SIGNÁLOVÉ PROCESORY A IMPLEMENTÁCIA ZÁKLADNÝCH BLOKOV SPRACOVANIA SIGNÁLOV

Fakulta elektrotechniky a informatiky

Miloš Drutarovský

NÁZOV: Signálové procesory a implementácia základných blokov spracovania signálov
AUTOR: doc. Ing. Miloš Drutarovský, CSc.
VYDAVATEĽ: Technická univerzita v Košiciach
ROK: 2017
ROZSAH: 129 strán
NÁKLAD: 50 ks
VYDANIE: prvé
ISBN: 978-80-553-3176-8

Rukopis neprešiel jazykovou úpravou.
Za odbornú a obsahovú stránku zodpovedá autor.

PREDSLOV

Bloky **číslicového spracovania signálov (ČSS)** sú významné stavebné prvky vložených elektronických systémov už po niekoľko desaťročí. Aj keď ich prítomnosť nie je pre koncového užívateľa typicky viditeľná, ich využitie a zložitosť v typických zariadeniach neustále narastá. Napr. typický moderný mobilný telefón obsahuje desiatky blokov ČSS a ich implementácia je v súčasnosti realizovaná v moderných viacjadrových procesoroch a špecializovaných obvodoch ASIC (Application Specific Integrated Circuit). So zvyšujúcou sa hustotou integrácie elektronických súčiastok narastá aj výkonnosť a zložitosť moderných procesorov využívaných vo vstavaných systémoch. Napr. kedysi špecializované jednočipové mikropočítače optimalizované pre číslicové spracovanie signálov – tzv. **signálové procesory** (**DSP** – Digital Signal Processor) sa v súčasnosti začleňujú do mikroprocesorov rôzneho výkonu ako špecializované časti a stavajú sa aj súčasťou výkonnejších jednočipových mikropočítačov (MCU), ktoré sú v posledných rokoch na trhu dostupné. Samozrejmosťou je aj hardvérová podpora blokov ČSS v moderných procesoroch Intel a ARM pre osobné počítače, notebooky a tablety. Úspešné nasadenie blokov ČSS v moderných technických prostriedkoch tak vyžaduje aj zvládnutie špecifických programovacích nástrojov a techník. Medzi najdôležitejšie patrí pochopenie súvislosti medzi technickými prostriedkami a vytváraným programom ako aj zvládnutie špecifických vývojových nástrojov ktoré pre oblasť ČSS a DSP existujú.

Jedným z trendov na vysokých školách je široké využívanie vysoko-úrovňových nástrojov ako napr. Matlab. Tieto nástroje umožňujú rýchly vývoj a testovanie nových algoritmov ČSS a majú nezastupiteľné miesto. Absolventi študijných programov zameraných na elektroniku, telekomunikačnú techniku a počítačové inžinierstvo sú však typicky konfrontovaní s potrebou uplatnenia základných znalostí z programovania v jazyku C, ktoré zvládli na začiatku štúdia, v oblasti programovania algoritmov ČSS. Na to, aby boli pri programovaní moderných technických prostriedkov efektívni aj z pohľadu kvality vytváraného kódu, musia pochopiť a zohľadniť špecifické hardvérové obmedzenia a nároky, ktoré vytváraný program musí spĺňať. Aj keď súčasný efektívny vývoj programov pre uvedené platformy využíva predovšetkým vyššie programovacie jazyky (v súčasnosti predovšetkým jazyk C a C++), zvládnutie aspoň základných princípov programovania v asembleri výrazne zvyšuje flexibilitu a potenciálnu kvalitu vytváraných programov. Je pre nich tiež výhodné ak pochopia aj nízko-úrovňové súvislosti technických prostriedkov pomocou ktorých je vyvíjaný program vykonávaný.

Predkladaný učebný text vznikol na základe skúsenosti autora vo výučbe predmetu Signálové procesory na Katedre elektroniky a multimediálnych telekomunikácií (KEMT) FEI TU v Košiciach. Vychádza zo skúsenosti, že aj napriek vysokému inovačnému tempu vo vývoji DSP, kedy sa na trhu každoročne objavujú nové modely DSP, je výhodné ak študenti počas štúdia pochopia predovšetkým základné princípy a súvislosti, pričom tieto si môžu okamžite otestovať s využitím vhodných vývojových nástrojov vrátane otestovania pomocou reálneho hardvéru. V rámci tohto učebného

materiálu je ako cieľový DSP využívaný produkt firmy Analog Devices ADSP BF533 s jadrom Blackfin a vývojové prostredie VisualDSP++ od toho istého výrobcu. Výber tohto DSP je daný bohatým technickým vybavením, ktorým katedra pre tento typ DSP už dlhodobo disponuje. Aj napriek tomu, že architektúra ADSP BF533 je viac ako desaťročie stará, DSP s jadrom Blackfin patria medzi stále aktívne vyvíjané a podporované DSP (http://www.analog.com/en/products/processors-dsp/blackfin.html). DSP s jadrom Blackfin sú optimalizované a nasadzované práve vo vstavaných aplikáciách. Samotná firma Analog Devices zaradzuje procesory s jadrom Blackfin medzi procesory, ktoré integrujú vlastnosti DSP a MCU.

Vzhľadom na to, že samotné jadro Blackfin resp. periférie procesorov ADSP firmy Analog Devices sú dostatočne opísané v množstve existujúcich dokumentov, nie je cieľom tohto učebného textu opisovať detaily procesorov ADSP a jadra Blackfin. Cieľom predloženého učebného textu je ukázať s využitím ADSP BF533 ako je možné úspešne realizovať základné bloky ČSS – číslicové filtre FIR (Finite Impulse Response), IIR (Infinite Impulse Response) a algoritmus FFT (Fast Fourier Transform) pomocou špecializovaného programovateľného hardvéru. Na KEMT sú FIR a IIR filtre ako aj algoritmus FFT preberané v špecializovaných predmetoch typicky s využitím vysoko-úrovňových nástrojov ako Matlab a LabView. Predložený učebný text si kladie za cieľ posunúť tieto skúsenosti bližšie k cieľovej hardvérovej platforme, rozšíriť praktické skúsenosti študentov s programovaním v jazyku C a demonštrovať tiež špecifické súvislosti, ktoré pri nasadení uvedených metód ČSS je potrebné v praxi zohľadňovať.

Košice, máj 2017 Miloš Drutarovský

OBSAH

ÚVOD

Číslicové spracovanie signálov (ČSS) sa počas uplynulých desaťročí zmenilo z vedeckej disciplíny, ktorá bola známa len úzkemu okruhu pracovníkov z univerzít a výskumných ústavov na všeobecne známy pojem, ktorý významným spôsobom ovplyvňuje náš každodenný život. Nárast využívania elektronických súčiastok využívajúcich princípy číslicového spracovania signálov v *spotrebnej elektronike*, *telekomunikačných zariadeniach* a *počítačovej technike* je tak výrazný, že ČSS je v súčasnosti jednou z hlavných síl rozvoja polovodičového priemyslu. Tento výrazný nárast v rozsahu využitia a výkonnosti týchto súčiastok má pôvod v rozvoji obvodov veľmi vysokej hustoty integrácie (VLSI –Very Large Scale Integration, ULSI – Ultra Large Scale Integration) a exponenciálnom náraste úrovne integrácie známom ako Moorov zákon.

Počas predchádzajúcich desaťročí sa úroveň integrácie VLSI obvodov zvyšovala štvornásobne každé tri roky, pričom hodinová frekvencia týchto obvodov taktiež neustále rastie. Tieto faktory priamo ovplyvňovali a stále ovplyvňujú výkonnosť číslicových integrovaných obvodov a je predpoklad, že tento nárast sa udrží minimálne aj v nasledujúcom desaťročí. Po dosiahnutí určitej hustoty integrácie bolo koncom 70-tych rokov možné vytvoriť monolitický čip – *číslicový signálový procesor* (DSP – Digital Signal Processor), ktorý významným spôsobom ovplyvnil prenikanie metód číslicového spracovania do technickej praxe. Možnosť monolitickej integrácie bola hlavným faktorom, ktorý umožnil významné zníženie ceny, príkonu a poruchovosti zariadení využívajúcich číslicové spracovanie signálov a tým aj ich masové nasadenie.

Úroveň mikroelektroniky však nie je jediným faktorom, ktorý ovplyvňuje rozvoj súčiastok a zariadení na báze ČSS. Číslicové spracovanie signálov sa stalo samostatnou vedeckou disciplínou a naďalej sa veľmi rýchlo rozvíja. Bola rozpracovaná samostatná teória algoritmov ČSS a nájdené efektívne postupy pre ich efektívny výpočet. Pre súčasné obdobie je charakteristická vzájomná súvislosť *teórie algoritmov* ČSS a *architektúr ULSI obvodov*. Tieto dve oblasti sa vzájomne podnecujú a doplňujú, pričom výsledky tejto vzájomnej symbiózy sa najviac prejavujú pri rozpracovaní paralelných algoritmov ČSS na pravidelne sa opakujúce základné elementy, ktoré sa dajú pomocou ULSI čipov s využitím dostupnej technológie pomerne jednoducho realizovať.

Teória paralelných počítačových systémov je známa a dobre rozpracovaná už niekoľko desaťročí. Veľmi populárne sú napr. *paralelné architektúry* SIMD, MIMD a MISD. Teória ČSS tiež významným spôsobom prispela do oblasti paralelných systémov rozpracovaním *systolických architektúr* už na začiatku 80-tych rokov.

Počas viac ako štyridsaťročného vývoja sa monolitické DSP vyvíjali spočiatku klasickým evolučným vývojom, pričom vývoj vychádzal predovšetkým z technologických zlepšení dostupnej CMOS technológie a prejavoval sa zmenšovaním dĺžky inštrukčného cyklu a zväčšovaním interných pamätí. V polovici deväťdesiatych rokov minulého storočia začal vývoj DSP nadobúdať (minimálne z hľadiska integrácie nových prvkov) znaky revolučného rozvoja a bol charakteristický predovšetkým väčším

využívaním paralelizmu s cieľom zvýšiť priepustnosť systémov na báze DSP. Existuje niekoľko základných spôsobov využívania paralelizmu, ktoré sa v súčasnosti v oblasti DSP uplatňujú. Je to jednak zvýšenie paralelizmu integrovaním viacerých funkčných jednotiek na jeden čip ako aj trend využívania predovšetkým SIMD. V poslednom desaťročí sa bloky pôvodne využívané len v DSP začali integrovať aj do obvodov MCU ako aj do výkonných procesorov firiem ako napr. Intel a ARM. Výpočtová výkonnosť týchto kategórií procesorov z pohľadu algoritmov ČSS sa tak posúva na úroveň, ktorá bola pred niekoľkými rokmi nepredstaviteľná. Princípy navrhnuté a využívané pôvodne pre DSP tak majú v súčasnosti podstatne širšie využitie. V ďalšej časti textu však budeme na označenie cieľovej programovateľnej platformy používať označenie DSP.

Využívanie paralelizmu je z pohľadu počítačových architektúr značne prepracovaná metóda zvyšovania výpočtového výkonu, v oblasti čipov pre vstavané aplikácie je však potrebné splniť veľmi špecifické požiadavky. Medzi najkritickejšie patria minimalizácia príkonu a množstva využívaných pamätí (umožňuje zníženie ceny čipu), čo vyžaduje výraznú modifikáciu už existujúcich princípov, prípadne aj vývoj úplne nových riešení.

Neoddeliteľnou súčasťou programovateľných DSP je ich *programové vybavenie*, ktoré je prostriedkom na *transformáciu* algoritmov ČSS do vykonateľného programu pre programovateľné DSP. Základným cieľom pri využívaní DSP je *efektívne mapovanie algoritmov* ČSS do architektúry DSP a programové prostriedky sú prostriedkom na dosiahnutie tohto cieľa. Vzhľadom na špecifickú architektúru týchto technických prostriedkov je často potrebné pôvodné algoritmy ČSS optimalizovať pre cieľovú architektúru DSP. Úspešné praktické využitie DSP tak vyžaduje zvládnutie troch relatívne samostatných oblastí:

- architektúr DSP,
- ich programového vybavenia,
- teórie ČSS.

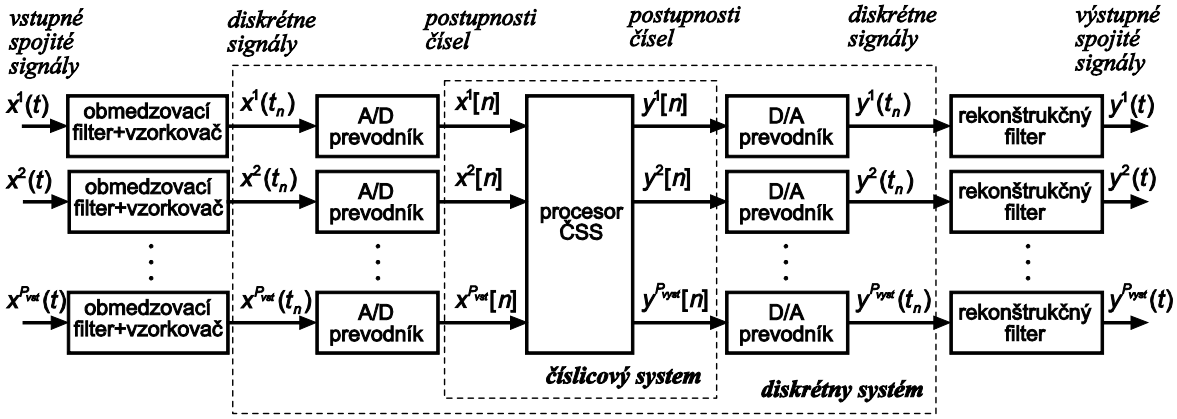
Predkladaný učebný text sa snaží o priblíženie tejto činnosti študentom tak, aby demonštroval využitie získaných znalostí z predmetov ako Programovanie, Signály a sústavy, Číslicové spracovanie signálov a Mikroprocesorová technika na mapovanie základných algoritmov ČSS to cieľového DSP hardvéru. Učebný text je rozdelený do jedenástich kapitol. Prvé tri kapitoly opisujú základnú architektúru procesora ČSS, niektoré charakteristické parametre týchto procesorov, historický vývoj DSP a základné možnosti využitia paralelizmu v moderných DSP. Štvrtá kapitola opakuje základné informácie a návrhové postupy pre návrh číslicových filtrov. Piata až deviata kapitola sú venované práci s vývojovým prostredím VisualDSP++ pre DSP s jadrom Blackfin ako aj implementácii FIR a IIR filtrov na DSP s jadrom Blackfin. Záverečné dve kapitoly sú venované algoritmu FFT a možnostiam jeho optimalizácie. Ďalšie informácie a kódy implementovaných projektov je možné nájsť na stránke predmetu Signálové procesory, ktorá je uvedená v zozname použitej literatúry.

1 SYSTÉMY ČÍSLICOVÉHO SPRACOVANIA SIGNÁLOV

Číslicové spracovanie signálov hlboko preniklo a intenzívne preniká do najrôznejších odborov ľudskej činnosti. Tomu výrazne napomáha technologický rozvoj súčiastkovej základne, vývoj nových architektúr a návrhových postupov. Napríklad klasické číslicové obvody sú nahradzované hradlovými poliami (PLD – Programmable Logic Device) a užívateľom rekonfigurovateľnými obvodmi (FPGA – Field Programmable Gate Array). Sú zlepšované parametre používaných štandardných polovodičových technológií (CMOS) a zavádzané nové materiály pre výrobu veľmi rýchlych číslicových obvodov. Vzniká rad nových zákazníckych, polozákazníckych a špeciálnych obvodov ako sú napr. grafické a multimediálne, signálové procesory, FFT procesory, programovateľné číslicové filtre a pod. Cieľom tejto kapitoly je definovať základné pojmy a všeobecné charakteristiky systémov na báze ČSS.

1.1 ZÁKLADNÝ MODEL ČÍSLICOVÉHO SPRACOVANIA SIGNÁLOV

Aj keď systémy ČSS môžu mať rôznorodé stvárnenie, model znázornený na Obr. 1.1 je dostatočne všeobecný na opísanie širokej triedy algoritmov ČSS.



Obr. 1.1 Model číslicového spracovania signálov

Analógové signály[1] $x^i(t),\quad i=1,2,\ldots,P_{vst}$ vstupujú do *obmedzovacích filtrov*, ktoré predstavujú analógové dolné priepusty s medznou frekvenciou F_m^i a ich úlohou je

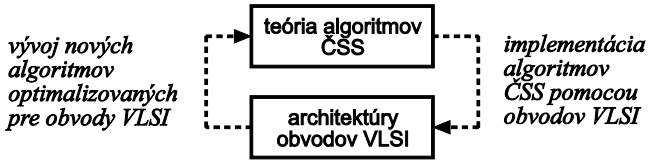
[1] Počet vstupných analógových signálov je závislý na konkrétnej aplikácii a môže byť aj nulový (napr. v prípade digitálnej syntézy).

obmedziť frekvenčný rozsah spracovávaných analógových signálov tak, aby po ich vzorkovaní nedošlo k prekrytiu spektra (tzv. aliasing). Filtrované signály sú v blokoch *analógovo-číslicových* (A/D) *prevodníkov* konvertované na číslicové vzorky $x^i[n]$. Rýchlosť vzorkovania je riadená frekvenciou vzorkovania F_{vz}^i, ktorá musí podľa Shanon-Koteľnikovho teorému splňovať podmienku:

$$F_{vz}^i > 2F_m^i, \qquad i = 1,\ldots,P_{vst} \tag{1.1}$$

Číslicové vzorky vstupujú do *procesora* ČSS, ktorý realizuje algoritmus číslicového spracovania signálov, ktorý môže byť lineárny alebo nelineárny, časovo invariantný alebo závislý od času a generuje výstupné číslicové vzorky $y^j[n], \quad j = 1,\ldots,P_{vyst}$. Procesor ČSS je možné definovať ako *číslicový procesor*, ktorý dokáže vykonávať všetky matematické operácie využívané realizovaným algoritmom ako napr. sčítania, odčítania, násobenia a pod. V prípade potreby[2] je možné konvertovať číslicové signály $y^j[n]$ na zodpovedajúce analógové signály $y^j(t)$ pomocou *číslicovo-analógových* (D/A) *prevodníkov* a *rekonštrukčných filtrov*, ktorých úlohou je potlačiť zrkadlové kmitočty nad medznými frekvenciami jednotlivých kanálov.

V teórii algoritmov ČSS, napr. pri tvorbe nových algoritmov ČSS je často možné pracovať s procesorom ČSS ako s abstraktným číslicovým procesorom. Typickým príkladom je algoritmus výpočtu rýchlej Fourierovej transformácie (FFT), ktorý bol v podstate objavením rýchleho matematického algoritmu na výpočet diskrétnej Fourierovej transformácie (DFT). Praktické nasadenie metód ČSS však výrazne ovplyvňujú praktické možnosti reálnych technických prostriedkov. Pre súčasné obdobie je dokonca charakteristická vzájomná súvislosť teórie algoritmov ČSS a architektúr VLSI obvodov, ktorá je znázornená na obr. 1.2 .



Obr. 1.2 Vzájomné ovplyvňovanie teórie algoritmov ČSS a architektúr obvodov VLSI

Táto vzájomná súvislosť sa prejavuje napr. tvorbou nových algoritmov ČSS, ktoré pre dostupné technické prostriedky minimalizujú počet matematických operácií a prístupov do pamätí. Na druhej strane požiadavky technológie VLSI vytvorili aj nové smery v oblasti teórie algoritmov ČSS. Typickým predstaviteľom je napr. vývoj algoritmov pre systolické polia. Základné vlastnosti reálnych systémov ČSS, ktoré umožňujú ich široké praktické využitie ako aj ich základné charakteristiky sú uvedené v nasledujúcej časti.

1.2 VLASTNOSTI A CHARAKTERISTIKY REÁLNYCH SYSTÉMOV ČSS

Kľúčovým predpokladom pre úspešné (masové) nasadenie systémov na báze ČSS je poskytnutie vlastností, ktoré nie je možné dosiahnuť s použitím analógových

[2] Existujú aj systémy, ktoré využívajú len číslicový výstup (napr. systém rozpoznávania reči s textovým výstupom).

technológií, resp. poskytnutím riešenia, ktorého cena je nižšia ako cena porovnateľného analógového riešenia. Medzi základné výhodné vlastnosti systémov na báze ČSS patria:

- ***Stabilita a necitlivosť na vplyv prostredia***. Číslicové systémy sú principiálne menej citlivé na vplyv prostredia, čo sa plne prejavuje aj v systémoch ČSS. Okrem toho využitie metód ČSS umožňuje znížiť vplyv prostredia aj na klasicky analógové bloky systému ČSS ako sú obmedzovacie a rekonštrukčné filtre a A/D a D/A prevodníky.
- ***Predikovateľnosť a opakovateľnosť***. Použitím systémov ČSS je možné eliminovať vplyv tolerancií súčiastok na cieľovú funkciu zariadenia. Najjednoduchším príkladom sú rôzne typy frekvenčne selektívnych filtrov, ktorých analógová implementácia je extrémne náročná.
- ***(Re)programovateľnosť***. Niektoré konštrukčné riešenia procesorov ČSS umožňujú definovanie, resp. zmenu funkcie systému ČSS zmenou *programu*. Táto vlastnosť má významný vplyv na efektívne využitie systémov ČSS. Je napr. možné využiť jeden systém ČSS na implementáciu značne odlišných zariadení, ktoré v optimálnom prípade vyžadujú len zmenu programu. Táto vlastnosť sa ukazuje ako *komerčne veľmi výhodná* a umožňuje napr. distribuovať zariadenia na báze ČSS k zákazníkom ešte pred schválením finálnych verzií noriem, podľa ktorých majú tieto zariadenia pracovať. Po prijatí noriem je možné realizovať potrebné zmeny priamo u zákazníka (alebo samotným zákazníkom). V súčasnosti, keď je doba prijímania nových (napr. telekomunikačných) štandardov rýchlejšia, než doba životnosti zariadení, je táto možnosť veľmi výhodná.
- ***Nízky príkon***. Pokrok v technológii VLSI umožňuje realizovať vzrastajúci počet systémov na báze ČSS, ktoré poskytujú minimálne rovnaké funkčné možnosti ako porovnateľné analógové systémy, pri podstatnom znížení príkonu. Táto vlastnosť umožňuje napr. u prenosných zariadení predlžovať napr. dobu medzi výmenou batérií a tým zvyšuje ich *úžitkovú hodnotu*. Z globálneho hľadiska nezanedbateľným efektom je aj *úspora energie*.
- ***Spoľahlivosť***. Zvyšovanie hustoty integrácie číslicových obvodov umožňuje znižovanie počtu súčiastok (integrovaných obvodov), ktoré sú potrebné na realizáciu cieľového zariadenia. Cieľom (v mnohých prípadoch realistickým) je realizovať *jednočipové riešenie* celého systému, čo zvyšuje spoľahlivosť výsledného zariadenia.
- ***Nízka cena***. Všeobecný trend znižovania ceny číslicových obvodov umožňuje zvyšovanie úžitkovej hodnoty zariadení na báze ČSS pri zachovaní ceny a často ju dokonca umožňuje znižovať. Podstatný vplyv tu zohráva aj znižovanie výrobných nákladov využívaním univerzálnych (často reprogramovateľných) súčiastok.
- ***Nové algoritmy***. Algoritmy ČSS umožňujú realizovať aj zariadenia, ktoré nemajú analógový ekvivalent. Ako príklad je možné uviesť filtre s konečnou impulzovou odpoveďou (FIR – Finite Impulse Response), metódy kompresie rečových, audio a video signálov, prípadne algoritmy z oblasti zabezpečovacích kanálových kódov.

Existuje niekoľko základných charakteristík, ktoré sú spoločné pre všetky systémy ČSS:

a) Algoritmus

Systémy ČSS sú často charakterizované použitým *algoritmom*. Algoritmus špecifikuje vykonávané aritmetické operácie, často však nešpecifikuje ako sú tieto operácie

implementované. Tieto môžu byť implementované napr. pomocou štandardných mikroprocesorov, programovateľných signálových procesorov, alebo pomocou zákazníckych obvodov. Výber implementačnej technológie je do značnej miery určovaný potrebnou rýchlosťou aritmetických operácií a potrebnou *aritmetickou presnosťou*.

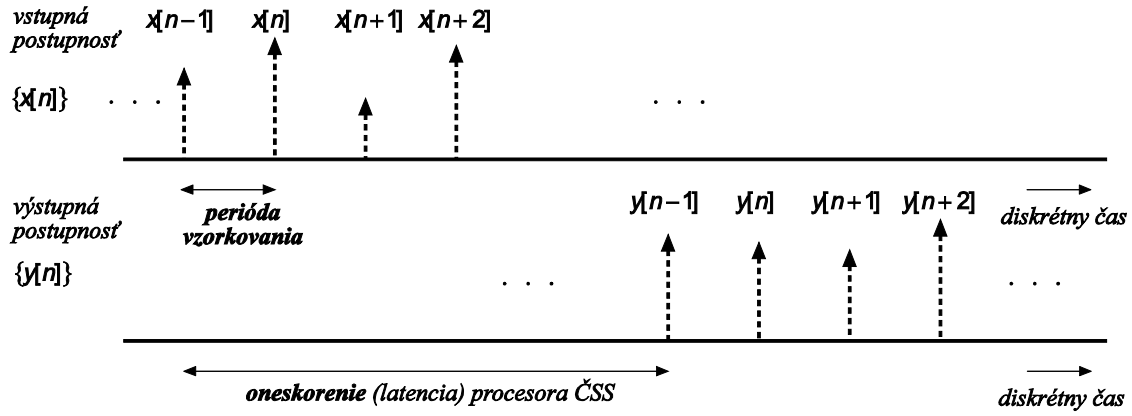
Medzi základné algoritmy ČSS s významným praktickým využitím patria:

- číslicová filtrácia,
- rýchla Fourierova transformácia,
- diskrétna konvolúcia a korelácia.

Teória týchto základných algoritmov je dobre prepracovaná a existujú efektívne postupy pre ich výpočet. Oblasť algoritmov ČSS je v súčasnosti veľmi rozsiahla a základné algoritmy ČSS sú na KEMT preberané v špecializovaných predmetoch.

b) Frekvencia vzorkovania

Jednou z kľúčových charakteristík systémov ČSS je *frekvencia vzorkovania* – t.j. rýchlosť s akou prichádzajú vstupné vzorky, resp. s akou sú generované výstupné vzorky. Spolu so zložitosťou implementovaného algoritmu ČSS, určuje frekvencia vzorkovania rýchlosť potrebnej hardvérovej resp. implementačnej technológie. Na obr. 1.3 sú znázornené vstupné a výstupné vzorky procesora ČSS s jedným vstupom $x[n]$ a jedným výstupom $y[n]$.



Obr. 1.3 Frekvencia vzorkovania a oneskorenie procesora ČSS

Časový interval medzi príchodom za sebou idúcich vzoriek vstupného signálu sa nazýva *perióda vzorkovania* T_{vz}, pričom pre frekvenciu vzorkovania F_{vz} platí vzťah

$$F_{vz} = \frac{1}{T_{vz}} \tag{1.2}$$

Časový interval medzi príchodom vstupnej vzorky a zodpovedajúcou výstupnou vzorkou je definovaný ako *výpočtové oneskorenie* (computational latency). V typických aplikáciách je minimálna dosiahnuteľná perióda vzorkovania zvyčajne dôležitejším faktorom ako oneskorenie procesora ČSS. Samozrejme systém ČSS môže využívať (a v

praxi aj veľmi často využíva) viac ako jednu frekvenciu vzorkovania a takýto systém sa nazýva *viacrýchlostný* (multirate) systém ČSS.

Z hľadiska praktických aplikácií je často veľmi dôležité, aby spracovanie signálov bolo realizované v *reálnom čase*. Tento pojem vyjadruje takú činnosť procesora ČSS, pri ktorej sa z postupnosti vstupných vzoriek spracuje každá vzorka. Táto definícia má rôzne dôsledky pre *rekurzívne* a *blokové algoritmy* výpočtu.
Medzi rekurzívne algoritmy výpočtu patria algoritmy, ktoré pracujú spôsobom:

- čítanie vstupnej vzorky $x[n]$ z A/D prevodníka,
- výpočet výstupnej vzorky $y[n]$ v procesore ČSS,
- zápis výstupnej vzorky $y[n]$ do D/A prevodníka,

a ich typickým predstaviteľom je napr. číslicová filtrácia[3]. Pokiaľ trvá realizácia výpočtu v procesore ČSS kratšiu dobu, než je perióda vzorkovania T_{vz}, potom je výpočet realizovaný v reálnom čase. V opačnom prípade sa niektoré vzorky vstupnej postupnosti nespracujú, sú vynechané a systém nepracuje v reálnom čase.

Blokové algoritmy, ako vyplýva z názvu, spracovávajú vstupné vzorky po blokoch a nie jednotlivo. Typickým predstaviteľom je napr. algoritmus FFT, ktorý zahrňuje v blokovom intervale $[b]$ tri operácie:

- zozbieranie bloku $[b]$ s dĺžkou N vstupných vzoriek,
- výpočet FFT z predchádzajúceho bloku $[b-1]$,
- výstup výsledkov výpočtu FFT z bloku $[b-2]$.

Pokiaľ trvá výpočet jedného bloku s N vstupnými vzorkami dobu kratšiu, než je jeho načítanie, je výpočet realizovaný v reálnom čase.

Rozsah frekvencií vzorkovania, ktoré sa používajú v prakticky využívaných systémoch ČSS je obrovský a presahuje 12 rádov. Samozrejme algoritmy pre najvyššie (aktuálne realizovateľné) frekvencie vzorkovania patria z hľadiska zložitosti medzi najjednoduchšie, pretože cena a zložitosť technickej realizácie s rastúcou frekvenciou vzorkovania prudko narastajú. Požiadavka na stále vyššiu výpočtovú výkonnosť procesorov ČSS je jedným z faktorov, ktoré vytvárajú tlak na okamžité využívanie pokroku v technológii obvodov VLSI na zvyšovanie výkonnosti procesorov ČSS.

c) Hodinová (taktovacia) frekvencia
Štandardné číslicové elektronické systémy sú charakterizované ich hodinovou frekvenciou, ktorá obvykle vyjadruje rýchlosť, s akou číslicový systém realizuje najzákladnejšiu operáciu v systéme. Pre systémy ČSS pomer

$$P_{frek} = \frac{hodinová\ \ frekvencia\ systému}{vzorkovacia\ \ frekvencia} \tag{1.3}$$

charakterizuje, ako je možné systém ČSS implementovať. Táto hodnota charakterizuje predovšetkým množstvo a typ technických prostriedkov potrebných na realizáciu algoritmu ČSS s určitou zložitosťou, ktorý by pracoval v reálnom čase. Pre hodnoty $P_{frek} \to 1$ je jedinou možnosťou využitie paralelizmu technických prostriedkov. Naopak

[3] Číslicovú filtráciu je však možné realizovať aj blokovými algoritmami napr. s využitím rýchlej konvolúcie, prípadne využitím špeciálnych paralelných štruktúr.

pre hodnoty $P_{frek} >> 1$ je možné využiť napr. štandardné programovateľné číslicové procesory.

d) Číslicová reprezentácia

Jadrom algoritmov ČSS sú predovšetkým operácie sčítania, odčítania a násobenia. Veľké množstvo algoritmov ČSS je možné efektívne realizovať pomocou tzv. MAC (Multiply and ACcumulate) operácie, ktorá realizuje matematickú operáciu:

$$A = B + C * D \tag{1.4}$$

Medzi základné číslicové reprezentácie v oblasti procesorov ČSS patrí reprezentácia *v pevnej rádovej čiarke* (fixed point representation), pričom v oblasti ČSS je preferovaná jej tzv. *zlomková reprezentácia* (fractional representation) v tvare M bitov (dĺžka slova) $b_m \in \{0,1\}$, $m = 0,1,2,\ldots,M-1$, ktorá vyjadrujú číslo $x_{pevná} \in \langle -1,1)$ v tvare

$$x_{pevná} = (-1)b_0 + \sum_{m=1}^{M-1} b_m 2^{-m} \tag{1.5}$$

ktorý reprezentuje záporné čísla v druhom doplnku. Čísla mimo intervalu $\langle -1,1)$ nie je možné v zlomkovej reprezentácií reprezentovať a prípadné výsledky mimo intervalu $\langle -1,1)$ sú v procesoroch ČSS najčastejšie *obmedzené* (saturated), t.j. nadobúdajú najväčšiu kladnú $(1-2^{-M})$, alebo najmenšiu zápornú hodnotu (-1). Menej časté je potlačenie dodatočných bitov, ktoré vzniknú pri pretečení aritmetických operácií (wrap around mode), ktoré je typické pre systémy ČSS na báze štandardných univerzálnych procesorov.

Reprezentácia v *pohyblivej rádovej čiarke* (floating point representation) výrazne zvyšuje dynamický rozsah (t.j. pomer medzi najväčším a najmenším reprezentovateľným číslom) použitím mantisy a exponentu a vyjadrením čísla v tvare

$$x_{pohyblivá} = mantisa \times 2^{\exp onent} \tag{1.6}$$

Tento formát bol vzhľadom na široké využitie štandardizovaný v norme IEEE 754.a jej novšej verzii IEEE 754 – 2008. Táto reprezentácia výrazne znižuje pravdepodobnosť *pretečení*, *podtečení* a *potreby zmeny mierky*, ktoré sú typické pre reprezentáciu v pevnej rádovej čiarke, čo sa odráža v zjednodušení návrhu algoritmov a ich realizácii pomocou programovateľných procesorov ČSS. Nevýhodou tejto reprezentácie je zložitejšia konštrukcia aritmetickej jednotky. V praktických aplikáciách je často možné využiť reprezentáciu v tzv. *blokovej pohyblivej čiarke*, ktorá využíva spoločný exponent pre blok (vektor) dát a využíva výhodné vlastnosti obidvoch predchádzajúcich systémov.

Okrem týchto klasických reprezentácií sa v procesoroch ČSS často využívajú aj menej tradičné číselné reprezentácie a aritmetiky a to predovšetkým

- distribuovaná aritmetika (DA – Distributed Arithmetic),
- číselný systém zvyškových tried (RNS – Residue Number System),
- aritmetika v konečných poliach (FFA – Finite Field Arithmetic).

Podstatou DA a RNS je odstránenie potreby využívať zložitú paralelnú násobičku a možnosť využitia paralelizmu, ktoré tieto netradičné aritmetiky poskytujú. Tieto prístupy nachádzajú uplatnenie predovšetkým v oblasti špecializovaných jednoúčelových procesorov ČSS a umožňujú dosahovať rádovo vyššie rýchlosti ako

klasické aritmetiky, ktoré využívajú zložité paralelné násobičky, pri relatívne malej zložitosti a ploche čipu procesora ČSS. Ukazuje sa, že DA a RNS je možné s úspechom uplatniť napr. v procesoroch ČSS na báze rekonfigurovateľných obvodov FPGA.

Uplatnenie FFA v procesoroch ČSS je v súčasnosti aktuálne predovšetkým pri implementácii blokových zabezpečovacích kódov ako aj vybraných kryptografických algoritmov, pričom je však možné realizovať v tejto aritmetike aj klasické algoritmy ČSS. Tieto typy algoritmov sa v súčasných, predovšetkým telekomunikačných aplikáciách široko využívajú.

e) Výkonnosť a paralelizmus

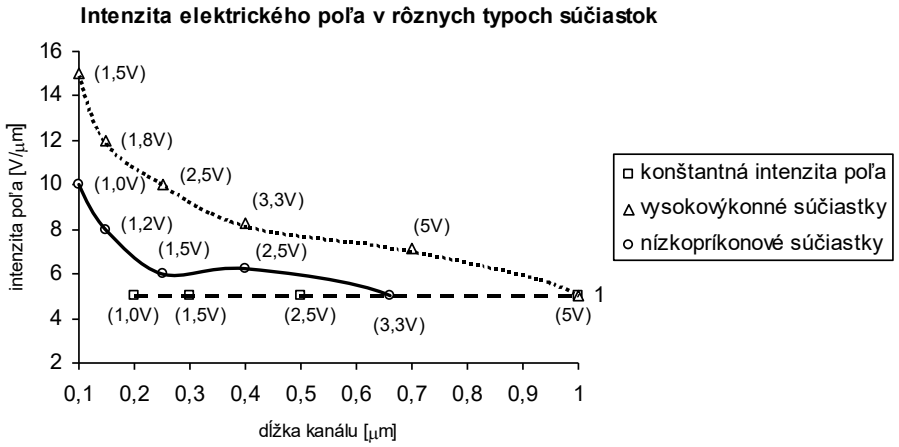
Ďalšie zvyšovanie výpočtovej výkonnosti procesorov ČSS je a (v minimálne blízkej) budúcnosti bude realizované predovšetkým rozsiahlejším využívaním *paralelizmu*, čo je spôsobené rýchlejším nárastom hustoty integrácie oproti nárastu hodinovej frekvencie obvodov VLSI.

f) Príkon

Príkon procesorov ČSS (na realizáciu konkrétneho algoritmu) sa vo všeobecnosti znižuje, čo sa prejavuje zníženou úrovňou napájacieho napätia vo všeobecnosti a samozrejme aj celkového príkonu. Rýchlosť obvodov taktiež priebežne narastá. Tento trend je možné pozorovať v obidvoch, z pohľadu optimalizácie CMOS VLSI obvodov (dosiahnutej rozdielnou selektívnou zmenou mierky) odlišných oblastiach:

- ***vysokovýkonné*** (high performance) súčiastky,
- ***nízkopríkonové*** (low power) súčiastky,

čo je znázornené na obr. 1.4 .



Obr. 1.4 Nárast intenzity elektrického poľa, U / L , v závislosti na dĺžke kanálu pre rôzne typy súčiastok CMOS

Typickým predstaviteľom nízkopríkonových súčiastok CMOS sú práve monolitické DSP optimalizované pre telekomunikačné aplikácie a spotrebnú elektroniku. Do kategórie vysokovýkonných súčiastok patria napr. procesory pre personálne počítače.
Na porovnanie rôznych procesorov ČSS z pohľadu príkonu sa veľmi často používa charakteristika

$$\frac{príkon}{operácia} \qquad \{\text{mW/MIPS, mW/MOPS, mW/MFOPS…}\} \tag{1.7}$$

prípadne vyjadrenie spotreby prúdu na operáciu pri udanej hodnote napájacieho napätia, čo bude zapisované v tvare $mA/operáciu\ @\ napätie$. Skratky MIPS, MOPS, MFOPS znamenajú milióny inštrukcií, operácií resp. operácií v pohyblivej rádovej čiarke za sekundu. Táto charakteristika síce do značnej miery odráža úroveň použitej technológie zahrňuje však v sebe aj vplyv použitej architektúry a je preto vhodná predovšetkým na porovnanie procesorov ČSS s rovnakou architektúrou. Z pohľadu implementovaného algoritmu je často používaná charakteristika

$$\frac{príkon}{aloritmus} \qquad \{\text{mW/FFT, mW/CELP…}\} \tag{1.8}$$

pričom algoritmus musí byť presne špecifikovaný (napr. rozmer FFT, typ CELP kodéra a pod.).

g) Ekonomický aspekt
Analýza dlhodobých údajov ukazuje, že cena kremíkových čipov na jednotku plochy (samozrejme pri použití aktuálnej technológie) vyjadrená v $\$/cm^2$ mierne narastá a dosahuje hodnoty $5-6\,\%$ za rok, čo zabezpečuje neustály pokles ceny čipov vykonávajúcich ekvivalentnú funkciu. Na porovnanie ceny rôznych čipov sa používa charakteristika

$$\frac{operácia}{cena} \qquad \{\text{MIPS/\$, MOPS/\$, MFOPS/\$…}\} \tag{1.9}$$

Táto charakteristika, spolu s predchádzajúcimi umožňuje zhodnotiť vhodnosť konkrétnych technických prostriedkov pre využitie v konkrétnom zariadení a v praxi sa veľmi často využívajú.

d) Integrácia
Exponenciálny nárast v počte integrovaných prvkov v čipoch VLSI umožní realizovať kompletné systémy ČSS na jednom monolitickom čipe, pričom takéto jednočipové systémy budú obsahovať na čipe aj A/D a D/A prevodníky a vstupno-výstupné senzory. V ďalšej fáze budú do monolitickej formy integrované kompletné systémy (tzv. SoC – System on Chip), ktoré budú využívať systémy ČSS na zvýšenie úžitkovej hodnoty a tým aj zvýšenie ich podielu na trhu. Aspekt masovosti je pre rozvoj technológie VLSI extrémne dôležitý, pretože veľké výrobné náklady je nutné amortizovať do veľkého počtu vyrobených čipov. Takúto masovosť nasadenia môžu zabezpečiť predovšetkým *spotrebná elektronika, automobilový priemysel, počítačová* a *telekomunikačná technika.* Je možné očakávať zvyšujúci sa podiel prenosných zariadení, ktoré budú založené na nízkopríkonových modifikáciách VLSI technológie.

