1. mérés

Jelfeldolgozó processzor alkalmazása

Molnár Károly, Sujbert László
BME Méréstechnika és Információs Rendszerek Tanszék

1. Bevezetés

Beágyazott információs rendszerekben gyakran alkalmaznak jelfeldolgozó processzorokat (DSP-ket). Ezeknek általában az a feladatuk, hogy a jelfolyamba beavatkozva ténylegesen valamilyen jelfeldolgozást valósítsanak meg, vagy egy magasabb szintű eljárás számára (pl. beszédfeldolgozás) készítsék elő a jelet. Hasonlóan fontos szerephez jutnak a jelprocesszorok szabályozó rendszerekben, ahol a szabályozó algoritmus a DSP-n fut. A mérés során egy fixpontos DSP programozását, fejlesztői környezetét, a jelfeldolgozó programok futásának tesztelését ismerhetjük meg.

2. Elméleti összefoglaló

A DSP-k mikroprocesszorok, és általában tartalmaznak minden olyan funkciót, amelyet egy általános célú mikroprocesszor is tartalmaz. A DSP-k azonban igen fejlett – esetleg különféle célokra több – aritmetikai egységgel is rendelkeznek. A DSP-k utasításkészlete olyan, hogy támogassák a leggyakrabban előforduló műveletek minél gyorsabb végrehajtását. Az alábbiakban a legfontosabb DSP tulajdonságokat tekintjük át. Természetesen ez a rövid bevezető nem elegendő ahhoz, hogy teljes képet adjunk a DSP-kről, de a mérés szempontjából kielégítő.

A mérés során a DSP programozása alapvetően C nyelven történik, de szükség lehet assembly nyelvű kódrészletek megírására is. Ez akkor indokolt, ha a megvalósítandó feladatnak szigorú valós idejű követelményeknek kell megfelelnie, vagy ha a processzor lehetőségeit maximálisan ki akarjuk használni.

2.1. Struktúra, memória

A DSP-k a szokásos Neumann-architektúra helyett ún. Harvard-architektúrával rendelkeznek. Ez azt jelenti, hogy külön memória szolgál az adat és a program tárolására, sőt egyes DSP-kben két külön adatmemória is van. Ez a megosztás, és az ezzel járó külön buszrendszer teremti meg a lehetőségét annak, hogy egy utasításciklus alatt fizikailag egy időben lehessen hozzáférni az utasításkódhoz, és akár több operandushoz is. A mérésben szereplő ADSP-BF537 DSP FIR szűrő programjának elemzésekor látni fogjuk, hogy egy utasításciklus alatt a processzor összeszoroz két számot, hozzáadja egy harmadikhoz és a szorzó egység bemeneti regisztereibe beolvassa a memóriából a következő két tényezőt. Ez a *Multiply and Accumulate (MAC)* művelet, amelyet tipikusan minden DSP végre tud hajtani. A művelet egy cikluson belüli végrahajtásának lehetőségét a nagymértékben párhuzamosított architektúra adja. Már a korai DSP-kben is volt önálló szorzó egység, külön címaritmetika (lásd később), stb. Az adatok áramlását több párhuzamos busz szolgálja ki.

A DSP-k fontos eleme a belső vagy on-chip memória. A fenti nagy párhuzamosság ugyanis csak a processzoron belül igaz teljes egészében, ha a DSP külső memóriához fordul, általában már bizonyos hozzáférések csak szekvenciálisan valósíthatók meg. Programozásnál tehát ügyelni kell arra, hogy az egyes változókat, de különösen a tömböket milyen memóriaterületre tesszük. A modern DSP-kben általában néhány tíz Kszó belső memória van, ami elegendő a legtöbb on-line alkalmazáshoz.

