1 FFT A BLOKOVÉ SPRACOVANIE V JAYZKU C POMOCOU PROCESORA ADSP218X

1.1 ÚVOD

Algoritmus rýchlej Furierovej transformácie (Fast Fourier Transform – FFT) patrí medzi základné **blokové algoritmy** číslicového spracovania signálov. Oproti klasickým FIR a IIR filtrom preberaným v predchádzajúcich cvičeniach blokové algoritmy vyžadujú odlišný prístup predovšetkým k načítavaniu vzoriek. Spracovanie bloku vzoriek je možné začať až po načítaní celého bloku, a teda výpočet FFT nie je možné v prípade väčších blokov realizovať v obsluhe prerušenia. V rámci cvičenia bude naznačený špeciálny blokový mód[1] seriového portu – tzv. autobufering, ktorý umožní relatívne jednoduchým spôsobom tento problém vyriešiť.

Pretože implementácia algoritmu FFT je oproti FIR a IIR výrazne zložitejšia, v rámci cvičenie ukážeme princíp využitia vyššieho programovacieho jazyka C a jeho optimalizovaných knižničných funkcií pre FFT. V praxi je tak možné aj bez detailnej znalosti pomerne komplikovanej implementácie podprogramu FFT jeho ľahké využitie v aplikačných programoch. Aj keď jazyk C je oproti dobre optimalizovanému programu v asembleri menej efektívny, v prípade blokového spracovania väčších blokov (v našom prípade rozmeru FFT) je vplyv prekladača na efektivitu výsledného programu pomerne malý.

Opísaný zdrojový kód zatiaľ neumožní prácu pomocou reálnych technických prostriedkov (vývojovou doskou EZ-KIT2181 Lite) a jeho funkčnosť bude overovaná pomocou simulátora.

1.2 AUTO-BUFROVANIE SÉRIOVÉHO KANÁLU

V predchádzajúcich cvičeniach sme využívali prerušenie od sériový portu SPORT1. SPORT bol nakonfigurovaný tak, že vždy keď sa do prijímacieho registra RX1 prečítala nová vzorka, vyvolalo sa príslušné prerušenie. V obsluhe tohto prerušenia sa prijatá vzorka z registra RX1 prečítala a spracovala. Takéto spracovanie predstavuje klasický spôsob obsluhy prerušenia. Sériové porty SPORT0 a SPORT1 však umožňujú aj využitie špeciálneho DMA módu, tzv. autobufrovania (autobuffering) [1]. V tomto móde po príjme vzorky do prijímacieho registra RX sú prijaté dáta **automaticky** presunuté do dátovej pamäte (DM)[2], pričom na adresovanie sa používa niektorý

[1] V podstate ide o využitie špeciálneho DMA módu pre sériový port.

[2] DM do ktorej sa zapisujú prijaté dáta z registra RX musí byť modulo bufer. Prerušenie od prijímača sa realizuje vždy po prechode z konca bufra na jeho začiatok, t.j. po načítaní celého bloku dát. V obsluhe prerušenia je preto potrebné tento blok spracovať. Typický spôsob spracovania je napr.

vyhradený register DAG jednotky[3]. Pojem vyhradený znamená, že tento register (voľba ktorého sa realizuje počas konfigurácie príslušného portu) už **nie je možné** v samotnom programe využívať. Táto skutočnosť samozrejme reprezentuje určité **obmedzenie**[4] a preto je potrebné využívať tento mód činnosti len v obmedzenej miere.

Ďalším výrazným obmedzením, ktoré **komplikuje** použitie autobufrovania spolu s C kompilátorom a optimalizovanými knižnicami pre FFT je **bitovo-reverzné adresovanie** jednotky DAG1. Podprogramy FFT využívajú hardvérovo podporované bitovo-reverzné adresovanie pre zrýchlenie výpočtu FFT. Keďže však registre i2, i3 sú práve v jednotke DAG1, ich využitie súčastne s využitím podprogramov FFT by spôsobilo zapisovanie na bitovo-reverzne modifikované adresy do DM. Po ukončení podprogramu FFT by pokračovalo zapisovanie v štandardnom lineárnom móde. Samozrejme toto by spôsobilo nesprávnu činnosť prenosu medzi sériovým portom a pamäťou. Možný spôsob riešenia tohto obmedzenia je opísaný v [3] a je použitý aj v tomto cvičení.

Podstata riešenia [3] spočíva v kombinácii **autobufrovania** v čase keď sa podprogram FFT nevyužíva a použití **klasickej obsluhy** prerušenia počas používania podprogramu FFT[5].