1.3 KLASIFIKÁCIA TECHNICKÝCH PROSTRIEDKOV PRE ČSS

Implementačná báza systémov pre ČSS je veľmi široká a ponúka možnosti využitia technických prostriedkov v rôznych kvalitatívnych, kvantitatívnych a cenových

oblastiach. Neexistujú jednoznačné kritéria pre klasifikáciu procesorov ČSS. Tieto by sa dali klasifikovať napr. podľa stupňa integrácie použitých technických prostriedkov, konštrukcie aritmetickej jednotky, použitej aritmetiky, výkonnosti, ceny, príkonu a pod. Žiadne z týchto hľadísk však nevystihuje to podstatné, t.j. spôsob riadenia procesora ČSS. Spôsob riadenia určuje predovšetkým flexibilitu procesora ČSS a ukazuje sa ako dostatočne univerzálne hľadisko pre klasifikáciu procesorov ČSS. Podľa spôsobu riadenia sa procesory ČSS delia do troch skupín:

1. Procesory s pevnou logikou
Do tejto skupiny patria procesory ČSS u ktorých je riadenie algoritmu spracovania signálov realizované *pevným obvodovým zapojením* jednotlivých stavebných prvkov. Podľa technologickej úrovne môžu tieto prvky predstavovať diskrétne obvody na báze technológie LSI alebo VLSI, prípadne štandardné bunky obvodov ASIC. Nevýhodou procesorov tejto skupiny je *malá flexibilita*, pretože zmena algoritmu sa realizuje zmenou obvodového zapojenia. Ich výhodou je však vysoká rýchlosť a relatívna jednoduchosť, pretože obsahujú iba prvky nevyhnutné pre realizáciu vybraného algoritmu ČSS. Do tejto skupiny patria špecializované, jednoúčelové zákaznícke a polozákaznícke procesory ako napr. FFT procesory, FIR filtre, MPEG dekodéry, modemové čipové sady a pod.

2. Programovateľné procesory
Tieto procesory majú program realizovaného algoritmu ČSS uložený *v reprogramovateľnej inštrukčnej pamäti* a sú *značne flexibilné*. Zmena implementovaného algoritmu ČSS sa realizuje preprogramovaním inštrukčnej pamäte. Programovateľné procesory sa delia do troch skupín:

- ***Procesory z diskrétnych funkčných blokov***. Medzi diskrétne funkčné bloky patria násobičky, sčítačky, mikroprocesorové rezy, viacportové pamäte, radiče programov a pod. Vzhľadom na pokrok technológie VLSI sú tieto procesory využívané len vo veľmi špeciálnych aplikáciách, kde je požadovaná vysoká rýchlosť, pričom otázka ceny a príkonu je druhoradá. V súčasnosti je však možné dosiahnuť podstatne väčšiu výkonnosť vytvorením týchto blokov v štruktúrach moderných FPGA obvodov.
- ***Jednočipové signálové procesory (DSP)***. Obsahujú na čipe všetky obvody potrebné na implementáciu veľkej triedy algoritmov ČSS. Procesory tejto skupiny sú vzhľadom na ich univerzálnosť značne rozšírené. Ich relatívne vysoká rýchlosť je dosiahnutá pomocou *optimalizovanej architektúry* a *optimalizovanej redukovanej inštrukčnej sady*. Procesory z tejto kategórie sú vyrábané viacerými významnými výrobcami ako napr. Texas Instruments, NXP, Analog Devices ako bežne dostupné súčiastky. Z nárastom integrácie sa však čoraz častejšie stretávame s aj DSP blokmi začleňovanými do obvodov SoC.
- ***Univerzálne procesory***. Vzhľadom na svoju univerzálnosť môžu tieto procesory realizovať aj algoritmy ČSS (zložitosť implementovateľných algoritmov ČSS sa s technologickým pokrokom samozrejme mení). Výkonnosti jednočipových signálových procesorov môžu konkurovať typicky len pri rádovo vyššom príkone a vyššej cene.

3. Iné procesory
Do tejto skupiny je možné zaradiť procesory ČSS v ktorých je vykonávanie algoritmov vykonávané netradičným spôsobom. Patria sem napr. procesory riadené tokom dát (data

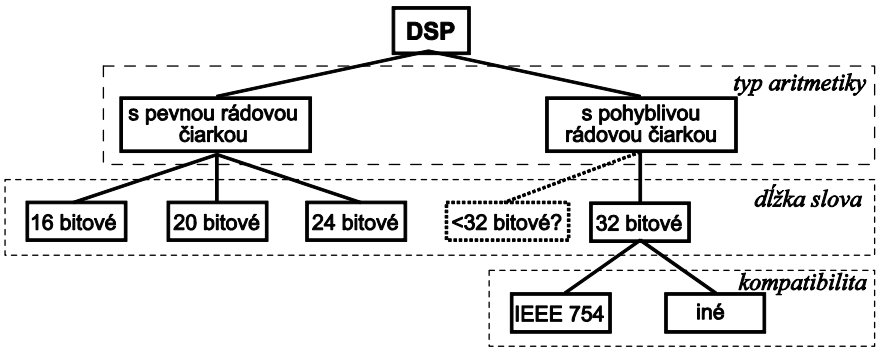
flow processors), systolické, celoparalelné a neurónové procesory, ktoré dosahujú vysokú rýchlosť spracovania predovšetkým s využitím masívneho paralelizmu. S pokrokom technológie VLSI je možné očakávať aj implementáciu tejto skupiny procesorov v monolitickej forme, pričom už v súčasnosti existujú špecializované čipy využívajúce niektoré z uvedených princípov. Pre širšie nasadenie týchto systémov bude veľmi dôležité *umožnenie programovateľnosti* týchto procesorov, prípadne zahrnutie týchto procesorov ako špecializovaných koprocesorov v rámci moderných programovateľných procesorov.

2 KLASICKÉ SIGNÁLOVÉ PROCESORY

Na určitom stupni vývoja integrácie polovodičovej technológie (na konci 70-tych rokov) bolo možné integrovať do monolitického čipu všetky základné stavebné bloky procesora ČSS a vznikol programovateľný monolitický signálový procesor – DSP, ktorého charakteristické vlastnosti:

- špecializovaná architektúra,
- krátky inštrukčný cyklus,
- malý ale výkonný inštrukčný súbor,
- nízka spotreba,
- zbernicová orientácia,
- malé rozmery,
- vysoká spoľahlivosť,

mu získali veľmi rýchlo významnú pozíciu v oblasti technických prostriedkov pre ČSS. Architektúra DSP sa síce postupne vylepšovala a prenikla aj do architektúr štandardných procesorov, približne do polovice 90-tych rokov sa však stále používali DSP s jednou aritmetickou jednotkou. Takéto DSP budú označované v ďalšom texte ako *klasické* DSP, prípadne len DSP. Z hľadiska použitej aritmetiky, ktorá výrazne ovplyvňuje vhodnosť využitia DSP pre konkrétnu aplikáciu je možné komerčne dostupné DSP rozdeliť do hierarchickej štruktúry zobrazenej na obr. 2.1 .



Obr. 2.1 Rozdelenie komerčných DSP na základe numerickej reprezentácie

V tejto kapitole je opísaný historický vývoj, základné stavebné bloky a modifikácie architektúr klasických DSP s orientáciou na segment DSP s pevnou rádovou čiarkou, ktoré sú dominantné pre aplikácie v telekomunikačnej technike a spotrebnej elektronike. Súčasťou tejto kapitoly je aj uvedenie základných pojmov a princípov využívaných v architektúrach DSP, ktoré budú využívané v ďalších častiach.

2.1 HISTORICKÝ VÝVOJ

DSP boli logickým vyústením dvoch trendov v oblasti procesorov ČSS, ktoré boli využívané koncom 70-tych rokov. Prvý trend bol vývoj špeciálnych jednoúčelových jednočipových procesorov ČSS určených pre špecifické aplikácie. Druhý trend bol vývoj multičipových systémov ČSS predovšetkým pre vysokovýkonné vojenské aplikácie, ktoré boli zvyčajne založené na technológii bitových rezov. Tieto dva trendy spolu s pokrokom v oblasti mikroprocesorovej techniky viedli k vytvoreniu DSP.

Prvý DSP – AMI S2811 bol ohlásený v roku 1978, ale nebol komerčne dostupný počas niekoľkých nasledujúcich rokov vzhľadom na problémy s VMOS technológiou. Prvým komerčne dostupným DSP sa tak stal v roku 1979 procesor Intel i2920. Koncepčne zaujímavou vlastnosťou na tomto procesore bola integrácia jedného vstupného 9-bitového A/D a jedného výstupného D/A prevodníka spolu so vstupnými a výstupnými analógovými multiplexormi pre 4 vstupné a 8 výstupných kanálov. Tie umožňovali aj principiálnu realizáciu viackanálového spracovania analógových signálov v jednočipovom vyhotovení. Novšie DSP s integrovanými A/D a D/A prevodníkmi (aj keď na kvalitatívne vyššej úrovni s využitím prevodníkov na báze *sigma-delta modulácie*) sa objavili až o niekoľko rokov neskôr. Intel i2920 umožňoval realizovať jednočipové riešenie kompletného systému ČSS pomocou čipu s 28 vývodmi. Procesor však nemal *hardvérovú násobičku*, ktorá sa stala typickým znakom všetkých novších DSP, čo sa odrazilo v jeho rýchlom nahradení novšími typmi DSP. Najväčším prínosom i2920 bolo, že ukázal na medzeru v technických prostriedkoch, ktorú rýchlo vyplnili iné firmy generáciou DSP, ktoré už mali architektúry podstatne lepšie prispôsobené k požiadavkám systémov ČSS. Zaujímavým faktom je tiež skutočnosť, že firma Intel po uvedení i2920 svoje aktivity v oblasti komerčných DSP na nasledujúcich 30 rokov zastavila.

Prvým komerčne dostupným DSP s hardvérovou násobičkou bol japonský NEC7720 uvedený na trh v roku 1980. Čip využíval *harvardskú architektúru*, ktorá sa stala ďalšou charakteristickou vlastnosťou klasických DSP. Jeho aritmeticko logická jednotka (ALU) však ešte nepodporovala saturačnú aritmetiku. Procesor NEC7720 umožňoval vykonávanie inštrukcií len z vnútornej PROM (EPROM) pamäte čo bolo pre zložitejšie aplikácie značným obmedzením.

V roku 1980 firma AT&Tvyvinula maskou programovateľný DSP1 vychádzajúci ešte z klasickej v*on Neumanovej architektúry*, ktorý však bol určený len pre interné účely.

Prvý DSP umožňujúci vykonávanie inštrukcií z externe pripojenej pamäte bol TMS32010 od firmy Texas Instruments (TI) vyrobený v roku 1982. Možnosť pripojenia externej programovej pamäte posunula DSP z hľadiska *programovacieho modelu* bližšie k univerzálnym mikroprocesorom. Táto vlastnosť spolu s veľkým dôrazom kladeným na vývojové prostriedky a optimalizované knižnice, ktorými je firma TI známa, viedli k masovému rozšíreniu DSP.

Vyššie uvedené DSP využívali aritmetiku v pevnej rádovej čiarke. Ďalším logickým krokom bolo využitie aritmetiky s pohyblivou rádovou čiarkou v procesore Hitachi HD61810 z roku 1982, ktorý využíval 12-bitovú mantisu a 4-bitový exponent. Aj keď niektoré úvahy naznačovali, že využitie aritmetiky s pohyblivou rádovou čiarkou, ktorá by využívala skrátenú mantisu a exponent by mohlo byť zaujímavé, dĺžka slova v tejto triede DSP sa veľmi rýchlo ustálila na hodnote 32 bitov. V roku 1984 firma AT&T uviedla na externý trh svoj prvý komerčne dostupný DSP s pohyblivou rádovou

čiarkou využívajúci 24-bitovú mantisu a 8-bitový exponent. Prvý DSP, ktorý podporoval aritmetiku IEEE 754 bol v roku 1987 procesor MB86232 od firmy Fujitsu.

V druhej polovici 80-tych rokov vstúpili na trh s DSP aj firmy Motorola a Analog Devices so svojimi DSP DSP56001 a ADSP2100.

Z hľadiska naznačeného historického vývoja sú klasické DSP často klasifikované do *troch generácií*, pričom vybraní predstavitelia jednotlivých generácií sú uvedení v tab. 2.1 . DSP z prvej a druhej generácie pracovali s pevnou rádovou čiarkou, pričom DSP z druhej generácie mali nižšiu spotrebu (využívali už technológiu CMOS), boli rýchlejšie a mali väčšiu kapacitu vnútorných a podporovaných vonkajších pamätí. DSP z tretej generácie používali aritmetiku s pohyblivou rádovou čiarkou, často obsahovali interné radiče DMA a boli prispôsobené pre prácu v multiprocesorových systémoch.

Tab. 2.1 Klasifikácia vybraných DSP

Generácia	Výrobca - typ (rok oznámenia)			
1. generácia	INTEL i2920 (1979)	NEC μPD7720 (1980)	TI TMS32010 (1982)	NEC μPD7725 (1988)
2. generácia	TI TMS320C25 (1985)	NEC μPD77220 (1985)	AT&T DSP16A (1988)	MOTOROLA DSP56001 (1988)
3. generácia	NEC μPD77230 (1986)	TI TMS320C30 (1987)	AT&T DSP32A (1988)	MOTOROLA DSP96002 (1988)

Tento, v staršej literatúre veľmi často používaný spôsob klasifikácie DSP, však môže viesť k chybným záverom. Skutočne, koncom 80-tych rokov bol často uvádzaný názor, že s postupom úrovne integrácie obvodov VLSI budú DSP s pevnou rádovou čiarkou v mnohých praktických aplikáciách nahradené DSP z tretej generácie, t.j. procesormi s pohyblivou rádovou čiarkou. To malo predovšetkým uľahčiť tvorbu programov minimalizovaním problémov s pretečením. Tento predpoklad sa však nesplnil a DSP s pevnou rádovou čiarkou sa vyvíjajú rýchlejšie (minimálne z hľadiska počtu typov a vyrobených kusov) ako DSP s pohyblivou rádovou čiarkou. Napr. firma Motorola, jeden z popredných svetových výrobcov DSP tej doby, po uvedení DSP 96002– svojho prvého DSP s pohyblivou rádovou čiarkou (bol dokonca jedným z prvých DSP, ktorý pracoval so štandardom IEEE 754) už počas ďalších troch desaťročí nevyrobila žiadny nový DSP s pohyblivou rádovú čiarkou. Počas tejto doby sústredila svoj výrobný a vývojový potenciál do oblasti DSP s pevnou rádovou čiarkou a vyvinula a vyrobila viac ako dve desiatky rôznych typov DSP tejto kategórie. Podobne aj ostatní svetoví výrobcovia DSP, ktorí tvoria tzv. veľkú štvorku[1] (okrem Motoroly ešte Lucent, TI a Analog Devices) vyrábajú nepomerne viac kusov a typov DSP s pevnou rádovou čiarkou. Toto platí aj v súčasnosti viac ako tridsať rokov po uvedení prvého DSP s pohyblivou rádovou čiarkou. Tento stav je možné *zdôvodniť požiadavkami trhu po stále lacnejších a energeticky efektívnejších DSP*, ktoré sa používajú v produktoch s veľkou sériovosťou. Menšia zložitosť a vo všeobecnosti *kratšia dĺžka slova* týchto procesorov, ktoré priamo vplývajú na *nižšiu cenu a príkon* DSP, predstavujú stále dominantný

[1] Niektorí výrobcovia menili v priebehu rokov svoje názvy a predávali segment DSP iným výrobcom. Typickým príkladom bola napr. Motorola. Segment DSP sa časom oddelil a vznikla firma Freescale. Aktuálne portfólio firmy Freescale odkúpila a vyrába firma NXP.

faktor. Otázka vyšších nákladov na programové prostriedky, ktoré sa amortizujú do veľkého počtu výrobkov je až druhoradou otázkou (aj keď stále dôležitou).

DSP s pohyblivou rádovou čiarkou nachádzajú uplatnenie predovšetkým v oblasti špecializovaných vysokovýkonných paralelných systémov ČSS, ako aj pri nízko sériovej výrobe zariadení na báze ČSS, kde otázka nákladov na vývoj programových prostriedkov je kritickým faktorom. V posledných rokoch sa DSP s pohyblivou rádovou čiarkou presadzujú aj v oblasti spracovania audio-signálov, kde je v súčasnosti preferovaný 32-bitový formát.

2.2 ARCHITEKTÚRY PROGRAMOVATEĽNÝCH DSP

Medzi najčastejšie používané algoritmy ČSS patria číslicová filtrácia a výpočet FFT. Základná architektúra DSP bola prakticky od začiatku ich vzniku vytvorená s cieľom dosiahnuť vysokú rýchlosť implementácie práve pre tieto dva typy algoritmov. Najjednoduchším číslicovým filtrom je FIR filter, ktorý je možné opísať matematickým vzťahom

$$y[n] = \sum_{k=0}^{N-1} h_k x[n-k] \tag{2.1}$$

pričom N je rád FIR filtra, h_k sú koeficienty filtra a $x[n]$, $y[n]$ sú vstupné resp. výstupné vzorky FIR filtra v diskrétnom čase n. Je zrejmé, že FIR filter je možné efektívne implementovať pomocou operácie MAC. Podobne je možné pomocou tejto operácie a špeciálneho rozkladu implementovať aj algoritmus FFT. Vzhľadom na potrebu maximálnej výpočtovej rýchlosti, prakticky všetky významné klasické DSP využívajú harvardskú architektúru s oddeleným spracovaním inštrukcií a dát, prípadne jej modifikácie, umožňujúce zvýšenie rýchlosti s využitím paralelizmu, ktoré táto architektúra poskytuje.

2.2.1 SÚBEŽNÉ SPRACOVANIE

Súbežné spracovanie (concurrent processing) je všeobecnou základnou metódou umožňujúcou výrazné zvýšenie výkonnosti technických prostriedkov a má v DSP dve základné formy – *paralelizmus* a *zreťazenie*, ktoré sú schematicky znázornené na obr. 2.2 a charakterizované nasledujúcimi vlastnosťami:

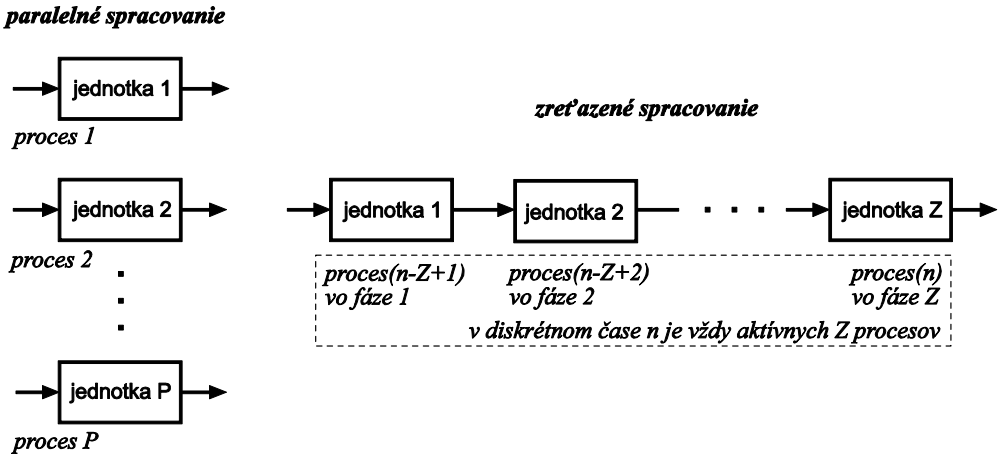
a) Paralelizmus
Ako paralelný sa označuje systém, v ktorom môže prebiehať niekoľko procesov súčasne (paralelne)[2]. Dôvod na uplatnenie paralelizmu v číslicových systémoch je treba hľadať predovšetkým v snahe zvyšovať ich výkonnosť nad hranicu, ktorá je realizovateľná aktuálne použitou úrovňou technológie VLSI obvodov. Vzhľadom na trendy vývoja technológie VLSI je využitie paralelizmu v moderných DSP jedným z kľúčových faktorov zvyšovania ich výkonnosti.

b) Zreťazenie
Spracovávanie informácie je obvykle realizované ako postupnosť jednoduchých úkonov – elementárnych operácií. Tieto úkony je možné zoradiť tak, aby na seba nadväzovali a

[2] Táto definícia paralelného systému je veľmi všeobecná a závisí na pojme proces. V tejto kapitole bude proces vyjadrovať spracovávanie inštrukcie prípadne dát v DSP. V ďalších kapitolách však procesom bude napr. aj časť úlohy ako napr. číslicová filtrácia vo filtračnom koprocesore.

vzájomne sa doplňovali v súlade s požadovaným algoritmom spracovania. Určitý operand[3] (inštrukcia alebo dáta) tak postupne prechádza postupnosťou transformácií, pričom každá operácia musí čakať na ukončenie všetkých operácií, ktoré ju predchádzajú. Pokiaľ sa jeden operand postupne spracováva v Z rôznych blokoch, je v optimálnom prípade možné súčasné spracovanie pomocou všetkých Z blokov tak, že v dobe, keď bol prvý operand spracovaný v prvom bloku a prechádza do druhého bloku, začne prvý blok spracovávať druhý operand, atď. V určitom okamihu sa tak bude v každom bloku spracovávať iný operand, ktorý bude po ukončení spracovania odovzdaný na ďalšie spracovanie do bloku s vyšším poradovým číslom. Takýto spôsob spracovania sa označuje termínom *zreťazené spracovanie* (pipelining).

Zreťazené spracovanie sa v procesorovej technike široko využíva, vzhľadom na možnosť zvýšenia výkonnosti pri relatívne nízkych nákladoch (obyčajne umiestnením vhodných vyrovnávacích registrov). Zreťazenie však môže výrazne skomplikovať programovanie procesora a pri konštrukcii je snaha tento vplyv eliminovať.



Obr. 2.2 Dve formy súbežného spracovania

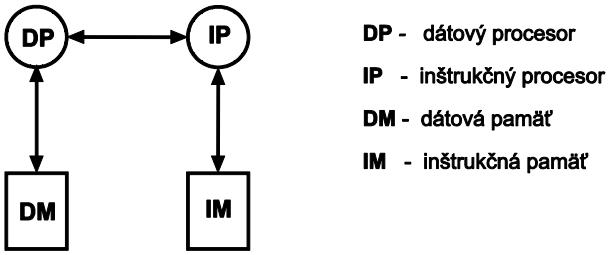
2.2.2 HARVARDSKÁ ARCHITEKTÚRA

V technike je veľmi častým javom, že niektoré princípy boli známe omnoho skôr ako došlo k ich širokému využitiu. Vývoj harvardskej architektúry patrí do tejto kategórie. Architektúra prvého významného elektromechanického počítača mala oddelené pamäte pre program a dáta, čím bol umožnený súčasný prístup k týmto pamätiam. Táto architektúra bola vyvinutá na konci 30-tych rokov Howardom Aikenom z Harvardskej univerzity a prvý počítač Harvard Mark 1 bol skonštruovaný v roku 1944. Na báze harvardskej architektúry pracoval aj známy počítač ENIAC (Electronic Numerical Integrator and Calculator) postavený v rokoch 1943 – 1946 na Pennsylvanskej univerzite. Základná štruktúra harvardskej architektúry je znázornená na obr. 2.3 . *Inštrukčný procesor* (IP) je funkčná jednotka, ktorá interpretuje inštrukcie a riadi *dátový procesor* (DP), ktorý modifikuje príslušné dáta. Obe tieto funkčné jednotky majú

[3] Zreťazeným spôsobom môže byť spracovávaná inštrukcia programu v inštrukčnom procesore tak aj dáta v dátovom procesore. Tieto procesory sú základnými blokmi harvardskej architektúry.

samostatné pamäte. *Dátová pamäť* (DM) uchováva dáta pre DP *a inštrukčná pamäť* (IM) uchováva inštrukcie pre IP.

Vzhľadom na zložitosť systému s oddelenými zbernicami však získala v počiatkoch vývoja procesorov omnoho väčší význam *von Neumannova architektúra*, ktorá dostala názov po jednom z konzultantov ENIAC projektu, matematikovi maďarského pôvodu, ktorého meno bolo John von Neumann**Error! Bookmark not defined.**. Táto architektúra s unifikovanou programovou a dátovou pamäťou, aj keď principiálne pomalšia, sa stala štandardom vo vývoji najrozšírenejších počítačových architektúr na viac ako 40 nasledujúcich rokov po jej publikovaní v roku 1946.



Obr. 2.3 Základná harvardská architektúra

DSP však vzhľadom na potrebu vysokej výpočtovej výkonnosti využili potenciálne rýchlejšiu, aj keď zložitejšiu, harvardskú architektúru, pričom v oblasti DSP je možné identifikovať niekoľko jej základných modifikácií.

2.2.3 MODIFIKÁCIE HARVARDSKEJ ARCHITEKTÚRY

V oblasti DSP využívané modifikácie harvardskej architektúry je možné klasifikovať predovšetkým z pohľadu organizácie pamätí. Cieľom jednotlivých modifikácií je *zvýšenie priepustnosti* DSP pri súčasnej snahe *optimalizovať* ďalšie parametre ako je *plocha čipu, príkon a zložitosť riadenia*. Základným cieľom je umožniť viacoperandové[4] inštrukcie pri minimalizácii nárokov na rýchlosť resp. počet pamätí v DSP. Hlavný dôraz je kladený na umožnenie dvojoperandových inštrukcií, čo vyplýva z faktu, že napr. pri implementácii FIR filtra (2.1) (základného algoritmu pre ktorý sú DSP optimalizované) je potrebné čítať dva vstupné operandy (koeficient h_k a vstupnú vzorku $x[n-k]$ N-krát častejšie ako zápis výstupnej vzorky). Podobné úvahy je možné využiť aj pre FFT. Samozrejme možnosť realizácie troj a viacoperandových inštrukcií je vítaná, je však zvyčajne spojená s dodatočnými nárokmi na rýchlosť resp. zložitosť DSP.

V klasických DSP sa využívalo niekoľko základných modifikácií harvardskej architektúry:

a) Základná harvardská architektúra

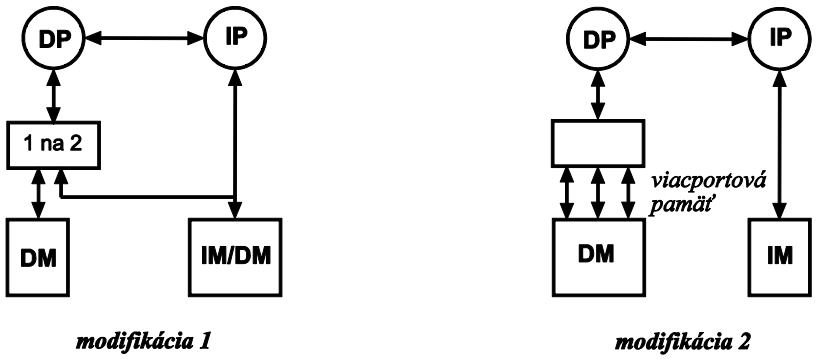
Táto architektúra (obr. 2.3) využíva jednu DM a jednu IM. Každá z pamätí využíva samostatné adresové a dátové zbernice. Výber inštrukcie z IM a dát z DM je realizovaný paralelne. Pre jednoslovné inštrukcie je dĺžka inštrukčného cyklu *rovná*

[4] Pri týchto úvahách sú uvažované operandy umiestnené v pamätiach DSP a nie v jeho registroch, pretože práve tieto operandy vyžadujú zvýšenú priepustnosť zbernicového systému.

dobe prístupu do pamäte. Príkladom DSP, ktorý využíval túto architektúru je TMS320C10 od TI.

b) Modifikácia 1

Modifikácia 1 zobrazená na obr. 2.4 umožňuje uloženie dát a inštrukcií v IM. Inštrukcie a dáta nemôžu byť v prípade dvojoperandových inštrukcií získané paralelne. Táto nevýhoda je kompenzovaná využitím pamätí, ktorých doba prístupu je *rovná polovici inštrukčného cyklu*. Na základe tejto vlastnosti je možné za optimálnych podmienok realizovať dokonca v jednom inštrukčnom cykle aj trojoperandové inštrukcie. Túto modifikáciu využíval napr. procesor DSP32C od AT&T, ktorý obsahuje dve banky RAM pamätí a dátovú zbernicu, ktorá v jenom inštrukčnom cykle umožňuje 4 prístupy (po 2 do/z každej pamäte RAM) a teda DSP32C umožňuje realizovať výber dvoch operandov, MAC operáciu a zápis výsledku[5] v jednom inštrukčnom cykle, pokiaľ operandy, inštrukcia a výsledok sú vhodne umiestnené v RAM pamätiach.



Obr. 2.4 Harvardská architektúra – modifikácia 1 a modifikácia 2

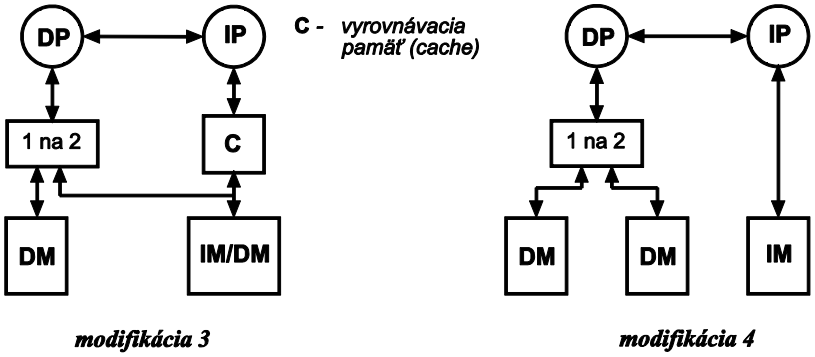
c) Modifikácia 2

V prípade, že DM je *viacportová pamäť* (multi-port memory), je možné realizovať niekoľko prístupov v rámci jedného inštrukčného cyklu. Táto modifikácia je znázornená na obr. 2.4 . Príkladom DSP, ktorý využíval trojportovú DM je MB86232 od Fujitsu čo umožňuje tomuto procesoru realizovať trojoperandové inštrukcie v jednom inštrukčnom cykle. Vo všeobecnosti sú viacportové pamäte drahé a sú, vzhľadom na obmedzenia počtu vývodov, prakticky realizovateľné len v rámci interných pamätí.

d) Modifikácia 3

V prípade modifikácie 1 nastáva konflikt, pokiaľ je potrebné pristúpiť súčasne k dátam a inštrukcii, ktoré sú umiestnené v IP. Dodatočná *vyrovnávacia pamäť* (cache) použitá v modifikácii 3 na obr. 2.5 umožňuje uloženie inštrukcií a realizáciu *opakovaných krátkych slučiek* (veľmi typických pre algoritmy ČSS) bez opakovaného čítania inštrukcií z IM a teda IM môže byť využitá pre druhý operand. Túto modifikáciu využívajú napr. DSP16 od AT&T a ADSP2100 od Analog Devices, ktoré využívajú vyrovnávacie pamäte pre 15 resp. 16 inštrukcií, čo potvrdzuje optimalizáciu pre veľmi krátke segmenty programov.

[5] Vzhľadom na využívanie zreťazenia v IP je to výsledok niektorej (v závislosti na úrovni zreťazenia) z predchádzajúcej operácie.



Obr. 2.5 Harvardská architektúra – modifikácia 3 a modifikácia 4

e) Modifikácia 4
Namiesto využitia vyrovnávacej pamäte niektoré DSP využívajú oddelenú IM a dve samostatné DM. Táto modifikácia bola využitá napr. v DSP5600x a DSP96002 od Motoroly. DSP využívajúce túto modifikáciu môžu realizovať prístup k dvom údajom a jednej inštrukcii aj v prípade, ak je prístup k pamäti rovný inštrukčnému cyklu. Táto modifikácia, znázornená na obr. 2.5 sa niekedy zvykne označovať ako *duálna harvardská architektúra*.

2.3 ZÁKLADNÉ BLOKY SIGNÁLOVÝCH PROCESOROV

Architektúry typických DSP obsahujú niekoľko paralelných blokov, ktoré síce vzájomne veľmi úzko spolupracujú, realizujú však výrazne odlišné činnosti. Pri podrobnejšej analýze je výhodné tieto bloky opísať samostatne.

2.3.1 DÁTOVÉ CESTY

Dátové cesty (data paths) sú miestom, kde sa realizujú základné aritmetické a logické operácie s dátami. Dátové cesty typických DSP obsahujú paralelnú násobičku, *aritmeticko-logickú jednotku*[6] (ALU – Arithmetic Logic Unit), jeden alebo viac posúvačov (shifters), operačné registre, akumulátory a prípadné ďalšie špecializované jednotky.

a) Násobička
Je základným stavebným blokom DSP. Prakticky všetky typy DSP môžu poskytnúť výsledok násobenia v každom inštrukčnom cykle. Niektoré DSP však využívajú interné zreťazenie, čo spôsobuje, že výsledok je dostupný až o niekoľko cyklov neskôr a teda dosahujú výkonnosť jedného násobenia/inštrukciu len pre dlhú opakovanú postupnosť násobení. Niektoré typy DSP majú násobičku integrovanú spolu so sčítačkou a realizujú priamo MAC operáciu (napr. DSP5600x), iné typy (napr. DSP16xx od AT&T) využívajú oddelenú násobičku a sčítačku a tak výsledok MAC operácie je oneskorený o jednu inštrukciu. Násobičky typicky poskytujú výsledok, ktorý má dvojnásobnú dĺžku ako vstupné operandy a v prípade operácie MAC je výsledok akumulovaný v *akumulátoroch*, ktorých dĺžka je zväčšená o tzv. *ochranné bity* (typicky 8 bitov, menej

[6] Niektorí výrobcovia označujú termínom ALU všetky časti dátových ciest, v tomto dokumente je termínom ALU označovaná aritmetická jednotka na realizáciu sčítania, odčítania a logických operácií.

často 4 bity), čo vedie k typickej dĺžke akumulátorov 40 bitov (16-bitové DSP) a 56 bitov (24-bitové DSP).

b) ALU
Táto jednotka vykonáva základné aritmetické operácie (sčítanie, odčítanie, inkrementovanie, dekrementovanie, negovanie) a logické operácie (logický súčin, súčet a negáciu). Dĺžka operandov pre aritmetické a logické operácie môže byť rovná dĺžke akumulátora (napr. DSP16xx), alebo môže byť len časťou akumulátora (napr. DSP5600x). Niektoré typy DSP používajú ALU aj na realizáciu MAC operácie, iné typy DSP pre tento účel využívajú oddelenú sčítačku (napr. ADSP21xx).

c) Posúvač (shifter)
Posúvač je v DSP možné typicky nájsť bezprostredne za násobičkou a ALU, niektoré typy DSP umožňujú posúvať aj vstupné dáta týchto jednotiek. Typická je možnosť realizovať posun výsledku o 1 bit doľava, doprava alebo posun nerealizovať, pričom je operácia posunu realizovaná paralelne s operáciami ALU a násobičky. Klasické viacbitové posuny vyžadujú väčší počet inštrukcií čo niektoré DSP nahradzujú využívaním špeciálnej jednotky *viacbitového posúvača* (barrel shifter).

2.3.2 RIADIACA JEDNOTKA

Typické riadiace jednotky DSP využívajú viacúrovňové zreťazenie, najčastejšie je základné *trojúrovňové* (výber inštrukcie, dekódovanie a vykonanie), prípadne *štvorúrovňové* (výber inštrukcie, dekódovanie, výber operandu a vykonanie). Existujú však aj DSP s dvoj a päťúrovňovým zreťazením. Vo všeobecnosti čím vyššia úroveň zreťazenia, tým je možné dosiahnuť vyšší výkon DSP, výrazne sa však komplikuje programovanie DSP. V DSP sú využívané nasledujúce modely riadenia zreťazenia:

a) Časovo stacionárny (time stationary) model
Inštrukcie procesora *špecifikujú činnosť výkonných jednotiek* (násobička, ALU adresové jednotky…) v každom inštrukčnom cykle. Typickým príkladom je DSP16xx, pre ktorý má MAC inštrukcia tvar

```
a0=a0+p        p=x*y          y=*r0++x=*pt++
```

a vykoná akumuláciu predchádzajúceho súčinu (p), vynásobí obsahy registrov x, y a uloží výsledok do registra p. Zároveň načíta obsahy pamätí uložených na adresách r0 a pt do registrov x,y a inkrementuje registre r0 a pt. Každá časť inštrukcie pracuje s odlišnými operandami.

b) Dátovo stacionárny (data stationary) model
Inštrukcie procesora *špecifikujú operácie, ktoré sa majú vykonať*, ale nešpecifikuje čas, kedy sa tieto operácie vykonajú. Typickým príkladom je kód pre DSP32xx

```
a1=a1+(*r5++ = *r4++) * *r3++
```

ktorý zabezpečí vynásobenie dát uložených v pamäti na adrese r3 a r4, uloží obsah pamäťového miesta s adresou r4 na adresu r5, inkrementuje registre r3, r4, r5 a výsledok súčinu pripočíta k akumulátoru a1. V rámci tohto modelu riadenia inštrukcia *opisuje prechod množiny dát sekvenciou operácií*.

c) Model s využitím blokovania (interlocking)
Technické prostriedky v rámci riadiacej jednotky *sledujú konflikty a zabezpečujú korektné vykonávanie* operácií (zvyčajne oneskorením konfliktných inštrukcií). Tento spôsob je pre programátora plne transparentný a je pomocou neho možné výrazne zjednodušiť programovanie DSP. Typickým príkladom je DSP563xx od firmy Motorola, ktorý vzhľadom na využitie blokovania umožňuje binárnu kompatibilitu s procesormi DSP560xx, ktoré využívajú časovo stacionárny model riadenia. Maximálne využitie DSP však vyžaduje minimalizáciu konfliktov dosahovanú pomocou preusporiadania konfliktných inštrukcií. Tento spôsob riadenia využíva aj ADSP BF533, využívaný v experimentálnych cvičebniach.

2.3.3 ADRESOVÉ GENERÁTORY A PAMÄŤOVÝ SYSTÉM

Vysoká výkonnosť DSP je umožnená okrem výkonných dátových ciest aj veľkou priepustnosťou pamäťového systému. Typická realizácia FIR filtra (2.1) vyžaduje prenos dvoch údajov (koeficientu a vzorky) v jednom inštrukčnom cykle. Zápis výsledku je realizovaný *N*-krát pomalšie a je teda z pohľadu prístupu do pamäte menej kritický. Typické DSP obsahujú dve (existujú však aj DSP len s jednou jednotkou) špeciálne jednotky – *adresové aritmetické jednotky*, (AAU – Address Arithmetic Units) ktoré umožňujú generovať dve adresy pre prístup k dvom dátovým pamätiam v jednom inštrukčnom cykle. Tieto jednotky môžu pracovať paralelne s dátovými cestami pričom typicky podporujú nasledujúce spôsoby adresovania

- ***lineárne*** – klasické adresovanie,
- ***modulo*** – vhodné pre vytváranie napr. oneskorovacích liniek číslicových filtrov,
- ***bitovo reverzné*** – vhodné na efektívne preusporiadanie výstupu FFT algoritmu.