2.2. Aritmetika

A modern DSP-kben összeadó (kivonó), szorzó, léptető, logikai egységek vannak, ezek biztosítják a vonatkozó műveletek egy utasításciklus alatti végrehajtását. általában létezik egy olyan egység, amely korlátozott pontosságú (pl. 9 bites) osztást is képes végrehajtani. Az ehhez tartozó utasítás egy iterációs osztó rutin alapja lehet. A processzor számábrázolási pontosságának megfelelő osztást kb. 10 utasítás alatt lehet végrehajtani. Az osztás tehát időkritikus művelet, ha előre ismert számmal kell osztani, célszerű inkább a reciprokával szorozni.

Az adott processzorra jellemző, hogy az aritmetikai egységei milyen bitszámú adatot fogadnak, illetve az eredmény milyen bitszámú regiszterben képződik. Ebből a szempontból (is) elkülönülnek a fixpontos és a lebegőpontos processzorok, ezért a továbbiakban ezeket külön tárgyaljuk.

2.2.1. Fixpontos számábrázolás

A számábrázolás nevét adó „fix pont” azt jelenti, hogy a „kettedespont” (bináris pont), tehát az egészeket szimbolizáló számjegyeket a törtrésztől elválasztó pont helyzete nem változik, a memóriában vagy regiszterben tárolt szám egy helyi értéke állandó. Ez a helyi érték azonban többféle lehet. Tegyük fel, hogy a szám 16 bites, mint a legtöbb fixpontos DSP-ben. A számábrázolás lehet:

- előjel nélküli egész, ebben az esetben az ábrázolt számok: $0..2^{16}-1$,
- előjeles egész, ebben az esetben az ábrázolt számok: $-2^{15}..2^{15}-1$,
- előjel nélküli törtszám, ebben az esetben az ábrázolt számok: $0..1-2^{-16}$,
- előjeles törtszám, ebben az esetben az ábrázolt számok: $-1..1-2^{-15}$.

Az előjeles számokat általában kettes komplemens módon ábrázolják. Az egész számok ábrázolása triviális. Törtszámok esetében a legfelső bit helyi értéke (ha nem volt előjelbit) 1/2, a legalsóé 2^{-16}. Előjeles törtszámok esetében a legfelső bit az előjelbit, ezért a legalsó bit helyi értéke 2^{-15}.

Az, hogy a számábrázolás milyen, nem csupán intervallum-hozzárendelési kérdés. A számábrázolás akkor válik fontossá, ha két számot összeszorzunk, és az eredményt értelmezni akarjuk, illetve helyesen eltárolni vagy továbbszámolni vele. Két bináris szám összeszorzásakor az eredmény bitszáma a két bitszám összege. A követhetőség kedvéért most 3 bites számokon mutatjuk be a számábrázolás jelentőségét.

Tekintsük először az 100, 001 előjel nélküli számokat! Ezek önmagukkal vett szorzata szerepel a következő táblázatban.

operandus	szorzat egész számként	szorzat törtszámként
100 (4, 1/2)	010 **000** (16)	**010** 000 (1/4)
001 (1, 1/8)	000 **001** (1)	**000** 001 (1/64)

A táblázatban a vastagon szedett számok mutatják az eredeti helyi értékeknek megfelelő számmezőt, zárójelben pedig a számok decimális megfelelője szerepel. Látható, hogy az első esetben az egész, a második esetben a törtszámábrázolás kivezet az ábrázolt tartományból. Egész számábrázolás esetén a magasabb, törtszámábrázolás esetén az alacsonyabb helyi értékek felé „nő” a szorzat bitszáma. Egész számábrázolás esetén általában az alsó, törtszámábrázolás esetén a felső szót visszük tovább.

Most tekintsük a 010-át mint előjeles számot! A következő táblázatban ismét az önmagával vett szorzat szerepel. Jól látható, hogy az előjel nélküli esethez képest újabb probléma vetődött fel: a szorzás eredménye itt

operandus	szorzat egész számként	szorzat törtszámként
010 (2, 1/2)	000 **100** (4)	**000** 100 (1/8) **!**
		001 000 (1/4)

nem is ad jó „bitmintát”, a helyes eredményhez (amely a második sorban szerepel) el kell tolni a szorzatot egy helyi értéket balra.

