicht verfügbar sind. Die Deklaration `__builtin_aligned` stellt sicher, dass die Pointer auf die Arrays word aligned sind und damit ein effizienter Datentransfer möglich ist (2×16 Bit Worte auf einmal durch ein 32 Bit Wort übertragen). Dieses Alignment ist speziell bei lokalen Variablen notwendig, die in Funktionen verwendet werden. Die Alignment-Deklaration ermöglicht dem Compiler bei der Optimierung den Einsatz von Befehlen, die er sonst aus Sicherheitsgründen nicht verwenden würde. Allerdings muss der Programmierer beim Anlegen der Variablen gewährleisten, dass diese tatsächlich word aligned sind.

```
// C function of FIR filter routine (for use with compiler optimizer)
#include <fract.h>

frac32 fir_sum(frac16 *x, frac16 *h, int nfilt, int n)
{
   int     k;
   fract32 sum = 0;

   __builtin_aligned(x,4);
   __builtin_aligned(h,4);

   for (k = 0, j = n; k < nfilt; k++, j--)
      sum = add_fr1x32(sum, mult_fr1x32(h[k]*x[j]));

   return sum;
}
```

Die Summenbildung in der Schleife des C-Programms erfolgt mit Built-in Functions für Multiplikation und Addition, wobei mit `_frlx32` 32 Bit Ergebnisse berechnet werden. Bei Verwendung von arithmetischen Built-in Functions versucht der Compiler bei der Optimierung auch den Einsatz von Parallelbefehlen, um z.B. Datentransfers parallel zu den Rechenoperationen durchzuführen (wie bei den Assemblerbeispielen des FIR-Filterprogramms).

3 Musterprogramm für einfache DSP-Anwendungen

Mit dem folgenden Musterprogramm werden die Abtastwerte einzeln, also ohne Blockverarbeitung, vom ADC interrupt-gesteuert mit DMA direkt an den Prozessor geleitet. Im Hauptprogramm `main()` wartet der DSP nach der Initialisierungsphase in einer while-Schleife auf neue Abtastwerte und kehrt nach deren Verarbeitung mit der Funktion `Process_Data()` wieder in die Warteschleife zurück.

```
1  /* main function of seminar prototype program
2
3    based on BF537 C Talkthrough I2S by Analog Devices, Inc.
4    modified by G.Doblinger, TU-Wien to allow processing of
5    fract16, and/or fract variables
6    (see also new version of Process_data.c)
7
8  mod. G.Doblinger, TU-Wien, 3-2008 (variable DAC_init), 3-2013
9  */
10
11 #include "Seminar.h"
12
13 // left input data from AD1871
14 #pragma align 4
15 int iChannel0LeftIn;
16 // right input data from AD1871
17 #pragma align 4
18 int iChannel0RightIn;
19 // left ouput data for AD1854
20 #pragma align 4
21 int iChannel0LeftOut;
22 // right ouput data for AD1854
23 #pragma align 4
24 int iChannel0RightOut;
25 // SPORT0 DMA transmit buffer
26 int iTxBuffer1[2];
27 // SPORT0 DMA receive buffer
28 int iRxBuffer1[2];
29
30 short DAC_init;
31
32 void main(void)
33 {
34     DAC_init = 1;        // avoids spurious output during setup of ADC/DAC
35
36     Init_Flags();
37     Audio_Reset();
38     Init_Sport0();
39     Init_DMA();
40     Init_Interrupts();
41     Enable_DMA_Sport0();
42
43     /* insert initialization of DSP algorithm here (e.g. set buffers) */
44
45     DAC_init = 0;        // enable processing of data in Process_Data()
```

```
46
47      while(1);                   // wait for interrupt to call Process_Data()
48  }
```