Vysoká priepustnosť pamäťového systému je v rámci DSP čipu umožnená *integráciou* IM a DM spolu s podpornými adresovými a dátovými zbernicami *priamo do čipu DSP*, čo umožňuje realizáciu jednočipových systémov ČSS. Novšie DSP umožňujú aj využitie externých IM a DM, pričom interné zbernice sú typicky *multiplexované* do jednej externej adresovej a jednej externej dátovej zbernice, čo oproti využivaniu len interných pamätí výrazne znižuje priepustnosť pri práci s externými pamäťami.

2.3.4 PERIFÉRNE OBVODY

Periférne obvody v DSP zabezpečujú predovšetkým komunikáciu s externými zariadeniami. Prakticky od počiatku vývoja DSP sú súčasťou DSP výkonné *sériové rozhrania* pre pripojenie A/D a D/A prevodníkov, čo umožňuje zníženie počtu vodičov potrebných na pripojenie externých prevodníkov. Tieto rozhrania je často možné využiť aj na prepojenie viacerých procesorov a vytvorenie jednoduchších sietí DSP.

Medzi ďalšie štandardné periférne obvody patria paralelné rozhrania pre pripojenie k nadradeným systémom, typicky k štandardným 8 a 16-bitovým mikroprocesorom. Aj keď základné vlastnosti týchto obvodov sú prakticky rovnaké u všetkých typov DSP, detailné vlastnosti sú špecifické pre každý typ DSP a ich opis v manuáloch DSP často zaberá viac priestoru ako opis jadra DSP.

2.3.5 PREVODNÍKY NA BÁZE SIGMA-DELTA MODULÁCIE

Aj keď A/D a D/A prevodníky by bolo možné zahrnúť medzi periférne obvody, ich postavenie z pohľadu systému ČSS je veľmi špecifické, pretože umožňuje realizovať z číslicového procesora (t.j. číslicového systému) systém diskrétny. Existuje niekoľko základných princípov využívaných v A/D a D/A prevodníkoch. V oblasti spracovania rečových a audio signálov (teda oblastí, kde majú v súčasnosti DSP dominantné postavenie) však v posledných desaťročiach dominuje použitie prevodníkov na báze sigma-delta modulácie. Tieto prevodníky využívajú princíp *prevzorkovania* a *tvarovania spektra* (noise shaping) kvantizačného šumu, ktorý sa realizuje sigma-delta modulátorom a. Z pohľadu DSP majú sigma-delta prevodníky tieto výhodné vlastnosti:

- vyžadujú minimum presných analógových prvkov, čo výrazne znižuje ich cenu,
- využívajú princípy ČSS, ich veľká časť môže byť implementovaná lacnou CMOS VLSI technológiou a integrovaná na jednom čipe spolu s DSP,
- dosahujú vysoké rozlíšenie (typicky 16 bitov, často až 24 bitov pri frekvencii vzorkovania 8 až 100 KHz,
- princíp prevodu zabezpečuje vysokú linearitu prevodu,
- umožňujú znížiť nároky na externý obmedzovací a rekonštrukčný analógový filter.

Výhodné vlastnosti tejto triedy prevodníkov boli využité napr. v rámci niektorých DSP (napr. DSP56156 od Motoroly, ADSP 2159 od Analog Devices) na integráciu A/D a D/A prevodníkov priamo do čipu DSP.

Určitou nevýhodou sigma-delta prevodníkov je ich relatívne nízky rozsah vzorkovacích frekvencií, čo je dané vysokým prevzorkovaním (často 256 a viac) s ktorým tieto prevodníky interne pracujú. Aj v tejto oblasti je však možné očakávať výrazný pokrok predovšetkým pri spracovaní úzkopásmových vysokofrekvenčných signálov, ktoré je potrebné pri realizácii tzv. *softvérového rádia*.

2.4 POROVNÁVANIE VÝKONNOSTI – BDTIMARK

Porovnávanie výpočtovej výkonnosti rôznych procesorov je úloha značne komplikovaná. Napr. v oblasti univerzálnych mikroprocesorov sa používajú rôzne typy skúšobných úloh (tzv. benchmarks) ako napr. Wheatstone, Dhrystone alebo Linpack. Jednoduché charakteristiky, ktoré výrobcovia DSP typicky uvádzajú vo forme MIPS, MOPS, MFOPS a pod. sú pre prax často nepostačujúce, pretože výrobcovia často interpretujú ich význam rôzne. Okrem toho napríklad výkonnosť MIPS značne závisí od architektúry DSP, veľkostí interných pamätí alebo mechanizmu riadenia vyrovnávacích pamätí a tak často DSP s nižším parametrom MIPS môže v prípade praktických aplikácií dosiahnuť vyšší výpočtový výkon. Druhým extrémom je porovnávanie výkonnosti na základe efektívnosti implementácie *kompletných aplikácií* ako sú napr. implementácia CELP vokodéra, V34 modemu a pod. Kompletné algoritmy často odrážajú skôr efektivitu a zručnosť programátora a sú ťažko reprodukovateľné. Ako pre prax optimálne kompromisné riešenie sa ukazuje porovnávanie výkonnosti pomocou súboru *jadier základných algoritmov*, prevažne z oblasti ČSS. V oblasti DSP je najznámejší systém testovacích úloh firmy Berkeley Design Technology, Inc. (www.bdti.com), ktorý zaviedol *testovaciu metriku*, ktorej jednotkou je tzv. *BDTImark*, pričom tento testovací systém má nasledujúce vlastnosti:

- ***relevantnosť*** – táto metrika odráža výkonnosť procesorov pre typicky používané jadrá algoritmov ČSS, ktorých zoznam so stručnou charakteristikou je uvedený v tab. 2.2 ,
- ***jednoduchosť*** – vyjadrenie hodnoty metriky jediným číslom umožňuje rýchle a ľahké porovnanie rôznych procesorov, pričom hodnoty BDTImark vyjadrujú výkonnosť v lineárnej mierke a vyššia hodnota vyjadruje vyšší výpočtový výkon,
- ***aplikovateľnosť*** – metrika je aplikovateľná pre ľubovoľný typ programovateľného procesora a je nezávislá na jeho architektúre,
- ***nezávislosť*** – metrika je vyhodnocovaná *nezávislou firmou*, čo zaručuje, že výsledky sú vypočítavané korektne,
- ***dostupnosť*** – hodnoty metriky sú bezplatne verejne dostupné na WWW stránke firmy.

Tab. 2.2 Zoznam algoritmov použitých pri vyhodnocovaní hodnoty BDTImark

Funkcia	Význam	Príklad aplikácie
Blokový reálny FIR	Bloková realizácia filtra s konečnou impulzovou odpoveďou pre reálne dáta	Spracovanie reči (napr. algoritmus G728)
Blokový komplexný FIR	Bloková realizácia FIR filtra pre komplexné dáta	Ekvalizácia kanálu v modemoch
Reálny FIR	FIR filter pre spracovanie jednej reálnej vzorky	Spracovanie reči, všeobecná filtrácia
LMS adaptívny FIR	Gradientný stochastický adaptívny algoritmus pre spracovanie jednej reálnej vzorky	Ekvalizácia kanálov, riadenie servopohonov, lineárne predikčné kódovanie
IIR	Rekurzívny filter s nekonečnou impulzovou odpoveďou pre jednu reálnu vzorku	Spracovanie audiosignálov, všeobecná filtrácia
Vektorový súčin	Súčet výsledkov komponentného násobenia zložiek dvoch vektorov	Konvolúcia, korelácia, násobenie matíc, viacrozmerné spracovanie signálov
Vektorový súčet	Komponentný súčet dvoch vektorov, výsledkom je tretí vektor	Grafika, kombinovanie audio signálov a obrazov, vektorové prehľadávanie
Maximum vo vektore	Nájdenie maximálnej hodnoty a jeho polohy vo vektore	Algoritmy protichybového zabezpečenia, aritmetika s blokovou pohyblivou čiarkou
Konvolučný kodér	Aplikácia konvolučných zabezpečovacích kódov na blok bitov	Americký štandard digitálnej mobilnej telefónnej siete (štandard IS 54)
Konečný automat	Klasické riadenie (testovanie, vetvenie) a bitové manipulácie	Prakticky všetky DSP aplikácie obsahujúce určitú formu riadenia
256 bodová, radix 2 FFT	Rýchla Fourierova transformácia	Radary, sonary, MPEG audio kompresia, spektrálna analýza

Samozrejme tento výkonnostný test má v praxi aj obmedzenia. Ako hodnota závislá na množine testovacích algoritmov, hodnota BDTImark neodráža napr. špecializáciu

procesora pre konkrétne aplikácie (pre ktoré napr. DSP môže obsahovať špeciálne koprocesory). Hodnoty BDTImark odrážajú rýchlosť testovaného procesora pre všeobecne používané typy algoritmov ČSS a na ich hodnoty nemajú vplyv ďalšie pre prax často dôležité vlastnosti ako je cena, príkon a pod. Všetky testovacie algoritmy sú optimalizované v asembleri príslušného procesora a využívajú prirodzenú dĺžku slova testovaného procesora. Pri interpretácii hodnôt BDTImark je preto potrebné vziať do úvahy, že procesory s pohyblivou rádovou čiarkou poskytujú presnejší výsledok.

3 MODERNÉ SIGNÁLOVÉ PROCESORY

Z predchádzajúcich úvah je zrejmé, že aj keď taktovacie frekvencie DSP sa stále zvyšujú a teda vplyvom tohto faktora tiež rastie aj ich výpočtový výkon, výraznejšie zvýšenie výpočtového výkonu je možné len zvýšeným využitím *paralelizmu*. V oblasti DSP sa využívajú dva základné spôsoby zvýšenia paralelizmu

- zväčšenie počtu operácií vykonaných každou inštrukciou,
- zväčšenie počtu inštrukcií vykonaných v každom cykle.

Prvý spôsob sa realizuje zahrnutím nových vlastnosti do dátových ciest a je typický predovšetkým u výrobcov DSP, ktorí sa snažia zvýšiť výkonnosť DSP bez radikálnej zmeny celkovej architektúry. Tieto DSP budú v ďalšej časti nazývané *rozšírené klasické* DSP (enhanced conventional DSP). Naopak druhý spôsob, paralelné vykonávanie viacerých inštrukcií je typické pre najnovšie architektúry DSP – *DSP s paralelným vykonávaním inštrukcií*, pričom v tejto triede DSP sa často využívajú aj vylepšenia, ktoré kombinujú obidva spôsoby zvýšenia paralelizmu. Cieľom tejto kapitoly je na základe naznačenej klasifikácie opísať trendy v rozvoji moderných DSP.

3.1 ROZŠÍRENÉ SIGNÁLOVÉ PROCESORY

Približne v polovici 90-tych rokov sa objavili nové varianty DSP, ktoré na základe evolučného vývoja architektúr klasických DSP poskytli oproti klasickým DSP zvýšený výpočtový výkon. Toto zlepšenie bolo dosiahnuté okrem technologického pokroku, predovšetkým *aplikačne orientovanými vylepšeniami*. Do tejto kategórie DSP patria napr. TMS320C54x od TI, DSP563xx a DSP566xx od Motoroly, DSP16xxx od firmy Lucent a čiastočne aj AD218x od Analog Devices. Z pohľadu klasifikácie je ich možné zaradiť do 3. generácie DSP s pevnou rádovou čiarkou.

3.1.1 TECHNOLOGICKÉ ZLEPŠENIA

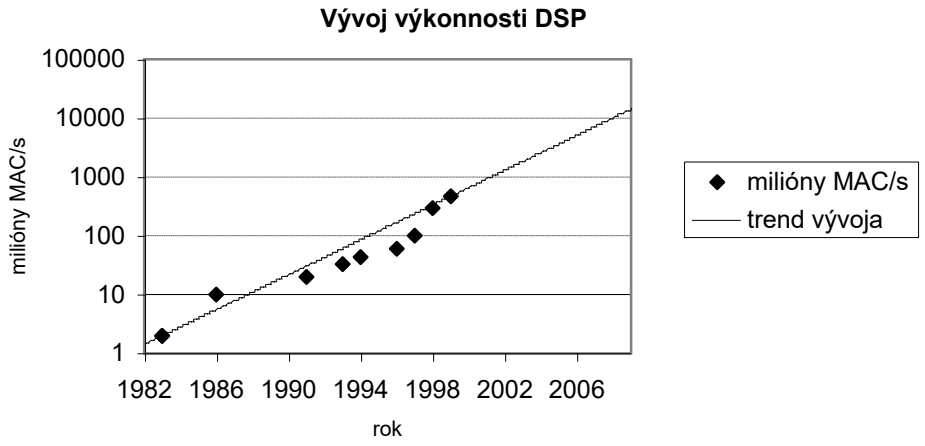
Technologický pokrok CMOS VLSI technológie sa prejavil (a v tejto kategórii DSP sa stále prejavuje) predovšetkým v týchto oblastiach:

a) Zvyšovanie taktovacej frekvencie
Najvyššia taktovacia frekvencia procesorov tejto kategórie v súčasnosti dosahuje[1] stovky MHz a sú ohlasované stále vyššie taktovacie frekvencie (napr. ADSP BF533 využívaný v experimentálnych cvičeniach ma taktovaciu frekvenciu 600 MHz). Naviac oproti DSP z druhej generácie je veľmi časté, že počet vykonaných inštrukcií za

[1] V súčasnosti však výrazne zaostáva za taktovacími frekvenciami najvýkonnejších procesorov napr. pre stolné počítače. Dôvodom je predovšetkým výrazná orientácia DSP na nízkopríkonové aplikácie.

sekundu je zhodný s použitou taktovacou frekvenciou (takzvaná *1X architektúra*). Prakticky ako štandard sa v tejto kategórií DSP využíva generovanie taktovacej frekvencie priamym násobením na čipe buď pomocou násobičiek alebo, stále častejšie, konfigurovateľnými obvodmi PLL.

Nárast výpočtového výkonu DSP vyjadrené v miliónoch MAC operácii za sekundu a predpokladané trendy[2] sú znázornené na obr. 3.1 .



Obr. 3.1 Nárast výkonnosti DSP vyjadrený v miliónoch MAC operácií za sekundu

b) Zvyšovania úrovne zreťazenia

Zreťazenie základných operácii riadiacej jednotky (výber, dekódovanie a vykonanie) bolo využívané už u 1. a 2. generácie DSP. Využitie zreťazených dátových ciest však využívali len niektorí výrobcovia DSP a napr. Motorola u DSP5600x a Analog Devices u AD218x vystačili s nezreťazenými jednotkami MAC. Architektúra 1X však vzhľadom na technologickú úroveň už vyžaduje použitie zreťazenia aj v rámci dátových ciest (jednotkách MAC, ALU…). DSP typicky využívajú v riadiacich jednotkách väčšiu úroveň zreťazenia (napr. 7 úrovní v DSP563xx ako v MAC jednotkách (napr. 2 úrovne v DSP563xx), čo je však stále podstatne menej ako napr. v oblasti procesorov pre pracovné stanice, kde sa typicky využíva zreťazenie s niekoľkými desiatkami úrovni zreťazenia.

Relatívne nízka úroveň zreťazenia sa pozitívne prejavuje v zachovaní malej zložitosti programovacieho modelu aj pre túto kategóriu DSP[3]. Vzhľadom na nutnosť použitia ručnej optimalizácie kódu (minimálne pre najkritickejšie časti kódu) pre túto triedu DSP je udržanie nízkej úrovne zreťazenia prakticky nutnosťou. Aj keď zavedením zreťazenia do jednotky MAC vznikajú nové obmedzenia (typu výsledok je dostupný až o x cyklov neskôr), existuje často snaha tento problém eliminovať na

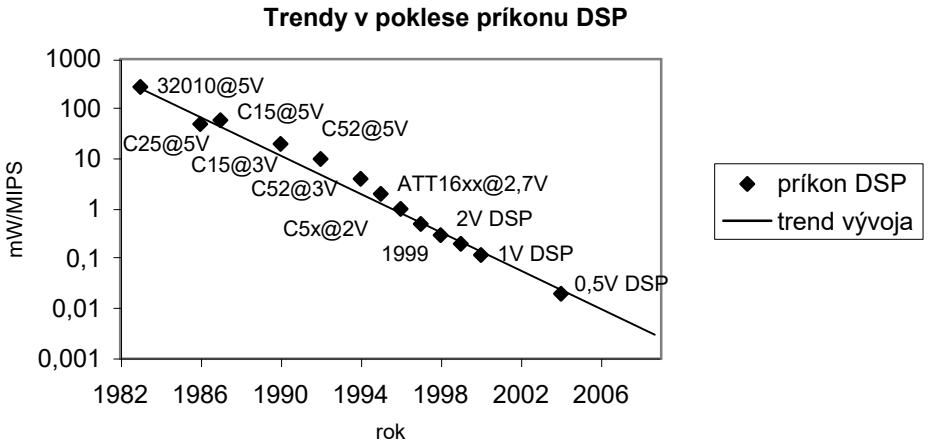
[2] Nárast výpočtovej výkonnosti od roku 1998 je už však zabezpečovaný predovšetkým zvýšeným využitím paralelizmu.

[3] Aj keď to znie paradoxne, pretože od 1. generácie DSP je rozšírený názor, že programovanie DSP je veľmi zložité. Ručná optimalizácia kódu napr. pre procesor Intel Pentium, prípadne novších, je však v súčasnosti podstatne zložitejšia, čo je dané jeho zložitejším programovacím modelom.

úrovni technických prostriedkov využitím modelu zreťazenia s využitím vzájomného blokovania (využíva napr. aj ADSP BF533). Takéto riešenie môže dokonca zabezpečiť kompatibilitu na úrovni binárneho kódu s procesormi predchádzajúcej generácie. Objektívne však treba povedať, že v prípade požiadavky dosiahnutia maximálneho výpočtového výkonu je potrebné pri optimalizácii kódu všetky obmedzenia vziať do úvahy a minimalizovať prípady zablokovania vhodným preusporiadaním inštrukcií v procese optimalizácie kódu[4]

c) Znižovanie napájacieho napätia

Napájacie napätie tejto triedy DSP už výrazne pokleslo pod hodnotu 3.3 V, ktorá bola typická v polovici 90-tych rokov. Vzhľadom na potrebu zabezpečenia kompatibility s externými obvodmi dochádza k rozdeleniu napájacieho napätia pre jadro DSP (v súčasnosti typicky okolo 1 V) a externé obvody DSP (typicky 3,3 V, prípadne 2,5 V alebo dokonca 1,2 V). Tento trend je všeobecný a je ho možné pozorovať napr. aj v oblasti procesorov pre pracovné stanice ako aj v oblasti zložitých obvodov FPGA. V oblasti DSP pre mobilné zariadenia (napr. GSM telefóny) sa napájacie napätie blíži k hranici 1 V, čo je dosahované využívaním najnovších nízkopríkonových CMOS technológií. Pokles napájacieho napätia a príkonu DSP spolu s vylepšeniami v architektúre vo všeobecnosti vedie k poklesu parametrov príkon/algoritmus. Samozrejme parametre mA/MIPS a mW/MIPS taktiež pre nízkopríkonové DSP klesajú (napr. 0,54 mW/MIPS pre TMS320UVC5402 pri napájacom napätí 1,2 V). Tieto trendy sú naznačené na obr. 3.2 . Znižovanie príkonu a napájacieho napätia DSP vytvára v súčasnosti výrazný tlak predovšetkým na optimalizáciu externých analógových obvodov využívaných v telekomunikačnej technike.



Obr. 3.2 Energetické trendy v oblasti DSP

d) Zväčšovania veľkostí interných pamätí

Interné pamäte majú z hľadiska výkonnosti pre DSP veľký význam. Takmer všetky dostupné DSP dosahujú pri práci s internými pamäťami podstatne vyšší výpočtový

[4] Tento jav bude demonštrovaný na príklade vysoko optimalizovaného kódu pre IIR filter implementovaný na procesore ADSP BF533.

výkon, pretože prístup k externým pamäťovým bankám harvardskej architektúry je z dôvodu zníženia počtu I/O vývodov *multiplexovaný*. Súčasný prístup k dvom alebo až trom externým pamäťovým bankám tak vyžaduje dva až tri inštrukčné cykly. Naviac pri zvyšujúcej sa taktovacej frekvencii a využívaní 1X architektúry požiadavky na rýchlosť prístupu k externým (typicky SRAM pamätiam stále narastajú a pri taktovacích frekvenciách viac ako 500 MHz sú potrebné pamäte s dobou prístupu pod 2 ns, čo je v prípade externých pamätí značný problém. V praxi tak musí byť prístup k pomalším externým pamätiam často realizovaný s využitím dodatočných čakacích stavov, čo ďalej znižuje výpočtový výkon DSP. V minulosti bolo potrebné venovať značné úsilie vývoju optimalizovaných algoritmov ČSS pre DSP s malými internými pamäťami. Typický príklad algoritmu výpočtu FFT s využitím menších FFT bude demonštrovaný časti venovanej FFT.

V súčasnosti majú DSP s najväčšími internými pamäťami na čipe implementované SRAM pamäte s veľkosťou niekoľko desiatok Mbitov[5]

e) Integrácia nových typov pamätí

Okrem štandardných SRAM pamätí a PROM pamätí u maskou programovateľných verzií DSP sa začínajú objavovať verzie s FLASH pamäťami (napr. DSP1609F). Tento trend, ktorý je výrazný predovšetkým v oblasti univerzálnych jednočipových mikropočítačov, je logickým dôsledkom o zníženie počtu čipov pri realizácii kompletného systému DSP na jednom čipe, čo vyžaduje umiestnenie programu do vnútra čipu DSP. Vzhľadom na rýchlosť dostupných FLASH pamätí je to stále len suboptimálne riešenie. Veľké nádeje sú vkladané napr. do oblastí feroelektrických pamätí, ktoré by mali poskytnúť výrazne kratšiu dobu prístupu než je doba prístupu FLASH pamätí, zatiaľ však tento typ technológie nebol v DSP použitý.

f) Integrácia radičov DMA

Pôvodne boli radiče DMA umiestňované len do výkonnejších DSP s pohyblivou rádovou čiarkou a určené predovšetkým na podporu multiprocesorovej komunikácie. Vzhľadom na výrazne zväčšenie interných a externých pamäťových priestorov ako aj umiestňovanie výkonných periférií priamo na čipoch DSP (napr. štandardné sériové a paralelné kanály, ale aj špecializované DSP koprocesory a pod.) je výhodné na presuny veľkého objemu dát využívať mechanizmus kanálov DMA a tým zvýšiť priepustnosť celého DSP. Naviac, oproti klasickým DSP bez radičov DMA nie je potrebné na prenos (realizovaný napr. v klasických DSP pomocou prerušení) alokovať časť registrov DSP a tieto potom môžu byť využité na efektívnejšiu implementáciu algoritmov ČSS v samotnom jadre DSP. Počet DMA kanálov v moderných DSP je rôzny (napr. ADSP218x má 1-kanálový radič DMA, DSP563xx má 6-kanálový radič DMA) a najmodernejšie implementácie radičov DMA môžu pracovať v prípade interných DSP zdrojov paralelne s DSP jadrom bez vzájomného ovplyvňovania[6].

DMA kanály predstavujú unifikovaný spôsob prístupu k periférnym obvodom DSP a úspešne tak nahradzujú rôzne špecializované prístupy spolupráce s periférnymi obvodmi klasických DSP. Samozrejme technológia DMA radičov je dnes široko využívaná už aj v MCU.

[5] Uvádzanie veľkostí pamätí v Mbitoch je zvyčajne reklamnou záležitosťou a objektívnejším údajom je počet slov, čo je napr. pre 16-bitový procesor 16-krát menej.

[6] Napr. interná pamäť DSP563xx je rozdelená do baniek s veľkosťou 256 slov. Pokiaľ jadro DSP a radič DMA pristupujú súčasne do rôznych pamäťových baniek, jadro DSP a radič DMA pracujú úplne nezávisle.

g) Integrácia ladiacich obvodov
Pri stále sa zvyšujúcej taktovacej frekvencii a využívaní SMD technológie je využívanie klasických emulačných techník, kedy sa cieľový procesor nahradil emulačnou hlavicou a jeho činnosť prevzal hardvérový emulátor, prakticky nerealizovateľné. V praxi sa tento problém rieši integráciou špeciálnych ladiacich obvodov priamo do každého čipu DSP, čo umožňuje realizovať emuláciu bez nutnosti vyberať čip z testovanej dosky plošného spoja. Na pripojenie je vyhradených niekoľko vývodov. V súčasnosti sú tieto vývody často zároveň využívané aj ako vývody štandardizovaného rozhrania JTAG, ktoré umožňuje predovšetkým testovanie správneho osadenia a porúch plošných spojov.

h) Vylepšené vyrovnávacie pamäte
DSP založené na modifikácii 3 harvardskej architektúry využívali vyrovnávacie pamäte veľmi malej veľkosti (typicky okolo 16-64 slov), pričom využitie tejto pamäte bolo automaticky zabezpečené pre opakované vykonávanie tej istej inštrukcie (zvyčajne po zadaní inštrukcie REP), prípadne v tzv. hardvérových slučkách (zvyčajne inštrukcia DO). Z hľadiska veľkosti vyrovnávacej pamäte je modifikácia 4 podstatne vhodnejšia, pretože veľkosť internej programovej pamäte bola minimálne niekoľko sto slov (napr. 512 slov v DSP5600x). Niektoré typy najnovších DSP na báze modifikácie 4 (napr. DSP563xx) aj keď využívajú podstatne zväčšené interné pamäte, majú naviac možnosť vyhradiť časť internej programovej pamäte (veľkosti 1 až 2 Kslov) ako vyrovnávaciu pamäť. Existuje niekoľko režimov činnosti tejto pamäte od režimu, ktorý je plne transparentný pre programátora, až po možnosť *riadenia uzamknutia* prípadne *uvoľnenia* jednotlivých sektorov vyrovnávacej pamäte pomocou inštrukcií vo vykonávanom programe. Tento nový (v oblasti DSP) prístup je výhodný predovšetkým v prípade, že sú pripojené pomalé externé programové pamäte a v prípade veľkých programov minimalizuje potrebu využívania techniky *dynamických prekryvných modulov* (dynamic overlays). Tieto fakty naznačujú, že zložitosť programov, pre ktoré sú najnovšie typy DSP *určené výrazne presahuje zložitosť programov* typicky implementovaných pomocou klasických DSP.

i) Multičipové moduly
Priamym spôsobom využitia paralelizmu, ktorý poskytuje technológia VLSI je umiestnenie viacerých čipov DSP do jedného puzdra – *multičipového modulu*. Z hľadiska koncového užívateľa je prakticky nepodstatné, či sa jedná o multičipový modul alebo jednočipovú integráciu viacerých procesorov. Logické je využitie multičipových modulov v oblasti vysokovýkonných DSP, ktoré sú originálne navrhnuté pre multiprocesorové moduly a multičipový modul môže výrazným spôsobom zlepšiť technické parametre koncových zariadení. Typickým príkladom bol napr. AD14060 od Analog Devices , ktorý integroval 4 čipy s pohyblivou rádovou čiarkou – AD21060 spolu s ich vzájomným prepojením alebo TMS320C8x, ktorý integroval 2 až 4 DSP s pevnou rádovou čiarkou a riadiaci RISC procesor. Tieto multičipové moduly boli dostupné už v polovici 90-rokov, kedy viacjadrové procesory Intel ešte len začínali využívať dvojjadrové procesory. Dnes je samozrejme situácia odlišná, keďže úroveň integrácie a ekonomické aspekty umožňujú vyrábať výkonne procesory Intel Xenon s niekoľkými desiatkami jadier.

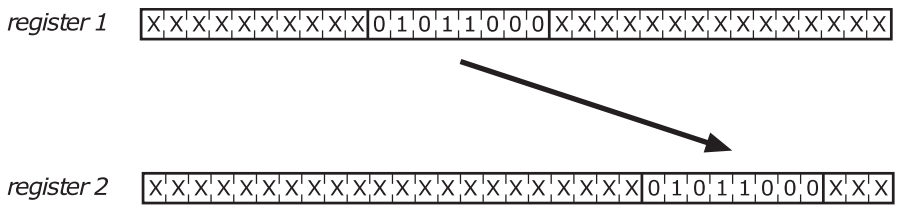
3.1.2 OPTIMALIZOVANÉ DÁTOVÉ CESTY

Klasické DSP mali architektúru výrazne optimalizovanú pre štandardné všeobecné algoritmy ČSS – číslicovú filtráciu a FFT. Ukázalo sa, že pre tieto typy algoritmov je potrebné zrýchliť MAC operáciu a tak sa MAC inštrukcia (podporovaná technickými prostriedkami priamo v dátových cestách) stala integrálnou súčasťou DSP. Podobne potreba špecializovaného prístupu k dátovým štruktúram v prípade číslicových filtrov (modulo adresovanie) a FFT (reverzné adresovanie viedli k priamej podpore týchto režimov v adresových aritmetických jednotkách.

Široké nasadenie DSP predovšetkým v telekomunikačnej oblasti prispelo k rozšíreniu triedy algoritmov, pre ktoré výrobcovia DSP integrovali do dátových ciest nové bloky a zaviedli nové inštrukcie.

a) Jednotky pre bitovú manipuláciu - Bit Field Units)
Bitové manipulácie sú v telekomunikačnej oblasti extrémne často využívané, pretože pri prenose informácie sa často pracuje s jednotlivými bitmi. Typickým príkladom je realizácia presusporiadania jednotlivých bitov v bloku interlívera, CRC zabezpečenia a pod. Už staršie DSP mali často možnosť realizovať základné bitové operácie ako napr. testovanie jednotlivých bitov v pamäti resp. v registroch. Novšie DSP už integrovali špeciálne viacbitové posúvače, ktoré realizovali viacbitové posuny v jednom inštrukčnom cykle. Tieto operácie sú veľmi časté predovšetkým pri výpočte exponentu prípadne pri *normovaní čísel*, ktoré je využívané napr. pri implementácií rečových kodekov na báze CELP algoritmov. Podpora normovania sa objavila najskôr u 16-bitových DSP, kde pre mnohé algoritmy ČSS je potrebné realizovať výpočty v blokovej pohyblivej rádovej čiarke a eliminovať tak relatívne malú dĺžku slova tejto triedy DSP.

Ďalším krokom bola integrácia nových jednotiek pre bitové manipulácie, ktoré už okrem uvedených bitových operácii umožnili realizovať v jednom inštrukčnom cykle aj operácie vloženia a extrakcie bitových reťazcov do/z iných bitových reťazcov. Na obr. 3.3 je tento typ operácie znázornený na príklade extrakcie bitového reťazca z registra 1 do registra 2. Zrýchlenie týchto typov operácií prispelo k výraznému zvýšeniu rýchlosti veľkej triedy telekomunikačných algoritmov.



Obr. 3.3 Princíp vloženia a extrakcie bitového reťazca

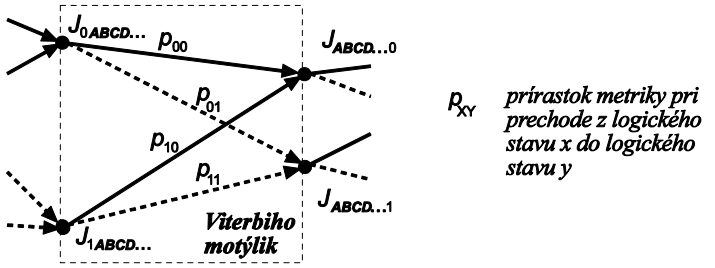
b) Podpora pre Viterbiho algoritmus
Viterbiho algoritmus (VA) je rýchly dekódovací algoritmus originálne navrhnutý v roku 1967 na optimálne dekódovanie konvolučných kódov. Konvolučné kódy patria v súčasnosti medzi najčastejšie používané kódy pre protichybové zabezpečenie a predovšetkým v oblasti mobilnej telekomunikačnej techniky tvoria prakticky štandardný algoritmus. Ďalšou typickou aplikáciou VA je ekvalizácia prenosového kanálu, ktorá sa taktiež široko využíva v telekomunikačnej technike, ale aj v menej

tradičných oblastiach ako napr. v riadiacich jednotkách pevných diskov. Význam VA v týchto oblastiach je porovnateľný s významom FFT v spektrálnej analýze. Tento rekurzívny algoritmus vyhľadáva minimálnu cestu v tzv. *trellise*[7], pričom základnou operáciou na najnižšej úrovni algoritmu je operácia tzv. *Viterbiho motýlika* (Viterbi butterfly) znázornená na obr. 3.4 , ktorá realizuje operáciu

$$J_{ABCD\ldots 0} = \min\left(J_{0\,ABCD\ldots} + p_{00}, J_{1ABCD\ldots} + p_{10}\right) \tag{3.1}$$

$$J_{ABCD\ldots 1} = \min\left\{J_{0\,ABCD\ldots} + p_{01},\ J_{1ABCD\ldots} + p_{11}\right\} \tag{3.2}$$

pričom $J_{ABCD\ldots X}$ a $J_{YABCD\ldots}$ sú tzv. *akumulované metriky* v stavoch $ABCD\ldots X$, $YABCD\ldots$, $X, A, B, C, D\ldots, Y \in \{0,1\}$ a motýlik tvoria stavy, ktoré majú rovnaké hodnoty $ABCD\ldots$



Obr. 3.4 Štruktúra základného bloku Viterbiho algoritmu – Viterbiho motýlika

Výrobcovia DSP pridali do inštrukčnej sady nové inštrukcie (napr. VSL pre DSP563xx a DSP566xx), ktoré umožnili zrýchliť spracovanie Viterbiho motýlika. Niektorí výrobcovia (napr. TI v TMS320C54x, Lucent v DSP16xxx) pridali nové bloky priamo do dátových ciest. Ich význam je závislý na type procesora, cieľ je však rovnaký – zrýchlenie VA. Využitím týchto optimalizovaných inštrukcií resp. rozšírení dátových ciest je možné dosiahnuť zrýchlenie najčastejšie sa opakujúcich segmentov kódu (napr. na TMS320C54x je možné realizovať algoritmus Viterbiho motýlika počas 4 taktov).

c) Špeciálne aritmetické režimy

Okrem podpory blokovej pohyblivej rádovej čiarky, ktorá sa využíva na zvýšenie presnosti výpočtu predovšetkým FFT na 16 a dokonca aj 24-bitových DSP a dátové cesty ich podporujú už od 2. generácie DSP, nové DSP vylepšujú aj podporu výpočtov *v dvojnásobnej presnosti*. Vo všeobecnosti platí, že tieto výpočty vyžadujú zvýšený počet inštrukčných cyklov. Cieľom nových inštrukcií je tento počet znížiť. U 16-bitových DSP bola táto podpora od počiatku značne prepracovaná, ukazuje sa však, že aj v oblasti 24-bitových DSP sa táto podpora stále vylepšuje. Napr. u radu DSP5600x bolo potrebné prepnúť dátové cesty do špeciálneho režimu dvojnásobnej presnosti (čo samozrejme vyžaduje dodatočné inštrukcie) u novšieho radu DSP563xx je síce z dôvodu spätnej kompatibility tento režim zachovaný, boli však dodané nové inštrukcie pre aritmetiku v dvojnásobnej presnosti, ktoré pracujú bez nutnosti prepnutia do režimu s dvojnásobnou presnosťou.

Aj keď saturačná aritmetika je stále podporovaná ako základný aritmetický režim nových DSP, v oblasti telekomunikačnej techniky sa veľmi často *používa klasická*

[7] Jeden zo spôsobov opisu konvolučných kódov.

aritmetika s pretečením. Napríklad v oblasti zdrojového kódovania rečových signálov existujú štandardné kompresné algoritmy (rôzne varianty CELP kodekov), ktorých implementácie musia poskytovať rovnaké výsledky (s bitovou presnosťou) ako referenčný model podľa normy. Keďže normy typicky používajú štandardnú aritmetiku, čo výrazne komplikuje programovanie klasických DSP, výrobcovia DSP začali v najnovších DSP podporovať nové aritmetické režimy, ktoré využívajú štandardnú 16-bitovú aritmetiku s pretečením (v prípade 24-bitových DSP teda dochádza dokonca k strate ôsmych najmenej významových bitov).

d) Pridanie druhej jednotky MAC
Keďže MAC operácia je najvýkonnejšia a najčastejšie využívaná inštrukcia v DSP, niektorí výrobcovia sa snažia zvýšiť výkon DSP integrovaním *sekundárnej* jednotky MAC. Napr. firma DSPGroup v rozšírenom jadre Teak DSP využilo sekundárnu jednotku MAC . Podobne DSP16xxx od firmy Lucent využíva duálnu MAC architektúru . Tento DSP (bol prvým po oddelení firmy Lucent od AT&T) však už využíva aj ďalšie výkonné rozšírenia a naznačuje orientáciu na širšie využívanie paralelizmu. Pridanie sekundárnej MAC jednotky je tak možné považovať len za *prechodný stupeň* k vývoju výkonnejších DSP, z komerčného hľadiska však môže v čase uvedenia zabezpečiť výrobcom DSP určitú výhodu na trhu. Aj procesory s jadrom Blackfin (vrátane ADSP BF533) využívané v experimentálnej časti využívajú duálnu MAC jednotku na zvýšenie ich výpočtového výkonu pri spracovaní algoritmov ČSS.

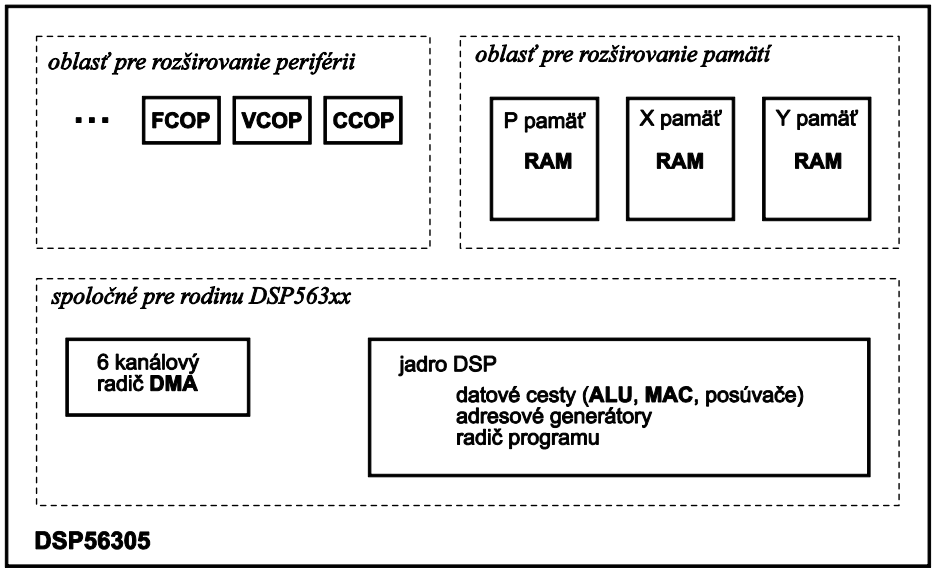
e) Ďalšie trendy
Ďalším zaujímavým trendom v oblasti aritmetických režimov je podpora *výpočtov v konečných (Galoisových) poliach*. Táto aritmetika sa využíva predovšetkým v blokových zabezpečovacích kódoch (napr. Read-Solomonove kódy, BCH kódy…), ktoré sú často využívané v moderných telekomunikačných aplikáciách. Výpočty v konečných poliach je principiálne možné realizovať pomocou ľubovoľných programovateľných procesorových architektúr, pokiaľ však neexistuje podpora na úrovni technických prostriedkov, je rýchlosť výpočtu nízka. Tieto operácie využívajú aj viaceré kryptografické algoritmy a okrem výrobcov najnovších DSP (napr. TI a Analog Devices) v súčasnosti využívajú napr. aj proceory Intel (nové inštrukcie pre podporu šifračného algoritmu AES alebo podpora GCM módu blokových šifier).