A szorzó egységben tehát a számábrázolástól függő operációt kell végrehajtani. A legtöbb processzoron programozható, hogy a szorzó egység milyen számként kezelje az operandusokat. Jelfeldolgozási célra legtöbbször az előjeles törtszámábrázolást alkalmazzuk, ugyanis ez illeszkedik jól a fizikai képhez (a jel az A/D átalakító tartományán belül van), és a szorzat figyelembe nem vett része hiba jellegű mennyiség. általában lehetőség van kerekítésre is, legtöbbször szintén többféle kerekítés állítható be.

Fixpontos számábrázolás esetén a 16 bit – mint említettük – általános, de nagyobb pontossági igények esetén nem kielégítő. Ilyen problémák merülnek fel pl. IIR szűrők megvalósításakor. Léteznek olyan DSP-k, amelyek magasabb bitszámú (pl. 24, 32 bites) szavakkal dolgoznak.

C-ben történő programozás esetén a fordító elrejti előlünk ezeket a részleteket, és garantáltan jól működik, amennyiben szabványos C adattípusokkal dolgozunk (`float, double, int, long, short`). Ezek használata azonban sokszor közel sem optimális, mert pl. 32 bites lebegőpontos (`float`) számok szorzása a 16 bites fixpontos processzoron akár több száz assembly utasításra fordul le. A 16 bites processzorhoz leginkább a short típus használata illeszkedik (16 bites előjeles egész szám).

A szabványos C-ben nincs meg a fent említett törtszám ábrázolás, ezért az Analog Devices definiálta a `fract16` és `fract32` típusokat, amelyek 16 illetve 32 bites törtszámok ábrázolására hivatottak. Ezen felül a C nyelvnek létezik kiterjesztése (Extensions to support embedded processors ISO/IEC Technical Report 18037), amely tartalmaz tört számábrázolást támogató típusokat, például `fract` és `accum` típusok. A `fract16` és `fract32` típusok továbbra sem szabványos C típusok, tehát pl. két fract16 összeszorzása a $*$ operátorral helytelen eredményre vezet! A probléma megoldására az Analog Devices biztosít C könyvtári függvényeket, amelyekkel a `fract16` és `fract32` típusú számok közötti műveletek elvégezhetők. A `fract` és `accum` típusokra a normál C nyelvű operátorok alkalmazhatók.

2.2.2. Lebegőpontos számábrázolás

A lebegőpont azt jelenti, hogy az ábrázolandó számot $m2^e$ alakban ábrázoljuk, ahol m jelenti a mantisszát, e az exponenst. Az exponenst mindig úgy állapítják meg, hogy a mantissza 0.5..1 közötti szám legyen. Mind az exponens, mind a mantissza általában előjeles szám, de természetesen a különböző kombinációk is előfordulnak. Ebben az esetben egész és törtszámok vegyesen is ábrázolhatók. Pl. a 2.5 decimális számot – 3-3 bites előjel nélküli exponenst és mantisszát feltételezve – 101 2^{010} alakban ábrázolhatjuk (101 $2^{010} = (\frac{1}{2}+\frac{1}{8})\cdot 2^2 = 0.625\cdot 4 = 2.5$).

A lebegőpontos számábrázolással részleteiben nem foglalkozunk. Ennek oka egyfelől, hogy a mérés keretében nem foglalkozunk lebegőpontos DSP-vel, másrészt a lebegőpontos számábrázolás jóval kevesebb problémát vet fel, mint a fixpontos. Az is igaz viszont, hogy amennyiben probléma adódik a számábrázolással, az analízis és a megfelelő megoldás megtalálása nehéz feladat. (Gondoljunk csak arra, hogy a kvantálás itt nem egyenletes.)

A modernebb lebegőpontos processzorok megfelelnek az *IEEE* 754-es szabványában leírt formátumoknak, és igyekeznek az ott leírt kerekítési módokat alkalmazni. (pl. *IEEE single precision*: 9 bites előjeles exponens és 24 bites előjeles mantissza. Mivel a mantissza vezető bitje mindig 1, azt nem ábrázolják fizikailag, így 32 bites szavak elegendőek.)