Im folgenden Programmsegment werden die 24 Bit Abtastwerte des linken und rechten Analogeingangs als 16 Bit Daten in `x16` (Struktur mit zwei 16 Bit Daten `x16.l` und `x16.r`) gespeichert. Je nach Aufgabenstellung werden diese Daten dann verarbeitet und als Variable `y16` an den DAC weitergeleitet. Im Beispielprogramm wird ein Monosignal erzeugt und direkt an beide Analogausgänge gesendet. Die Abtastfrequenz ist fix auf 48 kHz eingestellt und kann nicht verändert werden.

```
1   /* signal processing function
2   G. Doblinger, TU-Wien, 3-2008, 3-2013
3
4   NOTE: sampling frequency is 48 kHz,
5   NOTE: input voltage must not exceed 0.8 Vpp
6   */
7
8   #include "Seminar.h"
9
10  /* use unbiased rounding with fract data types */
11
12  #pragma FX_ROUNDING_MODE UNBIASED
13
14  // stereo input and output signals x.l = left channel, x.r right channel
15
16  volatile stereo_fract16 x16, y16;
17  volatile stereo_fract   xf, yf;
18  extern short            DAC_init;
19
20  void Process_Data(void)
21  {
22     fract   c;
23     fract16 c16;
24
25     if (DAC_init == 1)       // mute output DAC during setup
26        {
27           iChannel0LeftOut = 0;
28           iChannel0RightOut = 0;
29           return;
30        }
31
32     /* get 16 bit (short) input signals from high words */
33
34     x16.l = high_of_fr2x16(iChannel0LeftIn);
35     x16.r = high_of_fr2x16(iChannel0RightIn);
36
37     /* insert DSP algorithm here */
38
39     /* example with native fraction data types
40     Note: fract16 is typedef to short, fract is 1.15 fraction! */
41
42     xf.l = rbits(x16.l);        // convert fract16 to fract
```

```
43      xf.r = rbits(x16.r);
44
45      c = 0.5r;                    // c as 1.15 fraction
46      yf.l = c*(xf.l+xf.r);        // monaural signal
47      yf.r = yf.l;
48
49      y16.l = bitsr(yf.l);         // convert fract to fract16
50      y16.r = bitsr(yf.r);
51
52      /* example with frac16 data types
53         Note: native fixed-point arithmetic cannot be used with fract16 data
54         (uncomment the following section) */
55
56  /*
57      c16 = float_to_fr16(0.5f); // convert float to fract16
58      y16.l = add_fr1x16(x16.l,x16.r);
59      y16.l = multr_fr1x16(c16,y16.l);
60      y16.r = y16.l;
61  */
62
63      /* send 16 bit output signals (shifted to high words) to DAC */
64
65      iChannel0LeftOut = (int) (y16.l << 16);
66      iChannel0RightOut = (int) (y16.r << 16);
67  }
```

Im Beispielprogramm können sowohl Daten vom Typ `fract` als auch vom Typ `fract16` verwendet werden. Der Datentyp `fract` repräsentiert 1.15 (1 Vorzeichenbit, 15 Nachkommabits) Zweierkomplementdaten. Mit diesen Datentyp kann die Festkommaarithmetik des DSP direkt verwendet werden (siehe Beispiel). Bei der Verwendung von `fract16` müssen die Built-in Functions verwendet werden (siehe auskommentiertes Programmsegment). Trotz der damit verbundenen Unübersichtlichkeit wird für den einzufügenden Signalverarbeitungsalgorithmus die Verwendung von `fract16` Daten empfohlen, da alle optimierten Funktionen der DSP-Library ohne Datentypumwandlung verwendet werden können [6]. In Zukunft werde ich die Musterprogramme aber auf `fract` Daten umstellen. Gleitkommadaten und Gleitkommafunktionen der DSP-Library sollten nicht verwendet werden, da die Emulation der Gleitkommaarithmetik mit einem Festkommarechner für die Echtzeitverarbeitung der Signale zu langsam ist.

Als Beispiel für die Verwendung des Musterprogramms habe ich ein FIR-Filterprogramm geschrieben, das durch die Verwendung der Built-in Functions sehr effizient ist. Das FIR-Filterprogramm ist auf den Seminarrechnern installiert und zeigt auch die Programmierung von zyklischen Puffern in C und weitere Feinheiten.