1.2.1 MODIFIKÁCIA TABUĽKY VEKTOROV PRERUŠENÍ V C PROSTREDÍ

Obsah tabuľky vektorov prerušení je v prostredí prekladača jazyka C definovaný v súbore 218x_int_tab.asm, ktorý je možné nájsť v adresári

VisualDSP\218x\lib\src\lib_src

Modifikovaná verzia tohto súboru je súčasťou[6] projektu **FFT_test.zip** [4]. Obsluha prerušení v C prostredí je štandardne realizované pomocou špeciálnej funkcie (C interrupt dispatcher) a vektor prerušenia napr. pre príjem od SPORT0 je definovaný v pôvodnom súbore 218x_int_tab.asm v tvare

prepnutie na druhý prijímací bufer. Počas príjmu do druhého bufra je možné obsah prvého spracovať. Tento spôsob spracovania je využitý aj v tomto cvičení.

[3] Identický mechanizmus je možné využiť aj pre vysielací register TX1(2) sériových portov SPORT1(2).

[4] V DAG jednotkách (DAG1 a DAG2) je len 8 registrov (i0,i1,i2,i3,i4,i5,i6,i7), ktoré môžu byť využité pre adresovanie DM. Naviac napr. v prípade využitia C prekladača je možnosť výberu výrazne obmedzená, pretože kompilátor a optimalizované knižnice (ako napr. FFT, FIR, IIR, ...) už využívajú prevažnú časť registrov [2]. Jediné i* registre, ktoré môžu byť vyhradené pre autobufrovanie (a samozrejme nie sú ani použité v optimalizovaných knižniciach pre ČSS) sú registre i2 a i3.

[5] Samozrejme výsledný programový kód je výrazne komplikovanejší ako kód, ktorý sme používali v predchádzajúcich cvičeniach. Kód kombinuje pomerne komplikované spracovanie v obsluhe prerušenia (optimalizované v asembleri) s pomerne zložitým hlavným programom (napísaný v jazyku C a využívajúci optimalizované knižničné funkcie pre FFT). Je však potrebné si uvedomiť, že tento kód je potrebné napísať len raz a potom je možné okamžite začať využívať všetky výhody jazyka C. Tvorba výsledných aplikácií je tak podstatne efektívnejšia a prehľadnejšia.

[6] Ak potrebujeme modifikovať niektoré originálne zdrojové kódy, stačí zahrnúť modifikované zdrojové kódy do aktuálneho projektu. Vo finálnom kóde budú linkované preklady týchto modifikovaných kódov a nie originálnych knižničných funkcií. Tento mechanizmus umožňuje modifikovať časti prostredia C prekladača a jeho funkcií bez nutnosti rekompilovať celý systém.

```
.section/code IVsport0recv;
.global ___sport0recv;
___sport0recv:
        DM(I4+=M7)=AX1;             // ulozenie  registra do softveroveho zasobnika v DM
        AX1=SIGSPORT0RECV;
        JUMP __lib_int_determiner;  // funkcia zabezpeci aj navrat z podprogramu pomocou RTI instrukcie
        NOP;                        // zarovnanie na 4 instrukcie/prerusenie (originalny kod obsahuje NOP)
```

Modifikovaný súbor 218x_int_tab.asm ktorý je súčasťou projektu modifikuje obsluhu tohto prerušenia (t.j. obchádza celý mechanizmus a réžiu C prostredia) priamym volaním funkcie obsluhy prerušenia v tvare

```
/* deklaracia extrernej referencie pre obsluhu prerusenia od prijimaca SPORT0 */
.extern SPORT0_RX_Interrupt;
...
...
/* volanie optimalizovanej obsluhy prerusenia */

.section/code IVsport0recv;
.global ___sport0recv;
___sport0recv:
        JUMP SPORT0_RX_Interrupt;
                RTI;
                RTI;
                RTI;                // opat zarovnanie na 4 instrukcie/prerusenie
```