C programokban megengedett a szabványos lebegőpontos számok használata (`float, double`), de a 2.2.1 pontban említett erőforráskihasználási okokból ezek használata nem célszerű.

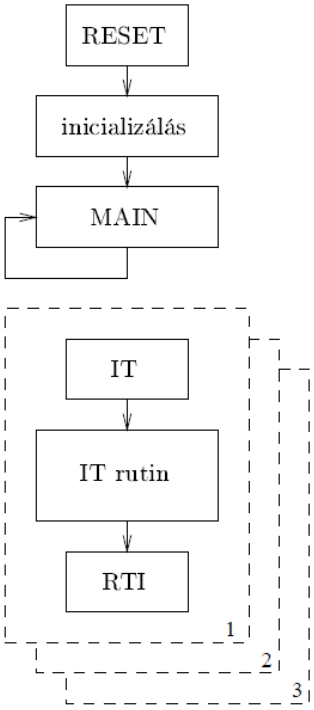
2.3. Címzés

A DSP-kben is létezik direkt és indirekt címzés. Az alábbiakban az indirekt címzésről lesz szó.

A korábban említett nagy párhuzamosság eléréséhez az indirekt címzés és a vele összekapcsolódó címaritmetika is hozzájárul. Az indirekt címzésre néhány (általában 8..16) regiszter áll rendelkezésre. A címregiszterek értékének módosítására a címaritmetikai egységben kerül sor. A módosítás lehet inkrementálás/dekrementálás (címzés előtt vagy után); vagy más, a címregiszterekhez rendelt egyéb regiszterek alapján történő módosítás. Általában két ilyen „hozzárendelt” regiszter van: egy módosító vagy ofszet regiszter, amelynek 1-től különböző értékével lehet módosítani a címregisztert; illetve a címregiszter által címzett ún. cirkuláris buffer hosszát meghatározó regiszter.

A cirkuláris buffer egy olyan memóriaterület, amelynek elemeit ciklikusan olvassuk ki vagy ciklikusan írunk bele. Ilyen buffer lehet pl. egy FIR szűrő együtthatóit tartalmazó buffer, amelyre minden kimeneti minta kiszámolásakor szükség van. A címaritmetika gondoskodik arról, hogy a cirkularitás megvalósítása ne igényeljen plusz utasítást.

A diszkrét Fourier-transzformáció kiszámítására legtöbbször valamilyen FFT algoritmust alkalmaznak. Ennek tulajdonsága, hogy vagy a bemeneti, vagy a kimeneti vektor ún. bitfordított sorrendben van. Ehhez a DSP-k



1. ábra.

támogatást adnak, vagy a címbusz bitjeinek fizikai megfordításával, vagy a kezdőcím ügyes módosításával.

2.4. Programszervezés

Jelfeldolgozó programok gyakran tartalmaznak olyan ciklusokat, amelyek sokszor futnak le két mintavétel között. Ezeknél a ciklusváltozó vizsgálata nagyon lassítaná a program futását. Ilyen esetekben hasznos a DSP-k hardver alapú ciklusszervezése, amely azt jelenti, hogy van egy dedikált hadver egység, amely a ciklusok gyors futtatását biztosítja. Ennek használatakor nem kell külön gondoskodni ciklusváltozóról, csak a ciklus elejét és végét, valamint a cilus lefutásainak számát kell megadni. C programokban előforduló `for` ciklusok esetén a fordító megkísérli kihasználni ezt a lehetőséget.

A gyors végrehajtást szolgálják a párhuzamos utasítások. Általában azok az utasítások, amelyek külön egységet és külön buszt használnak, párhuzamosan is végrehajthatók.

A DSP-kben fontos szerepe van a megszakításoknak. Legtöbbször az A/D átalakító is megszakítást kér, ha egy újabb érvényes adat van a kimenetén. A megszakítások vektorosak, prioritásuk a forráshoz kötött, de egyes esetekben programozható.