4 Musterprogramm für die FFT-Filterbank

Das Musterprogramm für die FFT-Filterbank entspricht jenem des ADSP-21065L (siehe dazu die Seminarunterlage [7]). Es erfolgt kein Blocktransfer von Abtastwerten mit DMA wie beim ADSP-21369 Filterbankprogramm [8]. Die Initialisierungsfunktionen müßten

dazu komplett geändert werden. Der Nachteil ist die laufende Unterbrechung des DSP-Programms durch Interrupts des ADC. Da der verwendete Blackfin® DSP jedoch vergleichsweise schnell arbeitet, ist das für die vorgesehenen Anwendung nicht weiter problematisch.

Im folgenden Initialisierungsprogramm werden alle benötigten Speicher mit Anfangswerten versehen, mit Startadressen die ein Vielfaches von 4 sind. Das wird für die Optimierungsmethoden des C-Compilers benötigt. Alle Puffer verwenden 16 Bit Fraction-Variable, da die meisten Funktionen der DSP-Library für diese Datentypen optimiert sind.

```
1  #include "fib.h"
2
3  /* initialization file for FFT filterbank project
4
5     G. Doblinger, TU-Wien, 3-2006 (BF535), 3-2008 (BF537), 3-2013
6  */
7
8  /* global input/output buffers */
9
10 #pragma align 4
11 stereo_fract16 adc[N];
12 #pragma align 4
13 stereo_fract16 dac[M];
14
15 /* global filterbank variables */
16
17 #pragma align 4
18 fract16 hann[N];                    // time window function
19 #pragma align 4
20 complex_fract16 x[N], y[N];         // cyclic FFT input and overlap-add
       buffer
21 #pragma align 4
22 complex_fract16 w[N/2];             // fft twiddle factors
23 unsigned int  i_adc, i_ybuf;       // index for cyclic ADC and overlap-add
       buffer
24 unsigned int  i_dac;               // DAC buffer index
25
26 /* function to initialze DSP algorithm variables
27    (called in function main, before processing of samples) */
28
29 void Init_DSP_Program(void)
30
31 {
32
33 short n;
34
35    __builtin_aligned(x, 4);
36    __builtin_aligned(y, 4);
37    __builtin_aligned(hann, 4);
38    __builtin_aligned(w, 4);
39    __builtin_aligned(adc, 4);
40    __builtin_aligned(dac, 4);
41
```

```
42      /* initialize ADC/DAC buffers */
43
44      for (n = 0; n < N; n++)
45         adc[n].left = adc[n].right = 0;
46      for (n = 0; n < M; n++)
47         dac[n].left = dac[n].right = 0;
48
49      /* initialize FFT variables */
50
51      for (n = 0; n < N; n++)
52        x[n].re = x[n].im = y[n].re = y[n].im = 0;
53
54      twidfftrad2_fr16(w, N);
55      gen_hanning_fr16(hann,1,N);
56
57      /* initizialize buffer indices */
58
59      i_adc = 0;
60      i_dac = 0;
61      i_ybuf = 0;
62  }
```