1.2.2 OPTIMALIZOVANÁ OBSLUHA PRERUŠENIA OD PRIJÍMAČA SPORT0

Optimalizovaný kód musí v prípade povolenia autobufrovania (t.j. keď nie je využívaný podprogram FFT) zabezpečiť bez vyvolania prerušenia správne spracovanie prerušenia, ktoré sa vyvolá až po prechode registra (v našom prípade i2) z konca cirkulačnej pamäte na jej začiatok. Dĺžka pamäte je definovaná príslušným registrom (v našom prípade l2) v DAG jednotke. V prípade, že je využívaný podprogram FFT, kód musí zabezpečiť spracovanie prerušenia po každom prijatí novej vzorky. Nasledujúci kód (súbor SPORT_irq.asm v projekte) spĺňuje uvedené požiadavky a je vysvetlený v príslušných komentároch.

```
#define N 512

/* .h subor, ktory definuje vstupne body a makra */
#include <asm_sprt.h>
#include <def2181.h>

// deklaracia premennych (ako externe) ktore su definovane  v C
.extern _Buffer1;
.extern _Buffer0;
.extern _db_tgl;

.extern _Bit_Reversal_On;
.extern _DoFFT;

// deklarovanie sekcie mena funkcie, meno funkcie sa musi zhodovat s funckiou definovanou v tabulke preuseni
.section /pm program;


.global SPORT0_RX_Interrupt;
SPORT0_RX_Interrupt:
      /* Najskor povolime tienovu banku registrov, aby nedoslo k modifikacii registrov
         pouzivanych v C prostredi */
      ena SEC_REG;

      /* Testovanie ci interupt bol vyvolany mechanizmom autobufrovania (t.j. navratom
         registra i2 na zaciatok cirkulacnej pamate), alebo doslo k prijmu novej vzorky pri
         vypnutom autobufrovani? */
```

```
        ax0 = dm(_Bit_Reversal_On);
        ar = pass ax0;
        if EQ jump      Autobuffering_Interrupt;
Non_Autobuffering_Interrupt:
      /* V tomto bode vieme, ze autobufrovanie bolo vypnute (v hlavnom programe) a je potrebne
         spracovat prijatu vzorku */

        /* Zakazanie bitovo-reverzneho adresovania (v DAG1), takze data mozu byt spravne ulozene */
        dis     BIT_REV;
        /* Softverova nahrada cinnosti, ktora sa realizovala automaticky pocas autobufrovania */
        ax0     = rx0;
        ay1 = i2;                   // ay1 = i2 pred zapisom do DM
        dm(i2,m1) = ax0;
        ax1 = i2;                   // ay1 = i2 po zapise do DM

        /* Kontrola pretecenia i2 registra spat na zaciatok cirkulacnej pamate, ak ano,
           je potrebne generovat obsluhu spracovania autobufrovania */
        ar = ax1 - ay1;
        if LT jump Autobuffering_Interrupt;

        /* Pri vyvolani prerusenia su stavove registre ulozene do stavoveho zasobnika.  Pri navrate
           z prerusenia su tieto hodnoty automaticky obnovene a hoci sme povolili tienovu banku
           registrov a zakazli bitovo-reverzne adresovanie, nie je potrebne ich obnovovat, pretoze
           toto zabezpeci (automatickym obnovenim hodnot zo zasobnika) RTI instrukcia! */
        rti;

Autobuffering_Interrupt:
/* Pri obsluhe prerusenia od autobufrovania sa vyuziva tzv. Dvojnasobne bufrovanie pomocou tzv.
   ping-pong metody, ked sa cyklicky prepina medzi prijmom do Buffer0 a Buffer1.  Udaje v aktualne
   nezapisovanej pamati su subezne spracovavane v hlavnom programe */
        ax0 = 1;
        dm(_DoFFT) = ax0;

        ax0 = dm(_db_tgl);
        ar = pass ax0;
        IF NE jump Buffer0_Use;

Buffer1_Use:
        i2 = _Buffer1;
        ar = NOT ar;
        dm(_db_tgl) = ar;
        rti;

Buffer0_Use:
        i2 = _Buffer0;
        ar = NOT ar;
        dm(_db_tgl) = ar;
        rti;
```

1.3 INICIALIZÁCIA SYSTÉMU V C PROSTREDÍ

Nasledujúci kód v jazyku C inicializuje sériový port, autobufrovanie a povoľuje prerušenia. Uvedený kód je v súbore hlavného programu **main.c**, ktorý je súčasťou projektu FFT_test.zip.

Nastavenie sériového portu (v našom príklade SPORT0) je realizované pomocou makier v hlavičkovom súbore **sport.h** (\VisualDSP\218x\include) priamo pomocou príkazov jazyka C, pričom periférie sú zapisované ako pamäťovo mapované C štruktúry.

```
#include <sport.h>
#include <circ.h>
...
/* Nastavenie serioveho portu SPORT0
   Nastavenie 10 Mhz internych hodin SCLK s 80Mhz hodinami */
   sport0.sclkdiv = COMPUTE_SCLKDIV(80000000, 10000000);

/* Nastavenie RFS tak, aby bol generovany kazdych 20 SCLK periods
   RFS je ramcovy synchronizacny signal (Receive Frame Synchronization),
   ktory definuje presne okamihy, kedy prichadzaju platne data. SCLK potom
   definuje jednotlive bity a kedze RFS je aktivne kazdych 20 period SCLK,
   je mozne preniest (aj s rezervou) aj 16-bitove vzorky */
   sport0.rfsdiv = 20;
/* Nastavenie riadiaceho registra SPORT0 */
   sport0.control.single.isclk      = 1;    // interne sclk
   sport0.control.single.irfs       = 1;    // interne RFS
   sport0.control.single.rfsr       = 1;    // RFS je vyzadovane
   sport0.control.single.slen       = 15;   // 16 bitove slova
```

Ďalšie príkazy nastavujú podobným spôsobom autobufrovanie portu SPORT0.

```
/* Nastavenie autobfrovacich registrov */
   sport0.autobuffer.receive_enable= 1;    // povoli autobfrovanie prijmu od SPORT0
   sport0.autobuffer.rmreg          = 1;    // M3 (=1 prednastavene v C prostredi)
   sport0.autobuffer.rireg          = 2;    // I2 (register je potrebne rezervovat)
/* Nastavenie DAG registra i2 pre autobufrovanie a jeho inicializacia tak, aby ukazoval
   na _Buffer0 s dlzkou 512
   toto makro je mozne najst v circ.h */
   circ_setup(2, Buffer0, 512);
```