A jelfeldolgozó programok egy általánosan alkalmazott struktúráját mutatja az 1. ábra. A program futása az inicializáló résszel kezdődik: az inicializálás jelenti a processzor beállításainak rögzítését (pl. IT prioritások, számábrázolási mód), de itt kapnak kezdőértéket a változók is. Az inicializálás alatt a megszakítsok tiltva vannak.

Miután az összes beállítás megtörtént, és a megszakítások is engedélyezve lettek, a „főprogram” következik, amely azonban nem csinál semmit, csak magára ugrik vissza. Amíg megszakítás nem érkezik (pl. az A/D átalalkító felől), addig a processzor vár. Megszakítás esetén a processzor a kérést kiszolgálja, majd visszatér a „főprogram”-ba.

Jelfeldolgozó programoknál az adatvesztés úgy kerülhető el, hogy az IT rutin biztosan hamarabb lefut, mint

hogy a következő A/D felől érkező megszakítás megérkezne. A jelfeldolgozó algoritmusok (átlagolás, szűrés, valamilyen szabályozási feladat) mindig felírhatók úgy, hogy egy tetszőleges időpontban a bemenet aktuális és korábbi értékeiből hogyan határozható meg a kimenet aktuális mintája. Ez – egyszerűbb esetben – a jól ismert állapotváltozós leírás. Az IT rutinnak ezt az algoritmust kell megvalósítania.

Ilyen programszervezés mellett az indokolt, hogy a RESET után egyszer lefutó inicializálást C nyelven írjuk meg, mivel ennek futási ideje nem kritikus. A jelfeldolgozást végző rutin esetén viszont előfordulhat hogy a C nyelven írt rutin futási ideje nagyobb lenne mint a mintavételi idő, ekkor ezt assembly nyelven kell megírni, törekedve a processzor párhuzamosítási és egyéb speciális hardver gyorsítási lehetőségeink teljes kihasználására.

3. Az ADSP BF537 jelfeldolgozó processzor

A processzor az Analog Devices cég terméke. Az alábbiakban az elméleti összefoglalóban megismert sorrendben tekintjük át a tulajdonságait. A programozáshoz, hibakereséshez természetesen szükség van a felhasználói kézikönyvekre [1, 2], itt csak a legfontosabb ismereteket közöljük.

3.1. Struktúra, memória

A processzor egy program- és egy adatmemóriát címez meg. A processzor 48 KByte belső programmemóriát és 64 KByte adatmemóriát tartalmaz. Az utasítások 16 és 32 bitesek lehetnek, az adatok pedig 8, 16 vagy 32 bitesek. A processzor on-chip memóriájával bővebben a programozási kézikönyv [2] 6. fejezete foglalkozik.

3.2. Aritmetika

Az aritmetikai műveleteket 3 különböző egység (ALU – aritmetikai-logikai, MAC – szorzó-összeadó, Shifter – léptető) végzi. Ezek részletes leírása a programozási kézikönyvben [2] található meg, a 2. fejezetben. Assembly programozás esetén nagyon fontos ezen kívül a 15. fejezetben bemutatott szintaktika betartása, mert esetenként nagyon kötött hogy az egyes műveletekhez melyik regiszterek használhatók.

3.3. Címzés

A címzésre használható regisztereket csak assembly nyelvű programozás esetén kell közvetlenül kezelnünk, C programozás esetén a fordító gondoskodik erről.

Az indirekt címzésre a P0..5 regiszterek használhatók. Cirkuláris bufferek megvalósítására a B0..3, I0..3, L0..3 és az M0..3 regisztereket kell használni. A buffer kezdőcímét az egyik B (Base) regiszterben, hosszát a hozzá tartozó L (Length) regiszterben, a léptetés értékét pedig valamely M (Modify) regiszterben kell megadni. Az I (Index) regiszterben található az éppen aktuális memóriacím, tehát léptetéskor ez a cím változik az M módosító regiszter értékével.