3.1.3 DSP KOPROCESORY

Využitie *koprocesorov* v mikroprocesorovej technike je veľmi časté. Typickým príkladom sú numerické koprocesory v oblasti personálnych počítačov (napr. Intel x87), jednočipových mikropočítačov (napr. Siemens 8xC537) alebo šifračných koprocesorov v čipoch pre čipové karty. Z principiálneho hľadiska nie je podstatné, či je koprocesor umiestnený na čipe, alebo je tvorený samostatným čipom. Jeho hlavnou úlohou je poskytnutie zvýšeného výpočtového výkonu v oblasti, v ktorej hlavný procesor (alebo jeho jadro) nie je dostatočne výkonný. Optimálne je, pokiaľ koprocesor tvorí samostatnú časť, ktorá môže pracovať paralelne s procesorom (alebo jeho jadrom).

Vzhľadom na uvedené trendy bolo len otázkou času, kedy sa objavia v čipoch DSP samostatné DSP koprocesory. Značným prekvapením však bolo, že medzi prvými bol tzv. *filtračný koprocesor*, ktorý umožňoval realizovať algoritmus FIR filtra (t.j. algoritmus, pre ktorý boli DSP optimalizované od počiatku ich vývoja). DSP koprocesory široko využíva v DSP predovšetkým firma NXP, ktorá prevzala DSP

portfólio produktov firmy Freescale (predtým Motorola). Priekopníkom v tejto oblasti bol procesor Motorola DSP5305, ktorý v polovici 90 rokov obsahoval okrem jadra DSP aj tri koprocesory – filtračný koprocesor (FCOP), *Viterbiho koprocesor* (VCOP) a *cyklický koprocesor* (CCOP). Bloková štruktúra DSP56305 je zobrazená na obr. 3.5 .



Obr. 3.5 DSP koprocesory v štruktúre DSP56305

Aj keď DSP56305 bol optimalizovaný pre implementáciu algoritmov, ktoré sú používané v systéme GSM, jednotlivé koprocesory sú značne univerzálne a môžu byť využité aj v iných algoritmoch ČSS. Vysokú výkonnosť DSP56305 bolo možné využiť predovšetkým v bázových staniciach mobilných telekomunikačných sieti, kde je možné v jednom procesore súbežne spracovať niekoľko samostatných kanálov.

Uvedené koprocesory sú z hľadiska programovacieho modelu externé periférie, ktorých riadiace, stavové a dátové registre sú mapované do dátovej pamäťovej oblasti harvardskej architektúry a môžu plne využívať vysokú výkonnosť kanálov DMA ako aj štandardný prerušovací mechanizmus, ktorými sú procesory DSP563xx vybavené.

Tieto vlastnosti naznačujú, že koprocesory poskytujú podporu pre širokú triedu algoritmov ČSS využívaných v telekomunikačnej technike. V najnovších DSP pre telekomunikačný segment môžeme nájsť vysoko-výkonné DSP, ktoré integrujú napr. koprocesory pre FFT a IFFT, Turbo a Viterbiho dekódovanie, CRC výpočet a pod. Typickým príkladom sú napr. DSP firmy NXP na báze jadra Starcore (napr. 6-jadrový MSC8156), ktoré patria medzi DSP s paralelným vykonávaním inštrukcií.

3.2 DSP S PARALELNÝM VYKONÁVANÍM INŠTRUKCIÍ

Do tejto kategórie patria v súčasnosti najvýkonnejšie jednočipové DSP a všetci hlavní výrobcovia vyrábajú produkty založené na tomto princípe. Tieto DSP využívajú *paralelizmus na úrovni inštrukcií* (ILP – Instruction Level Parallelism).

Technické prostriedky môžu využívať ILP rôznymi spôsobmi:

- niekoľko *rôznych* funkčných jednotiek v procesore môže pracovať paralelne,
- zvýšením počtu *rovnakých* funkčných jednotiek je možné ďalšie zvýšenie paralelizmu,
- funkčné jednotky môže využívať *zreťazenie.*

3.2.1 ZÁKLADNÁ KLASIFIKÁCIA

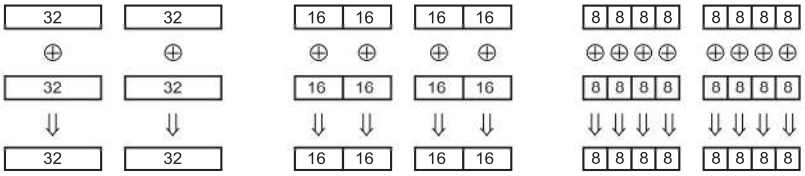
Z pohľadu riadenia procesorov, ktoré využívajú ILP je možné procesory rozdeliť do dvoch základných skupín:

a) Superskalárne procesory
Do tejto kategórie patrí väčšina moderných výkonných procesorov pre všeobecné použitie. Riadenie poradia a paralelizmu vykonávaných inštrukcií je u týchto procesorov realizované technickými prostriedkami umiestnenými priamo na čipe vo forme *plánovacej jednotky* (scheduling hardware) a je realizované počas behu programu. Tento princíp umožňuje spracovanie pôvodne sekvenčného kódu a teda zachovanie *spätnej kompatibility* s procesormi nižších generácii, čo je napr. v oblasti osobných počítačov kľúčová požiadavka. Nevýhodou tejto triedy procesorov je zložitá konštrukcia plánovacej jednotky, ktorá často využíva napr. *špekulatívne vykonávanie inštrukcií*, ktoré môže viesť z hľadiska systémov pracujúcich v reálnom čase k neakceptovateľným oneskoreniam.

b) Procesory s veľkou dĺžkou inštrukčného slova (VLIW processors)
V tejto triede procesorov je plánovanie poradia a paralelizmu inštrukcií realizované počas prekladu a riadiace jednotky na čipe sú oproti superskalárnym procesorom podstatne jednoduchšie. Oproti superskalárnym procesorom je ich hlavnou nevýhodou nezabezpečenie spätnej kompatibility (kód je potrebné pre rôzne VLIW procesory prekladať) a nemožnosť dynamickej zmeny plánovania poradia a paralelizmu inštrukcií v závislosti na aktuálnych podmienkach, čo napr. umožňuje v niektorých prípadoch superskalárnym procesorom využiť skrytý paralelizmus algoritmov.

Z pohľadu algoritmov ČSS, ktoré využívajú relatívne *statické riadenie programu* je aplikácia VLIW architektúry v DSP procesoroch veľmi výhodná. Naviac otázka spätnej kompatibility binárneho kódu má v oblasti pomerne DSP malý význam a optimalizácia a preklad kódu sa stali v tejto triede procesorov prakticky štandardným vývojovým postupom.

Ďalšou perspektívnou modifikáciou, ktorá je využívaná v oblasti superskalárnych aj VLIW procesorov je využitie paralelizmu na báze *architektúry SIMD*, ktoré je v moderných DSP realizované predovšetkým vo forme *SIMD inštrukcií*. Tieto inštrukcie sú efektívnym a kompaktným spôsobom reprezentácie inštrukcií, ktoré realizujú rovnaké operácie nad rôznymi dátami. V minulosti boli využívané predovšetkým v oblasti špeciálnych vektorových počítačov. V súčasnosti sa SIMD inštrukcie využívajú aj priamo v rámci procesorov predovšetkým pri spracovaní multimediálnych dát s menšou dĺžkou slova (typicky 8 alebo 16 bitov), ktoré sú zoskupené do slov väčších rozmerov (niekedy označovaných ako *kontajnery*) a využitím špeciálneho režimu aritmetickej jednotky, ktorá realizuje paralelne požadované operácie nad celým slovom, čo je znázornené na obr. 3.6 .



Obr. 3.6 Princíp využitia SIMD v aritmetických operáciách

Dĺžka slova (kontajnera) je typicky 32 resp. 64 bitov, pričom aritmetické jednotky a registre príslušnej dĺžky sú už zvyčajne v rámci existujúcich dátových ciest obsiahnuté napr. v rámci podpory aritmetiky s dvojnásobnou presnosťou. Tieto SIMD rozšírenia sa zvyknú označovať aj pojmom *multimediálne rozšírenia* prípadne tiež *zbalená aritmetika* (packed arithmetic) a sú využívané napr. aj v procesoroch INTEL (MMX), SUN (VIS inštrukcie), HP (PA-RISC multimedia extensions) a ARM (NEON SIM rozšírenia). Z pohľadu DSP je zaujímavé porovnanie *klasickej* VLIW architektúry s *klasickými* SIMD rozšíreniami.

Nevýhody VLIW architektúry oproti SIMD rozšíreniam:

- ***Funkčné jednotky VLIW procesora sú drahšie***. SIMD rozšírenia využívajú štruktúru existujúcich dátových ciest (predovšetkým v rámci aritmetickej jednotky), ktoré vyžadujú len malú modifikáciu. VLIW architektúra vyžaduje na dosiahnutie tej istej úrovne ILP niekoľko funkčných jednotiek s plnou presnosťou, čo vyžaduje väčšiu plochu čipu.
- ***Kódovanie VLIW inštrukcií je náročnejšie na pamäť***. Pomocou SIMD rozšírenia je možné reprezentovať jedinou inštrukciou *S* nezávislých operácii, pričom *S* je počet zbalených slov. V klasickej VLIW architektúre je na vykonanie rovnakého počtu operácií potrebných minimálne *S* inštrukcií, ktoré sú v optimálnom prípade umiestnené v jedinej VLIW inštrukcii (pokiaľ má VLIW procesor dostatočný počet funkčných jednotiek). To vyžaduje podstatne väčšiu inštrukčnú pamäť, čo je pre jednočipové aplikácie značná nevýhoda. Techniky *kompresie inštrukčnej pamäte* používané vo VLIW DSP sú jednou z metód, ktoré sa snažia túto základnú nevýhodu klasickej VLIW architektúry eliminovať.

Výhody VLIW architektúry oproti SIMD rozšíreniam:

- ***VLIW nevyžaduje explicitné určenie oblastí ILP***. V prípade SIMD rozšírení je naopak potrebné identifikovať tieto oblasti (často ručne) a realizovať volanie SIMD inštrukcií.
- ***VLIW inštrukcie sú flexibilnejšie***. Za cenu zvýšených nárokov na dekódovacie obvody a programové pamäte VLIW architektúry nekladú obmedzenia na typy operácií, ktoré môžu byť realizované paralelne, čo je naopak kritické v prípade SIMD rozšírení, kde je možné paralelne vykonávať len určité typy operácií.

Táto vlastnosť VLIW architektúry je výhodná predovšetkým v prípade algoritmov, ktoré nemajú regulárnu štruktúru.

Superskalárna architektúra je používaná v oblasti DSP len vo výnimočných prípadoch (napr. ZSP16400 od firmy ZSP) a v ďalšej časti nebude podrobnejšie analyzovaná. V súčasnosti najprogresívnejší smer v oblasti programovateľných DSP je využitie VLIW architektúry, modifikovanej s cieľom dosiahnuť úsporu programovej pamäte. SIMD rozšírenia sú typicky často využívané na zvýšenie výkonnosti klasických mikroprocesorov prípadne aj VLIW DSP.

3.2.2 ARCHITEKTÚRA VLIW

Táto architektúra bola pôvodne navrhnutá a využívaná v oblasti superpočítačov. V oblasti jednočipových procesorov bola VLIW architektúra prvýkrát využitá v roku 1995 v multimediálnych procesoroch Trimedia a MPACT od firiem Philips resp. Chromatics. Aj keď multimediálne procesory neboli pre trh klasických a rozšírených DSP v tom čase priamym konkurentom (boli orientované na odlišný segment trhu), uvedenie TMS320C62xx firmy TI v roku 1997 znamenalo rádový výkonnostný skok v oblasti univerzálnych DSP a naznačilo trend, ktorý nasledovali (a stále nasledujú) aj ďalší výrobcovia.

ZÁKLADNÝ PRINCÍP VLIW ARCHITEKTÚRY

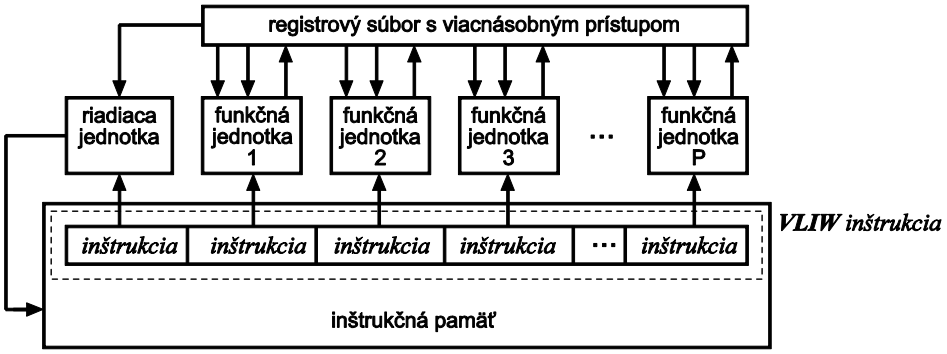
Architektúry DSP boli medzi prvými programovateľnými súčiastkami, ktoré využívali *dlhé inštrukčné slovo* (LIW – Long Instruction Word). Tento prístup vychádzal z horizontálneho mikroprogramovania, ktoré sa používalo v oblasti mikroprocesorových rezov. V architektúrach, ktoré využívajú LIW má procesor *jeden programový čítač*, pričom *inštrukcia riadi niekoľko* funkčných jednotiek. V jednom inštrukčnom cykle klasický DSP umožňuje prístup k dvom pamäťovým miestam, vykonáva MAC operáciu a modifikuje obsahy registrov v adresových aritmetických jednotkách, pričom napr. LIW inštrukcia 32-bitového procesora TMS320C30

```
        MPYI3  src2, src1, dst1
||      STI            src2, dst2
```

má formát zobrazený na obr. 3.7 , pričom dĺžka jednotlivých polí inštrukcie súvisí s počtom a typom registrov procesora ako aj s jeho architektúrou. Táto inštrukcia realizuje paralelné vykonanie (symbolicky označované znakom ||) celočíselného násobenia (MPYI3) a celočíselného presunu (STI) a v závislosti na forme parametrov src a dst aj operácie v adresových aritmetických jednotkách a teda jedna inštrukcia LIW je využívaná viacerými funkčnými jednotkami.

Obr. 3.7 Formát inštrukcie MPYI3 || STI procesora TMS320C30

Pokroky v technológii VLSI umožnili zvýšiť počet funkčných jednotiek nad rámec využiteľný v tradičných DSP. VLIW DSP zvyšujú využívanie paralelizmu priradením samostatnej časti (slotu) v celkovej inštrukcii pre každú funkčnú jednotku, čo je principiálne znázornené na obr. 3.8 .



Obr. 3.8 Princíp architektúry VLIW

Klasická architektúra VLIW je z pohľadu DSP zaujímavá predovšetkým z týchto dôvodov:

a) Využívanie paralelizmu
Inštrukcie v jednotlivých slotoch VLIW inštrukcie sú vyberané, dekódované a vykonávané paralelne, čo umožňuje výrazné zvýšenie výkonnosti VLIW DSP. Inštrukcie v jednotlivých slotoch sú vykonávané v samostatných funkčných jednotkách. Počet funkčných jednotiek je v súčasnosti typicky okolo 10-20 (TMS320C62x má 8), pričom existujú aj multimediálne procesory s 27 funkčnými jednotkami (Trimedia TM-1000). Dĺžka VLIW inštrukcie súčasných jednočipových VLIW DSP – rádovo stovky bitov (napr. $32 \times 8 = 256$ bitov pre TMS320C62x) je síce z pohľadu klasických jednočipových procesorov vysoká, je však potrebné zdôrazniť, že v oblasti superpočítačov je bežne využívaná dĺžka inštrukčného slova niekoľko tisíc bitov, takže je možné očakávať, že v budúcnosti by sa dĺžka inštrukčného slova v prípade jednočipových VLIW DSP ešte mohla zvýšiť.

b) Využívanie zreťazenia
Taktovacie frekvencie VLIW DSP majú zvyčajne hodnotu[8] minimálne niekoľko stoviek MHz, čo vyžaduje široké využívanie zreťazenia. Naviac VLIW DSP procesory typicky využívajú architektúru 1X podobne ako rozšírené DSP.

c) Riadenie (plánovanie) súbežnosti
V prípade architektúry VLIW je analýza a plánovanie paralelizmu realizované *počas prekladu*, čo výrazne zjednodušuje technickú konštrukciu VLIW DSP a umožňuje dosahovať vysoký výpočtový výkon pri relatívne malých nárokoch na zložitosť čipu.

d) RISC inštrukcie
Inštrukcie v jednotlivých slotoch *klasickej architektúry* VLIW predstavujú jednoduché a nezávislé inštrukcie, ktoré vychádzajú z princípu *redukovanej inštrukčnej sady*. Tento typ inštrukcií uľahčuje preklad programov z vyšších programovacích jazykov, čo je vzhľadom na zložitosť programovania VLIW architektúry veľmi dôležité.

Podobne ako v klasických RISC procesoroch sú aj vo VLIW DSP aritmetické operácie realizované ako operácie typu register-register nad *registrami zo súboru registrov* a prístup do pamäte je realizovaný samostatnými inštrukciami.

[8] V súčasnosti (rok 2017) majú najvýkonnejšie verzie procesorov TI a NXP s VLIW architektúrou taktovacie frekvencie typicky presahujúce 1 GHz.

e) Ortogonalita architektúry
Ľubovoľný register zo súboru registrov môže byť operandom v ľubovoľnej inštrukcii. Aj keď funkčné jednotky v architektúre VLIW DSP nie sú identické, ich štruktúra vychádza z princípu, že najfrekventovanejšie inštrukcie sa môžu realizovať vo väčšine funkčných jednotiek. Veľmi často sú funkčné jednotky zoskupené do dvoch, prípadne viacerých *identických* dátových ciest, čo zvyšuje pravdepodobnosť výskytu voľnej funkčnej jednotky, ktorá môže danú inštrukciu vykonať.

f) Využitie prekladačov
Aj keď to znie na prvý pohľad paradoxne, jedným z cieľov VLIW DSP je umožniť písanie (efektívnych) programov vo vyšších programovacích jazykoch. Aj keď je architektúra VLIW DSP zložitejšia ako architektúra klasických DSP, využívanie RISC inštrukcií a ortogonalita architektúry umožňuje prekladačom (typicky z jazyka C, prípadne C++) založených na najnovších *optimalizačných algoritmoch*, dosahovať *vysokú účinnosť* generovaného kódu, ktorá je v rámci klasických DSP nedosiahnuteľná.

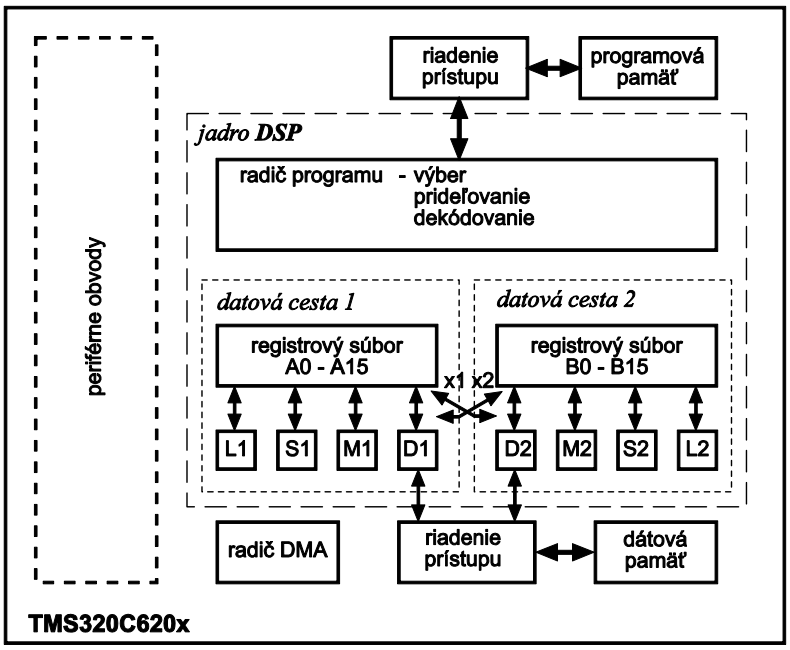
PRÍKLADY ARCHITEKTÚR VLIW DSP

Aj keď jednotliví výrobcovia VLIW DSP využívajú základný princíp architektúry VLIW, snažia sa eliminovať predovšetkým jej základnú nevýhodu – výrazné zvýšenie pamäťových nárokov, čo je v oblasti architektúr jednočipových DSP stále jeden z kritických parametrov. V ďalších častiach sú stručne uvedené základné architektúry VLIW DSP of firiem TI a Lucent naznačiť ich základné vlastnosti a princíp. Podrobnejšie a aktuálne informácie o dosahovaných parametroch týchto DSP je možné nájsť na aktuálnych web stránkach výrobcov.

VELOCITI – TEXAS INSTRUMENTS

Do kategórie VLIW DSP, ktoré TI označuje termínom *VelociTI architektúra*, patria DSP z rodiny TMS320C6x, ktoré z pohľadu klasických DSP (už) nemajú štandardnú MAC jednotku ani špecializované adresové aritmetické jednotky. Ako prvý bol dostupný DSP s pevnou rádovou čiarkou – TMS320C6201, ktorý má 8 nezávislých funkčných jednotiek a pri taktovacej frekvencii 200 MHz v polovici 90-tych rokov umožňoval realizovať 1600 MIPS (alebo 200 VLIW MIPS). Ďalším členom bol DSP s pohyblivou rádovou čiarkou TMS320C6701, ktorý je objektovo aj pinovo kompatibilný s TMS320C6201 a obsahuje rovnakú množinu 8 funkčných jednotiek z ktorých je 6 rozšírených o podporu aritmetiky v pohyblivej rádovej čiarke.

Štruktúra dátových ciest procesora TMS320C62x je znázornená na obr. 3.9 . Je zložená z dvoch identických dátových ciest, z ktorých každá je tvorená funkčnými jednotkami L, S, M a D. Každá dátová cesta obsahuje šestnásť 32-bitových registrov. Registrový súbor má 16 portov (10 čítacích a 6 zapisovacích), pričom každá z dátových ciest má prístup k operandom zo susedného registrového súboru (porty 1X, 2X).



Obr. 3.9 Štruktúra dátových ciest procesora TMS320C62x od firmy Texas Instruments

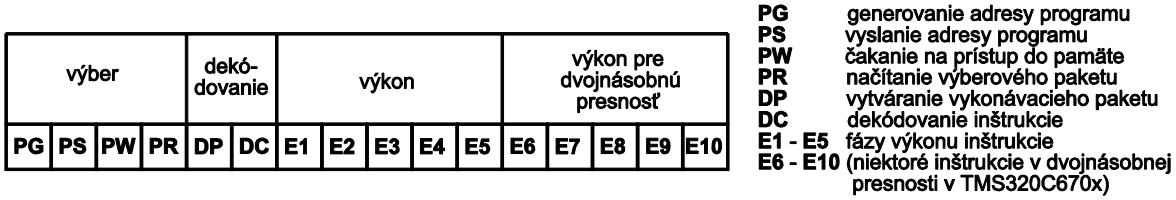
Každá z funkčných jednotiek môže vykonávať 32-bitové celočíselné operácie. Jednotky S a L môžu naviac vykonávať aj 40-bitové operácie a minimalizovať tak problémy s pretečením pri operáciách MAC. Väčšina funkčných jednotiek je ďalej rozdelená na subjednotky, ktoré vykonávajú rôzne činnosti uvedené v tab. 3.1 .

Tab. 3.1 Štruktúra funkčných jednotiek TMS320C62x

Jednotka L	Jednotka S	Jednotka D	Jednotka M
Sčítanie	Sčítanie	Sčítanie	Násobenie
Logické operácie	Logické operácie	Prístup do pamäte	
Bitové počítanie	Bitové manipulácie		
	Posuny		
	Konštanty		
	Vetvenie		

Všetky funkčné jednotky môžu začať inštrukciu v každom takte, pričom jednotlivé fázy zreťazenia zobrazené na obr. 3.10 sú rozdelené do troch kategórií:

- ***výber inštrukcie*** – 4 fázy zreťazenia
- ***dekódovanie inštrukcie*** – 2 fázy zreťazenia
- ***výkon inštrukcie*** – maximálne 10 fáz zreťazenia, pričom viac ako 90 % inštrukcií TMS320C62x využíva iba 5 fáz, posledných 5 fáz využívajú len inštrukcie pre podporu aritmetiky v dvojnásobnej presnosti.



Obr. 3.10 Fázy zreťazenia TMS320C62x

Úroveň zreťazenia je premenlivá (závislá na type inštrukcie a type funkčnej jednotky) a väčšia ako u klasických a rozšírených DSP.

Výkonnosť procesora TMS320C62x pri realizácii niektorých typických algoritmov ČSS je dokumentovaná v tab. 3.2 , pričom rýchlosť procesora je približne 5 až 10-násobne vyššia ako výkonnosť DSP založených na klasickej harvardskej architektúre.

Tab. 3.2 Výkonnosť TMS320C62x/200 MHz pre vybrané algoritmy ČSS

Algoritmus	Parametre	Zložitosť (cykly)	Hodnoty parametrov	Cykly	Čas pre 200 MHz
FIR	M výstupov N koeficientov	0,5MN	M = 100 N = 32	1618	8 μs
Komplexný FIR	M výstupov N koeficientov	2MN	M = 100 N = 32	6410	32 μs
LMS FIR	M výstupov N koeficientov	1,125MN	M = 100 N = 32	5105	25,5 μs
Lattice analýza	N koeficientov	1,5N	N = 10	25	125 ns
Lattice syntéza	N koeficientov	2N	N = 10	38	190 ns
IIR	N bikvadov	4N	N = 10	56	28 ns
Komplexná FFT	N bodov	$1{,}25N\log_2 N$	N = 1024	13228	66 μs
Vektor Max	N rozmerný	0,5N	N = 100	64	320 ns
8×8 DCT	–	–	–	230	1,15 μs
8×8 IDCT	–	–	–	226	1,13 μs
Viterbiho IS54 dekodér	N bodov	66N	N = 89	5874	29,67 μs

Táto vysoká výkonnosť je dosiahnutá jednak možnosťou realizácie dvoch operácií MAC v jednom inštrukčnom cykle, zvýšenou taktovacou frekvenciou procesora a prítomnosťou ďalších funkčných jednotiek, ktoré sú *značne univerzálne*, čo je principiálne odlišné od *úzko špecializovaných jednotiek* (napr. adresové aritmetické jednotky) DSP založených na harvardskej architektúre. Vysoká ortogonalita architektúry a pokrok v technológii prekladačov sa naviac odráža vo vysokej účinnosti kódu generovaného prekladom z jazyka C.

JADRO STAR CORE – LUCENT A MOTOROLA

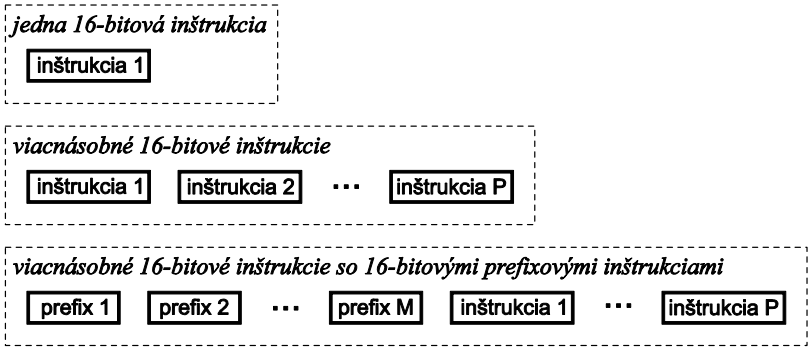
TI má oproti ostatným výrobcom DSP na trhu dlhodobo značnú prevahu. Prejavuje sa to jednak vo veľkom množstve rôznych typov DSP, ktoré TI vyrába ako aj určitým technologickým náskokom, ktorý má pred ostatnými výrobcami (uviedol napr. prvý univerzálny VLIW DSP podstatne skôr ako jeho konkurenti). Odpoveďou dvoch ďalších najväčších výrobcov DSP, firiem Lucent a Motorola bolo *vytvorenie spoločnej aliancie* v júni 1998 s cieľom vyvinúť konkurenčnú architektúru univerzálneho jadra

VLIW DSP, ktoré dostalo názov architektúra Star Core 100 . Cieľom je použitie jadra na báze tejto architektúry v rôznych produktoch, ktoré bude Motorola a Lucent vyrábať samostatne. Jadro Star Core 140 (ďalej len Star Core) bolo prvým ohláseným výsledkom ich spoločného vývoja. Aj procesor NXP MSC8156, jeden z najvýkonnejších VLIW DSP s výkonnými koprocesormi využíva princípy jadra StarCore. Toto jadro je z hľadiska architektúry VLIW DSP jadro, ktoré do oblasti VLIW DSP prináša niekoľko zaujímavých myšlienok resp. prístupov:

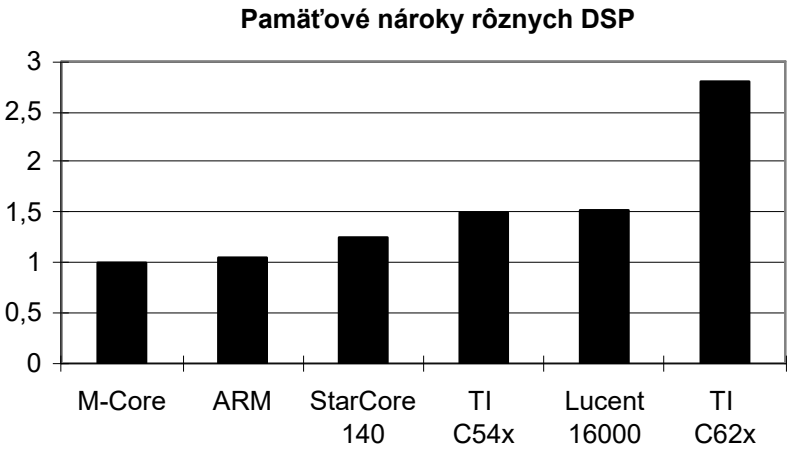
a) Minimalizácia potrebných programových pamätí

Základný problém VLIW architektúry rieši Star Core využitím inštrukcií s premenlivou dĺžkou (VLES – Variable-Length Execution Set) ktoré využívajú 16-bitové inštrukcie, čo je polovičná dĺžka oproti 32-bitovým inštrukciám vo VelociTI architektúre. Pri 16-bitovej šírke inštrukcií však nie je možné zakódovať všetky kombinácie inštrukcií a operandov, a Star Core používa premenlivý počet voliteľných „*prefixových*“ inštrukcií, ktoré sú súčasťou paralelne spracovávaných inštrukcií. Prefixové inštrukcie rozširujú funkčné možnosti základných 16-bitových inštrukcií (napr. umožnením prístupu k väčšej množine registrov). Vo všeobecnosti prefixové inštrukcie ovplyvňujú celú skupinu paralelných inštrukcií, ktoré nasledujú. Princíp VLES je znázornený na obr. 3.11 .

Princíp inštrukcií s premenlivou dĺžkou bol využitý už skôr v DSP16000 od firmy Lucent s cieľom kombinovať výkonnosť 32-bitových inštrukcií, ktorá je potrebná v kritických (z hľadiska rýchlosti) častiach programov s vysokou hustotou kódu a 16-bitové inštrukcie v ostatných častiach kódu. Star Core dosahuje väčšiu hustotu kódu ako DSP na báze harvardskej architektúry a blíži sa z hľadiska hustoty kódu k univerzálnym jednočipovým procesorom (M-Core, ARM), čo je naznačené na obr. 3.12 (tieto výsledky sú založené na súbore interných testov z oblasti algoritmov ČSS, kryptografických a riadiacich algoritmov používaných firmou Motorola).



Obr. 3.11 Princíp inštrukcií VLES využívaný v architektúre Start Core



Obr. 3.12 Porovnanie relatívnej hustoty kódu jednotlivých architektúr

b) Škálovateľnosť (scalability) architektúry

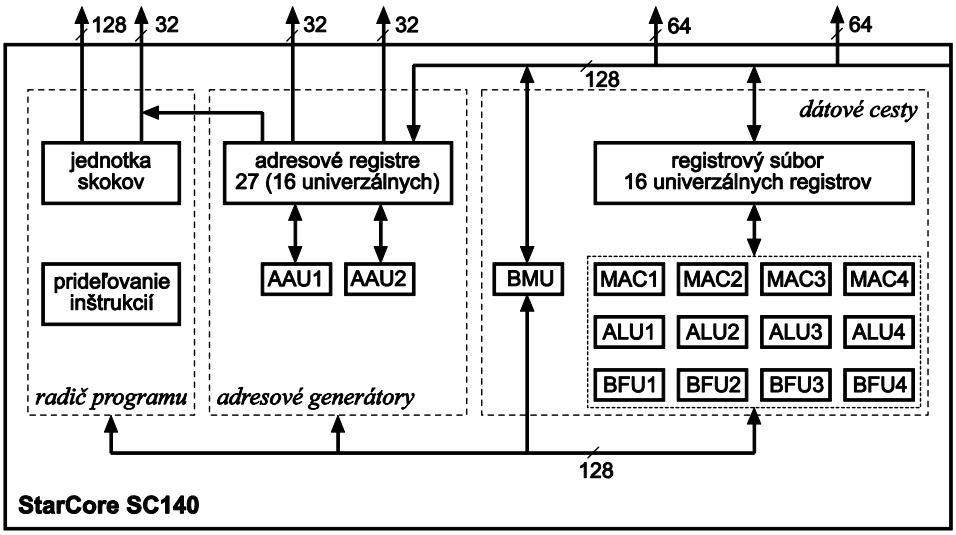
Jedným zo základných cieľov architektúry Star Core je poskytnúť jadro DSP, ktoré by mohlo byť využívané v širokej triede produktov od nízkopríkonových až po vysokovýkonné jednočipové systémy. Motívom k tomuto cieľu je snaha znížiť vývojové náklady na podporu stále sa rozširujúcich aplikácii ČSS, ktoré z hľadiska výrobcov čipov neustále narastajú. *Škálovateľnosť* zahrňuje možnosti zmeny:

- taktovacej frekvencie,
- šírky dát,
- počtu a typu funkčných jednotiek,
- počtu a šírky zberníc na čipe,
- priepustnosti pamäťového systému,
- počtu registrov,
- počtu a typu inštrukcií uskutočnených v jednom cykle.

c) Výkonná architektúra

Prvé ohlásené jadro DSP Star Core SC140 obsahovalo 12 výkonných funkčných jednotiek, z toho 4 ALU, 4 BFU a 4 MAC jednotky, čo je 2-krát viac ako v konkurenčnej VelociTI architektúre. Všetky štvorice jednotiek sú *identické*, čo prispieva k vysokej ortogonalite architektúry. V architektúre sú naviac 2 špecializované adresové aritmetické jednotky AAU[9]), jedna jednotka pre manipuláciu s bitmi (BMU – Bit Manipulation Unit) a jedna jednotku vetvenia (BU – Branch Unit). Architektúra jadra Star Core je znázornená na obr. 3.13 . V rámci jednej VLES inštrukcie môže jadro vykonať paralelne 6 inštrukcií, napr. 4 MAC inštrukcie a dva presuny. Táto kombinácia inštrukcií je ekvivalentná desiatim RISC inštrukciám vo VelociTI architektúre, ktorá na výpočet MAC operácie vyžaduje dve RISC inštrukcie. Predovšetkým prítomnosť štyroch MAC jednotiek umožňuje jadru dosiahnuť vyššiu relatívnu výkonnosť oproti konkurenčným výrobkom firmy TI, čo je znázornené v tab. 3.3 .

[9] Napr. VelociTI architektúra nemá žiadnu špecializovanú jednotku na generovanie adries.



Obr. 3.13 Blokový diagram jadra Start Core SC140

Tab. 3.3 Porovnanie výkonnosti jadra Star Core s produktmi TI

Algoritmus	TI C6x	TI C54x	SC 140	**SC140** oproti **C6x**	**SC140** oproti **C54x**
Reálny FIR	N/2	N	N/4	2x	4x
Komplexný FIR	2N	4N	N	2x	4x
LMS FIR	3N	3N	N	3x	3x
Bikvad (4 násobenia)	4N	5N	1,5N	2,67x	3,33x
Korelácia	N/2	2N	N/4	2x	8x
G1 vo VSELP	N	2N	N/2	2x	4x
Radix 2 FFT	4N	8N	2N	2x	4x

V každom takte je zo 16 funkčných jednotiek aktívnych *maximálne* 6, čo je z pohľadu tradičných VLIW značný odklon od pôvodnej koncepcie (*na čipe je viac paralelných jednotiek, ako môže VLES inštrukcia využiť*). Ďalšia novinka je v spôsobe prideľovania inštrukcií k jednotlivým funkčným jednotkám. Analýza a plánovanie paralelizmu je realizované počas prekladu ako u klasickej VLIW architektúry, mapovanie inštrukcií ku konkrétnym paralelným jednotkám je však realizované dynamicky počas vykonávania programu. Tento prístup je priamou podporou pre možnosť škálovateľnosti budúcich verzií procesorov, ktoré budú môcť využívať rôzny počet funkčných jednotiek.

Jadro používa 5-úrovňové zreťazenie tvorené fázami predvýberu inštrukcie, výberu inštrukcie, dekódovania/prideľovania (dispatch), generovania adresy a vykonania inštrukcie. Toto, relatívne nízkoúrovňové zreťazenie, prispieva k relatívne ľahšiemu programovaniu v asembleri a efektívnemu spracovaniu skokov a prerušení.

3.2.3 ARCHITEKTÚRA SIMD

TIGERSHARC – ANALOG DEVICES

Firma Analog Devices je tradičným výrobcom vysokovýkonných DSP s pohyblivou rádovou čiarkou, pričom samozrejme vyrába aj úspešný rad procesorov s pevnou rádovou čiarkou (vrátane DSP s jadrami Blackfin).

Jej najvýkonnejšie procesory s pohyblivou rádovou čiarkou pre paralelné systémy vychádzajú historicky z procesorov Sharc ADSP2106x. Tieto DSP obsahujú na čipe typicky veľké množstvo SRAM pamätí. Po uvedení TMS320C67x firmou TI však Analog Devices stratil čelnú pozíciu v tejto oblasti.

Analog Devices na tento stav zareagoval vývojom druhej generácie, čipu ADSP2116x, ktorý nazval *Hammerhead* a uviedol začiatkom roku 1999. Zvýšenie výpočtového výkonu u tohto čipu je dosiahnuté jednak zvýšením taktovacej frekvencie na 100 MHz a tiež pridaním druhej paralelnej dátovej cesty. Táto druhá dátová cesta však môže byť využitá len pomocou SIMD inštrukcií paralelne s prvou, pričom tento čip nedosahuje výkonnosť TMS320C67x. Určitou výhodou bola kompatibilta na úrovni asemblerovského kódu s ADSP2106x, pre maximálne využitie čipu však kód musí byť prepísaný s cieľom využiť SIMD inštrukcie. Ďalšou generáciou boli procesorov Sharc bol TigerSharc. Základné vlastnosti procesorov a porovnanie s konkurenciou v čase ich uvedenie (okolo roku 2000) sú dokumentované tab. 3.4 .