Po konfigurácií portu SPORT0 povolíme sériový port a spustíme prerušenie. Volanie funkcie **interrupt** je v skutočnosti volanie prázdnej funkcie SPORT_Receive_Int. Táto funkcia v skutočnosti nebude nikdy volaná, pretože obsluhu prerušenia od prijímača portu SPORT0 v jazyku C sme nahradili vlastnou obsluhou prerušenia. Volanie funkcie interrupt v našom kóde zabezpečí odmaskovanie prerušenia od prijímača SPORT0.

```
/* start SPORT0 */
   sport_start(0);

/* Definovanie C funkcie ktora by bola volana pri obsluhe prerusenia. V skutocnosti
   tato funkcia len povoli prislusne prerusenie. Pretoze v tabulke preruseni sme
   priamo definovali vlastnu funckiu, pri obsluhe prerusenie od prijimaca SPORT0 bude
   prerusenie spracovat nasa funkcia */
   interrupt(SIGSPORT0RECV, SPORT0_Receive_Int);
```

1.4 FFT A BLOKOVÉ SPRACOVANIE

Blokové spracovanie pomocou FFT využíva implementáciu FFT pomocou knižničnej funkcie **fft256(…)**, ktorej prototyp je definovaný v hlavičkovom súbore **fft.h** (\VisualDSP\218x\include) a zdrojový kód v súbore fft256.asm (\VisualDSP\218x\lib\scr\dsp_src\). Hlavný program najskôr inicializuje sériový port SPORT0 a povolí prerušenie od SPORT0. Potom je v nekonečnej slučke for(;;) realizované spracovanie prijatých vzoriek v pamätiach Buffer0 a Buffer1. Synchronizácia s obsluhou prerušenia je realizovaná pomocou premenných (príznakov) **Do_FFT**, **db_tgl** a **Bit_Reversal_On** opísaných v predchádzajúcich častiach. V hlavnej slučke procesor vstupuje do režimu so zníženým príkonom (inštrukcia **asm("idle;")**;). Z tohto režimu sa procesor dostane automaticky po obsluhe prerušenia (v našom prípade od prijímača SPORT0) a automaticky pokračuje spracovaním aktuálne načítaného bloku dát. V prvej časti sa určí spektrum pomocou funkcie fft256 a potom sa nájde spektrálna zložka s maximálnou amplitúdou.