Im Hauptprogramm `main()` werden zyklische Puffer für die Werte des ADC, des DAC und des Overlap-Add Puffers (der Synthesestufe der FFT-Filterbank, siehe [7]) verwendet. Diese Pufferspeicher werden alle M Abtastwerte aktualisiert (wenn in der while-Schleife `i_dac` gleich M ist). Vor dem Aufruf der eigentlichen Filterbankroutine `filterbank()` werden der ADC-Puffer in den FFT-Eingangspuffer und der FFT-Ausgangspuffer in den DAC-Puffer kopiert. Das erfolgt mit der Funktion `cp_cyc2lin()`, die in Assemblersprache entwickelt wurde.

```
1   /* overlap-add FFT filterbank implementation for BF 537 board
2      (ADC/DAC processing based on talk_through program)
3
4   G. Doblinger, TU-Wien, 3-2006 (BF535), 3-2008 (BF537), 3-2013
5   */
6
7   #include "fib.h"
8
9   // left input data from AD1871
10  #pragma align 4
11  int iChannel0LeftIn;
12  // right input data from AD1871
13  #pragma align 4
14  int iChannel0RightIn;
15  // left ouput data for AD1854
16  #pragma align 4
17  int iChannel0LeftOut;
18  // right ouput data for AD1854
19  #pragma align 4
20  int iChannel0RightOut;
21  // SPORT0 DMA transmit buffer
22  int iTxBuffer1[2];
```

```
23 // SPORT0 DMA receive buffer
24 int iRxBuffer1[2];
25
26 short no_interrupt;          // indicates termination of setup phase
27
28 void main(void)
29 {
30   short  m, n;
31
32    __builtin_aligned(x, 4);
33    __builtin_aligned(y, 4);
34    __builtin_aligned(adc, 4);
35    __builtin_aligned(dac, 4);
36
37    no_interrupt = 1;          // disable ADC/DAC interrupts during setup
         phase
38     Init_Flags();
39     Audio_Reset();
40     Init_Sport0();
41     Init_DMA();
42     Init_Interrupts();
43     Enable_DMA_Sport0();
44     Init_DSP_Program();
45          no_interrupt = 0;  // enable ADC/DAC interrupts
46
47     while(1)
48      {
49        if (i_dac == M)       // M samples already send to DAC */
50          {
51            i_dac = 0;        // reset DAC buffer index
52
53            /* copy N samples from ADC buffers to FFT input buffers
54               NOTE: both copy sections must not be interrupted
55               (therefore, assembly functions are used) */
56
57            cp_cyc2lin_buffer((int *) x, (int *) adc, N, N, i_adc);
58
59            /* copy M samples of previously updated cyclic buffers to DAC
                 buffers */
60
61            cp_cyc2lin_buffer((int *) dac, (int *) y, M, N, i_ybuf);
62
63           /* update cyclic buffer index by M samples modulo N */
64
65            i_ybuf = (i_ybuf+M) % N;
66
67            /* call FFT filterbank function */
68
69            filterbank(x, y);
70
71          }
72      }
73 }
```

Das Speichern der Abtastwerte in den ADC-Puffer bzw. die Ausgabe der verarbeiteten Werte in den DAC-Puffer erfolgt ähnlich wie beim einfachen Musterprogramm mit der Funktion `Process_Data()`. Diese Funktion wird durch den SPORT0-Interrupt einmal pro Abtastintervall aufgerufen und unterbricht laufend die Verarbeitung der Signale in der FFT-Filterbank.

```
1  /* signal processing function
2  G. Doblinger, TU-Wien, 3-2006 (BF535), 3-2008 (BF537)
3
4  NOTE: sampling frequency is 48 kHz
5  NOTE: input voltage must not exceed 0.8 Vpp
6  */
7
8  #include "fib.h"
9
10 void Process_Data(void)
11 {
12
13     if (no_interrupt == 1)       // mute output DAC
14         {
15             iChannel0LeftOut = 0;
16             iChannel0RightOut = 0;
17             return;
18         }
19
20     /* 16 bit (short) input signals */
21
22     adc[i_adc].left = high_of_fr2x16(iChannel0LeftIn);
23     adc[i_adc++].right = high_of_fr2x16(iChannel0RightIn);
24     i_adc %= N;
25
26     /* send 16 bit output signal to DAC */
27
28     iChannel0LeftOut = (int) ((dac[i_dac].left) << 16);
29     iChannel0RightOut = (int) ((dac[i_dac++].right) << 16);
30
31 }
```