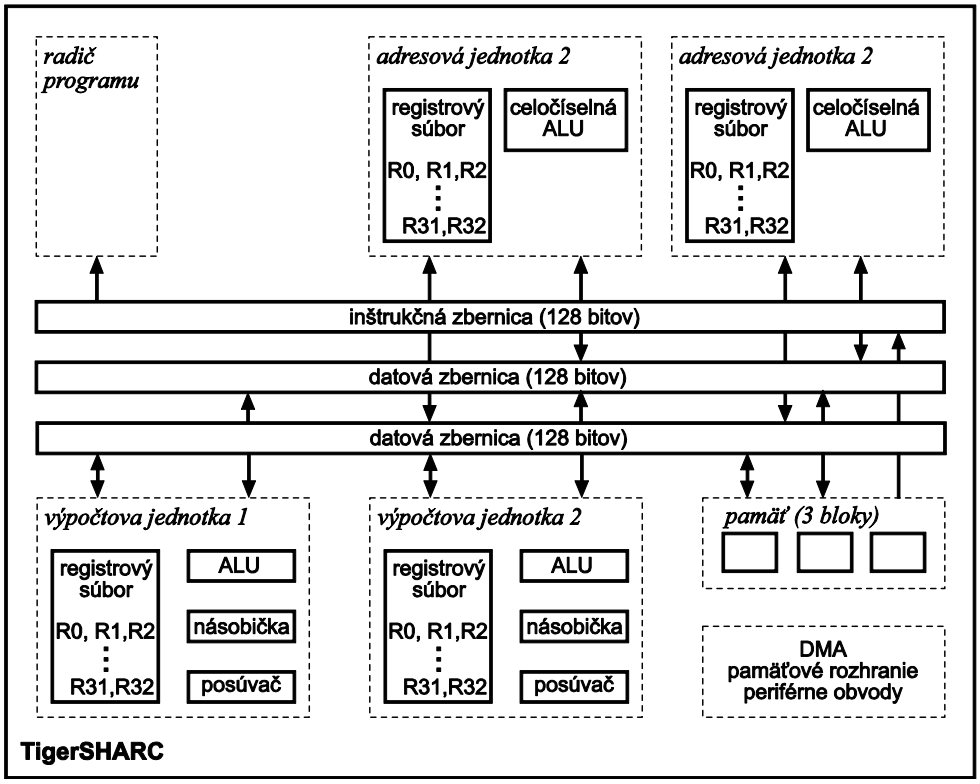
Tab. 3.4 Porovnanie architektúr Sharc, VelociTI a Star Core

Výrobca	Analog Devices			Lucent/ Motorola	Texas Instruments	
Typ	2106x	2116x	–	–	320C26x	320C27x
Názov	Sharc	Hammerhead	TigerSharc	StarCore	VelociTI	VelociTI
Dostupnosť	1994	1Q1999	Stred 1999	1Q2000	1Q1997	3Q1998
Kompatibilita	–	Asembler 2106x	–	–	–	Binárna s 320C62x
Architektúra	DSP	DSP+ SIMD	VLIW+ SIMD	VLIW	VLIW	VLIW
Takt. frekv.	60 MHz	100 MHz	250 MHz	300 MHz	250 MHz	167 MHz
16b MAC/s	60 miliónov	200 miliónov	2000 miliónov	1200 miliónov	500 miliónov	333 miliónov
FP MAC/s	60 miliónov	200 miliónov	500 miliónov	–	–	333 miliónov

Vysoká výpočtová výkonnosť procesora TigerSharc v oblasti aritmetiky s pevnou rádovou čiarkou je dosiahnutá SIMD rozšírením pôvodných dátových ciest pre pohyblivú rádovú čiarku. Architektúra procesora TigerSharc je znázornená na obr. 3.14 a obsahuje riadiacu jednotku, 2 adresové aritmetické jednotky, 2 výpočtové jednotky, pamäť, radič DMA a rôzne periférne jednotky.

Táto architektúra je prakticky totožná s architektúrou DSP na báze harvardskej architektúry (okrem dvoch výpočtových jednotiek, ktoré z nej vytvárajú VLIW architektúru), pričom vysoká priepustnosť v rámci čipu je podporovaná 128-bitovými zbernicami. Každá výpočtová jednotka obsahuje násobičku, ALU a posúvač, pričom tieto subjednotky môžu spracovať 32 alebo 64-bitové operandy a podporujú operácie v pevnej aj pohyblivej rádovej čiarke. V prípade 64-bitových operandov sa používajú dva

spojené 32-bitové registre. Z pohľadu architektúry VLIW DSP sú najväčšou novinkou SIMD rozšírenia. Dátové presuny a niektoré výstupy z násobičky môžu byť široké až 128 bitov, čo je dosiahnuté spojením štyroch 32-bitových registrov.



Obr. 3.14 Architektúra procesora TigerSharc

SIMD rozšírenia výpočtových jednotiek

Voliteľne môže byť výpočet v obidvoch dátových cestách riadený v jednom takte *jednou (spoločnou) inštrukciou*. Z tohto pohľadu je TigerSharc SIMD procesorom. Naviac však procesor využíva ďalší *hierarchický SIMD* prístup na úrovni jednotlivých registrov, pričom môže pracovať s hodnotou v ľubovoľnom 32-bitovom registri ako s 32-bitovou hodnotou v pohyblivej rádovej čiarke vo formáte IEEE 754, jednou 32-bitovou, dvomi 16-bitovými alebo štyrmi 8-bitovými hodnotami v pevnej rádovej čiarke. Takéto hierarchické usporiadanie je bežné predovšetkým u výkonných procesorov pre všeobecné použitie, v oblasti DSP však v čase uvedenia bolo nezvyčajné[10] a je široko využívané pri spracovaní multimediálnych dát akými sú zvuk, obraz a videosekvencie.

Každá výpočtová jednotka môže realizovať v každom cykle jednu 32-bitovú operáciu MAC v pohyblivej rádovej čiarke, dve $32\times32\rightarrow64$ alebo štyri $16\times16\rightarrow32$-bitové MAC v pevnej rádovej čiarke, čo vzhľadom na dve paralelné jednotky umožňuje dosiahnuť výpočtový výkon uvedený v tab. 3.4 . Internými dátovými zbernicami je možné preniesť 8 Gbajtov/s ($2\ \textit{jednotky}\times250\ \textit{MHz}\times128\ \textit{bitov}$), čo predstavuje priepustnosť šestnásť 16-bitových slov/cyklus. Cez programovú zbernicu je možné

[10] Tento trend naznačuje určitú konvergenciu DSP s multimediálnymi rozšíreniami univerzálnych a multimediálnych procesorov.

preniesť paralelne štyri 32-bitové inštrukcie a tak je možné v každej jednotke realizovať paralelne napr. aritmetické operácie a posuny.

Zreťazenie procesora je 8-úrovňové, využíva vzájomné blokovanie a operácie násobenia, sčítania a čítania dát majú oneskorenie 2 cykly. TigerSharc bol prvým univerzálnym VLIW DSP, ktorý výrazne využíval SIMD rozšírenia. To umožňuje architektúre TigerSharc dosiahnuť vysoký výpočtový výkon. Hierarchická SIMD architektúra však z pohľadu programovania znamená znásobenie problémov ktoré sú typické pre klasické DSP na báze harvardskej architektúry. Z pohľadu užívateľov je dôležitá predovšetkým podpora v oblasti efektívnych knižničných funkcií, ktoré budú využívať rozšírenia SIMD. V súčasnosti sú tieto procesory dodávané s taktovacou frekvenciou 500 MHz a dominujú hlavne v aplikáciách spracovania audio signálov.

4 ČÍSLICOVÉ FILTRE (OPAKOVANIE)

Číslicové filtre a efektívne algoritmy výpočtu **DFT** (Discrete Fourier Transform), ktoré využívajú rýchle algoritmy výpočtu založené na algoritme **FFT** (Fast Fourier Transform) patria medzi základné algoritmy ČSS. Všetky prakticky využívané **DSP** majú architektúru optimalizovanú minimálne práve pre tieto dva základné typy algoritmov ČSS. Pomocou týchto algoritmov je možné demonštrovať prakticky všetky základné vlastnosti jadier klasických DSP a tiež podstatnú časť vlastností moderných DSP. V rámci práce s reálnym DSP hardvérom budú tieto vlastností demonštrované na relatívne moderných procesoroch firmy **Analog Devices ADSP Blackfin** s 16-bitovou aritmetikou v pevnej rádovej čiarke.

Vzhľadom na to, že problematika číslicových filtrov a FFT je podrobne preberaná na KEMT FEI TU v Košiciach v iných špecializovaných predmetoch, cieľom je len zopakovanie základných pojmov z tejto oblasti a opísanie **niektorých praktických metód návrhu** číslicových filtrov v prostredí Matlab s nainštalovaným Signal Processing Toolboxom ako aj špecializovanom voľne dostupnom programu **FilterExpress** od firmy Systolix. Aj keď program FilterExpress je voľne dostupný, umožňuje navrhnúť a analyzovať aj pomerne zložité praktické číslicové filtre. V prípade extrémnych nárokov na parametre navrhovaných filtrov resp. ich analýzu je možné použiť komerčne dostupné balíky – napr. program Filter Solutions od firmy Nuhertz (licencia dostupná v laboratóriu vstavaných systémov na KEMT).

4.1 FILTRE S KONEČNOU IMPULZOVOU ODPOVEĎOV

Filtre s konečnou impulzovou odpoveďou (FIR- Finite Impulse Response) patria medzi najjednoduchšie číslicové filtre ako z teoretického, tak aj z realizačného hľadiska. FIR filter N -tého rádu je možné opísať **prenosovou funkciou**

$$H_{FIR}(z) = \sum_{k=0}^{N} h[k] z^{-k} \tag{4.1}$$

pričom $h[k]$ sú **koeficienty** FIR filtra. **Frekvenčná charakteristika** $H^f(\theta)$ FIR filtra (**komplexná funkcia** reálnej premennej θ) je určená vzťahom

$$H^f(\theta) = H_{FIR}(z)\Big|_{z=e^{j\theta}} \tag{4.2}$$

pričom $\theta \in \langle -\pi, \pi \rangle$ (alebo $\theta \in \langle 0, 2\pi \rangle$) je normalizovaná frekvencia aktuálnej frekvencie F (napr. v Hz, kHz a pod.)

$$\theta = 2\pi \frac{F}{F_{samp}} \tag{4.3}$$

a F_{samp} je použitá **frekvencia vzorkovania**.

Keďže frekvenčná charakteristika je komplexnou funkciou, v praxi sa využívajú amplitúdové a fázové frekvenčné charakteristiky, ktoré sú reálnymi funkciami reálnej premennej θ. V prostredí Matlab je možné zobraziť obidve charakteristiky pomocou príkazu *freqz.*

Príklad

Zobrazte v prostredí Matlab amplitúdovú a fázovú frekvenčnú charakteristiku FIR filtra s koeficientmi $h_i = 0.1, \quad i = 0,1,\ldots 9$.

Riešenie

Príklad vyriešime v prostredí Matlab zapísaním nasledujúcich príkazov

```
h=[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1];
freqz( h );
```

Medzi charakteristické vlastnosti FIR filtrov patrí ich **stabilita** (t.j. kvantovaním koeficientov sa FIR filter nemôže stať nestabilným) a možnosť realizácie FIR **filtra s lineárnou fázovo-frekvenčnou charakteristikou**. Nutnou a postačujúcou podmienkou aby FIR filter mal lineárnu fázovú frekvenčnú charakteristiku je symetria jeho koeficientov, ktorú je možné vyjadriť v tvare

$$h[k] = h[N-k], \quad k = 0,1,\ldots N \tag{4.4}$$

4.2 FILTRE S NEKONEČNOU IMPULZOVOU ODPOVEĎOV

Pre **filtre s nekonečnou impulzovou odpoveďou** (IIR – Infinite Impulse Response) je charakteristická prítomnosť spätnej väzby[1], čo sa v prípade prenosovej funkcie prejaví existenciou menovateľa prenosovej funkcie $H_{IIR}(z)$, ktorý je na rozdiel od $H_{FIR}(z)$ rôzny od 1 (pre jednoduchosť uvažujme, že rád čitateľa a menovateľa sú zhodné a rovné N):

$$H_{IIR}(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} \tag{4.5}$$

Implementáciu IIR filtra opísaného prenosovou funkciou (4.5) je možné realizovať viacerými spôsobmi. Najznámejšie a v praxi najčastejšie využívané sú realizácie založené na rozklade (4.5) do sekcií druhého rádu, tzv. **bikvadov** podľa vzťahu[2] (za predpokladu, že N je párne)

[1] Existuje však aj možnosť realizácie FIR filtrov pomocou štruktúr so spätnou väzbou. Typickým príkladom je realizácia tzv. hrebeňového (comb) FIR filtra v tvare $y(n) = y(n-1) + x(n) - x(n-N)$, ktorý za predpokladu vhodných počiatočných podmienok realizuje FIR filter $y(n) = x(n) + x(n-1) + \ldots + x(n-N+1)$.

[2] Niektoré literárne pramene resp. programové balíky používajú pre koeficienty a_k záporné znamienko. Pri praktickom návrhu IIR filtra je preto potrebné správne určenie znamienka, pretože jeho nesprávnym určením sa IIR filter zvyčajne stane nestabilným.

$$H_{IIR}(z)=\prod_{k=1}^{N/2}\left(\frac{b_{0k}+b_{1k}z^{-1}+b_{2k}z^{-2}}{1+a_{1k}z^{-1}+a_{2k}z^{-2}}\right)=\prod_{k=1}^{N/2}H_k(z) \tag{4.6}$$

pričom prenosová funkcia *k*-tého bikvadu $H_k(z)$ je určená vo všeobecnom prípade piatimi koeficientmi $b_{0k}, b_{1k}, b_{2k}, a_{1k}, a_{2k}$. IIR filtre sú charakteristické predovšetkým podmienenou stabilitou, pričom po kvantovaní koeficientov sa IIR filter môže stať nestabilným. Jedným z hlavných dôvodov rozkladu IIR filtra na kaskádne zapojenie bikvadov je predovšetkým podstatne nižšia citlivosť kaskády bikvadov na kvantovanie koeficientov IIR filtra. Naviac, v prípade bikvadov je možné jednoznačne definovať oblasť v ktorej sa musia nachádzať koeficienty bikvadov tak, aby boli všetky bikvady (a tým aj celý IIR filter) stabilné(ý).

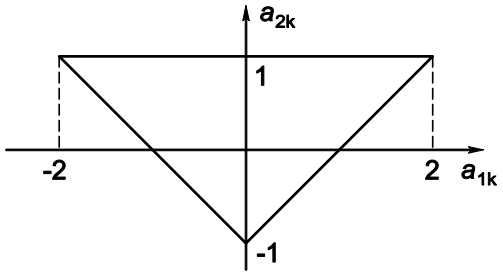
Nutnou podmienkou stability bikvadu je, aby koeficienty menovateľa $H_k(z)$ spĺňali podmienku

$$\left|a_{jk}\right|<2.0 \quad j=1,2 \quad k=1,2,\ldots,N/2 \tag{4.7}$$

Postačujúcou podmienkou stability je splnenie vzťahov

$$-1<a_{2k}<1, \quad 1+a_{1k}+a_{2k}>0, \quad 1-a_{1k}+a_{2k}>0 \tag{4.8}$$

ktoré definujú tzv. trojuholník stability zobrazený na obr.4.1.



Obr.4.1 Trojuholník stability bikvadu, body vo vnútri trojuholníka zodpovedajú stabilnému bikvadu

Z pohľadu implementácie pomocou DSP s pevnou rádovou čiarkou je hodnota 2.0 vo vzťahu (4.7) určitým problémom[3], pretože interval efektívne využiteľných čísel je u typických DSP obmedzený na interval $\langle -1,1)$. V prípade koeficientov $a_i \quad i=1,2,\ldots,N$ vo vzťahu (4.5) nie je možné sformulovať na kontrolu stability podmienky, ktoré by boli také jednoduché ako podmienka (4.7).

Na implementáciu sekcie druhého rádu je možné využiť niekoľko rôznych realizácií. Najznámejšia je **priama forma I**, ktorú je možné opísať diferenčnou rovnicou (index k je pre jednoduchosť vynechaný)

[3] Štandardné DSP tento problém typicky riešia s využitím špecializovaných módov aritmetickej jednotky. V ďalších cvičeniach bude riešenie tohto problému opísané pre ADSP Blackfin od firmy Analog Devices.

$$y(n) = \sum_{j=0}^{2} b_j x(n-j) - \sum_{j=1}^{2} a_j y(n-j) \tag{4.9}$$

Pri technickej realizácii sú často využívané realizácie (tzv. **kanonické formy**), ktoré využívajú minimálny počet stavebných prvkov (napr. oneskorovacích členov, násobičiek, ...). Medzi najčastejšie využívané patria **kanonická forma I** (tzv. **transponovaná priama forma**) opísaná diferenčnou rovnicou

$$y(n) = b_0 x(n) + w_1(n-1) \tag{4.10}$$

$$w_1(n) = b_1 x(n) - a_1 y(n) + w_2(n-1) \tag{4.11}$$

$$w_2(n) = b_2 x(n) - a_2 y(n) \tag{4.12}$$

a **kanonická formy II** (tzv. **priama forma II**) opísaná diferenčnou rovnicou

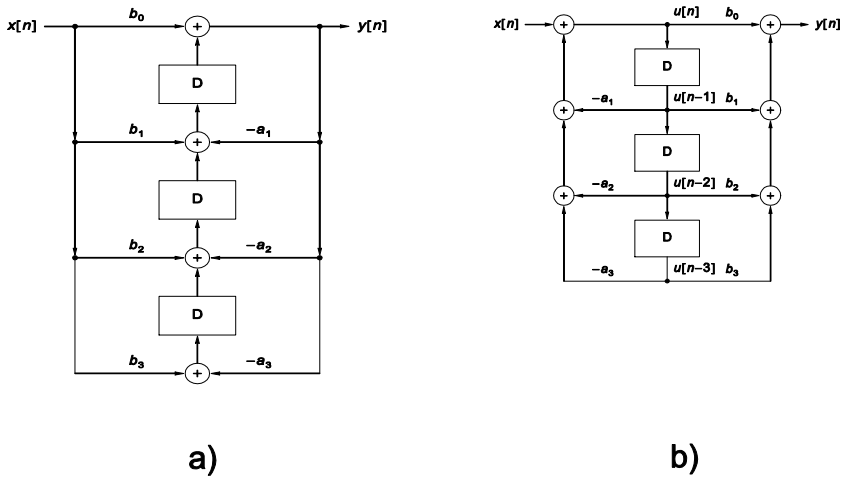
$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) \tag{4.13}$$

$$y(n) = \sum_{j=0}^{2} b_j w(n-j) \tag{4.14}$$

Výhodnou vlastnosťou IIR filtrov v porovnaní s FIR filtrami je menší počet koeficientov, ktoré sú potrebné na realizáciu filtra s určitými vlastnosťami (napr. požadovanou strmosťou medzi priepustným a nepriepustným pásmom).

Príklad
Na základe rovníc (4.10) - (4.14) nakreslite realizáciu kanonických foriem I a II pre IIR filter rádu $N = 3$ *(realizáciu bikvadu dostaneme pre* $N = 2$ *).*

Riešenie
Riešenie je zobrazené na obr. 4.2.



Obr.4.2 Kanonické formy IIR filtra – a) forma I, b) forma II

Príklad
Na základe rovnice (4.9) nakreslite realizáciu priamej formy[4].

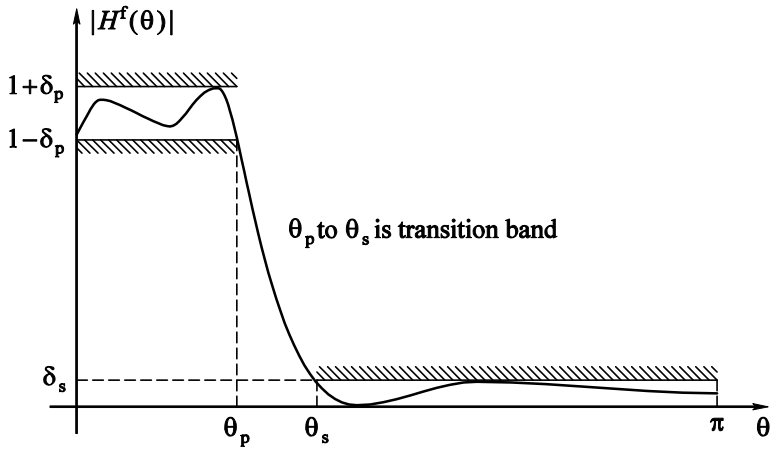
4.3 NÁVRH ČÍSLICOVÝCH FILTROV

V procese návrhu je potrebné určiť predovšetkým **koeficienty** filtra, prípadne aj **rád** filtra. Existujú rôzne **návrhové algoritmy**, ktoré sa líšia **kritériom a metódami optimalizácie**. Tieto algoritmy sú preberané v iných špecializovaných predmetoch a preto sa ich opisom nebudeme podrobnejšie zaoberať. Sústredíme sa predovšetkým na praktické zvládnutie niektorých vybraných návrhových algoritmov s cieľom navrhnúť číslicový filter (typu FIR aj IIR) s požadovanými parametrami.

Medzi najčastejšie požiadavky[5] v prípade návrhu číslicového filtra patrí **tolerančná schéma**, ktorá je v prípade DP filtra (dolný priepust) naznačená na obr.4.3, pričom δ_p je zvlnenie v priepustnom, δ_s je zvlnenie v nepriepustnom pásme a θ_p, θ_s sú normované frekvencie priepustného resp. nepriepustného pásma. Podobne je možné zakresliť tolerančné schémy pre HP, PZ a PP.

[4] Aj keď uvedená priama forma vyžaduje viac stavebných blokov ako kanonické formy, je často využívaná pri praktickej implementácii práve pomocou DSP.

[5] V praxi sa samozrejme číslicové filtre navrhujú aj podľa iných špecifikácií.



Obr.4.3 Tolerančná schéma DP filtra

NÁVRH ČÍSLICOVÝCH FILTROV V PROSTREDÍ MATLAB

Návrh číslicových filtrov FIR a IIR v prostredí Matlab budeme demonštrovať na príkladoch. Podrobnejší opis všetkých príkazov Matlab-u je možné získať príkazom

```
help prikaz
```

Príklad

Navrhnite DP FIR s pásmom priepustnosti do 3000 Hz, pásmom tlmenia od 5000 Hz, a maximálnym zvlnením v priepustnom a nepriepustnom pásme 0.01 ($20\log_{10}(1.01) \approx 0.09\,\text{dB}$*) resp. 0.001 (*$20\log_{10}(0.001) = -60\,\text{dB}$*). Pre návrh využite program Matlab, pričom frekvencia vzorkovania je 48000 Hz.*

Riešenie

Pre návrh FIR filtrov PP, PZ, DP, HP klasickou metódou oknových funkcií je možné využiť príkaz ***fir1,*** v prípade požiadavky na zložitejšie priebehy amplitúdovej frekvenčnej charakteristiky je možné využiť príkaz ***fir2***. Pri návrhu je potrebné zvoliť typ použitého okna. Metóda oknových funkcií však nie je optimálna (z hľadiska zložitosti navrhnutého filtra, t.j. počtu koeficientov[6] FIR filtra, ktoré sú potrebné na splnenie požiadaviek zadania) a v praxi je často výhodnejšie využiť zložitejšie návrhové algoritmy, ktoré umožňujú navrhnúť FIR filter s menším počtom koeficientov. V ďalšej časti využijeme tzv. **Remezov výmenný algoritmus**, ktorý patrí do kategórie pomerne výkonných návrhových algoritmov. Najskôr určíme rád FIR filtra, ktorý by mal[7] splniť zadané požiadavky:

```
[n,fo,mo,w] = remezord( [3000 5000], [1 0], [0.01 0.001], 48000 );
```

a potom učíme koeficienty navrhnutého FIR filtra:

[6] Počet koeficientov je často kritický predovšetkým pri realizácii filtrov v reálnom čase a preto sa snažíme minimalizovať počet koeficientov navrhovaného FIR filtra.

[7] Splnenie požiadaviek je potrebné spätne overiť a v prípade ich nesplnenia zvýšiť rád filtra.

```
h = remez(n,fo,mo,w);
```

navrhnutý FIR filter má

```
size(h)
```

koeficientov a má symetrické koeficienty podľa vzťahu (4.4) (je to teda FIR filter s lineárnou fázovo-frekvenčnou charakteristikou). Na spracovanie jednej vstupnej vzorky navrhnutý FIR filter vyžaduje $h = 62$ tzv. **MAC**[8] operácií. Koeficienty navrhnutého filtra je možné zobraziť príkazom

```
plot(h)
```

a amplitúdovú a fázovú frekvenčnú charakteristiku príkazom

```
freqz(h)
```

Príklad
Navrhnite DP IIR z predchádzajúceho príkladu a porovnajte zložitosť navrhnutého filtra s FIR filtrom navrhnutým v predchádzajúcom príklade.

Riešenie
Pre návrh IIR je možné opäť využiť niekoľko rôznych návrhových algoritmov. V ďalšej časti vyžijeme návrh tzv. **eliptických filtrov** (niekedy tiež nazývaných Cauerove). Tieto filtre využívajú Čebyševovskú aproximáciu v priepustnom aj nepriepustnom pásme a pri návrhu vyžadujú pomerne zložité výpočty (Jakobiho eliptické funkcie, eliptické integrály). S využitím Matlabu je však ich návrh pomerne jednoduchý. Najskôr určíme rád filtra:

```
Rp=20*log10(1.01);                                          % prepocet na decibely
Rs=-20*log10(0.001);                                        % prepocet na decibely
[n,w]=ellipord(3000/(48000/2), 5000/(48000/2), Rp, Rs);
```

a teraz určíme koeficienty filtra

```
[b,a]=ellip(n,Rp,Rs,w);
```

a jeho amplitúdové a frekvenčné charakteristiky

```
freqz(b,a)
```

Na spracovanie jednej vzorky navrhnutý IIR filter vyžaduje len 13 MAC operácií. Ak potrebujeme zobraziť koeficienty na viac desatinných miest ako 4, použijeme príkaz

```
format long
b
a
```

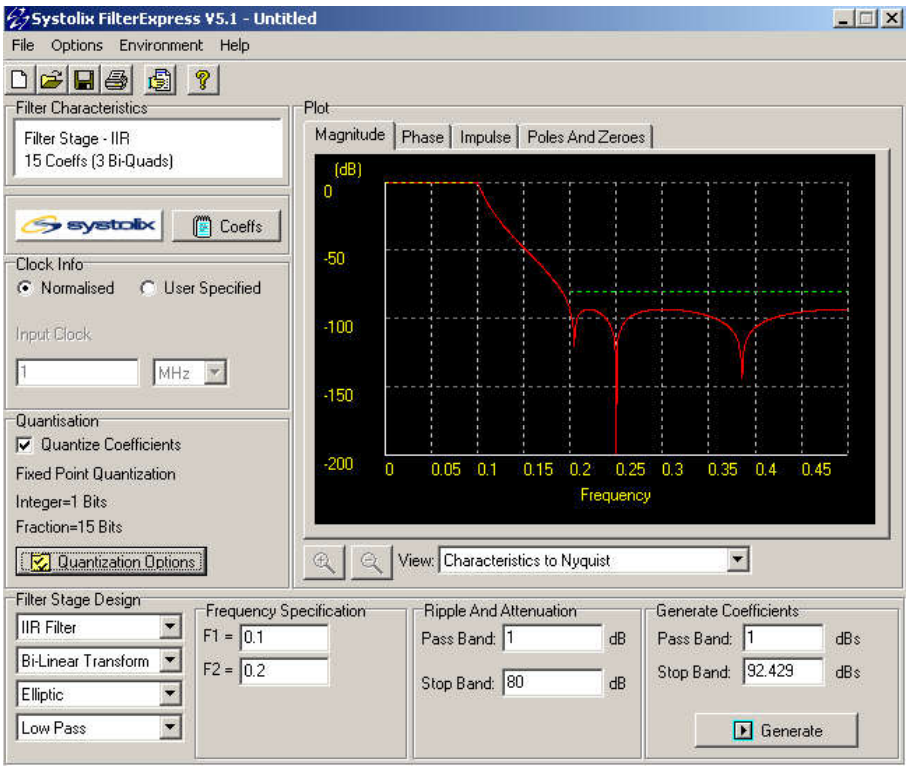
Pri rozklade na sekcie druhého rádu je potrebné zistiť korene polynómov, ktoré je možné nájsť pomocou príkazu **roots** a využiť základné znalosti z algebry o rozklade polynómov.

[8] MAC operácia (**M**ultiply **A**nd a**C**umulate) je operácia typu A=A+X*Y. Operácia MAC je základnou operáciou, pre ktorú sú všetky DSP optimalizované.

Príklad
Rozložte navrhnutý filter na sekcie druhého rádu.

NÁVRH ČÍSLICOVÝCH FILTROV V PROSTREDÍ FILTEREXPRESS

Inou možnosťou (podstatne pohodlnejšou) ako navrhnúť číslicové filtra je využitie špecializovaných návrhových programov, ktoré umožňujú navrhnúť číslicové filtre dokonca bez znalosti akýchkoľvek príkazov. Umožňujú tiež rozklad navrhnutých IIR filtrov na sekcie druhého rádu. Na počítačoch v učebni je dostupný program Systolix FilterExpress zobrazený na obr.4.4, ktorý umožňuje pohodlným spôsobom navrhnúť prakticky využiteľné FIR a IIR filtre, realizáciu rozkladu IIR filtrov na sekcie druhého rádu a kvantovanie koeficientov filtra. Ich ovládanie je intuitívne a nevyžaduje podrobnejší opis. Podrobnejšie informácie je možné získať z menu help.

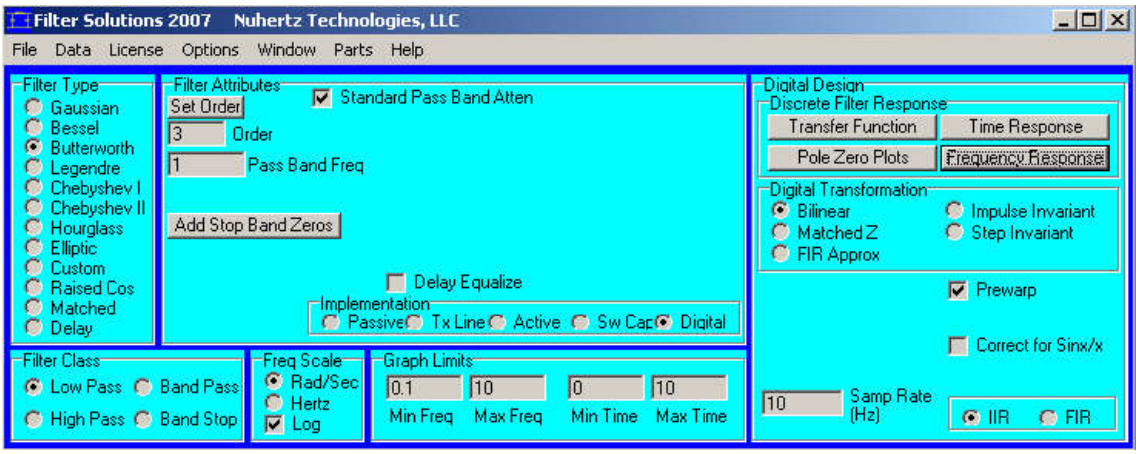


Obr.4.4 Grafické rozhranie programu FilterExpress

Príklad
Analyzujte príklad overenia navrhnutého IIR filtra (v prostredí Filter Express) pomocou simulácie v Matlabe. Nasledujúci m-súbor realizuje výpočet amplitúdovej frekvenčnej charakteristiky IIR filtra realizovaného ako kaskádna sekcia 3 bikvadov. Zároveň je realizovaná simulácia spracovania vstupného signálu tvoreného súčtom dvoch harmonických signálov (jeden z pásma priepustnosti a druhý z pásma tlmenia) príslušného IIR filtra.

```
% Demonstracia overenia navrhu IIR filtra (vystup z programu Filter
% Express, ktory je pouzivany v predmete SPvT).
% Vysledkom navrhu su koeficienty bikvadov. Kaskadnym zapojenim
% bikvadov dostavame IIR filter s pozadovanou charakteristikou.
% Realizacia IIR pomocou bikvadov je vyhodna jednak z pohladu
% stability realizacia v prostredi s konecnou presnostou, ako aj
% vzhladom na optimalizaciu architektur DSP prave pre realizaciu IIR
% filtrov pomocou bikvadov.
%
% SPvT, M.D. 26-02-2007, v.1.01

% Parametre navrhu (pre rpogram Filter Express v.5.1):
% Fvz = 48 kHz  (Input Clock = vzorkovacia frekvencia)
% Quantize Coefficients
% Fixed Point Quantization (Integer Bits = 1, Fraction Bits = 15)
% IIR filter
% Bi-Linear Transform
% Elliptic
% Low Pass
% F1 = 5 kHz (pasmo priepustnosti)
% F2 = 7 kHz (pasmo tlmenia)
% Pass Band 1 dB
% Stop Band 60 dB

fvz = 48000;              % vzorkovacia frekvencia

% Vysledkom navrhu su tieto koeficenty (3 bikvady)
% Program Filter Express realizuje navrh pre prenosovu
% funkciu bikvadu v tvare
% H(z) = (B0 + B1*z^-1 + B2*z^-2)/(1 - A1*z^-1 - A2*z^2)

% bikvad 1
B01 = 0.02215576171875;
B11 = 0.020538330078125;
B21 = 0.02215576171875;
A11 = 1.6365966796875;
A21 = -0.70147705078125;

% bikvad 2
B02 = 0.213592529296875;
B12 = -0.174652099609375;
B22 = 0.213592529296875;
A12 = 1.57125854492188;
A22 = -0.823760986328125;

% bikvad3
B03 = 0.49176025390625;
B13 = -0.58294677734375;
B23 = 0.49176025390625;
A13 = 1.54629516601563;
A23 = -0.946868896484375;

% Pre simulaciu uvazujeme signal zlozeny z 2 harmonickych zloziek
N = 1024;                 % pocet simulovanych vzoriek
f1 = 2000;                % frekvencia 1. harmonickej zlozky (z pasma priep.)
f2 = 8000;                % frekvencia 2. harmonckej zlozky (z pasma tlmenia)
a1 = 0.4;                 % amplituda 1. zlozky
a2 = 0.4;                 % amplituda 2. zlozky
n = (0:N-1);              % diskretny cas
% sucet zloziek je z int <-1,1)
x = a1*sin(2*pi*n*f1/fvz) + a2*sin(2*pi*n*f2/fvz);
```

Ďalším programom, ktorý umožňuje okrem napr. návrhu pasívnych a aktívnych analógových filtrov aj číslicové filtre je program Filter Solutions firmy Nuhertz (www.nuhertz.com), ktorého úvodné grafické rozhranie je zobrazené na obr.4.5. Tento program je k dispozícii v učebni počas cvičení, pričom zo stránky firmy Nuhertz je možné stiahnuť aj limitované resp. evaluačné verzie. Program patrí medzi špičkové komerčné programy pre návrh filtrov a študentom KEMT je k dispozícii v rámci univerzitného programu firmy Nuhertz.



Obr.4.5 Grafické rozhranie programu Filter Solutions firmy Nuhertz

5 PROCES TVORBY PROGRAMOV PRE DSP

Vysoký výpočtový výkon DSP je len jedným z nutných predpokladov pre úspešnú realizáciu výkonných systémov ČSS na báze DSP. Na dosiahnutie vysokého výpočtového výkonu v praktických aplikáciách má podstatný (a často dokonca dominantný) vplyv kvalita dostupných **vývojových prostriedkov**. Aj keď **klasické DSP** s aritmetikou v pevnej rádovej čiarke (napr. Analog Devices ADSP218x, Motorola DSP560xx, Texas Instrument TMS30C5x a pod.) sú v súčasnosti z hľadiska typov vývojových prostriedkov porovnateľné s univerzálnymi jednočipovými mikroprocesormi[1], vzhľadom na typickú požiadavku ich činnosti v reálnom čase je **nízko-úrovňové programovanie** v asembleri často jedinou reálnou alternatívou pri tvorbe programového vybavenia pre túto triedu DSP. V rámci cvičení v predchádzajúcich rokoch, keď bol na cvičeniach nosným DSP procesor Analog Devices ADSP2181, bol dôraz kladený predovšetkým na programovanie v asembleri, ktorý patrí medzi **základné prostriedky** na tvorbu programov.

Súčasné **moderné DSP**, medzi ktoré patria[2] aj procesory firmy **Analog Devices na báze jadra Balckfin** využívané v rámci cvičení, sú však podstatne výkonnejšie[3] ako klasické DSP. Táto skutočnosť vyžaduje **kvalitatívne odlišný prístup** k efektívnej tvorbe programového vybavenia. Programovanie v asembleri je využívané predovšetkým pre tvorbu efektívnych knižničných funkcií resp. pre optimalizáciu kritických častí kódu. Zvyšok aplikácie je v súčasnosti typicky realizovaný vo vyššom programovacom jazyku. Uvedenému trendu sú prispôsobené aj nasledujúce cvičenia, kde programovanie **v jazyku C** tvorí základ programovania cieľových DSP.

Príklady jednotlivých vývojových prostriedkov a väzieb medzi nimi budú demonštrované na vývojových prostriedkoch pre signálové procesory Analog Devices s jadrom Blackfin, ktoré sú súčasťou integrovaného vývojového prostredia **VisualDSP++**. Tieto prostriedky budú využívané aj v rámci ďalších cvičení pri vysvetľovaní základných vlastností jadra procesora Blackfin ako aj pri práci s reálnymi technickými prostriedkami (vývojová doska EZ-KIT21533 Lite od firmy Analog Devices).

[1] V oblasti univerzálnych (jednočipových) mikroprocesorov sú v súčasnosti široko využívané vyššie programovacie jazyky (typicky jazyk C, C++, Embedded C, …), operačné systémy reálneho času a pod. Tieto vývojové prostriedky existujú aj pre klasické DSP, ich využitie v reálnych systémoch na báze DSP je však do určitej miery limitované požiadavkou na prácu v reálnom čase resp. snahou o minimalizáciu veľkosti programových pamätí. V praktických aplikáciách sú preto tieto prostriedky zvyčajne kombinované s kódom v asembleri, ktorý je využitý napr. vo forme optimalizovaných knižníc resp. priamo vložených asemblerovských príkazov.

[2] Pre upresnenie je potrebné uviesť, že procesory Analog Devices s jadrom Blackfin sú kombináciou architektúry DSP a klasického RISC procesora. Uvedená kombinácia je v súčasnosti jednou z najlepších architektúr medzi DSP s aritmetikou s pevnou rádovou čiarkou.

[3] Taktovacie frekvencie moderných DSP sa blížia k hranici 1 GHz, existujú 2-jadrové a viac-jadrové verzie procesorov, procesory umožňujú adresovanie stoviek megabajtov adresového priestoru, ...

5.1 VÝVOJOVÉ PROSTRIEDKY PRE DSP

Do tejto kategórie patria predovšetkým asemblery, linkovacie programy, knižničné funkcie, simulátory, vývojové dosky a emulátory.

5.1.1 ASEMBLERY

Asembler je program, ktorý prekladá špecifické inštrukcie procesora zapísané v zdrojovom ASCII súbore do binárneho **objektového kódu** (object code) pre cieľový DSP. Zvyčajne tento objektový kód (pokiaľ je v tzv. **relatívnom tvare**) vyžaduje dodatočnú transformáciu (**relokáciu** a **linkovanie**), ktorá sa realizuje linkovacím programom (**linkerom**). Linker generuje vykonateľný binárny kód pre cieľový DSP. Vo fáze linkovania je možné použiť dostupné **knižničné funkcie**.

Väčšina súčasných asemblerov sú tzv. **macro asemblery**, ktoré umožňujú definovať (alebo používať preddefinované) parametrizovateľné bloky kódu (**makrá**), ktoré sú do zdrojového kódu vkladané počas prekladu. Makrá umožňujú programátorovi zmenšiť množstvo zdrojového kódu, ktorý je treba udržiavať (čím je možné do určitej miery zvýšiť spoľahlivosť programov) ako aj eliminovať nadbytočné inštrukcie, ktoré je potrebné použiť pri volaní podprogramov a sú preto v programoch pre DSP veľmi často využívané. Nasledujúci kód dokumentuje použitie asemblerovského makra pri realizácii FIR filtra pomocou klasického DSP Motorola DSP5600x, ktorý obsahuje jednu MAC jednotku:

```
; definícia makra „FIR", ktoré implementuje FIR filter s N koeficientmi
FIR     macro  N
        clr    a
        rep    #N-1
        mac    x0,y0,a          x:(r0)+,x0       y:(r4)+,y0
        macr   x0,y0,a (r0)-
        endm

; inicializácia smerníkov na koeficienty a dáta
        move   #data,r0
        move   #koeficienty,r4

; volanie makra pre FIR filter s 512 koeficientmi
        FIR    512
```

Už na prvý pohľad je vidno zásadný rozdiel medzi efektívnym programom pre DSP a programom pre typický jednočípový procesor (napr. na báze jadra Intel 8051). Asembler pre DSP umožňuje zápis podstatne „širšieho" operačného kódu (napr. inštrukcia mac), čo je dané vnútornou architektúrou dátových ciest v DSP.