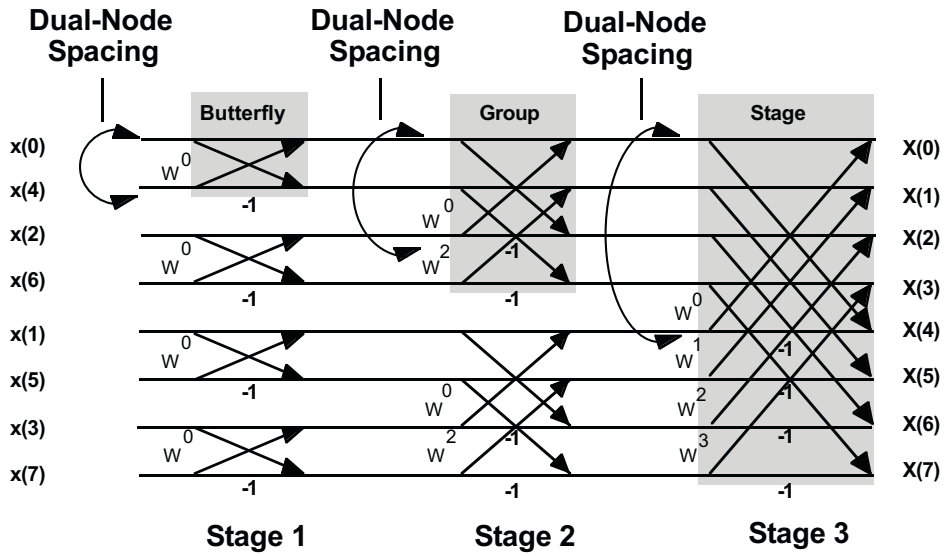
```
#include <ffts.h>
#include <signal.h>
#include <fract.h>

#include <sport.h>
#include <circ.h>

extern  int Real_Out[256], Imag_Out[256];

/* Prva polovica kazdeho buffra obsahuje len realne casti vzoriek, druha polovica imaginarne casti vzoriek */
extern  int Buffer0[], Buffer1[];

/* Tieto pointre su vyuzite na rozdelenie pamate s velkostou 512 vzoriek na 2 pamate
   s velkostou 256, druha polovica je pri vypocte FFT interpretovana ako imaginarna */
int * Buffer0_Imag;
int * Buffer1_Imag;

/* Priznak na prepinanie bufrovania. Vstupne data sa zapisuju cyklicky do dvoch vyrovnavacich
   pamati s velkostou 512, pricom priznak db_tgl urcuje, ktora pamat je aktivna. Do aktivnej
   sa udaje zapisuju pocas obsluhy prerusenia a druha je v tom istom case spracovavana v hlavnom
   programe pomocou (blokovej) FFT. */
int     db_tgl = 0;

/* Do_FFT priznak je nastaveny v case ked su platne udaje v jednej z cirkulacnych pamati. Tento
   priznak nastavuje obsluha prerusenia od prijimaca SPORT0 a informuje tak hlavny program, ze
   bol nacitany komletny blok udajov.
   Po nastaveni priznaku Do_FFT hlavny program spracuje prijaty blok a vynuluje priznak Do_FFT. */
int     DoFFT=0;

int max = 1;
int index = 1;
/* Nastavenie priznaku Bit_Reversal_On informuje obsluhu prerusenia, ze v hlavnom programe je
   pouzivana funkcia FFT, ktora pouziva bitovo-reverzne adresovanie. Hlavny program tak informuje
   obsluhu prerusenia o aktualnom nastaveni DAG1 jednotky, ktora pouziva bitovo reverzne adresovane.
   Podla tohto priznaku je riadene spracovanie preruseni od prijimaca portu SPORT0 */
int Bit_Reversal_On = 0;

/* Tato funckia je len parametrom volania funckie interrupt a v skutocnosti sa nevyvolava! */
void    SPORT0_Receive_Int( int sig_int )
{

}
int bexp;

main()
{
      int i;
      /* Nulovanie riadiaceho registra SPORT0 pre spravnu funkciu neskor pouzitych makier */
      asm("AX0=0; DM(0x3FF3)=AX0;");//r

      /* Nastavenie pointrov do stredu dvoch pracovnych bufrov Buffer0 a Buffer1 */
      Buffer0_Imag = Buffer0 + 256;
      Buffer1_Imag = Buffer1 + 256;

      /* Nastavenie serioveho portu SPORT0

        Nastavenie 10 Mhz internych hodin SCLK s 80Mhz hodinami */
      sport0.sclkdiv = COMPUTE_SCLKDIV(80000000, 10000000);

      /* Nastavenie RFS tak, aby bol generovany kazdych 20 SCLK periods
        RFS je ramcovy synchronizacny signal (Receive Frame Synchronization),
        ktory definuje presne okamihy, kedy prichadzaju platne data. SCLK potom
        definuje jednotlive bity a kedze RFS je aktivne kazdych 20 period SCLK,
        je mozne preniest (aj s rezervou) aj 16-bitove vzorky */
      sport0.rfsdiv = 20;

      /* Nastavenie riadiaceho registra SPORT0 */
      sport0.control.single.isclk       = 1;    // interne sclk
      sport0.control.single.irfs        = 1;    // interne RFS
      sport0.control.single.rfsr        = 1;    // RFS je vyzadovane
      sport0.control.single.slen        = 15;   // 16 bitove slova

      /* Nastavenie autobufrovacich registrov */
      sport0.autobuffer.receive_enable= 1;  // povoli autobfrovanie prijmu od SPORT0
      sport0.autobuffer.rmreg           = 1;  // M3 (=1 prednastavene v C prostredi)
```

```
        sport0.autobuffer.rireg             = 2;    // I2 (register je potrebne rezervovat)

        /* Nastavenie DAG registra i2 pre autobufrovanie a jeho inicializacia tak, aby ukazoval
           na _Buffer0 s dlzkou 512
           toto makro je mozne najst v circ.h */
        circ_setup(2, Buffer0, 512);

        /* start SPORT0 */
        sport_start(0);

        /* Definovanie C funkcie ktora by bola volana pri obsluhe prerusenia. V skutocnosti
           tato funkcia len povoli prislusne prerusenie. Pretoze v tabulke preruseni sme
           priamo definovali vlastnu funckiu, pri obsluhe prerusenie od prijimaca SPORT0 bude
           prerusenie spracovat nasa funkcia */
        interrupt(SIGSPORT0RECV, SPORT0_Receive_Int);

        for(;;)
        {
            asm("idle;");       // znizeny prikon, pokracuje sa po vyvolani a obsluhe prerusenia
            if (DoFFT)
            {
                /* v tomto bode je v jednej z dvoch cirkulacnych pamati nacitany
                   kompletny blok N=512 vzoriek */

                DoFFT = 0;  /* Vynulovanie priznaku (opat ho nastavi obsluha prerusenia
                               po nacitani dalsieho bloku) */
                if (!db_tgl)    // definuje v ktorej cirkulacnej pamati su platne data
                {

                    /* Zakaze autobufrovanie a oznami to obsluhe prerusenia SPORT0
                       nastavenim hodnoty Bit_Reversal_On na 1 */

                    Bit_Reversal_On = 1;
                    sport0.autobuffer.receive_enable = 0;    // zakazanie autobufrovania

                    bexp = fft256(Buffer1, Buffer1_Imag, Real_Out, Imag_Out);

                    /* Opatovne povolenie autobufrovaniaand a oznamenie obsluhe
                       prerusenia nastavenim Bit_Reversal_On value na  0 */

                    sport0.autobuffer.receive_enable = 1;    // povolenie autobuferovania
                    Bit_Reversal_On = 0;

                }
                else    // to iste spracovanie pre Buffer0
                {
                    Bit_Reversal_On = 1;
                    sport0.autobuffer.receive_enable = 0;

                    bexp = fft256(Buffer0, Buffer0_Imag, Real_Out, Imag_Out);

                    sport0.autobuffer.receive_enable = 1;
                    Bit_Reversal_On = 0;
                }

                /* Priklad najdenia maximalnej amplitudy v spektre vypocitanom pomocou FFT
                   Vysledky FFT (realna zlozka) su v nasledujucom vypocte prepisane  !!!*/
                max = 0;
                for (i=0;i<128;i++)
                {
                    Real_Out[i] = Real_Out[i]*Real_Out[i] + Imag_Out[i]*Imag_Out[i];
                    if (Real_Out[i] > max)
                    {
                        max = Real_Out[i];
                        index = i;
                    }
                }

            }
        }
    }
```