Die Signalverarbeitung in der Filterbank ist im folgenden Teil des Musterprogramms angegeben. Die Funktionsweise der FFT-Filterbank ist in [7] beschrieben. Die Verarbeitung des komplexwertigen Eingangssignalpuffers erfolgt in gleicher Weise wie im Musterprogramm für den ADSP-21369. In [8] ist genau erklärt, wie die Spektren des linken und rechten Stereosignals aus dem Spektrum des komplexwertigen Signals berechnet werden, um danach getrennt mit der Funktion `modify_spectra()` verarbeitet zu werden. Im Musterprogramm wird diese Funktion an die entsprechende Aufgabe angepasst. Dabei ist die FFT des aktuellen Eingangsblocks in `X[k].l` (linker Kanal) und `X[k].r` (rechter Kanal) gespeichert. Der Frequenzindex umfasst $k = 1, 2, \ldots, N-1$, wobei die Spektren bei $k = 0$ auf Null gesetzt sind (kein Gleichanteil). Im Gegensatz zum ADSP-21369 Musterprogramm werden hier eigene Arrays für die FFT des linken und des rechten Kanals verwendet. Im ADSP-21369 Musterprogramm werden diese im FFT-Puffer gespeichert,

was Speicherplatz spart, aber die Speicherung des Spektrums des rechten Kanals mit umgekehrter Indexreihenfolge bedingt.

```
1  /* filterbank.c
2
3     overlap-add filterbank function
4     (loop cylces for optimization enabled, taken from filterbank.s)
5
6      G. Doblinger, TU-Wien, 3-2006 (BF535), 3-2008 (BF537), 3-2013
7  */
8
9  #include "fib.h"
10
11 #pragma align 4
12 static complex_fract16  Xl[N], Xr[N];
13
14 void filterbank(complex_fract16 *x, complex_fract16 *y)
15
16 {
17   short       n;
18   int         be1, be2;
19   fract16     tr, ti;
20   fract2x16   t1, t2, t3;
21
22    __builtin_aligned(x, 4);
23    __builtin_aligned(y, 4);
24    __builtin_aligned(hann, 4);
25    __builtin_aligned(w, 4);
26    __builtin_aligned(Xl, 4);
27    __builtin_aligned(Xr, 4);
28
29   /* apply windowing of signal frame */
30
31   for (n = 0; n < N; n++)                        // 10 cycles
32     {
33       x[n].re = multr_fr1x16(x[n].re, hann[n]);
34       x[n].im = multr_fr1x16(x[n].im, hann[n]);
35     }
36
37   /* complex inplace FFT, Note: spectra are scaled to [-1,1) */
38
39   cfft_fr16(x,x,w,1,N,&be1,2);
40
41   /* separate spectra of left and right channel
42      Note: spectra at f = 0 set to zero (no DC component),
43            in addition, factor 1/2 is omitted */
44
45   Xl[0].re = 0;
46   Xl[0].im = 0;
47   Xr[0].re = 0;
48   Xr[0].im = 0;
49   for (n = 1; n < N; n++)                         // 12 cycles
50     {
51       t1 = compose_fr2x16(x[n].im, x[n].im); // t1_h = x[n].im,
```

```
52                                                  // t1_l = x[n].im
53          t2 = compose_fr2x16(x[N-n].im, x[N-n].im);
54          t3 = add_as_fr2x16(t1, t2);             // t3_h = t1_h+t2_h,
55                                                  // t3_l = t1_l-t2_l
56          Xr[n].re = low_of_2x16(t3);
57          Xl[n].im = high_of_2x16(t3);
58          t1 = compose_fr2x16(x[N-n].re, x[N-n].re);
59          t2 = compose_fr2x16(x[n].re, x[n].re);
60          t3 = add_as_fr2x16(t1, t2);
61          Xr[n].im = low_of_2x16(t3);