Zaujímavou alternatívou ku klasickým asemblerom sú tzv. *algebraické asemblery*, u ktorých je snaha používať štýl písania asemblerovských programov, ktoré pripomínajú algebraický zápis algoritmu. Tento prístup využíva napr. firma Analog Devices, čo dokumentuje nasledujúci kód FIR filtra (bez využitia makier) pre procesor ADSP21xx s jednou MAC jednotkou:

```
.ENTRY          fir;
fir:    MR = 0, MX0 =  DM( I0, M1),    MY0 = PM(I4, M5);
        DO sop UNTIL CE;
sop:            MR = MR + MX0*MY0( SS ), MX0=DM(I0, M1), MY0=PM(I4, M5);
        MR = MR+MX0*MY0(RND);
        IF  MV  SAT  MR;
        RTS
```

Dátové cesty moderných DSP obsahujú väčší počet MAC jednotiek. Nasledujúci kód zapísaný v algebraickom asembleri dokumentuje program pre realizáciu FIR filtra v jadre procesora Blackfin, ktoré obsahuje dve MAC jednotky:

```
...
                LSETUP(E_MAC_ST, E_MAC_END) LC1=P2>>1;                    //Loop 1 to Nc/2 - 1
                A1 = R2.L*R1.L, A0 = R2.H*R1.H  ||  R2.H=W[I2++]  ||  [I3++]=R3;
E_MAC_ST:       A1 += R0.L*R2.H,  A0 += R0.L*R2.L  || R2.L=W[I2++]  || R0=[I1--];
E_MAC_END:
                A1 += R0.H*R2.L, A0 += R0.H*R2.H   ||  R2.H=W[I2++] ;
```

Kód jasne odráža skutočnosť, že efektívne[4] programovanie v asembleri je pre moderné DSP značne náročná úloha a preto je v praxi žiaduce minimalizovať priame programovanie v asembleri.

5.1.2 KNIŽNIČNÉ FUNKCIE

Pre aplikácie na báze DSP majú optimalizované knižnice kľúčový význam. Aj keď typické jadrá algoritmov ČSS implementované pomocou DSP sú relatívne krátke, ich optimalizácia je z pohľadu programátora značne náročná úloha, ktorá vyžaduje dobrú znalosť architektúry cieľového DSP. Naviac je často potrebné zvážiť možnosti modifikovať samotný algoritmus ČSS do tvaru, ktorý je pre danú architektúru vhodnejší, čo zvyčajne vyžaduje špecifické znalosti z oblasti teórie ČSS. Je logické, že výrobcovia DSP poskytujú pre svoje procesory **optimalizované kódy**, ktoré umožňujú relatívne efektívnu implementáciu základných algoritmov ČSS (FIR, IIR, FFT, DCT…). Tieto kódy sú typicky dostupné v rámci WWW stránok jednotlivých výrobcov DSP, aplikačných príručiek a špecializovaných kníh.

Aj keď je často možné tieto kódy ďalej vylepšiť, sú uvedené kódy dobrým štartovacím bodom pre vývoj aplikačných programov. Je dokonca výhodné pri zoznamovaní sa s architektúrou DSP využívať hotové programy resp. funkcie. Klasický postup, t.j. počiatočné zvládnutie celej inštrukčnej sady a následná snaha o písanie vlastných efektívnych programov je málo účinným prístupom k zvládnutiu programovania DSP. V ďalších cvičeniach budeme preto analyzovať takéto (relatívne jednoduché) kódy pre FIR filter, IIR filter a FFT a na týchto príkladoch budeme vysvetľovať podstatné vlastnosti procesorov ADSP Blackfin.

Knižničné funkcie je možné pomocou asemblera preložiť do relatívneho objektového tvaru a spojiť pomocou programu – **knihovníka** do jedného súboru – **knižnice**. Knižnice je potom možné využívať napr. ako aplikačné balíky (napr. knižnica pre číslicovú filtráciu, FFT a pod) a výrazne tak skrátiť dobu vývoja cieľového firmvéru pre DSP. Vývojové nástroje často obsahujú štandardné knižničné funkcie pre ČSS. Súčasťou prostredia VisualDSP++ sú napr. aj funkcie pre FIR, IIR a FFT z tzv. DSP Run-Time Library:

```
void fir_fr16( const fract16 x[ ], fract16 y[ ], int n, fir_state_fr16 *s );
void iir_fr16( const fract16 x[ ], fract16 y[ ], int n, iir_state_fr16 *s );
void iirdf1_fr16( const fract16 x[ ],  fract16 y[ ], int n, iirdf1_fr16_state *s );
void cfft_fr16(const complex_fract16 in[ ], complex_fract16 t[], complex_fract16 out[ ],
            const complex_fract16 w[ ], int wst, int n, int block_exponent, int scale_method );
```

Časť funkcií z knižnice DSP Run-Time Library bude využitá v nasledujúcich cvičeniach resp. zadaniach. Podrobnejšie informácie o uvedených funkciách sú

[4] Pri programovaní je treba uvažovať aj závislosti medzi jednotlivými inštrukciami, ktoré môžu spôsobiť automatické vsúvanie NOP inštrukcií v riadiacej jednotke DSP. Typickým príkladom je čakanie na operand, ktorý bol zapísaný do pamäti v predchádzajúcej inštrukcii.

dostupné v on-line manuáloch prostredia VisualDSP++. Súčasťou prostredia sú aj optimalizované zdrojové kódy knižničných funkcií v asembleri.

Nasledujúci kód dokumentuje princíp využitia asemblera pri vytvorení vlastnej knižničnej C funkcie s prototypom

```
int a_dot_c_asm( int *a, int *c );
```

a telom funkcie optimalizovanom v asembleri:

```
/*************************************************************************
 kniznicna funkcia realizuje nasledujucu C funkciu

 const int N = 20;

 int a_dot_c_asm( int *a, int *c ) {
   int i;
   int output = 0.0;
   for( i=0; i<N; i++ )  {
                output += ( a[i] * c[i] );
   }
   return( output );
}
*************************************************************************/

// ASSEMBLY DOT PRODUCT FUNCTION
// File: dotprod_func.asm

.section my_asm_section;
.global _a_dot_c_asm;

_a_dot_c_asm:
        P0 = R0;                                        // adresa prvku a[0]
        I0 = R1;                                        // adresa prvku c[0]
        P1 = 19;                                        // N-1
        R0 = 0;                                         // output = 0
        NOP;
        R1 = [P0++];
        R2 = [I0++];
        LSETUP (begin_loop, end_loop) LC0 = P1;
begin_loop:     R1 *= R2;
                R2 = [I0++];
end_loop:       R0= R0 + R1 (NS) || R1 = [P0++] || NOP;         // NS -nesturovany vysledok
        R1 *= R2;
        R0 = R0 + R1;
        RTS;
_a_dot_c_asm.end:
```

Aj keď inštrukčná sada, direktívy[5] asembleru a programovací model jadra Balckfin neboli preberané, asemblerovský kód je pomerne čitateľný (detaily budú vysvetlené v rámci cvičenia). Zdrojové kódy reálnych knižničných funkcií sú zvyčajne zložitejšie, pretože okrem implementácie samotnej funkcie musia riešiť napr. kontrolu odovzdávaných parametrov, správne zarovnanie kritických sekcii kódu v pamäti a pod.

5.1.3 SIMULÁTORY

Simulátory sú programy, ktoré simulujú na cieľovej platforme (zvyčajne PC s operačným systémom Windows) prácu procesora na úrovni jednotlivých inštrukcií. Z

[5] Direktívy asembleru sú súčasťou zdrojových kódov. Na rozdiel od inštrukcií, ktoré sú príkazmi, ktoré realizuje po preklade do strojového kódu cieľový procesor, sú direktívy príkazmi pre program, ktorý realizuje samotný preklad. V predchádzajúcom príklade sú použité direktívy **.section**, **.global**, **LSETUP**.

pohľadu optimalizácie kódu pre DSP majú simulátory kľúčovú úlohu a umožňujú odhaliť problémy vznikajúce napr. vplyvom **zreťazenia**[6], prípadne zvýšiť efektivitu kódu identifikáciou inštrukcií, medzi ktorými sa vytvárajú tzv. „pipeline bubbles", t.j. okamihy, kedy zreťazené technické prostriedky nie sú plne využité. Vhodným preusporiadaním inštrukcií je často možné zvýšiť využitie prostriedkov procesora a tým aj rýchlosť implementovanej funkcie. V rámci cvičení bude využívaný simulátor, ktorý je súčasťou integrovaného prostredia VisualDSP++. Prostredie VisualDSP++ podporuje okrem klasickej simulácie aj tzv. **kompilovanú simuláciu** (Compiled Simulation). Klasická simulácia pracuje **interpretačným spôsobom**, kedy simulátor najskôr dekóduje a následne interpretuje každú simulovanú inštrukciu. Tento princíp simulácie je pre zložitejšie simulácie pomalý. Kompilovaná simulácia realizuje „preklad" simulovaného programu a nevyžaduje opakované dekódovanie inštrukcií. Výsledná simulácia je tak podstatne rýchlejšia.

5.1.4 VÝVOJOVÉ DOSKY, EMULÁTORY

Pri vývoji zariadení na báze DSP je cieľom realizovať systém ČSS, ktorý pracuje s reálnymi vstupnými resp. výstupnými dátami. V určitej etape vývoja je dôležité mať k dispozícii reálne technické zariadenie (často sa technické a programové prostriedky vyvíjajú paralelne a cieľové zariadenie nie je v úvodnej fáze vývoja často k dispozícii). Na overenie činnosti algoritmov ČSS v reálnom čase sú vhodné **vývojové dosky**, ktoré typicky obsahujú všetky základné bloky číslicového signálového procesora (obmedzovacie a rekonštrukčné filtre, AD a DA prevodníky, DSP, pamäte) . Všetci hlavní výrobcovia DSP poskytujú lacné vývojové moduly (v cenách 70-300 EUR), ktoré je možné dokonca v prípade menej náročných zariadení využiť aj ako konečné technické riešenie systému ČSS. Výhodné je, pokiaľ cieľový DSP podporuje **emuláciu na čipe**[7], čo zlepšuje možnosti ladenia priamo v reálnych podmienkach pri minimálnych nárokoch na dodatočné technické vybavenie.

Využitie klasických **emulátorov** je v praxi veľmi problematické predovšetkým vzhľadom na stále sa zvyšujúcu taktovaciu frekvenciu DSP a používanie prevažne SMD technológie, čo vylučuje nasadenie klasickej **emulačnej hlavice**.

5.2 VÝVOJOVÉ PROSTRIEDKY VYUŽÍVANÉ NA CVIČENIACH

V rámci cvičení budú využívané vývojové prostriedky pre signálové procesory na báze jadra Analog Devices Blackfin (ADSP BF533, ADSP BF561) dodávané firmou Analog Devices. Vývojové nástroje pre tieto DSP sú dostupné na WWW stránkach firmy Analog Devices

www.analog.com/dsp/tools

vo forme 90-dňovej testovacej verzie, ktorú je možné zdarma aktivovať po získaní aktivačného kódu (položka **VisualDSP++ Test Drive Registration**). Počas cvičení budú využívané predovšetkým tieto programy:

[6] Zreťazenie (pipelining) je jednou zo všeobecných metód zvýšenia výkonnosti mikroprocesorov a je v oblasti DSP široko využívaná.

[7] Špeciálna metóda ladenia, ktorá využíva ladiace obvody implementované priamo v čipe procesora. Ladiace obvody moderných DSP umožňujú aj prenos ladiacich informácií v reálnom čase bez prerušenia činnosti DSP. Firma Analog Devices používa pre uvedený prenos termín **spätný telemetrický kanál** (Background Telemetric Channel).

1) asembler **easmblkfn.exe**,
2) C/C++ prekladač **ccblkfn.exe**,
3) linker **linker.exe**,
4) Simulátor (dostupný z integrovaného prostredia podobne ako aj asembler, prekladač a linker).

Ďalšie programy ako knihovník, rôzne konverzné a pomocné programy, operačný systém reálneho času sú tiež dostupné v rámci prostredia VisualDSP++, počas cvičení však nebudú využívané[8]. Počas cvičení budú využívané aj optimalizované kódy pre procesory Balckfin, ktoré sú súčasťou prostredia VisualDSP++ resp. sú dostupné na WWW stránke firmy Analog Devices:

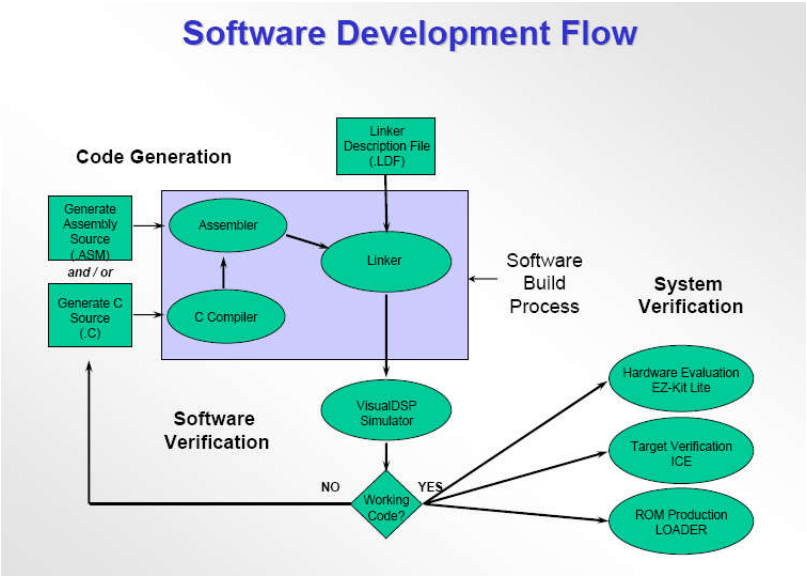
http://www.analog.com/en/design-center/processors-and-dsp.html/

Tieto kódy reprezentujú kódy pre algoritmy z rôznych oblastí ČSS. Veľká časť informácií je platná aj pre iné rodiny DSP firmy Analog Devices – ADSP21xx, ADSP SHARC a ADSP Tiger SHARC(5.1), pre ktoré je tiež možné využiť prostredie VisualDSP++.

Integrované vývojové prostredie VisualDSP++ od firmy Analog Devices využíva špecifické prípony pre generované výstupné súbory. Informácie o projekte sú uložené v súbore s príponou **.dpj**. Zdrojové kódy majú zvyčajne prípony **.asm**, **.h**, **.c**. Kód preložený do relatívneho objektového tvaru má príponu **.doj** a kód preložený do absolútneho objektového tvaru má príponu **.dxe**. Technické prostriedky pre ktoré je vytváraný výsledný program sú opísané v tzv. **Linker Description File** s príponou **.ldf**. Knihovník vytvára zlúčením viacerých súborov s príponami **.doj** knižnicu s príponou **.dlb**. Proces tvorby programového kódu pre signálové procesory Analog Devices, použité programové prostriedky a ich vzájomná väzba sú znázornené na obr 5.1.

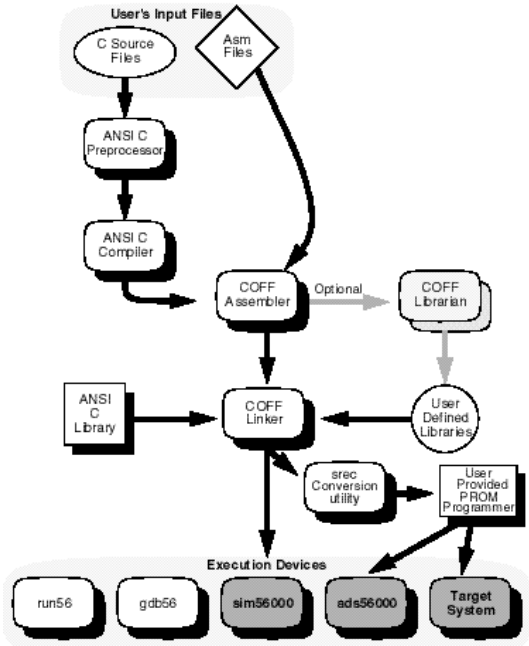
Tento obrázok reprezentuje možnosti a praktickú postupnosť využitia dostupných vývojových prostriedkov. Vykonateľný kód bude v počiatočných cvičeniach simulovaný pomocou simulátora a neskôr vykonávaný v cieľovom systéme – vývojovej doske EZ-KIT Lite pre procesory Balckfin ADSP BF533.

[8] Cieľom cvičení je vysvetliť základné vlastnosti DSP, čo budeme dokumentovať na relatívne jednoduchých príkladoch pri ktorých vystačíme s asemblerom, C prekladačom, linkerom a simulátorom. Programy ako napr. knihovník je výhodné využívať pri väčších projektoch (ako napr. diplomové úlohy), pri ktorých môžu výrazne zrýchliť a sprehľadniť generovanie výsledného kódu.



Obr.5.1 Štruktúra programových prostriedkov pre DSP firmy Analog Devices

Na obr.5.2 je pre porovnanie znázornený proces tvorby kódu pre klasické DSP firmy Motorola. Obrázok podrobnejšie naznačuje aj možné využitie **prekladača** (kompilátora) z vyššieho programovacieho jazyka C a **knihovníka** (Librarian) pre tvorbu užívateľských knižníc. Samozrejme tieto princípy sú využívané aj v prostredí VisualDSP++ resp. aj vo vývojových prostrediach pre jednočipové mikroprocesory.



Obr.5.2 Štruktúra programových prostriedkov pre DSP firmy Motorola

Príklad

*Precvičte ladenie priloženého C projektu **dotprod.zip** na výpočet skalárneho súčinu v prostredí VisualDSP++ zobrazenom na obr.5.3. Precvičte základné ladiace kroky:*

- *preloženie programu (**menu Project**)*
- *krokovanie programu (**menu Debug**),*
- *možnosti zobrazenia rôznych registrov procesora (**menu Register**),*
- *zobrazenie C premenných (menu **View->Debug Windows-> Expressions, Locals**),*
- *krokovanie kódu v okne **Disassembly,***
- *nastavenie bodov zastavenia (klik myšou na riadok C-kódu, **menu Settings->Breakpoints**),*
- *výpis funkcie printf( ) v okne **Output Window**.*

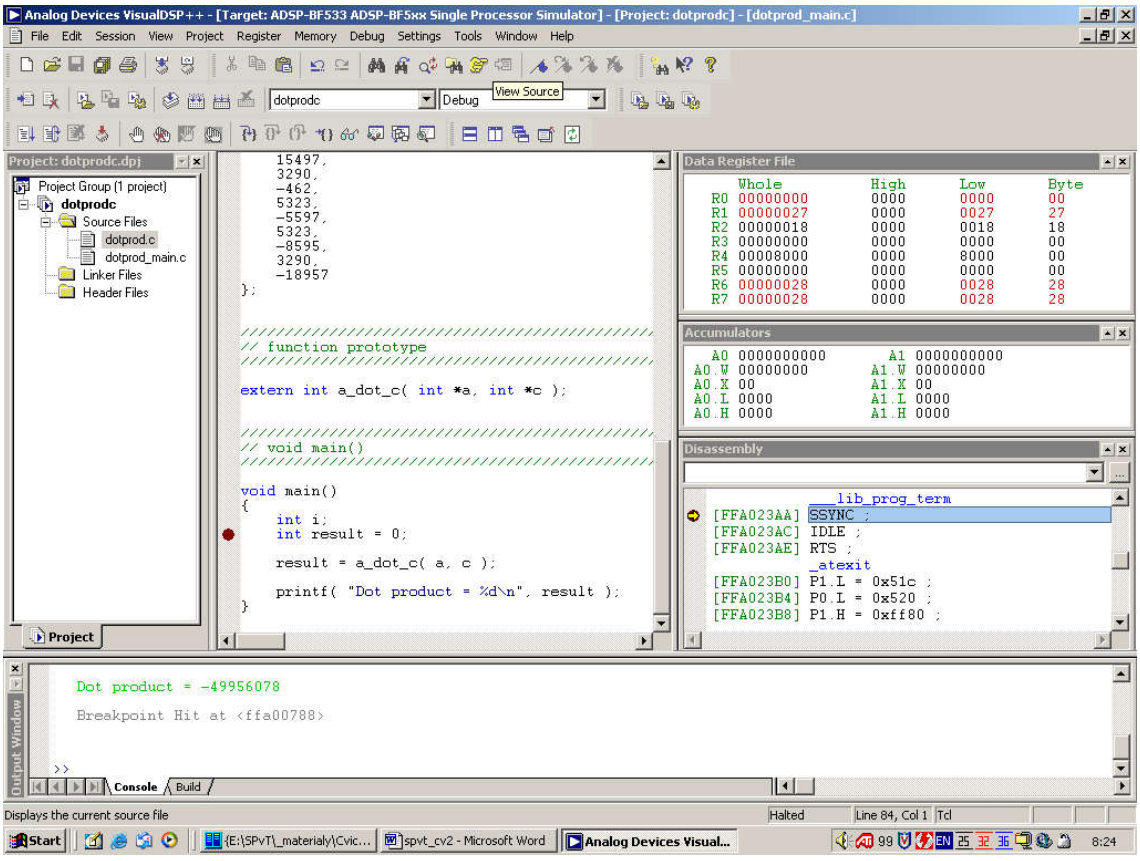
Pri ladení odpovedzte na nasledujúce otázky:

a) Koľko inštrukcií procesora je realizovaných v jednej iterácii funkcie a_dot_c( )? Porovnajte zistený počet s počtom inštrukcií v hlavnej slučke funkcie a_dot_c_asm( ) na str.61._

b) Čo sa stane po ukončení funkcie main( )?

c) Kde by bol v reálnej aplikácii smerovaný výstup funkcie printf ()?

d) Aký kód je vykonávaný procesorom pred spustením funkcie main? (processor začína vykonávať kód na adrese start: 0xFFA00000) _



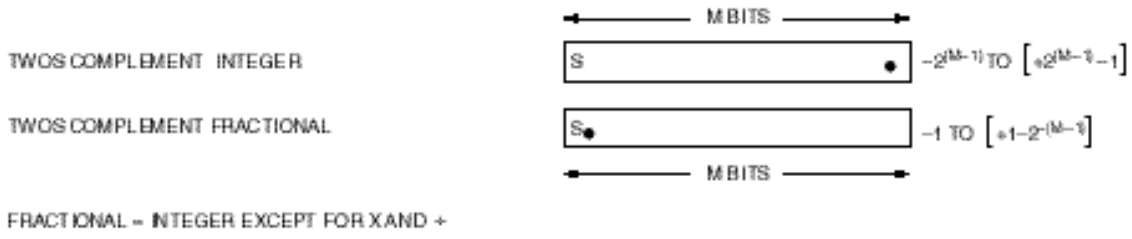
Obr.5.3 Ladenie príkladu skalárneho súčinu v prostredí VisualDSP++

5.3 ZLOMKOVÝ FORMÁT ČÍSEL V DSP

Základnou úlohou pre ktorú sú DSP optimalizované je matematické spracovanie číslicových údajov. Medzi typické algoritmy pre ktoré sú komerčne dostupné DSP optimalizované patria číslicová filtrácia a algoritmus FFT. Základným dátovým typom DSP **s pevnou rádovou čiarkou** (medzi ktoré patria aj procesory Analog Devices s jadrom Balckfin, ako aj staršie procesory Motorola DSP56xxx a Analog Devices ADSP21xx) patrí **zlomkový formát** (fractional format, firma Analog Devices používa pre ich 16-bitové DSP tiež označenie **fractional representation 1.15**), pomocou ktorého sú čísla v intervale $\langle -1,1)$ reprezentované v tvare M bitov (**dĺžka slova**) $b_m \in \{0,1\}, \quad m = 0,1,2,\ldots,M-1$, ktoré vyjadrujú číslo v tvare

$$x_{pevna} = (-1)b_0 + \sum_{m=1}^{M-1} b_m 2^{-m} \tag{5.1}$$

ktorý sa od klasického celočíselného formátu 16.0 (ktorý reprezentuje záporné čísla pomocou dvojkového doplnku) odlišuje polohou desatinnej čiarky, čo je dokumentované na obr. 5.4.



Obr.5.4 Porovnanie celočíselnej a zlomkovej reprezentácie

Formáty 1.15 a 16.0 sú formáty znamienkových čísel s desatinnou čiarkou maximálne posunutou vľavo resp. vpravo. DSP sú optimalizované pre zlomkový formát, principiálne však môžu využívať aj ďalšie formáty, ktoré sú pre $M = 16$ zobrazené na obr.5.5.

Dĺžka slova v komerčne využívaných DSP s pevnou rádovou čiarkou je $M = 16$ a $M = 24$ bitov[9]. Základným dôvodom, prečo je využívaná zlomková reprezentácia je jej tesnejšia väzba s formátom **pohyblivej rádovej čiarky**, ktorý sa bežne využíva pri vedecko-technických výpočtoch (medzi ktoré patria aj algoritmy ČSS). Napríklad všetky bežne dostupné programy pre návrh číslicových filtrov poskytujú koeficienty v pohyblivej rádovej čiarke, čo je zvyčajne možné pretransformovať do zlomkového formátu jednoduchou **zmenou mierky** (t.j. predelením hodnôt mierkovou konštantou).

[9] 16-bitové DSP sa typicky využívajú v telekomunikačnej technike a 24-bitové DSP sú dominantné predovšetkým pre audio aplikácie. Procesory Motorola DSP5600x a DSP563xx sú 24-bitové DSP. Všetky DSP firmy Analog Devices, ktoré využívajú aritmetiku s pevnou rádovou čiarkou sú 16-bitové.

Ranges for 16 bit Formats

FORMAT		Largest Positive Value (0x7FFF) In Decimal	Largest Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	Fractional	0.999969482421875	–1.0	0.000030517578125
2.14		1.999938964843750	–2.0	0.000061035156250
3.13		3.999877929687500	–4.0	0.000122070312500
4.12		7.999755859375000	–8.0	0.000244140625000
5.11		15.999511718750000	–16.0	0.000488281250000
6.10		31.999023437500000	–32.0	0.000976562500000
7.9		63.998046875000000	–64.0	0.001953125000000
8.8		127.996093750000000	–128.0	0.003906250000000
9.7		255.992187500000000	–256.0	0.007812500000000
10.6		511.984375000000000	–512.0	0.015625000000000
11.5		1023.968750000000000	–1024.0	0.031250000000000
12.4		2047.937500000000000	–2048.0	0.062500000000000
13.3		4095.875000000000000	–4096.0	0.125000000000000
14.2		8191.750000000000000	–8192.0	0.250000000000000
15.1		16383.500000000000000	–16384.0	0.500000000000000
16.0	Integer	32767.000000000000000	–32768.0	1.000000000000000

Obr.5.5 Číselné formáty pre znamienkové čísla a $M = 16$

Ďalšou výhodnou vlastnosťou zlomkového formátu je vlastnosť, že ak $x, y \in \langle -1,1)$, potom platí

$$z = x * y \in \langle -1,1) \tag{5.2}$$

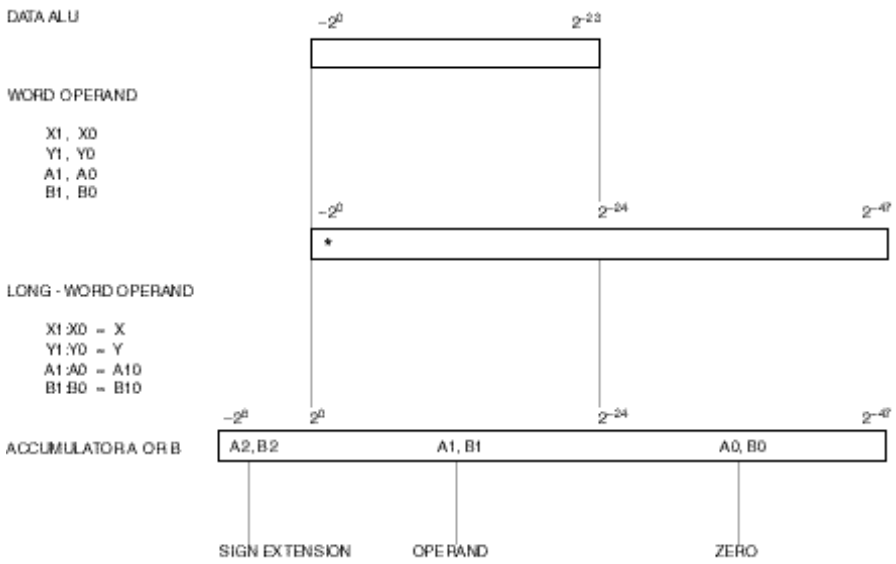
a teda pri operácií násobenia nedochádza k pretečeniu výsledkov mimo interval $\langle -1,1)$. K pretečeniu mimo zlomkový interval $\langle -1,1)$ dochádza len[10] pri operácií sčítania resp. odčítania.

V rámci štruktúry DSP (presnejšie **dátových registrov DSP**) typicky existuje niekoľko typov dátových registrov s rôznou presnosťou. V signálových procesoroch DSP56xxx ktoré využívajú **jednu MAC** jednotku napr. existujú 24-bitové registre (X0,X1,Y0,Y1) a 56-bitové registre (**akumulátory**) A, B, ktorých štruktúra a váhy jednotlivých bitov sú znázornené na nasledujúcom obr.5.6.

Akumulátory A=(A2:A1:A0) a B=(B2:B1:B0) sa skladajú z 24-bitových subregistrov A0, A1, B0, B1 a 8-bitových registrov A2, B2. Akumulátory A a B sa využívajú predovšetkým pomocou inštrukcie MAC, ktorá realizuje výpočet

$$Akum = Akum \pm \mathrm{Re}\, g\_X * \mathrm{Re}\, g\_Y \tag{5.3}$$

[10] Operácia delenia je z pohľadu DSP špeciálna operácia, ktorá je podstatne pomalšia ako operácie +,-,*. DSP typicky poskytujú špeciálne inštrukcie, pomocou ktorých je možné realizovať delenie iteračným spôsobom, t.j. na dosiahnutie výsledku delenia je potrebných niekoľko inštrukcií. V procesoroch ADSP je napr. možné využiť inštrukcie DIVS (divide sign) a DIVQ (divide quotient).



Obr.5.6 Štruktúra registrov v procesoroch Motorola DSP56xxx

Štruktúra registrov v procesoroch Blackfin je podstatne bohatšia, čo je dané predovšetkým **väčším počtom MAC jednotiek** (MAC0 a MAC1), registrovým súborom s ôsmimi 32-bitovými registrami R0, R1, R2, R3, R4, R5, R6, R7, existenciou jednotky posúvača (Barrel Shifter) a podporou SIMD inštrukcií pre video operácie s 8-bitovými operandami.

Vzhľadom na to, že procesory ADSP s aritmetikou v pevnej rádovej čiarke využívajú menší dynamický rozsah (16 resp. 40 bitov oproti 24 a 56 bitom u DSP5600x), obsahujú procesory ADSP podstatne lepšiu podporu pre aritmetiku v tzv. **blokovej pohyblivej rádovej čiarke** a v aritmetike s dvojnásobnou presnosťou.

Procesory Blackfin majú v jednotke MAC0 akumulátor A0=(A0.X:A0.H:A0.L) a v jednotke MAC1 akumulátor A1=(A1.X:A1.H:A1.L), ktoré majú 40 bitov. Akumulátory sú zložené zo 16-bitových subregistrov A0.H, A0.L, A1.H, A1.L a 8-bitových subregistrov A0.X, A1.X. Ich štruktúra je teda identická so štruktúrou akumulátorov A, B na obr.5.6 pre procesory Motorola. Jediný rozdiel je v dĺžke subregistrov a im zodpovedajúcim binárnym váham.

Akumulátory A0, A1 sa v procesoroch Blackfin využívajú predovšetkým pomocou MAC inštrukcie, ktorá realizuje všeobecný výpočet

$$A = A + X * Y \tag{5.4}$$

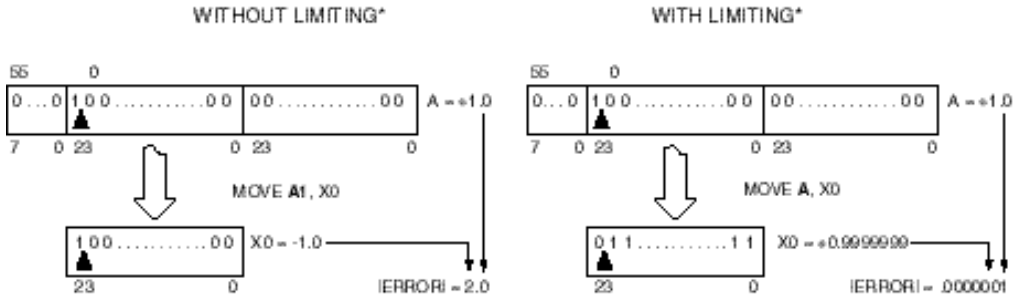
pričom vstupné registre X a Y sú 16-bitové subregistre z registrového súboru R0-7 (R0.L, R0.H, …R7.H) a výstupom A sú akumulátory A0, A1. Typické príklady zápisu inštrukcií v algebraickom asembleri pre procesor Blackfin:

```
A0 = R3.L * R4.H;                    // nasobenie bez akumulacie
A0 += R3.L * R4.H;                   // nasobenie s akumulaciou
A1 += R3.H * R4.H;
```

Prítomnosť dvoch paralelných MAC jednotiek v jadre procesorov Blackfin umožňuje paralelne vykonať dve MAC inštruckie ako napr.:

```
R3.H = (A1 += R1.H * R2.L),  R3.L = (A0 += R1.L * R2.L);
R3 = (A1 += R1.H * R2.L),  R2 = (A0 += R1.L * R2.L);
```

S aritmetikou v procesoroch DSP sú spojené dva nové pojmy – **saturačná aritmetika (saturačný mód v ALU a MAC)** a **konvergentné zaokrúhľovanie (convergent rounding[11]).** Tieto vlastnosti sú demonštrované na obr.5.7 a obr.5.8.



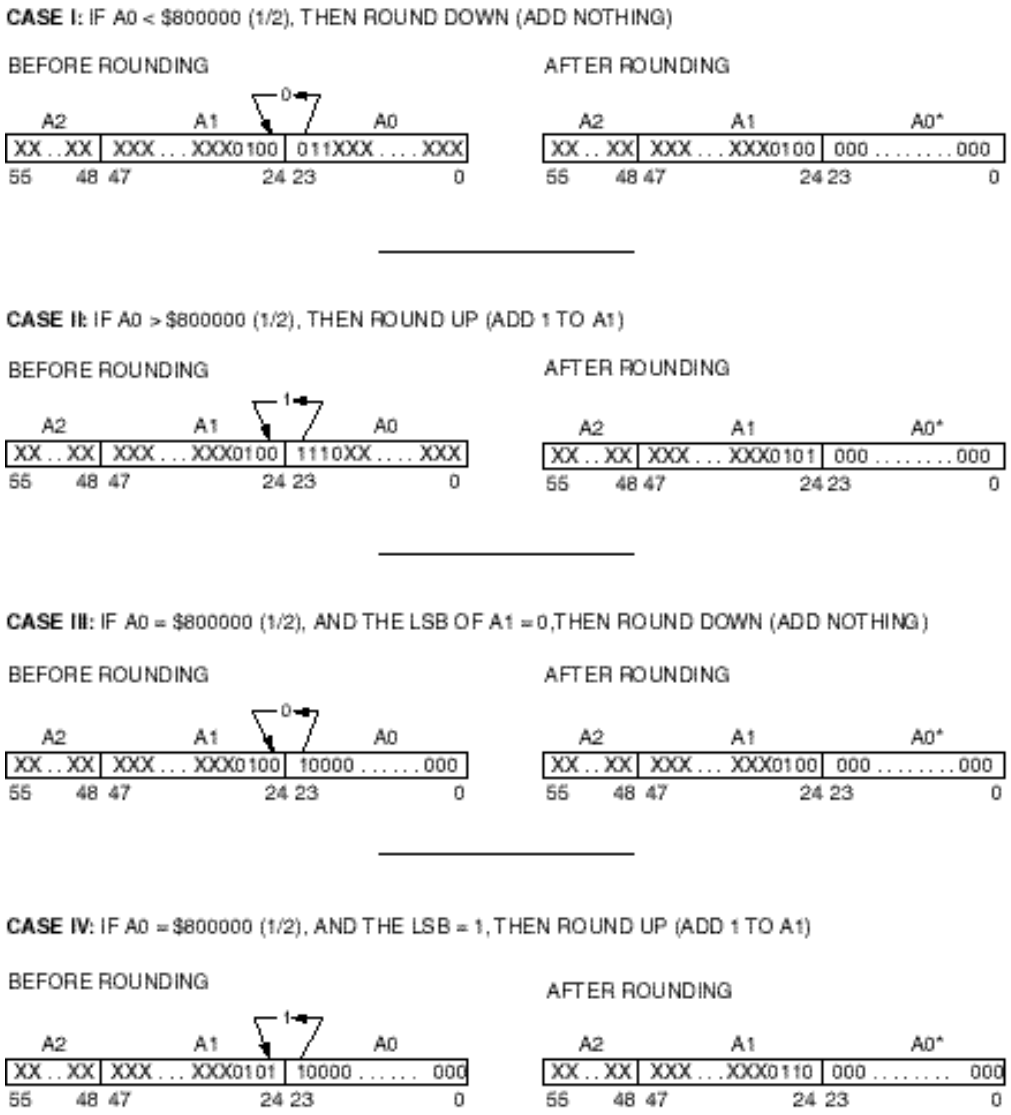
Obr.5.7 Princíp saturačnej aritmetiky v procesoroch Motorola (56-bitový akumulátor), zhodný princíp sa uplatňuje aj v prípade procesorov ADSP21xx a ADSP Blackfin (40-bitový akumulátor)

Využitie saturačnej aritmetiky v ALU jednotkách je možné v procesoroch Balckfin riadiť využitím modifikátorov **S** (saturate) a **NS** (non saturate)

```
R3 = R1 + R2 (NS);          // 32-bitovy vysledok nebude saturovany
R3 = R1 + R2 (S);           // 32-bitovy sucet bude saturovany
```

V MAC jednotkách je prevažná časť inštrukcií, ktoré pracujú so 40 bitovými akumulátormi A0, A1 automaticky saturovaná. U ostatných inštrukcií je saturácia závislá na type operácie.

[11] Novšie procesory vrátane procesorov Blackfin umožňujú aj využitie klasického zaokrúhľovania (tzv. **biased rounding**), ktoré sa využíva napr. pri implementácií kompresných rečových kodekov, ktoré musia poskytovať bitovo zhodný výstup s referenčným kodekom, ktorý je zvyčajne implementovaný v jazyku ANSI C s využitím celočíselnej aritmetiky.



Obr.5.8 Princíp konvergentného zaokrúhľovania v procesoroch Motorola (56-bitový akumulátor), zhodný princíp sa uplatňuje aj v prípade ADSP21xx (40-bitový akumulátor)

Zaokrúhľovací mód **MAC jednotiek** je v procesoroch Blackfin definovaný hodnotou bitu **RND_MOD** v registri **ASTAT**. Naviac Blakfin ALU poskytuje možnosť realizovať klasické zaokrúhlenie na konkrétnej bitovej pozícii bez ohľadu na nastavenie v registri ASTAT s využitím inštrukcií

```
R3.L = R4 (RND);          // klasicke zaokruhlenie na 16-tom bite
R3.L = R4 + R5 (RND12);   // 32-bitovy sucet a nasledne zaokruhlenie na 12-tom bite
R3.L = R4 + R5 (RND20);   // 32-bitovy sucet a nasledne zaokruhlenie na 20-tom bite
```

6 TVORBA A LADENIE PROGRAMOV V PROSTREDÍ VISUALDSP++

Prostredie VisualDSP++ poskytuje bohaté možnosti na tvorbu a ladenie kompletných projektov pre DSP firmy Analog Devices. Prostredie je identické pre rodiny procesorov Blackfin, SHARC a TigerSHARC[1] a v súčasnosti je aktuálna verzia VisualDSP++ v.5.1. Na stránkach firmy Analog Devices (www.analog.com) je možné stiahnuť inštalačné balíky najnovších verzií a Service Pack balíkov. Tieto verzie **sú identické** s verziami balíkov VisualDSP++, ktoré sú dostupné komerčne. Po inštalácii je pre správnu funkčnosť prostredia potrebné na stránkach firmy Analog Devices získať **bezplatné** testovacie licencie, ktoré umožnia využívať **plne funkčné verzie** prostredia VisualDSP++ po dobu 90 dní.