1.5 KNIŽNIČNÉ FUNKCIE FFT

Algoritmus FFT je ďalším algoritmom, pre ktorý sú prakticky všetky signálové procesory optimalizované. Algoritmus FFT má široké využitie v číslicovom spracovaní signálov a používa sa na výpočet spektra, rýchlu konvolúciu vo frekvenčnej oblasti a pod. Existuje **veľké množstvo algoritmov výpočtu FFT**, ktoré sa líšia metódou výpočtu (Cooleyov-Tookeyov algoritmus, Winogradov algoritmus, Goertzelov algoritmus, Chirp FFT, …). Medzi základné patrí **Cooleyov-Tookeyov algoritmus**, ktorý bol prebraný v rámci iných špecializovaných predmetov a tento algoritmus tvorí aj základ implementácii pomocou ADSP218xx. V tejto časti sa obmedzíme len na najdôležitejšie skutočnosti, podrobnejšie informácie je možné nájsť v literatúre z iných odborných predmetov, prípadne v [6]

Tento algoritmus využíva rekurzívny rozklad algoritmu FFT s rozmerom N na menšie algoritmy s rozmerom $N/2$ až po rozmer $N=2$, ktorý sa realizuje pomocou tzv. **motýlika** (butterfly). Existujú dva **základné rozklady** – **DIT** (decimácia v časovej oblasti) a **DIF** (decimácia vo frekvenčnej oblasti), ktoré sa líšia preusporiadaním vzoriek v časovej resp. vo frekvenčnej (transformovanej) oblasti. Ich základný princíp je znázornený na nasledujúcich obrázkoch pre RADIX 2 FFT s rozmerom $N=8$.

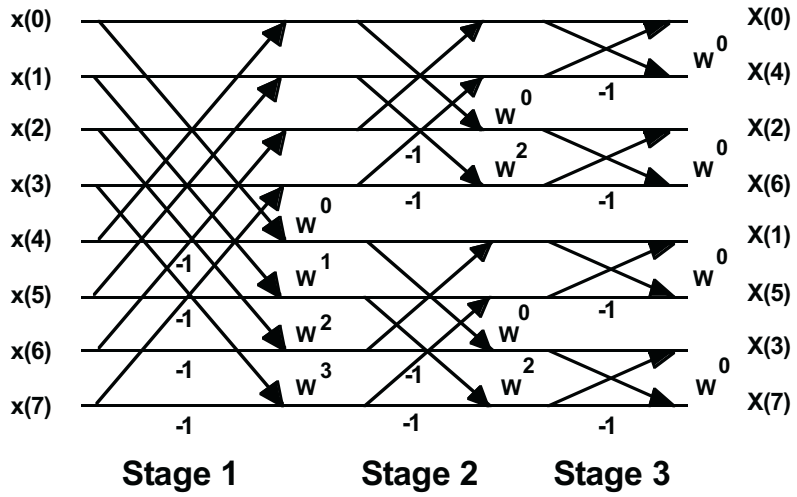


Obr.1 Štruktúra DIT FFT pre N=8

Preusporiadanie je možné opísať pomocou bitovej inverzie, t.j. vstupnému prvku s binárnym indexom $\langle b_2, b_1, b_0 \rangle_2$ zodpovedá výstupný prvok s indexom $\langle b_0, b_1, b_2 \rangle_2$ (napr. prvku $4 = \langle 1,0,0 \rangle_2$ zodpovedá prvok $1 = \langle 0,0,1 \rangle_2$). Práve pre tento spôsob adresovania sú optimalizované obvody adresovej aritmetickej jednotky DAG1 v prípade prepnutia do režimu **s reverzným prenosom** (reverse carry)[7]. Prepnutie adresovej