4. A fejlesztőrendszer

A DSP kártya programozása a VisualDSP++ integrált fejlesztői környezetben történik. Ennek egyik fő része a projekt editor, ahol a DSP program forrásfájljainak szerkesztése történik. Az egymáshoz rendelt forráskód fájlok alkotnak egy projektet, amelyet a VisualDSP++ fordít le, linkel és egy DSP-re letölthető fájlt generál. A forráskód fájlok készülhetnek assembly, C vagy C++ nyelven. A Linker számára az ún. Linker Description File (.ldf) tartalmaz utasításokat.

A fejlesztői környezet másik fontos része a debugger, amely lehetőség biztosít a megírt DSP programok kipróbálására futás közben. Lehetőség van a program futásának részletes vizsgálatára, töréspontok helyezhetők el, lehet léptetve végrehajtani az utasításokat, a memóriaterületek valamint a rendszer regisztereinek tartalma

folyamatosan monitorozható. A debugger "bementi adata" a projekt editor által előállított, a DSP-re letölthető fájl. A DSP program futtatható szimulátorban, vagy emulátor segítségével fizikai hardveren is. Ennek kiválasztása a Session menüben történik.

A mérés során a programokat ADSP BF537 EZ-KIT Lite kártyán fogjuk futtatni emulátor segítségével. A kártya USB porton csatlakozik a PC-hez. A DSP kártya tartalmaz egy ADSP BF537 600 MHz-es DSP-t, egy AD1871 sztereo DAC-t, valamint egy AD1854 sztereo ADC-t, amelyek jelei Jack aljzatokra vannak kivezetve. Mind az ADC, mind a DAC 48 kHz mintavételi frekvencián üzemel. A kártya sok egyéb funkcionális egységet is tartalmaz, de a mérés szempontjából csak ezek lényegesek.

5. Példa alkalmazások

A példa alkalmazások bemutatják az ADSP-BF537 legfontosabb utasításainak használatát és a programozási modellt. A programok ismertetése kapcsán kitérünk néhány, a fejlesztőrendszerre jellemző tulajdonságra is.

Az első példa a BlackFin_Frame projekt. A projekthez tartozó forrásfájlok az alábbiak (a fontosabbak forráskódját ld. a 6. fejezetben):

- **main.c** – Ez a főprogram, RESET után ez fut le először. Meghívja az inicializáló függvényeket, amelyben a megszakítási rutinok is inicializálásra kerülnek, majd végtelen ciklusba kerül.
- **Initialize.c** – A hardver inicializását végző kódot tartalmazza.
- **ISR.c** – A különböző megszakításokhoz rendeli hozzá a kiszolgáló rutinokat. Jelen esetben csak az AD-hoz tartozó megszakítást rendeli hozzá a Process_data függvényhez, azaz ha az AD-n egy új minta áll elő, akkor meghívódik a Process_data függvény.
- **Process_data.c** – Ez tartalmazza a Process_data függvényt, amely tehát a minták feldolgozását végzi. Ebben a függvényben célszerű megvalósítani a jelfeldolgozó algoritmust. Jelen esetben a bejövő mintákatat változtatás nélkül átmásoljuk a kimenetre, azaz a program egy egyszerű jeltovábbítást hajt végre.

Második példánk a BlackFin_FIR_asm projekt, ami az előzőtől csak a Process_data függvényben különbözik. Ez az alkamazás egy FIR szűrőt valósít meg. A FIR szűrő algoritmusa a következő:

$$y(n) = \sum_{i=0}^{N-1} w(i)x(n-i),$$

ahol $y(n)$ a kimenet az n. időpillanatban, $x(n-i)$ jelenti a bemenet aktuális és késleltetett értékeit, $w(i)$ pedig a szűrőegyütthatók. A szűrőnek tehát N együtthatója van. A fenti képlet szerinti számítást kell tehát a DSP-nek elvégeznie a megszakítást kiszolgáló rutinban. A fenti művelet, az ún. diszkrét konvolúció kitüntetett jelentőségű a digitális jelfeldolgozásban. Használatáról lásd pl. *The Scientist and Engineer's Guide to Digital Signal Processing* [3] ingyenesen letölthető gyakorlati DSP témájú könyv 6. fejezetét.