Prostredie VisualDSP++ pre procesory Balckfin bude využívané vo všetkých nasledujúcich cvičeniach pri vytváraní, ladení a testovaní programov pre DSP. Cieľom cvičení je prezentácia **základných vlastností** prostredia VisualDSP++, ktoré umožnia postupne vytvoriť reálnu DSP aplikáciu. Súčasťou prostredia je bohatý on-line help a kompletné manuály v PDF formáte, ktoré je možné využiť v prípade potreby zvládnutia zložitejších možností, ktoré prostredie VisualDSP++ poskytuje.

6.1 DSP PROJEKT S KOMBINÁCIOU C A ASM SÚBOROV

Kombinácia zdrojových súborov v jazyku C a ASM je typickou technikou, ktorá umožňuje dosiahnuť pomerne vysokú efektivitu vytvoreného programu s relatívne nízkym úsilím programátora. V nasledujúcej časti bude vytvorený kompletný projekt realizujúci výpočet skalárneho súčinu, ktorý bol realizovaný v predchádzajúcom cvičení pomocou C jazyka. Telo funkcie na výpočet skalárneho súčinu bude realizované optimalizovanou funkciou v ASM. Projekt tak umožní porovnanie rýchlosti obidvoch techník. Kompletný projekt v prostredí ViusalDSP++ bude vytvorený len zo zdrojových C a ASM súborov.

6.1.1 VYTVORENIE PROJEKTU A ZAČLENENIE ZDROJOVÝCH SÚBOROV

Súčasťou projektu sú dva zdrojové súbory (dotprodasm.zip):

dotprod_main.c
dotprod_func.asm

[1] Procesory SHARC a TigerSHARC sú výkonné 32-bitové procesory využívajúce aritmetiku s pohyblivou rádovou čiarkou.

```
//////////////////////////////////////////////////////////////////////
//      FILE: dotprod_main.c
//
//////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//////////////////////////////////////////////////////////////////////
// global variables
//////////////////////////////////////////////////////////////////////
int a[] = {
66, 140, 48, 4, -121, -178, -146, 14, 231, 383, 328,
-15, -607, -1286, -1827, 6160, -1827, -1286, -607, -15
};

int c[] = {
0, 18957, -3290, 8595, -5323, 5597, -5323, 462, -3290, -15497,
0, 15497, 3290, -462, 5323, -5597, 5323, -8595, 3290, -18957
};
//////////////////////////////////////////////////////////////////////
// prototype funkcie
//////////////////////////////////////////////////////////////////////

extern int a_dot_c_asm( int *a, int *c );

//////////////////////////////////////////////////////////////////////
// void main()
//////////////////////////////////////////////////////////////////////

void main() {
int i;
int result = 0;
result = a_dot_c_asm( a, c );
printf( "Dot product = %d\n", result );
}
```

```
// File: dotprod_func.asm

.section my_asm_section;
.global _a_dot_c_asm;

_a_dot_c_asm:
P0 = R0;
I0 = R1;
P1 = 19;
R0 = 0;
NOP;
R1 = [P0++];
R2 = [I0++];
LSETUP (begin_loop, end_loop) LC0 = P1;

begin_loop:     R1 *= R2;
R2 = [I0++];
end_loop:       R0= R0 + R1 (NS) || R1 = [P0++] || NOP;

R1 *= R2;
R0 = R0 + R1;
RTS;

_a_dot_c_asm.end:
```

ktorých funkčnosť je identická s projektom dotprod.zip, ktorý bol precvičovaný na predhádzajucom cvičení. Na rozdiel od projektu dotprod.zip je však výpočet realizovaný pomocou funkcie

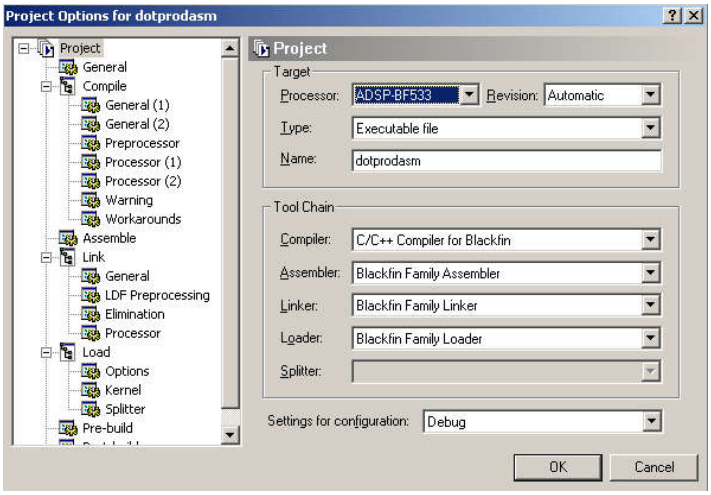
```
int a_dot_c_asm( int *a, int *c );
```

analyzovanou na minulom cvičení. Hlavný program je modifikovaný nasledujúcim spôsobom

```
extern int a_dot_c_asm( int *a, int *c );


void main()
{
      int i;
      int result = 0;

      result = a_dot_c_asm( a, c );

      printf( "Dot product = %d\n", result );
}
```

Súčasťou balíka **dotprodasm.zip** už však nie sú súbory VisualDSP++ projektu s príponami ***.dpj** a ***.mak**. Projekt v prostredí VisualDSP++ je možné vytvoriť nasledujúcim postupom[2]:

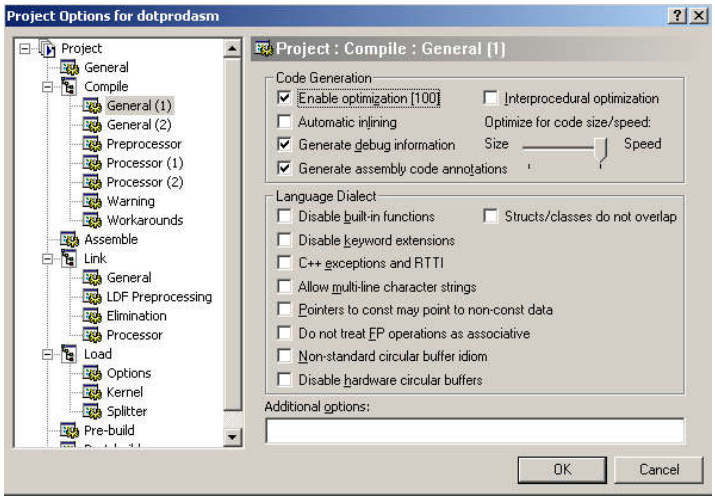
File->New->Project
v položke **General**
zadáme **meno** – napr. dotprodasm
zadáme **cestu** do pracovného adresára kde máme súbory
zvolíme voľbu Standard application
v položke Output Type
vyberieme cieľový procesor
v položke Add Startup Code/LDF
add an LDF and startup code
a stlačíme **Finish**
Project->Project Options
zvolíme cieľový procesor **ADSP-BF533,** prípadne ďalšie voľby podľa obr.6.1.



Obr.6.1 Voľby Project Options

[2] Uvedený postup umožňuje aj vytvorenie LDF súboru, modifikáciou ktorého je možné definovať napr. umiestnenie kódu prípadne dát do zvolených časí pamätí (napr. L1 CODE, L1 DATA, ...), čím je možné výrazne zvýšiť napr. rýchlosť programu. V prípade, že pri definovaní projektu nezvolíme pridanie LDF súboru, bude použitý preddefinovaný LDF súbor pre zvolený cieľový procesor.

V položke **Compile** zvolíme ďalšie položky podľa obr.6.2.



Obr.6.2 Voľby Project Compile

Potvrdíme stlačením **OK**.
V **okne Project** dodáme do položky **Source Files** (kliknutie myšou a voľba položky **Add File(s) to Folder ...**) zdrojové súbory *dotprod_main.c* a *dotprod_func.asm*.

6.1.2 VYTVORENIE LDF (LINKER DESCRIPTION FILE) SÚBORU

V predchádzajúcich krokoch boli definované zdrojové súbory, ktoré budú súčasťou projektu. Následne je potrebné zvoliť ich umiestnenie v pamäti procesora po preklade do binárneho tvaru. Často je táto činnosť realizovaná automaticky. Prostredie VisualDSP++ umiestni kód preložený zo zdrojového C kódu do preddefinovanej časti pamäti procesora Blackfin, ktorá je preddefinovaná[3] pre segment kódovej pamäti. Keďže ASM funkcia

```
int a_dot_c_asm( int *a, int *c );
```

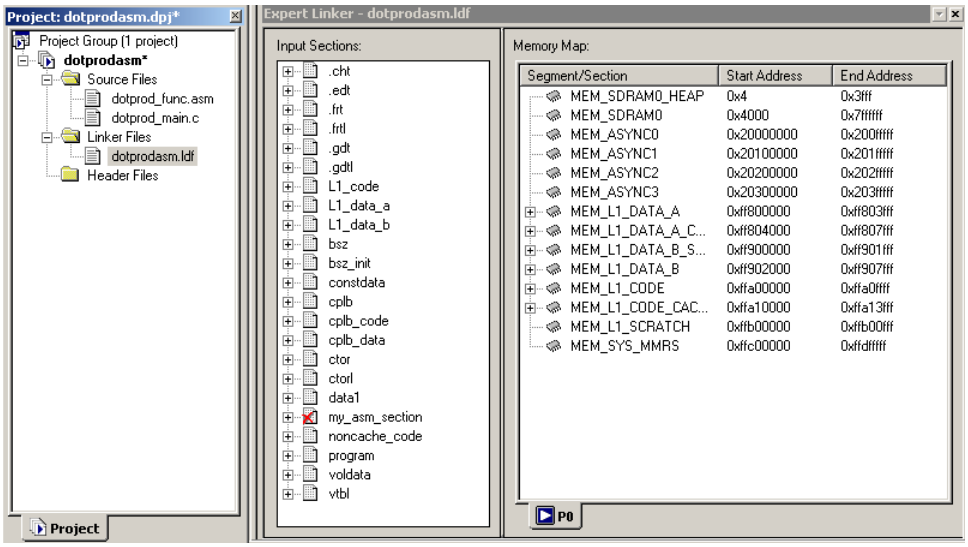
je umiestnená do **užívateľom definovaného** segmentu **my_asm_section** je potrebné definovať[4] do ktorej časti pamäte procesora Blackfin má byť uvedený segment umiestnený. Umiestnenie je v prostredí ViusalDSP++ definované v LDF súbore. Prácu s LDF súborom výrazne zjednodušuje grafické rozhranie – **Expert Linker**, ktorý využijeme[5].
Potvrdením preddefinovaných volieb sa v pracovnom adresári vytvoril súbor *dotprodasm.ldf*. Pomocou ľubovoľného editora je možné prezrieť si tento automaticky

[3] Samozrejme preddefinované umiestnenie je možné zmeniť.
[4] Voľbou niektorého z preddefinovaných segmentov by bolo možné vyhnúť sa nutnosti definovať konkrétne umiestnenie. Pre typické optimalizované ASM funkcie je však výhodné mať priamu kontrolu na procesom umiestňovania **programových** prípadne **dátových** sekcií do pamäte procesora. Je tým možné výrazne **zvýšiť rýchlosť kritických častí kódu**, čo je dané rozdielnymi rýchlosťami **hierarchického** pamäťového priestoru procesorov Blackfin (pamäť L1, L2, externá SDRAM, ...).
[5] LDF súbor je textový súbor, ktorý využíva preddefinované príkazy a syntax. Je ho teda možné vytvárať a modifikovať aj ručne pomocou štandardného editora. Tento prístup však vyžaduje hlbšie znalosti. Alternatívou je modifikácia LDF súboru pomocou grafického rozhrania Expert Linkera.

generovaný súbor. Grafická reprezentácia je dostupná po kliknutí na súbor *dotprodasm.ldf* v okne project. Kliknutím na pravú časť (Memory Map) a voľbou

View Mode ->Memory Map Tree
je možné získať grafické zobrazenie sekcií kódu a pamäte procesora zobrazené na obr.6.3. V okne Input Sections je červeným krížikom zvýraznené, že sekcia *my_asm_section* nemá zvolené umiestnenie v pamäti procesora.

Umiestnenie v pamäti je možné definovať presunutím (po kliknutí na +) položky **dotprod_func.doj**[6] do sekcie MEM_L1_CODE[7]. Do tejto sekcie bol automaticky mapovaný aj programový kód main( ) funkcie **dotprod_main.doj**.



Obr.6.3 Grafická reprezentácia súboru LDF (časť okna Memory Map je zvyčajne viditeľná až po rozšírení celého okna Expert Linker)

6.1.3 LADENIE PROJEKTU

Po definovaní LDF súboru je možné projekt preložiť voľbou **Project->Build (F7)** a následne ladiť podobne ako v predchádzajúcom cvičení (**Menu Debug**, výpis funkcie printf() v okne Output Window, …).

Príklad
Porovnajte rýchlosť výpočtu ASM funkcie a C funkcie z predchádzajúceho cvičenia. Na určenie počtu cyklov využite register ***CYCLES***[8] *dostupný v menu* ***Register->Core->Cycles***[9].

[6] Relatívny súbor (object code) funkcie definovanej v ASM súbore.
[7] Najrýchlejšia časť internej pamäte procesora Blackfin. Hierarchický pamäťový systém procesorov Blackfin bude podrobne analyzovaný v rámci prednášok.
[8] Registre CYCLES a CYCLES2 sú **reálne** 32-bitové registre v jadre procesorov Balckfin. Umožňujú **presné meranie** počtu cyklov nie len počas simulácie ale aj pri ladení pomocou reálnych technických prostriedkov.

Pokiaľ bol pri meraní rýchlosti C-projektu z predchádzajúceho cvičenia použitý pôvodný nemodifikovaný projekt, je rozdiel rýchlosti medzi C funkciou (*a_dot_c( )*) a ASM funkciou (*a_dot_c_asm*( )) na výpočet skalárneho výpočtu pomerne veľký. Na základe tohto príkladu však nie je možné tvrdiť, že ASM realizácia funkcie je výrazne rýchlejšia. Pri pohľade na kód ASM funkcie (predovšetkým na jej najvnútornejší cyklus) je zrejmé, že nie je využitý paralelizmus dvoch MAC jednotiek v jadre Blackfin. Primárnym dôvodom veľkého rozdielu je to, že v projekte z minulého cvičenia nebola povolená optimalizácia prekladača. Na druhej strane nie je možné očakávať, že zapnutie optimalizácie prekladača vždy zabezpečí generovanie kódu s rýchlosťou porovnateľnou s **kvalitne** ručne optimaliovaným kódom.

Príklad
*Porovnajte rýchlosť výpočtu ASM funkcie a **optimalizovanej** C funkcie z predchádzajúceho cvičenia. Optimalizácia prekladača je definovaná v menu **Project->Project Options->Compile** v položke **Enable optimization**.*

Príklad
Pri ladení funkcie a_dot_c_asm( ) zobrazte vnutorné rgistre procesora Blackfin (R0-R7, P a I registre, ...) a sledujte ich modifikácie počas krokovania programu .

6.2 DSP PROJEKT S VYUŽITÍM KNIŽNIČNÝCH C FUNKCIÍ

Tvorba efektívneho kódu pre moderné DSP je pomerne náročná úloha vyžadujúca veľmi dobré zvládnutie architektúry cieľového DSP, asemblera príslušného DSP a tiež podrobnú znalosť implementovaného algoritmu. Súčasťou prostredia VisualDSP++ je aj množina optimalizovaných knižničných funkcií (optimalizovaných v asembleri), ktoré je možné využívať v aplikačných C programoch a tak výrazne zjednodušiť vývoj reálnych aplikácií. Projekt FIR_LIB je príkladom, ako je možné s minimálnym úsilím realizovať efektívnu filtráciu bloku dát pomocou FIR filtra.

6.2.1 ZDROJOVÉ KÓDY PROJEKTU FIR_LIB

Projekt **fir_lib.zip** obsahuje 3 zdrojové súbory. Súbory **coefs.dat** a **in.dat** obsahujú[10] koeficienty použitého FIR filtra a vstupné vzorky ktoré spracováva hlavný program *main( )* v súbore **fir.c**:

[9] Podobne ako aj u iných registrov je možné **zvoliť tvar zobrazovania** (celočíselný, hexadecimálny, binárny, ...) pre každú skupinu registrov kliknutím myšou na príslušné okno. Hodnotu registrov je tiež možné **editovať**, čo je napr. výhodné pri meraní počtu cyklov, keď je možne pred vstupom do meranej časti kódu register CYCLES vynulovať.

[10] Zahrnutie vstupných vzoriek pomocou direktívy **#include "in.dat"** je pomerne netypické, z hľadiska syntaxe jazyka C však úplne korektné riešenie. Použité riešenie je výhodné predovšetkým v počiatočnej fáze ladenia aplikácií v simulátore. V reálnych aplikáciách sú zvyčajne vstupné resp. výstupné dáta čítané resp. zapisované cez vstupno-výstupné kanály (AD, DA prevodníky a pod.).

```
#include <fract.h>                          // definicie novych datovych typov
#include <filter.h>                         // prototypy pouzitych kniznicnej funkcii a makier

#define VEC_SIZE 256                        // dlzka spracovavaneho vstupneho vektora
#define NUM_TAPS 8                          // pocet koeficientov FIR filtra

fract16 in[VEC_SIZE] = {                    // definovanie vstupneho vektora
        #include "in.dat"
};
fract16 coefs[NUM_TAPS] = {                 // definovanie koeficientov pouziteho FIR filtra
        #include "coefs.dat"
};
fract16 delay[NUM_TAPS];                    // oneskorovacia linka pre FIR filter
fract16 out[VEC_SIZE + NUM_TAPS - 1];       // vector pre vystupne vzorky

fir_state_fr16 state;                       // struktura pre stavovu premennu FIR filtra

int main() {
        fir_init(state, coefs, delay, NUM_TAPS, 1);    // inicializacia stavovej premennej
        fir_fr16(in, out, VEC_SIZE, &state);           // filtracia vektora in FIR filtrom
}
```

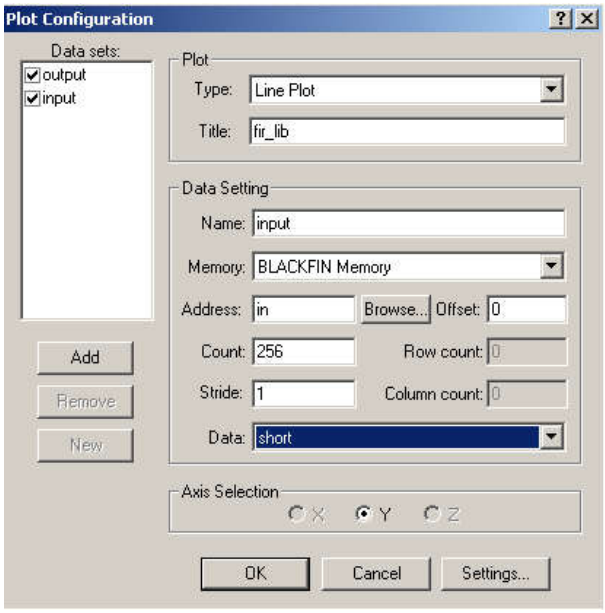
Použité knižničné funkcie resp. makrá sú definované hlavičkovým súborom **<filter.h>**. Zátvorky < > určujú, že sa jedná o systémový hlavičkový súbor. Systémové hlavičkové súbory sú uložené v adresári **..\VisualDSP 5.1.2\Blackfin\Include**
Podrobnejšie informácie o použitých knižničných funkciách je možné nájsť v súbore filter.h, on-line manuále a v **Error! Reference source not found.**.

6.2.2 LADENIE PROJEKTU FIR_LIB

Projekt FIR_LIB je možné preložiť a ladiť štandardným postupom opísaným v predchádzajúcej časti. Prostredie VisualDSP++ však poskytuje ďalšie ladiace nástroje pre **vizualizáciu väčšieho množstva dát**, ktoré je možné využiť pri ladení projektov typických DSP projektov pre ČSS.
Pomocou **okna Plot** (menu **View->Debug Windows -> Plot**) je možné graficky zobraziť obsah pamätí v ktorých sú uložené vektory dát. V našom prípade to je vektor vstupných (pole *in[ ]*) a výstupných (pole *out[ ]*) dát. Tieto údaje sú v našom prípade uložené ako 16-bitové premenné, čomu v danej implementácii jazyka C zodpovedá premenná typu *short*. Konfigurácia okna Plot je zobrazená na obr.6.4.

Po simulácii programu (Run alebo F5) je možné zobraziť graficky obsah pamäti. V okne plot je možné meniť rozlíšenie zobrazenia (zoom), zapnúť graficky kurzor, exportovať zobrazenie do JPEG súboru, meniť parametre zobrazenia (napr. farbu zobrazenia, vzorkovaciu frekvenciu pre zobrazenie na osi x, ...), typ zobrazenia (napr. zvoliť zobrazenie spektra, ...) a pod.

Príklad
S použitím zobrazenia spektra v okne Plot zistite aký je tvar vstupného signálu a ako sú modifikované FIR filtrom jeho jednotlivé zložky.



Obr.6.4 Konfigurácia Plot zobrazenia pre projekt FIR_LIB

7 IMPLEMENTÁCIA FIR FILTRA POMOCOU DSP BLACKFIN

FIR filter patrí z pohľadu signálových procesorov medzi najjednoduchší algoritmus ČSS. Prakticky všetky dostupné DSP sú optimalizované práve pre tento typ algoritmu. Jadro FIR filtra je u klasických DSP s jednou MAC jednotkou tvorené jedinou MAC inštrukciou. V asembleri je tak program pre FIR filter väčšinou pomerne jednoduchý, čo bolo demonštrované v jednom z predchádzajúcich cvičení. Knižničné funkcie pre jazyk C, ktoré sú optimalizované (v asembleri) pre prostredie jazyka C však musia riešiť aj väzbu medzi volajúcou funkciou v jazyku C a knižničnou funkciou:

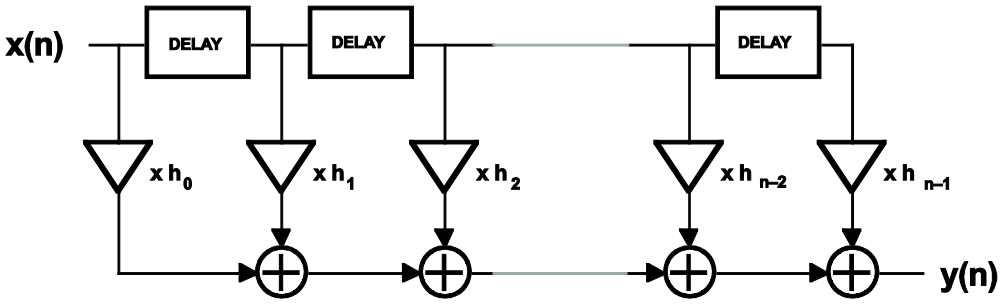
- odovzdávanie vstupných parametrov do volanej funkcie,
- uloženie používaných registrov pred začiatkom výkonnej časti knižničnej funkcie,
- spätné obnovenie používaných registrov pred návratom do volajúcej C funkcie,
- manažment zásobníka[1],
- odovzdanie návratovej hodnoty volajúcej C funkcie.

V prípade moderných DSP sa situácia ďalej komplikuje tým, že efektívne knižničné funkcie využívajú paralelný kód pracujúci s viacerými MAC jednotkami (v prípade jadra Blackfin s dvoma jednotkami MAC0 a MAC1). Zdrojové kódy knižničných funkcií sú tak, minimálne na prvý pohľad, pomerne zložité. Funkcia FIR filtra je najjednoduchšia funkcia ČSS a preto bude v rámci cvičenia detailne analyzovaná ako typický príklad knižničnej funkcie pre ČSS. Knižničné funkcie pre IIR filter a FFT budú v ďalších cvičeniach analyzované hlavne z aplikačného pohľadu.

7.1 KNIŽNIČNÁ FUNKCIA FIR FILTRA FIR_FR16

Štruktúra implementovaného FIR filtra, ktorý je realizovaný funkciou *fir_fr16()* je zobrazená na obr.7.1. FIR filter je definovaný rádom filtra, jeho koeficientmi a obsahom (stavom)[2] oneskorovacej linky.

[1] Zásobník je dátová štruktúra typu LIFO (Last In First Out), ktorá je vytváraná vo vyhradenej časti (dátovej) pamäte. Niektoré typy DSP používajú pre zásobník špeciálnu nezávislú pamäť (tzv. **hardvérový zásobník**). Ak je zásobník vytváraný v štandardnej pamäti, potom hovoríme o **softvérovom zásobníku**. Využitie zásobníka na odovzdávanie parametrov medzi volajúcou a volanou funkciou patrí medzi štandardné mechanizmy, ktoré sú používané v kompilátoroch jazyka C. Niektoré implementácie kompilátorov môžu **pri menšom počte** odovzdávaných parametrov použiť **namiesto zásobníka priamo vyhradené registre**. Týmto spôsobom je možné zjednodušiť manažment zásobníka a zrýchliť tak výsledný program.



Obr.7.1 Štruktúra transverzálnej realizácie FIR filtra

Funkcia *fir_fr16()* je súčasťou knižnice LIBDSP prostredia VisualDSP++. Zdrojové kódy aktuálnych knižničných funkcií (v asembleri) pre procesory Blackfin sú v
..\VisualDSP 5.1.2\Blackfin\lib\src\libdsp

Kompletný zdrojový kód knižničnej funkcie *fir_16asm()* zo staršej knižnice LIBDSP verzie VisualDSP 4.0 je v prílohe. Význam jednotlivých inštrukcií a štruktúra programu **budú podrobne analyzované počas cvičenia**. Aj bez detailnej analýzy je však z uvedeného kódu vidieť niektoré základné vlastnosti:

a) Kód v knižničnej funkcii spracováva odlišne párny a nepárny počet vstupných vzoriek a tiež FIR filter s párnym a nepárnym počtom koeficientov. Knižničná funkcia je optimalizovaná tak, aby maximálne využila prítomnosť dvoch MAC jednotiek, ktoré môžu pracovať paralelne. Kód knižničnej funkcie je optimalizovaný pre všetky možnosti a je tak podstatne dlhší ako kód pre DSP s jednou MAC jednotkou. V prípade, že by bolo zaručené, že funkcia bude spracovávať napr. len párny počet vstupných vzoriek, prípadne, že FIR filtre budú mať vždy párny počet koeficientov (v prípade potreby užívateľom doplnený o nulový koeficient), je možné kód funkcie výrazne skrátiť.

b) Kód obsahuje časti kódu, ktoré sú realizované podmieneným prekladom časti kódu ako napr.

```
#if defined(__WORKAROUND_CSYNC) || defined(__WORKAROUND_SPECULATIVE_LOADS)
```

alebo

```
#if defined(__WORKAROUND_BF532_ANOMALY38__)
```

Knižničná funkcia tak môže byť preložená v závislosti na definícii príslušných kľúčových slov. Prostredie VisualDSP++ obsahuje takto predkompilované knižnice. Programátor môže v menu projektu
Project-> Project Options->Project
okrem typu cieľového procesora Blackfin definovať aj tzv. **revíziu (Revision)** čipu.
V prípade potreby je možné presne definovať aj konkrétne **úpravy (Workarounds)** v menu
Project-> Project Options->Compile->Workarounds

[2] **Stav oneskorovacej linky** FIR filtra v diskrétnom čase n obsahuje vstupné vzorky FIR filtra v predchádzajúcich časových okamihoch $x(n-1)$, $x(n-2)$, $\ldots x(n-N+1)$. Ich uložením v štruktúre **fir_state_fr16** je možné v programe jednoducho realizovať FIR filtráciu blokovým spôsobom.

Analog Devices (podobne ako iní výrobcovia čipov) priebežne vylepšuje návrh čipov s cieľom eliminovať predchádzajúce konštrukčné chyby. Konkrétny procesor má na svojom čipe aj číslo aktuálnej revízie. Aby bolo možné používať bez problémov aj staršie revízie čipu, je možné používať verzie knižníc, ktoré tieto chyby eliminujú (typicky na úkor zníženia efektivity výsledného kódu). Zdrojové kódy knižníc preto obsahujú časti kódu pre všetky revízie čipov, ktoré môžu mať vplyv na správnu činnosť kódu. Za výber správnej revízie čipu je zodpovedný programátor. Vývojové dosky používané na cvičeniach používajú procesor s označením

Analog Devices
ADSP-BF5333
SKBC600
64 12.99. 1 **0.4** <- číslo revízie čipu

c) Úvodná časť kódu

```
[--SP]=(R7:4,P5:4);      //PUSH R7-R4,P5-P4 ON STACK
P5=[SP+36];              // ADDRESS OF FILTER STRUCTURE
```

zabezpečí uloženie registrov **R7:4, P5:4** (tzv. **Call Preserved Registers**) na softvérový zásobník. V ďalšej časti kódu funkcia môže uvedené registre bez obmedzenia modifikovať, pretože tieto registre sú pred návratom z funkcie zo zásobníka obnovené

```
(R7:4,P5:4)=[SP++];    //POP R7-R4,P5-P4 FROM STACK
RTS;
```

Kód funkcie využíva aj ďalšie R registre

R0,R1,R2,R3

ktoré sú modifikované bez uloženia na zásobník. Tieto registre patria do skupiny tzv. **Scratch Registers**, ktoré je možné v tele C funkcie modifikovať bez obmedzenia a nie je potrebné ich obnovovať. Do tejto skupiny patria aj registre

P0,P1,P2
LB0,LB1
LC0,LC1
LT0,LT1
ASTAT
A0,A1
I0,I1,I2,I3
B0,B1,B2,B3
M0,M1,M2,M3

Do skupiny tzv. **Dedicated Registers** patria registre

SP (P6) - ukazovateľ zásobníka (Stack Pointer)
FP (P7) - ukazovateľ rámca (Frame Pointer)
L0,L1,L2,L3 - definujú dĺžku kruhových bufrov (Circular Buffers)

Podrobnejšie informácie o využití jednotlivých registrov v rámci tzv. C++ Run-Time Model and Environment je možné nájsť v manuále C prekladača pre Blackfin DSP.

d) Funkcia *fir_fr16()* má prototyp

```
void fir_fr16(const fract16 x[],fract16 y[],int n,fir_state_fr16 *s);
```

Prvé tri parametre sú z dôvodu efektívnosti presunuté v registroch R0, R1 a R2, štvrtý parameter – adresa štruktúry *fir_state_fr16 *s* je odovzdávaná cez zásobník. Volajúca funkcia ukladá adresu na aktuálnu pozíciu zásobníka a volaná funkcia *fir_fr16()* ju vyčítava do registra P5 pomocou kódu

```
[--SP]=(R7:4,P5:4);      //PUSH R7-R4,P5-P4 ON STACK
P5=[SP+36];              // ADDRESS OF FILTER STRUCTURE
```

e) Kód funkcie *fir_fr16()* využíva kruhové bufre (modulo adresovanie) adresované pomocou DAG (Data Address Generator) jednotiek, ktoré umožňuje výrazné zefektívnenie kódu. Tento spôsob adresovanie je pre DSP typický. Nastavenie modulo adresovania je realizované zápisom dĺžky bufra do zodpovedajúcich L registrov jednotiek DAG, čo je realizované v úvodnej časti kódu.

Pred návratom z funkcie *fir_fr16()* je nevyhnutné nastaviť použité registre L späť na hodnotu 0, ktorá je vyžadovaná C++ Run-Time modelom. Hodnoty L=0 zabezpečia štandardné lineárne adresovanie DAG jednotiek.

7.2 TESTOVANIE FIR FILTRA S VYUŽITÍM IO FUNKCIÍ

Pri praktickom testovaní algoritmov je často potrebné overiť ich funkčnosť pre väčšiu množinu vstupných dát. Napr. testovanie funkcie *fir_fr16()* v predchádzajúcom cvičení je nedostatočné. Keďže bol spracovaný **len jeden** dátový blok, nebolo overené, či funkcia *fir_fr16()*, korektne uloží pred návratom všetky informácie o aktuálnom stave filtra do štruktúry *fir_state_fr16 state* a teda či následné ďalšie volania funkcie *fir_fr16()* **pre ďalšie** dátové bloky budú vracať korektné výsledky. Je teda potrebné simulovať spracovanie minimálne dvoch dátových blokov.

V praxi existujú aj aplikácie, kde je potrebné pri simulácií spracovávať veľké množstvo dát. Ako typický príklad je možné napr. uviesť analýzu zaokrúhľovacích chýb algoritmov ČSS. Takáto analýza typicky vyžaduje spracovanie veľkého množstva dát.

V ďalšej časti budú pri simulácii využité štandardné IO funkcie *read()* a *write()* dostupné v prostredí VisualDSP++.

7.2.1 IO FUNKCIE READ() A WRITE()

Prostredie VisualDSP++ umožňuje využitie niektorých IO funkcií z knižnice stdio. Tieto funkcie umožňujú realizovať čítanie a zápis do externých súborov uložených v PC (s využitím tzv. Default Device Driver Interface). Tento mechanizmus je možné využiť nielen v simulátore prostredia VisualDSP++, ale je pri ladení s vývojovými doskami EZ-Kit.

IO knižnice je možné tiež presmerovať do užívateľom definovaných periférii (napr. na rozhranie UARTu, s využitím mechanizmu rozšírenia o nové IO rozhrania). V ďalšej simulácii budú využité tri štandardné funkcie, ktorých prototypy sú definované v hlavičkovom súbore <stdio.h>:

```
FILE * fopen(const char *, const char *);
size_t fread(void *, size_t, size_t, FILE *);
size_t fwrite(const void *, size_t, size_t, FILE *);
```

7.2.2 TESTOVANIE FIR FILTRA S VYUŽITÍM EXTERNÝCH DÁTOVÝCH SÚBOROV

Pri ladení algoritmov ČSS je výhodné ak výstup simulácie je možné porovnať napr. s **vysoko-úrovňovou simuláciou** napr. v Matlabe[3]. Pre tento účel je výhodné ak je možné pripojiť k simulátoru **vstupné a výstupné súbory**, ktoré budú simulátorom čítané resp. do ktorých simulátor bude zapisovať výstupné hodnoty. Je výhodné ak formát týchto súborov je **prenositeľný** aj do iných systémov (napr. Matlabu).

Súčasťou projektu je **binárny súbor x.dat** so vstupnými vzorkami pre simuláciu. Každá vstupná vzorka je reprezentovaná 2 bajtmi a súbor x.dat bol generovaný v Matlabe pomocou m-súboru[4]:

```
% generuje data v binarnom formate (x.dat) pre overenie presnosti vypoctu FIR filtra
% v prostredi VisualDSP++

SCALE = 2^15;                % konverzia z 1.15 na 16.0
DATASIZE = 256*10            % pocet generovanych vzoriek

Fvz = 48000;                 % vzorkovacia frekvencia
A1 = 0.33;                   % amplituda 1. harmonickeho signalu
F1 = 2000;                   % frekvencia 1. harmonickeho signalu
A2 = 0.33;                   % amplituda 2. harmonickeho signalu
F2 = 8000;                   % frekvencia 2. harmonickeho signalu
A3 = 0.33;                   % amplituda 3. harmonickeho signalu
F3 = 10000;                  % frekvencia 3. harmonickeho signalu
n=(1:DATASIZE);              % diskretny cas

x = A1*sin(2*pi*F1*n/Fvz) + A2*sin(2*pi*F2*n/Fvz) + A3*sin(2*pi*F3*n/Fvz);
x = round( x*SCALE)';        % konverzia do formatu 16.0

outFile = fopen('x.dat','wb');
save x.txt x -ascii          % ulozenie vzoriek do suboru v ASCII formate (pre referencny vypocet)
fwrite(outFile,x,'short');   % ulozenie vzoriek v binarnom formate (2 bajty/vzorku) pre VisualDSp++
fclose(outFile);
```

Testovaný FIR filter bol navrhnutý v prostredí Filter Express na základe špecifikácie:

FIR Filter	Požiadavky:
Equiripple	Pass Band 0.3 dB
BandPass	Stop Band 65 dB
Fvz = 48 kHz	F1 = 5 kHz F2 = 6 kHz F3 = 9 kHz F4 = 10 kHz

pričom po návrhu a kvantovaní do formátu 1.15 boli dosiahnuté hodnoty

Počet koeficientov	122
Pass Band	0.4507 dB
Stop Band	59.731 dB

[3] Súčasťou projektu je aj testovací súbor firtest.m, pomocou ktorého je možné porovnať presnosť výpočtu FIR filtra v 16-bitovom procesore ADSP Blackfin a výpočtom FIR filtra v prostredí MATLAB s presnosťou 64 bitov.

[4] Súbor x.dat reprezentuje 16-bitové vzorky vstupného signálu zloženého z 3 sínusových signálov s rovnakými amplitúdami a frekvenciami zvolenými tak, že jedna zložka je z pásma priepustnosti testovaného FIR filtra.

Hlavný program *fir.c* je modifikáciou kódu z predchádzajúceho cvičenia, ktorý využíva knižničné funkciu *fir_fr16()* a *fir_init()*. Prístup k binárnym súborom **x.dat** a **y.dat** je realizovaný štandardným spôsobom pomocou funkcií *fopen()*, *read()* a *write()*:

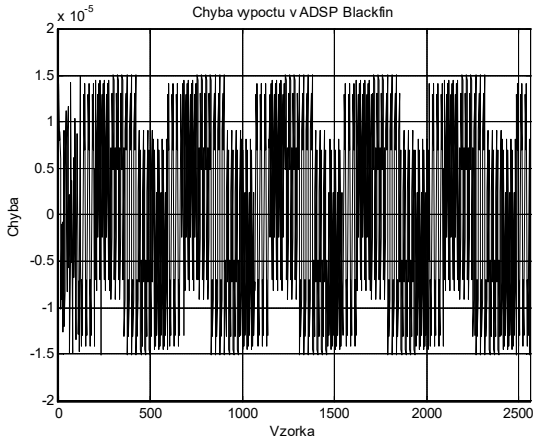
```
#include <stdio.h>
#include <fract.h>
#include <filter.h>

#define NUM_VEC      10      // pocet spracovavanych blokov
#define VEC_SIZE     256     // velkost spracovavaneho bloku
#define NUM_TAPS     122     // pocet koeficientov FIR filtra

fract16 coefs[NUM_TAPS] = {
        #include "coefs.h"        // subor s koeficientmi FIR filtra
};

fract16  delay[NUM_TAPS];
fir_state_fr16 state;             // declare filter state
fract16 in[VEC_SIZE];
fract16 out[VEC_SIZE];

FILE *inFile, *outFile;

int readInput(short *inBuf,int count) {
        int wordsRead=0;
        wordsRead = fread(inBuf,sizeof(short),count,inFile);
        return wordsRead;
}

int writeOutput(short *outBuf,int count) {
        int wordsRead=0;
        wordsRead = fwrite(outBuf,sizeof(short),count,outFile);
        return wordsRead;
}

void main( void ){
        int i;

        inFile = fopen("D:\\SPvT\\x.dat","rb");                    // nastavena aktualna cesta k
testovanym suborom
        outFile = fopen("D:\\SPvT\\y.dat","wb");                   // (zmenit podla aktualnej struktury
adresarov)

        fir_init(state, coefs, delay, NUM_TAPS, 1);        // inicializacia stavu filtra
        i = NUM_VEC;
        printf("Zaciatok testu\n");
        while( i-- ) {
                readInput(in, VEC_SIZE);                           // nacitanie vstupneho bloku
                fir_fr16(in, out, VEC_SIZE, &state);               // filtracia vstupneho bloku
                writeOutput(out,VEC_SIZE);                         // zapis vystupneho bloku
        }
        printf("Koniec testu");
        fclose( inFile );
        fclose( outFile );
}
```

Počas cvičenia bude vykonaná kompletná simulácia programu fir.c so vstupnými vzorkami zo súboru **x.dat**, ktoré sú súčasťou projektu.