62          Xl[n].re = high_of_2x16(t3);
63        }
64
65      /* modify spectra Xl, Xr */
66
67      modify_spectra(Xl, Xr);
68
69      /* combine modified spectra to one spectrum for inverse FFT,
70         x = 2(Xl +jXr) */
71
72      for (n = 0; n < N; n++)                     // 4 cycles
73        {
74          t1 = compose_fr2x16(Xl[n].im, Xl[n].re);
75          t2 = compose_fr2x16(Xr[n].re, Xr[n].im);
76          t3 = add_as_fr2x16(t1, t2);
77          x[n].re = low_of_2x16(t3);
78          x[n].im = high_of_2x16(t3);
79        }
80
81      /* complex inverse inplace FFT (left channel = x.re,
82         right channel = x.im) */
83
84      ifft_fr16(x,x,w,1,N,&be2,2);
85
86      /* carry out overlap-add of successive IFFT outputs
87         (overlap-add is done with a cyclic buffer y in order to store only N
               samples)
88         (note that frame length N is an integer multiple of frame hop size M)
               */
89
90      be2 += be1-LOG2N;                           // compensate block FP scaling
91                                                  // and apply 1/N of IFFT
92                                                  // and 1/2 to avoid overflow in ola
93      for (n = 0; n < N-M; n++)                   // overlap-add N-M samples,
94                                                  // 27 cycles
95        {
96          tr = shl_fr1x16(x[n].re, be2);          // compensate BFP scaling
97          ti = shl_fr1x16(x[n].im, be2);
98          y[i_ybuf].re = round_fr1x32(add_fr1x32((y[i_ybuf].re << 16),
                mult_fr1x32(tr, hann[n])));
99          y[i_ybuf].im = round_fr1x32(add_fr1x32((y[i_ybuf].im << 16),
                mult_fr1x32(ti, hann[n])));
100         i_ybuf++;
101         i_ybuf %= N;
```

```
102       }
103     for (n = N-M; n < N; n++)              // copy last M samples, 19 cycles
104       {
105         tr = shl_fr1x16(x[n].re, be2);     // compensate BFP scaling
106         ti = shl_fr1x16(x[n].im, be2);
107         y[i_ybuf].re = multr_fr1x16(tr, hann[n]);
108         y[i_ybuf].im = multr_fr1x16(ti, hann[n]);
109         i_ybuf++;
110         i_ybuf %= N;
111       }
112 }
113
114 /* ------------------------------------------------------------------ */
115
116 /* function for modification of short-time spectra of left and right
       channel */
117
118 void modify_spectra(complex_fract16 *X_left, complex_fract16 *X_right)
119
120 /* as an example, multiply spectra with gain = 0.5 */
121
122 {
123    short       n;
124    fract2x16   t1, t2;
125    fract16     gain = float_to_fr16(0.5f);
126
127    t1 = compose_fr2x16(gain, gain);
128    for (n = 1; n < N; n++)
129       {
130          t2 = compose_fr2x16(X_left[n].re, X_left[n].im);
131          t2 = multr_fr2x16(t1, t2);
132          X_left[n].re = low_of_2x16(t2);
133          X_left[n].im = high_of_2x16(t2);
134          t2 = compose_fr2x16(X_right[n].re, X_right[n].im);
135          t2 = multr_fr2x16(t1, t2);
136          X_right[n].re = low_of_2x16(t2);
137          X_right[n].im = high_of_2x16(t2);
138       }
139
140 }
```