- **process_data.c** – Ez a fájl tartalmazza a beérkezett adatok feldolgozását végző `Process_data()` függvényt. A bementi adatokat az input tömbben tárolja, a szűrőegyütthatókat pedig a coefs tömbbe olvassa be egy külső fájlból. A `section` előtagok biztosítják hogy a két tömb különböző memóriaterületre kerüljön, hogy egyidejűleg lehessen olvasni őket a konvolúció során. A konvolúciót a `conv_asm()` függvény végzi, amelynek forráskódja a conv_asm.asm fájlban található.
- **conv_asm.asm** – A `conv_asm()` függvény assembly nyelven készült, annak érdekében hogy kihasználja processzor párhuzamosítási lehetőségeit a számítás során. A függvény részletes működése a kommentekből érthető meg.

6. Forrásfájlok

6.1. main.c

```
/*********************************************************************************/
#include "LabFrameFIR.h"
#include <sysreg.h>
#include <ccblkfn.h>
/*********************************************************************************/
Variables
Description:  The variables ChannelxLeftIn and ChannelxRightIn contain
the data coming from the codec ADC (AD1871).  The (processed)
playback data are written into the variables
ChannelxLeftOut and ChannelxRightOut respectively, which
are then sent back to the DAC (AD1854) in the SPORT0 ISR.
*********************************************************************************/
// left input data from AD1871
int iChannel0LeftIn, iChannel1LeftIn;
// right input data from AD1871
int iChannel0RightIn, iChannel1RightIn;
// left ouput data for AD1854
int iChannel0LeftOut, iChannel1LeftOut;
// right ouput data for AD1854
int iChannel0RightOut, iChannel1RightOut;
// SPORT0 DMA transmit buffer
int iTxBuffer1[2];
// SPORT0 DMA receive buffer
int iRxBuffer1[2];

//------------------------------------------------------------------------
// Function:  main()
// Description:  After calling a few initalization routines, main() just
// waits in a loop forever.  The code to process the incoming
// data can be placed in the function Process_Data() in the
// file "Process_Data.c".
//------------------------------------------------------------------------
void main(void)
{
        Init_Flags();
        Audio_Reset();
        Init_Sport0();
        Init_DMA();
        Init_Interrupts();
        Enable_DMA_Sport0();
        while(1);
}
```

6.2. process_data.c

```
#include "Talkthrough.h"
//-----------------------------------------------------------------------------------------------------------
// Function:  Process_Data()
//
// Description:  This function is called from inside the SPORT0 ISR every
// time a complete audio frame has been received.  The new
// input samples can be found in the variables iChannel0LeftIn,
// iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn
// respectively.  The processed data should be stored in
// iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut,
// iChannel1RightOut, iChannel2LeftOut and iChannel2RightOut
// respectively.
//-----------------------------------------------------------------------------------------------------------
void Process_Data(void)
{
          iChannel0LeftOut = iChannel0RightIn;
          iChannel0RightOut = iChannel0RightIn;
}
```