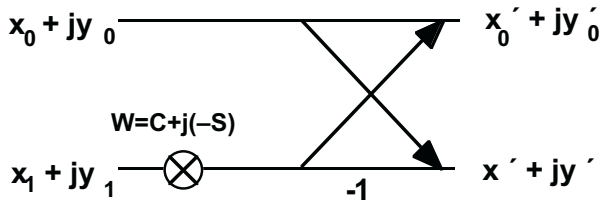
[7] Je to tretí režim DAG1 a využíva sa práve pri preusporiadaní počas výpočtu FFT. Určitým obmedzením procesorov ADSP21xx je skutočnosť, že pokiaľ je DAG1 prepnutá do módu s reverzným prenosom, všetky registre tejto jednotky (I0, I1, I2, I3) sú týmto módom ovplyvnené. Napr. V procesoroch Motorola DSP5600x je možne každému registru adresovej aritmetickej jednotky vnútiť tento mód samostatne.

jednotky DAG1 to tohto módu sa realizuje nastavením príslušného bitu v registri MSTAT.



Obr.2 Štruktúra DIF FFT pre N=8

Základný stavebný blok DIT FFT (ktorá je použitá v knižničných funkciách VisualDSP) – DIT motýlik je zobrazený na obrázku 3.



Obr.3 Štruktúra DIT motýlika

Pri implementácii DIT motýlika v aritmetike s pevnou rádovou čiarkou je hlavným problémom **možný nárast** hodnôt na výstupe motýlika. Aj keď sú všetky vstupné hodnoty $|x_0|,|y_0|,|x_1|,|y_1|<1$ a hodnota $|W|=|e^{-j2\pi/N}|=1$, výstupné hodnoty motýlika **môžu byť väčšie** ako 1 (teoreticky najväčšia hodnota na výstupe je 2,8, takýto nárast však nemôže nastať v dvoch po sebe idúcich motýlikoch). Tento problém sa v praktických implementáciach rieši nasledujúcimi spôsobmi

1. Zmenšením amplitúdy vstupného signálu tak, aby na výstupe nedošlo k pretečeniu výstupných hodnôt. V N-bodovej FFT stačí vstupný signál posunúť o $\log_2 N+1$ bitov doprava[8].

[8] Tento spôsob je využitý aj vo funkcii FFT256, ktorá je využitá v projekte FFT_test.zip. V prípade, že funkcia FFT256 vráti hodnotu rôznu od 0, sú výsledky funkcie nesprávne (zistené pomocou simulácie).

2. Automatickým zmenšením amplitúdy na výstupe každého motýlika na polovicu
3. Zmenšením amplitúdy všetkých motýlikov v sekcii (stage), pokiaľ aspoň jeden motýlik v danej sekcii má výstup väčší ako 1. Tento spôsob využíva reprezentáciu v tzv. **blokovej pohyblivej čiarke** (Conditional Block Floating Point – BFP).

Z pohľadu implementácie je metóda 1. najrýchlejšia, vyžaduje však podstatné zmenšenie vstupných signálov a teda stratu presnosti. Metóda 3. je naopak najpresnejšia, vyžaduje však výrazne dlhší výpočtový čas. Podrobne sú uvedené metódy opísané v [6] na str. 141-251. Počet cyklov pre 1024 bodové realizácie FFT opísané v [6] sú uvedené v nasledujúcej tabuľke[9]

Metóda výpočtu	Počet cyklov
Radix-2 DIT (metóda 1)	52911
Radix-2 DIT (metóda 3)	113482
Radix-4 DIT (metóda 1)	37021

1.6 LADENIE FFT PROJEKTU V PROSTREDÍ VISUALDSP

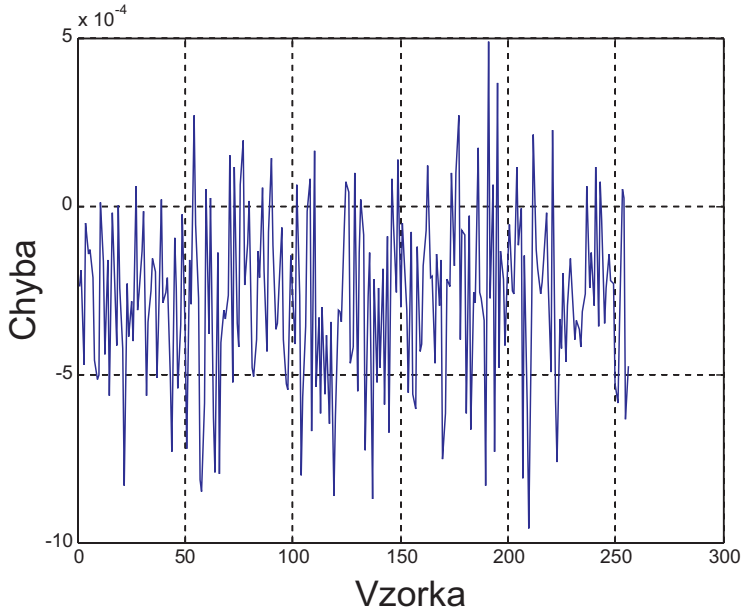
Počas cvičenia bude vykonaná kompletná simulácia programu main.c so vstupnými vzorkami zo súboru **x.dat**, ktoré sú súčasťou projektu[10] [4]. Súbor x.dat je potrebné pripojiť[11] pred začiatkom simulácie k prijímaču portu SPORT0. Jeho prvých 256 hodnôt tvorí reálnu časť a ďalších 256 hodnôt imaginárnu časť vstupu funkcie FFT256. Po výpočte FFT je možné vyčítať reálnu a imaginárnu časť výsledku (súbory **y_real.dat** a **y_imag.dat** v [4]) pomocou položky **\Memory\Dump\...** a uložiť hodnoty výsledku v DM do súborov[12]. Prečítanie výsledkov výpočtu FFT je potrebné realizovať pred určovaním maximálnej hodnoty, pretože **Real_Out[]** je v nasledujúcej časti programu **prepísaná**. Súčasťou projektu je aj program test.m pre Matlab, ktorý umožňuje porovnať výsledky zo simulácie ADSP2181 s referenčným výpočtom FFT filtra v Matlabe. Na nasledujúcom obrázku je zobrazená chyba výpočtu 256 bodovej FFT v procesore ADSP218x. Z obrázku je zrejmé, že maximálna chyba je podstatne väčšia ako 1 LSB $\doteq 2^{-15} = 3.1*10^{-5}$, čo dokumentuje skutočnosť, že počas výpočtu FFT dochádza (čo je predovšetkým dôsledok veľkého počtu matematických operácií realizovaných počas výpočtu FFT) k akumulácii chýb.