4.1 Anwendungen der FFT-Filterbank

Die vorgesehenen Anwendungen sind in [8] für den ADSP-21369 angegeben und können auch mit dem Blackfin® Prozessor realisiert werden. Gleitkommaoperationen in zeitkritischen Funktionen müssen jedoch durch Festkommaarithmetik ersetzt werden. Das bedingt umfangreiche Skalierungsoperationen, um den darstellbaren Zahlenbereich $[-1, 1)$ der Daten vom Typ `fract` zu gewährleisten. Falls erhöhte Rechengenauigkeit erforderlich ist, dann müssen in den kritischen Funktionen 32 Bit Variable vom Typ `int` oder `long fract` verwendet werden. Mit der Festkommaarithmetik und Fractions können auch die beiden 40 Bit Akkumulatoren eingesetzt werden um z.B. eine Summe von Produkten

ohne Genauigkeitsverlust und mit reduzierten Überlaufproblemen zu berechnen (Datentyp `accum` in [6]).

Die Abtastfrequenz von 48 kHz kann beim verwendeten Hardware-Entwicklungssystem nicht verändert werden, da die Abtastfrequenz fix durch den Oszillator des DAC eingestellt ist und die entsprechenden Anschlüsse zur softwaremäßigen Änderung der Abtastfrequenz deaktiviert sind. Für die meisten Filterbankanwendungen ist der DSP für diese Abtastfrequenz schnell genug. Einige der Anwendungen, wie z.B. Pitch Scaling oder Speech Enhancement, benötigen eine aufwendige Verarbeitung im Spektralbereich. Für Sprache und andere Audiosignale reicht normalerweise eine Bandbreite von 12 - 16 kHz vollkommen aus. Für diese Anwendungen kann der Aufwand (Anzahl der zu verarbeitenden Frequenzpunkte der FFT) dadurch reduziert werden, dass z.B. der Frequenzbereich auf 12 kHz reduziert wird. Der restliche Bereich von 12 kHz bis 24 kHz wird auf Null gesetzt. Das folgende MATLAB®-Beispiel zeigt eine Filterbank mit Bandbegrenzung.

```
function y = fft_fib_fc(N,x,fc,df)
%function y = fft_fib_fc(N,x,fc,df)
%
% analysis/synthesis FFT filterbank (oversampling factor L = 4)
% with bandlimitted processing (lowpass filtering with
% Kaiser step function in transition region, and cutoff frequency < Fs/2)
%
% N       window length = FFT length
% x       input signal vector
% fc      cutoff frequency normalized to Fs/2
% df      transition region width normalized to Fs/2
% y       output signal vector
%
% example: xi = [zeros(500,1) ; 1 ; zeros(2000,1)];
%          yi = fft_fib_fc(2048,xi,4/8,0.4/8);
%
% G. Doblinger, TU-Wien, 1-2001, 5-2004, 3-2013

if nargin == 0
   help fft_fib_fc
   return
elseif fc > 1
   error('fc must not be greater then 1');
elseif df > fc
   error('df must not be greater then fc');
end
if mod(N,2) == 1
   N = N+1;
   fprintf(1,'INFO: N must be even, set to %d\n', N);
end

L = 4;                          % oversampling factor
M = round(N/L);                 % filter bank hop size
if M < 1
   error('L must be less than N');
end
Nfh = N/2;

```

```
39 % create time window function
40
41 h = hanning(N);
42 h = sqrt(M/N)*h(:);
43
44 Nx = length(x);
45 if Nx < N
46    error('input signal length less then window length N');
47 end
48 x = x(:);
49 y = zeros(Nx,1);
50
51 % create freq. domain window W of transition region
52
53 if fc < 1
54    W = winstep(Nfh, df);
55    dN = length(W);
56    dNh = floor(dN/2);
57
58    Nlow = round(fc*Nfh)+1; % freq. index of cutoff frequency
59 end
60
61 for m = 1:M:Nx-N+1          % filterbank loop
62
63    % analysis filter bank
64
65    m1 = m:m+N-1;
66    X = fft(x(m1).*h,N);
67
68    % reduce bandwidth to fc using window function W to smooth step-like
69    % transition region (needed to achieve sufficient stop band attenuation)
70
71    if fc < 1
72       X(Nlow-dNh:Nlow+dNh) = W .* X(Nlow-dNh:Nlow+dNh);
73       X(Nlow+dNh+1:Nfh+1) = 0;
74       X(Nfh+2:end) = conj(X(Nfh:-1:2));
75    end
76
77    % synthesis filter bank
78
79    y1 = real(ifft(X,N));
80    y(m1) = y(m1) + y1;
81 end
82
83 % ------------------------------------------------------------------------
84
85 function W = winstep(N, df)
86
87 % window function to smooth step-like transition in frequency domain
88 % (Kaiser step function)
89 %
90 % N  number of frequency points in [0,Fs/2)
91 % df transition width normalized to Fs/2
92
```

```
93  dN = round(N*df);
94  dN = dN - 1 + rem(dN,2);    % dN must be odd
95
96  if dN <= 1
97     W = 0.5;
98     return
99  end
100 W = kaiser(dN,10);
101 W = cumsum(W(:));
102 W = 1-W/max(W);
```