6.3. Process_data.c

```
#include "LabFrameFIR.h"
#define N 401
         // Szűrőegyütthatók száma
section("L1_data_b") fract16 coefs[N] = #include "bp_fract16.dat";
         // Szűrőegyütthatók beolvasása külső fájlból
section("L1_data_a") fract16 input[N];
         // A "section" előtag szabja meg, hogy a deklarált változó melyik
         // memóriterületre kerüljön.  A coef és input buffereket érdemes
         // különböző területekre tenni, hogy a konvolúciónál párhuzamosan
         // lehessen beolvasni az értékeiket.
int i = 0;
extern int conv_asm( fract16 *, fract16 *, fract16 * );
//-------------------------------------------------------------------------//
// Function:  Process_Data()
//-------------------------------------------------------------------------//
void Process_Data(void)
{

         iChannel0LeftOut = iChannel0RightIn;
                  // A bal kimeneti csatornán változtatás nélkül
                  // megjelenik a jobb bemeneti csatorna jele

         input[i] = rbits((iChannel0RightIn >> 8));
                  // Konverzió 24-ről 16 bitre
         fract16 out = 0;

         out = conv_asm(coefs,&input[i],input);
                  // Konvolúció számítását végző függvény meghívása

         if (i==N){
                  i = 0;
                  }
                  // Számláló nullázása

         iChannel0RightOut = bitsr(out) << 8;
                  // Konverzió 16-ról 24 bitre }
```

6.4. conv_asm.asm

```
// ******************************************************* //
// ASSEMBLY CONVOLUTION FUNCTION
// Created by Karoly Molnar 2006.
// Declaration:
// extern int conv_asm( fract16 *, fract16 *, fract16 * );
// Usage:
// out = conv_asm(coefs,&input[i],input);
// ******************************************************* //
// Fontos megjegyzes:
// A két bemeneti vektor hosszát manuálisan kell megadni az
// N_TAPS konstans beállításával
// ******************************************************* //

#define N_TAPS 401
.section L1_code;
         // A kód az L1_code memóriaterületre kerül
.global _conv_asm;
         // A függvény globálisan hívható

_conv_asm:
         // Címke, a függvény kezdõcíme
         // A C függvény argumentumait az ASM függvény az R0, R1, R2
         // regiszterekben kapja meg (elõírás)

         P0 = R0; // A P0-ba indexregiszterbe kerül a coeffs kezdõcíme
         I0 = R1; // Az I0 cirkuláris bufferbe az aktuálisan bejött minta címe
         B0 = R2; // B0-ba az input buffer kezdõcíme
         P1 = 2 * N_TAPS;
                  // A 2-vel szorzás oka, hogy 16 bites adatokat használunk
         L0 = P1;
         M0 = 1;
         R0 = 0;
         NOP;
         NOP;

         R1 = W[P0++] (Z);
                  // Indirekt címzés, R1-be íródik a P0 által mutatott érték
                  // majd P0 értéke egyel nõ
         R2.l = W[I0++];
                  // Cirkuláris buffer használata, R2-be íródik az I0
                  // által mutatott érték majd P0 értéke egyel nõ

         A0 = 0;  // Az akkumuláló regiszter nullázása
         P1 = N_TAPS - 1;
                  // N-1-re kell állítani, mert a ciklust N-1-szer futtajuk le
```

```
        LSETUP (mac_loop,mac_loop) LC0 = P1;
                // Hardveres ciklusszervezés, a ciklus eleje és vége is
                // ugyanaz a címke (mac_loop), azaz 1 utasítás van a ciklusban


        mac_loop:
        A0 += R1.l * R2.l || R2.l = W[I0 ++] || R1 = W[P0++](Z);
                // Egy utasításon belül 1 szorzás, 1 összeadás
                // és 2 memóriamozgatás (MAC)
        A0 += R1.l * R2.l;
                // Az utolsó szorás és összeadás már cikluson
                // kívüli, mert ekkor már nincs memórimozgatás
        R0.l = A0;
                // Az visszatérési értéket az R0-regiszterben kell
                // tárolni (elõírás)
        RTS;

_conv_asm.end:
                // Címke, a függvény vége
```

Hivatkozások

[1] *C/C++ Compiler and Library Manual for Blackfin Processors.* http://www.analog.com/processors/blackfin/technicalLibrary/manuals/.

[2] *ADSP-BF53x/BF56x Blackfin Processor Programming Reference.* http://www.analog.com/processors/blackfin/technicalLibrary/manuals/.

[3] Stewen W. Smith. *The Scientist and Egineer's Guide to Digital Signal Processing.* 1997. http://www.dspguide.com.