[9] Táto tabuľka ukazuje, že implementácia pomocou BPF je takmer 2-krát pomalšia. Implementácia pomocou RADIX 4 realizácie je v ADSP218x výrazne rýchlejšia. Uvedené údaje reprezentujú hodnoty uvedené v [6] a neudávajú hodnoty pre implementáciu FFT v prostredí VisualDSP!

[10] V projekte je použitý prepínač **-reserve=I2,I3** (položka Project\Project Options\Compile), ktorý zabezpečí, že registre i2, i3 nie sú použité samotným kompilátorom a môžu byť použite v optimalizovaných asemblerovských funkciách.

[11] Dáta sú v zlomkovom formáte 1.15. Pri simulácii je **vhodné zaškrtnúť políčko Circular**, čím sa zabezpečí automatický prechod z konca na začiatok súboru x.dat. S touto voľbou je potom možné realizovať ľubovoľne dlhú simuláciu.

[12] VisualDSP uloží do prvého riadku súboru textovú informáciu. Pred načítaním do prostredia Matlab je potrebné tento riadok ručne vymazať.



Obr.4 Chyba výpočtu funkcie FFT256 v procesore ADSP218x

1.7 ZÁVER

Projekt blokového spracovania a FFT preberaný v rámci cvičenia poukázal na možnosti využitia jazyka C a jeho optimalizovaných knižničných funkcií. Aj keď na prvý pohľad je využitie C kompilátora zložitejšie ako predchádzajúce projekty pre FIR a IIR filtre, C prostredie umožňuje okamžité využívanie knižníc bez ich detailnej analýzy[13]. Takto je možné použiť napr. podprogramy FFT, ktoré už predstavujú pomerne zložité funkcie optimalizované v asembleri ADSP218x. Ich detailná analýza by nám zabrala omnoho viac času ako napr. analýza podprogramov pre FIR a IIR filtre a preto sme sa obmedzili len na ich základné vlastnosti. Naviac efektivita výsledného programu je pri blokovom spracovaní relatívne vysoká a celková rýchlosť algoritmu FFT dokumentuje efektívnosť architektúry ADSP218x pri realizácii aj tohto základného algoritmu ČSS.

[13] Je však potrebné upozorniť na to, že knižnice v systéme VisualDSP je možne ďalej vylepšovať a tak s využitím vlastných optimalizovaných knižníc dosiahnuť ešte lepšie výsledky.

LITERATÚRA

[1] ADSP-218x DSP Hardware Reference. Analog Devices, Inc., February 2001 (dostupné aj v elektronickej forme **\VisualDSP\Docs\218x_hwr*.pdf**).

[2] VisualDSP++ 3.0 C Compiler and Library Manual for ADSP-218x DSPs. Analog Devices, Inc.(dostupné aj v elektronickej forme **\VisualDSP\Docs\218x_ccm*.pdf**)

[3] Autobuffering, C and FFTs on the ADSP-218x. Engineer To Engineer Note EE-142. Analog Devices, Inc., 8/2001, www.analog.com/library/applicationNotes/dsp/applicationNotes.html

[4] (dostupné v elektronickej forme **\SPvT\Cvicenia\FFT_test.zip**)

[5] Číslicové filtre a FFT – opakovanie (dostupné v elektronickej forme **\SPvT\Cvicenia\spvt_2cv.pdf**)

[6] Mar, A.: *Digital Signal Processing Applications using the ADSP-2100 Family, Volume 1*. Prentice Hall, Englewood Cliffs, 1992 (dostupné aj v elektronickej forme **\SPvT\Knihy\Using_ADSP-2100\…**)