Der Übergang vom Durchlass- in den Sperrbereich erfolgt im Programm nicht abrupt sondern geglättet mit der Fensterfunktion `winstep()`. Dadurch wird eine wesentlich größere Sperrdämpfung erzielt. Das MATLAB®-Programm kann als Grundlage für die Erweiterung des Blackfin® Filterbankprogramms verwendet werden, wobei die benötigte Fensterfunktion mit MATLAB® erzeugt werden kann. Diese Modifikation des Filterbankprogramms ist bereits eine lohnende Aufgabe für den Einstieg in die Programmierung des Blackfin® DSP.

5 Hinweise zur Programmentwicklung

Zusammenfassend möchte ich die folgenden Hinweise für die Programmentwicklung mit dem Hardware/Software-Entwicklungssystem angeben:

- In den Projektoptionen der VisualDSP®++ Entwicklungsumgebung immer die Optimierungsfunktion des C-Compilers einschalten.
- Die Optimierung der Programmfunktionen mit dem Profiler von VisualDSP®++ kontrollieren.
- Variable, die global oder statisch definiert sind und z.B. in Interrupt-Serviceroutinen verändert werden, müssen mit dem Zusatz „volatile“ definiert werden, da sonst der Optimizer diese Variablen „wegoptimiert“.
- In zeitkritischen Funktionen die Built-in Functions für Arithmetik- und Transferoperationen verwenden, wie z.B. in den Musterprogrammen.
- Eine häufige Fehlerquelle ist die Verwechslung der speziellen Datentypen `fract` (bzw. `long fract`) mit `fract16` (bzw. `fract32`). Erstere (sog. Native Fixed-Point Types) werden als Fractions, also als 1.15 oder 1.31 Zahlen interpretiert. Letztere (`fract16`, `fract32`) sind Zahlen im Integer-Format. Operationen mit diesen Variablen müssen mit den Fractional Built-in Functions durchgeführt werden, wenn in C Fractional Arithmetic verwendet werden soll. Im Gegensatz dazu können Native Fixed-Point Types auch direkt in den Arithmetikoperationen von C verwendet werden. Lesen Sie dazu mehr im Abschnitt „Using Native Fixed-Point Types“ in [6].

Nicht alle Funktionen der DSP-Library und Run-Time Library sind derzeit für `fract` und `long fract` verwendbar. Daher sollten bei häufigem Gebrauch dieser Programmbibliotheken die Datentypen `fract16` bzw. `fract32` eingesetzt werden. Übersichtlicher und moderner ist die Verwendung von `fract` und `long fract`.

- Vor dem Programmieren empfehle ich das Lesen von Kapitel 2 „Achieving Optimal Performance From C/C++ Source Code“ in [6].

Literatur

[1] **www.analog.com**

[2] Gerhard Doblinger, „Signalprozessoren, 2. Auflage,“ J. Schlembach Fachverlag, 2004.

[3] Analog Devices, Inc. ADSP-BF537 Blackfin® Processor Hardware Reference, Revision 3.4, Februar 2013.

[4] Analog Devices, Inc. Ez-Kit Lite® Evaluation System Manual, Revision 2.5, Juli 2012.

[5] Analog Devices, Inc. Blackfin® Processor Programming Reference, Revision 2.2, Februar 2013.

[6] Analog Devices, Inc. VisualDSP®++ 5.0 C/C++ Compiler and Library Manual for Blackfin® Processors, Revision 5.4, Januar 2011.

[7] Gerhard Doblinger, „Digitale Signalverarbeitung (ADSP-21065L Prozessor),“ Skriptum zur LVA 389.066 und 389.067, TU-Wien, Februar 2011.

[8] Gerhard Doblinger, „Digitale Signalverarbeitung (ADSP-21369 Prozessor),“ Skriptum zur LVA 389.066 und 389.067, TU-Wien, März 2013.