Nasledujúci jednoduchý program v Matlabe umožňuje porovnať výsledky zo simulácie ADSP Blackfin s referenčným výpočtom FIR filtra v Matlabe.

```
% Subor testuje vystup FIR filtra vypocitany v prostredi VisualDSP

SCALE = 2^15;
DATASIZE = 256*10              % pocet testovanych vzoriek

inFile = fopen('x.dat','rb');  % subor so vstupnymi vzorkami (format 16.0)
[x,cnt] = fread(inFile,DATASIZE,'short');
x = x/SCALE;                   % prevod do formatu 1.15

outFile = fopen('y.dat','rb'); % subor s vystupnymi vzorkami z VisualDSP simulacie (format 16.0)
[y,cnt] = fread(outFile,DATASIZE,'short');
y = y/SCALE;                   % prevod do formatu 1.15

load coefs.txt;                % koeficienty FIR filtra vo formate (1.15)
h = coefs;
N = length(h);                 % pocet koeficientov FIR filtra

y_ref = filter(h,1,x);         % referencny vypocet FIR filtra v Matlabe
e = y - y_ref;                 % rozdiel medzi VisualDP a Matlabom

plot(e);                       % max chyba by mala byt radovo 2^(-15)
title('Chyba vypoctu v ADSP Blackfin');
xlabel('Vzorka');
ylabel('Chyba');
grid;
```

Na obrázku 7.2 je zobrazená chyba výpočtu FIR filtra v procesore ADSP Blackfin, pričom všetky údaje boli transformované do formátu 1.15. Z obrázku je zrejmé, že maximálna chyba pre daný vstupný signál je menšia ako 1 LSB $\doteq 2^{-15} = 3.1*10^{-5}$.



Obr.7.2 Chyba výpočtu FIR filtra v procesore ADSP Blackfin

Príklad

Realizujte kompletnú simuláciu FIR filtra pomocou funkcie fir_fr16()_a overte správnosť údajov na Obr.7.2. Sledujte výstup funkcie printf(), ktorý je presmerovaný do výstupného okna prostredia VisualDSP++.

Príklad

Určite rýchlosť výpočtu funkcie fir_fr16() pre párny a nepárny počet vstupných vzoriek a koeficientov FIR filtra. Asymptoticky by mal FIR filter v jadre DSP Blackfin na spracovanie jedného bloku dosahovať rýchlosť $\sim N_{FIR}N_{block}/2 = \sim 122*256/2 = 15616$ *cyklov/blok. Namerané výsledky sú výrazne odlišné.* ***Prečo?***

Príklad
Zopakujte predchádzajúci test po pridaní nasledujúcich riadkov (zvýraznený tučným písmom):

static segment("L1_data_b")
fract16 coefs[NUM_TAPS] = {
#include "coefs.h" // subor s koeficientmi FIR filtra
};

static segment("L1_data_a")
fract16 delay[NUM_TAPS];

Ako a prečo sa zmenia výsledky?

7.2.3 PREDKOMPILOVANÁ SIMULÁCIA

Predchádzajúce simulácie v prostredí **klasického simulátora** VisualDSP++ boli pomerne pomalé. VisualDSP++ umožňuje zrýchliť simulácie s využitím tzv. predkompilovanej simulácie (Compiled Simulation). Klasický simulátor využíva v počítači PC **opakované** dekódovanie a interpretovanie simulovaných inštrukcií. Predkompilovaná simulácia zrýchľuje simuláciu predkompilovaním celého simulovaného kódu, ktorý už počas simulácie nie je potrebné opakovane dekódovať, čo umožňuje výrazné zrýchlenie simulácie[5]. Nevýhodou predkompilovanej simulácie je obmedzená podpora simulačných vlastností prostredia VisualDPS++[6].

Predkompilovanú simuláciu je možné zvoliť v menu
Session\Select Sesion\ADSP-BF533 Blackfin Family Compiled Simulation

Ak ešte v prostredí VisualDSP++ predkompilovaná simulácia nebola využívaná, je potrebné zvoliť ju pre konkrétny cieľový procesor v menu
Session\New Session

Príklad
Porovnajte rýchlosť predkompilovanej simulácie a klasickej simulácie pri testovaní FIR filtra s využitím externých dátových súborov.

7.3 ĎALŠIE KNIŽNIČNÉ FUNKCIE PRE FIR FILTRÁCIU

Súčasťou knižnice LIBDSP sú knižničné funkcie pre decimáciu a interpoláciu[7] pomocou FIR filtrov:

[5] Aj keď moderné počítače PC sú pomerne výkonné, rýchlosť DSP sa tiež výrazne zvyšuje. Kompletná simulácia napr. DSP so 600 MHz taktovaním tak môže byť pri zložitejších simuláciách pomerne zdĺhavá.

[6] Nie je napr. podporovaný zápis resp. čítanie do/z rozhrania UART pomocou tzv. „streams“, ktoré budú využívané v nasledujúcich cvičeniach.

[7] Decimácia je proces, pri ktorom je znížená vzorkovacia frekvencia spracovávaného signálu. Decimačný FIR filter zabezpečí, že nedochádza k prekrytiu spektra. Interpolačný filter naopak zvyšuje

```
void fir_decima_fr16(const fract16 x[], fract16 y[], int  N,  fir_state_fr16  *s);
void fir_interp_fr16(const fract16 x[], fract16 y[], int  N,  fir_state_fr16  *s);
```

Prototypy uvedených funkcií sú v hlavičkovom súbore <filter.h> a zdrojové kódy knižničných funkcií (v asembleri) pre procesory Blackfin sú v

..\VisualDSP 5.1.2\Blackfin\lib\src\libdsp

podobne ako pre funkciu *fir_fr16()*. Všetky funkcie pre FIR filtráciu využívajú štruktúru *fir_state_fr16* opísanú v predchádzajúcej časti.

Implementovaný FIR filter vyžaduje na realizáciu FIR filtra s N koeficientmi približne[8] $N/2$ inštrukčných cyklov. Signálové procesory rodiny ADSP BF533-600, ktoré budú využívané na cvičeniach, môžu pracovať na frekvencii 600 MHz a realizovať 1200 MAC/s. Tieto hodnoty určujú maximálne frekvencie vzorkovania F_{vz} resp. pri danej frekvencii vzorkovania maximálny rád FIR filtra N, ktorý je možné implementovať. Napr. pre FIR filter s $N = 1000$ koeficientmi (čo už je pomerne zložitý FIR filter) je možné s procesorom, ktorý realizuje 1200 MAC/s pracovať až do

$$F_{vz} \approx \frac{600 \times 10^6}{N/2} = \frac{1200 \times 10^6}{1000} = 1,2\left[MHz\right] \tag{7.1}$$

čo jasne dokumentuje vysokú výkonnosť architektúry ADSP Blackfin pri realizácii tohto základného algoritmu ČSS.

vzorkovaciu frekvenciu spracovávaného signálu pričom interpolačný FIR filter zabezpečí zachovanie pôvodného spektra vstupného signálu. Knižničné funkcie podporujú len celočíselné decimačné a interpolačne faktory.

[8] Vzťah platí dostatočne presne pre veľké hodnoty N a optimálne umiestnenie bufrov (pre stavy filtra a jeho koeficienty) v oddelených interných datových pamätiach.

PRÍLOHA

```
/*****************************************************************************
C/opyright(c) 2000-2004 Analog Devices Inc.
*****************************************************************************
File Name         : fir_fr16.asm
Function Name  : __fir
Module Name    : FILTER Library
Description       : This function performs FIR filter operation on given input.
Operands          : R0 - Address of input vector,
                    R1 - Address of output vector,
                    R2 - Number of input elements
                    Filter structure is on stack.

Prototype:      void fir(const fract16 x[],fract16 y[],int n,fir_state_fr16 *s);

Structure of fir_state_fr16:
          {
               fract16 *h,              // filter coefficients
               fract16 *d,              // start of delay line
               fract16 *p,              // read/write pointer
               int k,                   // no. of coefficients
               int l                    // interpolation/decimation index
          } fir_state_fr16;

Registers used :

     R0, R1, R2, R3, R3, R4, R5, R6, R7

     I0 -> Address of delay line (used for updating the delay line)
     I1 -> Address of delay line (used for fetching the delay line data)
     I2 -> Address of filter coeff. h0, h1 , ... , hn-1
     I3 -> Address of output array y[]

     P0 -> Address of input array x[]
     P1, P2, P4
     P5 -> Address of structure s


     Code size: 354 bytes
_____________________________________________________________________________
     Computation time:

          a) Even Taps, Even no. inputs   : 64 + Ni/2*(3 + Nc)
            For Ni=256, L=8                    : 1472

          b) Odd Taps, Even no. inputs    : 70 + Ni/2*(3 + Nc)
            For Ni=256, L=9                    : 1606

          c) Even Taps, Odd no. inputs    : 64 + (Ni-1)/2*(3 + Nc) + Nc
            For Ni=257, L=8                    : 1480

          d) Odd Taps, Odd no. inputs          : 73 + (Ni-1)/2*(3 + Nc) + Nc
            For Ni=257, L=9                    : 1618
_____________________________________________________________________________


*****************************************************************************/

#if defined(__ADSPLPBLACKFIN__) && defined(__WORKAROUND_AVOID_DAG1)
#define __WORKAROUND_BF532_ANOMALY38__
#endif

.section program;
.global    __fir_fr16;
.align 2;
__fir_fr16:
     [--SP]=(R7:4,P5:4);          //PUSH R7-R4,P5-P4 ON STACK
     P5=[SP+36];                  // ADDRESS OF FILTER STRUCTURE
```

```
        P0=R0;                  // ADDRESS OF INPUT ARRAY
        I3=R1;                  // ADDRESS OF OUTPUT ARRAY
        CC=R2<=0;               // CHECK IF NUMBER OF INPUT ELEMENTS<=0
        P1=[P5++];              // POINTER TO FILTER COEFFICIENTS
        P2=[P5++];              // POINTER TO DELAY LINE
        P4=[P5++];              // ADDRESS TO READ/WRITE POINTER
        R1=[P5--];              // NUMBER OF COEFFICIENTS

        IF CC JUMP _fir_fr16_RET_END;
        CC=R1<=0;               // CHECK IF NUMBER OF COEFF. <=0
        IF CC JUMP _fir_fr16_RET_END;
        R5=R1;                  // STORE NUMBER OF FILTER COEFF. IN R5
        R1=R1+R1;               // DOUBLE R1 TO INCREMENT AS HALF WORD
        I2=P1;                  // INITIALISE I2 TO FILTER COEFF. ARRAY
        B2=P1;                  // MAKE FILTER COEFFS. AS CIRCULAR BUFFER
        L2=R1;
        I1=P4;                  // INDEX TO READ/WRITE POINTER
        B1=P2;                  // INITIALISE B1 AND L1
        L1=R1;                  // FOR  DELAY LINE CIRCULAR BUFFER
        I0=P4;                  // INDEX TO READ/WRITE POINTER
        B0=P2;                  // INITIALISE B0 AND L0
        L0=R1;                  // FOR  DELAY LINE CIRCULAR BUFFER

        P1=R2;                  // SET OUTER LOOP COUNTER
        P2=R5;                  // SET INNER LOOP COUNTER
        L3 = 0;

        R6=PACK(R2.H,R2.L) || I1+=4;
        CC=R6==1;

        IF CC JUMP _fir_fr16_SING_SAMP;

        CC=BITTST(R5,0);        // Check if the number of filter taps are odd
        P2+=-2;                 // Nc-2

        IF CC JUMP _fir_fr16_FIR_ODD;   //Odd tap FIR

#if defined(__WORKAROUND_CSYNC) || defined(__WORKAROUND_SPECULATIVE_LOADS)
        NOP;
        NOP;
        NOP;
#endif

/*****************************************************************************/
        /** Even number of input samples **/
        /** Even number of filter taps   **/

#if defined(__WORKAROUND_BF532_ANOMALY38__)

        /* ---------- Start of BF532 Anomaly#38 Safe Code ----------          **
        **                                                                    **
        ** which implements the IIR filter for an even number of samples     **
        ** and an even number of filter coefficients                          **
        **                                                                    */

        I0+=2 || R2=[I2++];
        R0=[P0++];
        I1+=2;

        LSETUP(_fir_fr16_E_FIR_START,_fir_fr16_E_FIR_END) LC0=P1>>1;
        // Loop 1 to Ni/2
_fir_fr16_E_FIR_START:
            A1=R0.H*R2.L,A0=R0.L*R2.L ||I1-=2 || W[I0--]=R0.L;
            R4=PACK(R0.H,R4.L) || R0.H=W[I1++];

            LSETUP(_fir_fr16_E_MAC_ST,_fir_fr16_E_MAC_END) LC1=P2>>1;
            //Loop 1 to Nc-1/2
_fir_fr16_E_MAC_ST:   R2.L=W[I2++];
                      A1+=R0.L*R2.H,A0+=R0.H*R2.H || R0.L=W[I1++];
                      R2.H=W[I2++];
_fir_fr16_E_MAC_END: A1+=R0.H*R2.L,A0+=R0.L*R2.L || R0.H=W[I1++];

            R3.H=(A1+=R0.L*R2.H),R3.L=(A0+=R0.H*R2.H) || R0=[P0++] || W[I0--]=R4.H;
_fir_fr16_E_FIR_END:
            R2=[I2++] || [I3++]=R3;
```

```
#else  /* End of BF532 Anomaly#38 Safe Code */

	/* ---------- Start of NON BF532 Anomaly#38 Safe Code ---------	**
	**                                                              **
	** which implements the IIR filter for an even number of samples	**
	** and an even number of filter coefficients                    **
	**                                                              */

	I0+=2 || R0.L=W[I1++];
	R0=[P0++] || R2=[I2++];

	LSETUP(_fir_fr16_E_FIR_START,_fir_fr16_E_FIR_END) LC0=P1>>1;
	// Loop 1 to Ni/2
_fir_fr16_E_FIR_START:
		A1=R0.H*R2.L,A0=R0.L*R2.L ||I1-=2 || W[I0--]=R0.L;
		R4=PACK(R0.H,R4.L) || R0.H=W[I1++] || NOP;

		LSETUP(_fir_fr16_E_MAC_ST,_fir_fr16_E_MAC_END) LC1=P2>>1;
		//Loop 1 to Nc-1/2

_fir_fr16_E_MAC_ST:    A1+=R0.L*R2.H,A0+=R0.H*R2.H || R2.L=W[I2++] || R0.L=W[I1++];
_fir_fr16_E_MAC_END: A1+=R0.H*R2.L,A0+=R0.L*R2.L || R0.H=W[I1++] || R2.H=W[I2++];
		R3.H=(A1+=R0.L*R2.H),R3.L=(A0+=R0.H*R2.H) || R0=[P0++] || W[I0--]=R4.H;
_fir_fr16_E_FIR_END:
		R2=[I2++] || [I3++]=R3;

#endif		/* End of Alternative to BF532 Anomaly#38 Safe Code */

	R0.L=W[I0--] || I2-=4;
	R0=I0;
	R2.L=W[I1--];
	P0+=-4;

	JUMP _fir_fr16_DATA;

/*****************************************************************************/
		/** Even number of input samples **/
		/** Odd number of filter taps    **/

_fir_fr16_FIR_ODD:

	B3=I3;
	R2>>=1;
	R2=R2 << 2 || R6.H=W[I1++];
	L3=R2;
	R5=[P0];

#if defined(__WORKAROUND_BF532_ANOMALY38__)

	/* ---------- Start of BF532 Anomaly#38 Safe Code ----------	**
	**                                                              **
	** which implements the IIR filter for an even number of samples	**
	** and an odd number of filter coefficients                     **
	**                                                              */

	R2=[I2++] || I3-=4;
	R3=[I3];

	LSETUP(_fir_fr16_O_FIR_START,_fir_fr16_O_FIR_END) LC0=P1>>1;
	//Loop 1 to Ni/2
_fir_fr16_O_FIR_START:
		R4.H=W[I0] || W[I0++]=R5.H;
		R0=PACK(R6.H,R5.L) || R7=[P0++];
		A1=R5.H*R2.L,A0=R5.L*R2.L || R5=[P0] || W[I0--]=R5.L;

		LSETUP(_fir_fr16_O_MAC_ST,_fir_fr16_O_MAC_END) LC1=P2>>1;
		//Loop 2 to Nc/2-1
_fir_fr16_O_MAC_ST:   A1+=R0.L*R2.H,A0+=R0.H*R2.H || R2=[I2++];
			R0.L=W[I1++];
_fir_fr16_O_MAC_END: A1+=R0.H*R2.L,A0+=R0.L*R2.L || R0.H=W[I1++];

		A1+=R0.L*R2.H,A0+=R0.H*R2.H || R2.H=W[I2++] || I0-=4;
		//Done only for Odd number of coeffs
		R3.L=(A0+=R4.H*R2.H) || R6.H=W[I1++] ||  [I3++]=R3 ;
```

```
_fir_fr16_O_FIR_END:
        R3.H=(A1+=R0.H*R2.H) || R2=[I2++];

#else           /* End of BF532 Anomaly#38 Safe Code */

    /* ---------- Start of NON BF532 Anomaly#38 Safe Code ----------    **
    **                                                                  **
    ** which implements the IIR filter for an even number of samples   **
    ** and an odd number of filter coefficients                        **
    **                                                                  */

    R4.H=W[I0] || I3-=4;
    R3=[I3] || R2=[I2++];

    LSETUP(_fir_fr16_O_FIR_START,_fir_fr16_O_FIR_END) LC0=P1>>1;
    //Loop 1 to Ni/2
_fir_fr16_O_FIR_START:
        R0=PACK(R6.H,R5.L) || R7=[P0++] || W[I0++]=R5.H ;
        A1=R5.H*R2.L,A0=R5.L*R2.L || R5=[P0] || W[I0--]=R5.L;

        LSETUP(_fir_fr16_O_MAC_ST,_fir_fr16_O_MAC_END) LC1=P2>>1;
        //Loop 2 to Nc/2-1
_fir_fr16_O_MAC_ST:    A1+=R0.L*R2.H,A0+=R0.H*R2.H || R2=[I2++] || R0.L=W[I1++];
_fir_fr16_O_MAC_END:  A1+=R0.H*R2.L,A0+=R0.L*R2.L || R0.H=W[I1++];

        A1+=R0.L*R2.H,A0+=R0.H*R2.H || R2.H=W[I2++] || I0-=4;
        //Done only for Odd number of coeffs
        R3.L=(A0+=R4.H*R2.H) || R6.H=W[I1++] ||  [I3++]=R3 ;
_fir_fr16_O_FIR_END:
        R3.H=(A1+=R0.H*R2.H) || R2=[I2++] || R4.H=W[I0];

#endif          /* End of Alternative to BF532 Anomaly#38 Safe Code */

    L3 = 0;
    R0 = I0;
    I2-=4;
    [I3++] = R3 || I1-=2;

/*****************************************************************************/

_fir_fr16_DATA:
    CC=BITTST(R6,0);
    IF !CC JUMP _fir_fr16_RET_END1;
    //If even number of input samples, jump to ret_end

/*****************************************************************************/
        /** Process the last sample separately **/
        /** for an odd number of input samples **/

    P2+=2;

_fir_fr16_SING_SAMP:

#if defined(__WORKAROUND_BF532_ANOMALY38__)

    /* ---------- Start of BF532 Anomaly#38 Safe Code ----------          **
    **                                                                   **
    ** which processes the last sample separately for an odd number of   **
    ** number of input samples                                           **
    **                                                                   */

#if defined(__WORKAROUND_CSYNC) || defined(__WORKAROUND_SPECULATIVE_LOADS)
    NOP;
    NOP;
    NOP;
#endif

        A1=A0=0 || R0=W[P0] (Z);
        R4 = PACK(R0.H,R0.L) || R2.H=W[I0++];

        LSETUP(_fir_fr16_L_MAC_ST,_fir_fr16_L_MAC_END)LC1=P2;//Loop 1 to Nc
_fir_fr16_L_MAC_ST:    R2.L=W[I2++];
_fir_fr16_L_MAC_END:   R3.L=(A0+=R0.L*R2.L) || R0.L=W[I1++];

#else           /* End of BF532 Anomaly#38 Safe Code */
```

```
        /* ---------- Start of NON BF532 Anomaly#38 Safe Code ----------          **
        **                                                                       **
        ** which processes the last sample separately for an odd number of      **
        ** number of input samples                                               **
        **                                                                       */

        A1=A0=0 || R0=W[P0] (Z) || R2.L=W[I2++];
        R4 = PACK(R0.H,R0.L) || R2.H=W[I0++];

        LSETUP(_fir_fr16_L_MAC_ST,_fir_fr16_L_MAC_ST)LC1=P2;//Loop 1 to Nc
_fir_fr16_L_MAC_ST:   R3.L=(A0+=R0.L*R2.L) || R2.L=W[I2++] || R0.L=W[I1++];

#endif              /* End of Alternative to BF532 Anomaly#38 Safe Code */

        W[I0--]=R4.L;
        W[I3++]=R3.L || I0-=2;
        R0=I0;

_fir_fr16_RET_END1:
        [P5]=R0;

/*****************************************************************************/

_fir_fr16_RET_END:
        L0=0;
        L1=0;
        L2=0;
        (R7:4,P5:4)=[SP++];     //POP R7-R4,P5-P4 FROM STACK
        RTS;

.__fir_fr16.end:
```

8 VÝVOJOVÝ MODUL ANALOG DEVICES ADSP-BF533 EZ-KIT LITE

Výrobcovia DSP typicky poskytujú pre svoje DSP vývojové moduly, ktoré umožňujú otestovať činnosť programov pomocou reálneho hardvéru. Tieto moduly umožňujú otestovať činnosť algoritmov ČSS v reálnych podmienkach pri využití signálov z reálnych rozhraní (typicky audio AD a DA prevodníky, UARTy, video prevodníky a pod.). Typicky sú moduly externe[1] pripojiteľné k počítaču PC, čo výrazne zvyšuje flexibilitu práce s uvedenými modulmi.

Firma Analog Devices ponúka pre všetky typy svojich DSP **bohato vybavené** vývojové moduly – tzv. EZ KITy. Ich aktuálny prehľad je možné nájsť na stránkach firmy Analog Devices.

Súčasné vývojové DSP moduly sú už pomerne zložité procesorové systémy osadené veľkými kapacitami externých SRAM a FLASH pamätí[2] a vybavené výkonnými rozhraniami. Nízko-úrovňové programovanie týchto rozhraní vyžaduje okrem dobrého zvládnutia periférnych obvodov v cieľovom procesore aj detailnú znalosť externých obvodov (napr. AD a DA kodekov). Pri práci s uvedenými modulmi tak na cvičeniach vzniká dilema. Môžu byť analyzované detailné vlastnosti jednoduchších programov, avšak uvedený postup vzhľadom na obmedzené množstvo času neumožní opis a zvládnutie systémových možností práce s modernými DSP modulmi.

V rámci cvičení je preto priorita kladená na **opis a zvládnutie systémových** vlastností cieľového DSP modulu. Detaily použitých programov je možné zvládnuť v rámci semestrálnych projektov, diplomových prác, prípadne počas samostatnej práce v rámci zadaní zo špecializovaných predmetov, ktoré DSP využívajú.

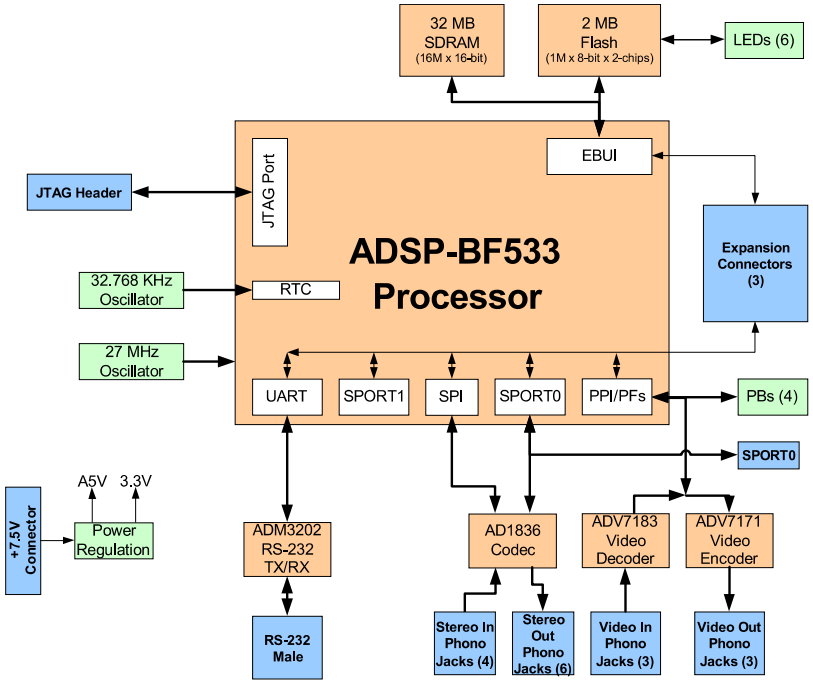
8.1 VÝVOJOVÝ MODUL ADSP-BF533 EZ-KIT LITE

Vývojový modul **ADSP BF533 EZ-KIT Lite** je vývojovým prostriedkom, ktorý umožňuje ladenie programov pre procesory Blackfin ADSP-BF533[3]. Bloková štruktúra EZ-KIT Lite modulu je zobrazená na obr.8.1.

[1] Staršie moduly najčastejšie využívali sériové (RS232) a paralelné (LPT) rozhrania počítačov. V súčasnosti sú najčastejšie využívané USB rozhrania.

[2] Kapacity osadených pamätí dnes bežne dosahujú niekoľko desiatok Megabajtov, čo je o niekoľko rádov viac ako kapacita pamätí typických vývojových DSP modulov z pred 10 rokov.

[3] Moduly ADSP-BF533 EZ-KIT Lite využívané v rámci cvičení sú osadené 600 MHz procesorom v 160-vývodovom puzdre Mini-BGA. Procesor využívajú revíziu jadra v.0.4.



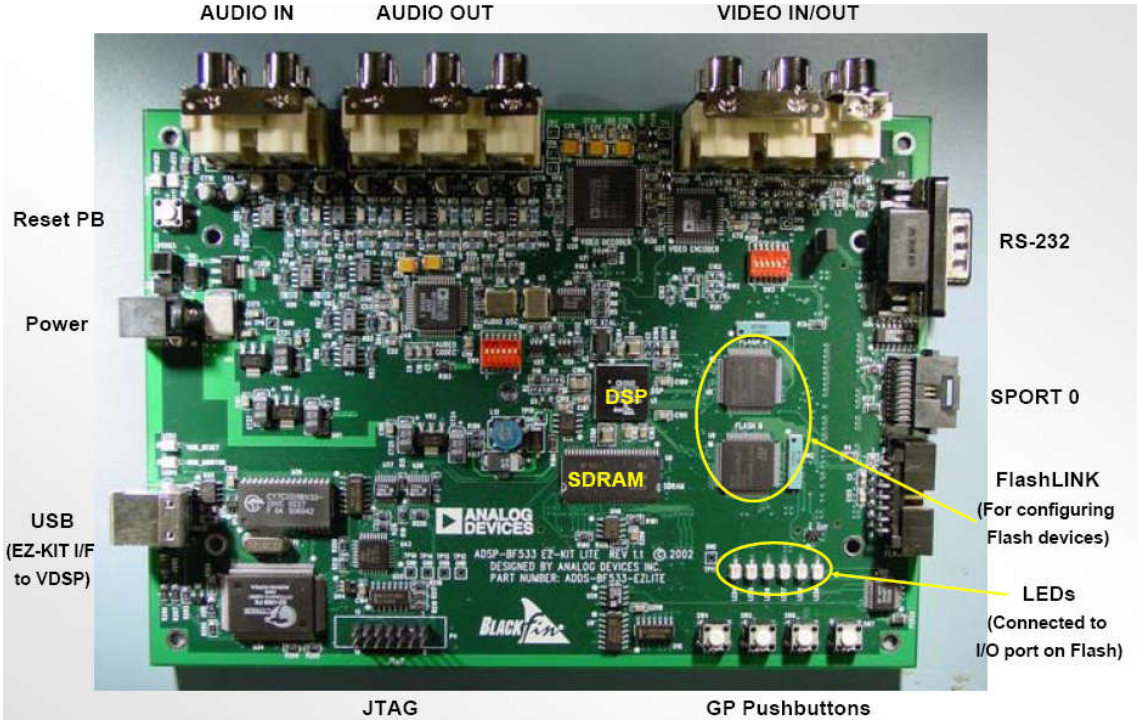
Obr.8.1 Bloková štruktúra modulu ADSP BF-533 EZ-KIT Lite

Modul je osadený 64 MB SDRAM (Synchronous Dynamic Random Access Memory) pamäťou v konfigurácii 32M × 16bitov a FLASH pamäťou s kapacitou 2 MB (512K×16×2 čipy).

Audio rozhranie modulu využíva 96 kHz **16-bitový** multimediálny AD/DA **kodek AD1836** od firmy Analog Devices, ktorý spolu s procesorom Blackfin umožňuje realizovať klasický systém ČSS s viackanálovými analógovými vstupmi a výstupmi. Pretože AD a DA prevodníky kodeku využívajú **sigma-delta moduláciu**, sú vstupné a výstupné analógové obvody relatívne jednoduché. Kodek má 4 audio vstupy (2 kanály) a 6 audio výstupov (3 kanály). Video rozhrania implementované na doske sú tvorené video dekóderom ADV7183 a video kodérom ADV7171.

Rozhranie UART s obvodom ADM3202 je využiteľné na prepojenie DSP modulu a počítača PC pomocou rozhrania RS232.

Modul je ďalej osadený 14-pinovým JTAG konektorom pre pripojenie externého emulátora, USB konektorom pre pripojenie k nadradenému PC, 10-timi LED diódami a 5-timi tlačidlami. Na obr.8.2 je zobrazené rozloženie jednotlivých komponentov modulu ADSP BF-535 EZ-KIT, kompletné schémy modulu sú dostupné na stránkach výrobcu.



Obr.8.2 Rozmiestnenie jednotlivých komponentov a rozhraní modulu Analog Devices ADSP BF-533 EZ-KIT Lite

8.2 VYUŽITIE ŠTANDARDNÝCH IO FUNKCIÍ S DSP MODULOM

8.2.1 IO FUNKCIE A ŠTANDARDNÝ VSTUP/VÝSTUP

V predchádzajúcich cvičeniach boli pri simuláciách využívané štandardné IO funkcie napr. na diagnostický výstup pomocou funkcie *printf()* resp. čítanie a zápis z/do súborov v nadradenom PC počítači. Prostredie VisualDSP++ umožňuje transparentné využitie uvedených ladiacich postupov aj s EZ-KIT modulom. Jediná zmena, ktorú je pri ladení s DSP modulom nevyhnutné vykonať je nastavenie ADSP BF-533 EZ-LITE v menu
Session\Select Sesion\ADSP-BF533 ADSP-BF533 EZ-KIT Lite via Debug Agent[4]

Ak ešte v prostredí VisualDSP++ uvedený modul nebol využívaný, je potrebné zvoliť ho v menu
Session\New Session

[4] EZ-KIT obsahuje obvody umožňujúce ladenie pomocou JTAG rozhrania aj bez JTAG adaptéra. Výrobca EZ-KITu tak umožnil použitie výkonných ladiacich techník (napr. krokovanie programu, prenos informácií cez spätný telemetrický kanál a pod.) aj bez nutnosti použiť drahý externý JTAG adaptér. Oproti plnohodnotnému externému JTAG adaptéru je však výkonnosť tohto riešenia znížená. Obvodové riešenie USB-JTAG prepojenia nie je súčasťou voľne dostupných schém modulu ADSP BF533 EZ-LITE.

Príklad
Overte funkčnosť vybraných štandardných IO funkcií pri testovaní knižničnej funkcie fir_fr16() z predchádzajúceho cvičenia. Na testovanie použite vývojový modul ADSP BF-533 EZ-KIT Lite.

8.2.2 PRESMEROVANIE IO FUNKCIÍ NA ROZHRANIE UART

Rozhranie UART je pri ladení mikroprocesorových aplikácií často využívané[5] na diagnostický vstup/výstup. Programátor pomocou neho môže ovplyvňovať činnosť cieľového hardvéru (napr. čakaním na údaje z klávesnice terminálového programu na PC, ktorý posiela výsledky na sériový port), prípadne posielať pomocou funkcie *printf()* z aplikácie diagnostické výstupy na obrazovku terminálového programu na PC.

Flexibilnosť IO knižnice pre procesory Blackfin umožňuje jednoduché presmerovanie štandardných vstupov a výstupov na ľubovoľné zariadenie, pre ktoré existuje nízko-úrovňový softvérový drajver. Aj keď súčasťou „run-time“ knižníc prostredia VisualDSP++ nie sú žiadne nízko-úrovňové drajvre, požiadavky a mechanizmus ich činnosti je podrobne opísaný v príslušných manuáloch proestredia VisualDSP++.

Projekt STDIO UART je príkladom ako je možné presmerovať štandardné IO funkcie na rozhranie UART. Projekt sa skladá z programu *main.c* (uvedený v skrátenej forme) projektu STDIO UART:

```
#include <stdlib.h>
#include <device.h>
#include <device_int.h>
#include <stdio.h>
#include <string.h>

unsigned short CLKIN = 27;      // CLKIN frekvencia je v ADSP-BF533 EZ-Kits 27 MHz

#define BAUDRATE     2400    // je mozne zmenit na lubovolnu hodnotu poporovanu terminalovym programom

#define UART_DEVICEID  2
#define MAX_BUF        128
void backwards ( char *buf );
extern DevEntry UART_DevEntry;

int main () {

        // vytvori novy identifikator zariadenia - device ID
        int result = 0;
        static char buf[ MAX_BUF ];
        UART_DevEntry.DeviceID = UART_DEVICEID;
        UART_DevEntry.data     = (void *) BAUDRATE;

        // zapis “new device” do tabulky zariadeni (device table)
        result = add_devtab_entry ( &UART_DevEntry );
        set_default_io_device( UART_DEVICEID );
```

[5] Rozhranie UART bolo v počiatkoch využívania mikroprocesorovej techniky „vysoko-úrovňovým” ladiacim prostriedkom, pomocou ktorého bolo možné odladiť aj náročné aplikácie bez využitia simulátorov a emulátorov, ktoré neboli často pre mnohých vývojárov dostupné.

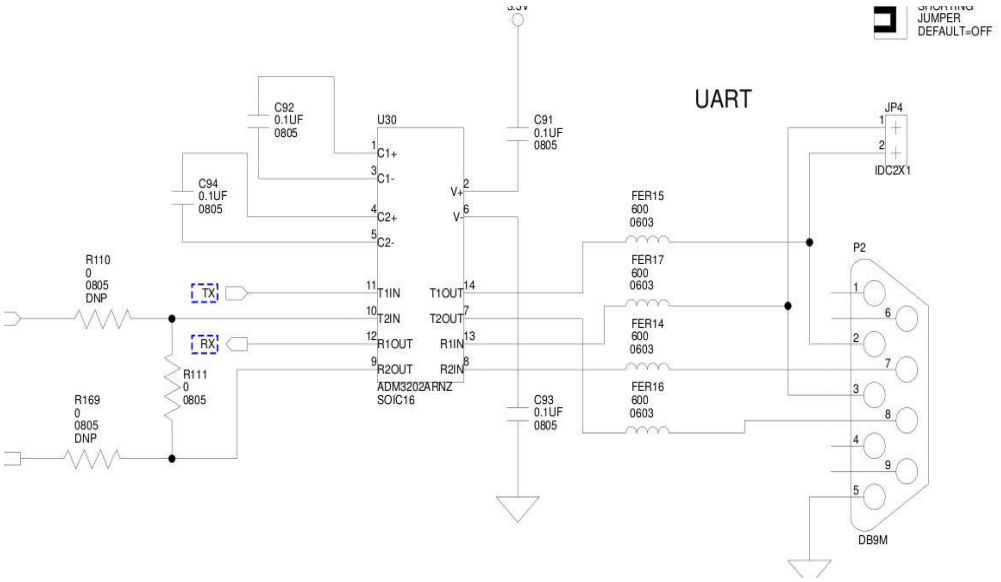
```
        // presmerovanie STDIO pre funkcie printf a scanf
        FILE *fp;

        fp = freopen("", "a+", stdin);
        fp = freopen("", "a+", stdout);

        printf("\n\r================\n\r");
        printf("Hello, World!\n\r");
        printf("Type in something, press <Enter>, and I'll repeat it backwards!\n\r");

        scanf ( "%s", buf );
        backwards ( buf );

        return 0;
}
```

Uvedený program je príkladom ako je možné pomocou začlenenia nového drajvra dynamicky (počas činnosti programu) presmerovať vstupy a výstupy IO knižnice. Kompletný kód drajvra pre presmerovania vstupov a výstupov IO knižnice na UART procesora Blackfin je v zdrojovom kóde *stdio_uart.c*. Uvedený kód je pomerne jednoduchý, obsahuje však už nízko-úrovňové časti kódu, ktoré pracujú s perifériou UART procesora Blackfin. Ako typický príklad je možné uviesť nízko-úrovňové funkcie na zápis (vysielanie) pomocou rozhrania UART procesora Blackfin:

```
// vyslanie znaku s vyuzitim testovania (polling) bitu THRE v LSR registri
void UART_putc(char c) {
        while (!(*pUART_LSR & THRE)) { };       // cakaj
        *pUART_THR = c;
}

void UART_puts(char *c) {
        while (*c)        {
                UART_putc(*c);
                c++;
        }
}
```

Príklad

Overte funkčnosť presmerovania funkcii printf() a scanf() na rozhranie UART pomocou projektu STDIO UART. Na testovanie použite vývojový modul ADSP BF-533 EZ-KIT Lite a terminálový program Hyperterminal na počítači PC. Na obr.8.3 je zobrazené obvodové riešenie RS232 rozhrania implementované v module EZ-KIT Lite. Na prepojenie z počítačom je potrebné použiť ***priamy (tzv. patch) RS232 kábel*** *a nie bežne používaný krížený RS232 kábel.*



Obr.8.3 Obvodová realizácia RS232 rozhrania na doske ADSP BF-533 EZ-KIT Lite

8.2.3 LADENIE PERIFÉRIÍ V PROSTREDÍ SIMULÁTORA VISUALDSP++

Pri ladení programov je často výhodné (minimálne v počiatočných fázach vývoja) pracovať v **simulátore**[6]. Prostredie VisualDSP++ poskytuje výkonný ladiaci mechanizmus s využitím tzv. **streams**, ktorý umožňuje presmerovať do/zo súborov nie len štandardné výstupy/vstupy knižnice stdio.h, ale aj **výstupy/vstupy periférnych obvodov a pamätí procesora Blackfin**. Tento mechanizmus tak umožňuje realizáciu aj pomerne náročných simulácií[7].

Zápis resp. načítavanie do/zo súborov je prístupný v menu
Settings\Streams

Po výbere simulovanej periférie alebo pamäte je možné zvoliť pripojený súbor a formát dát s ktorými simulátor bude pracovať (celočíselný, zlomkový, hexadecimálny, ...).

Príklad
Overte možnosť využitia „streams" pri simulácii presmerovania výstupu a vstupu funkcií printf() a scanf() na rozhranie UART z predchádzajúceho príkladu.

8.3 AUDIO ROZHRANIE

Typickým príkladom, ktorým je možné demonštrovať základný systém ČSS pre spracovanie analógových signálov, je vstup a výstup analógových dát s využitím AD

[6] Prostredie VisualDSP++ poskytuje najlepšiu podporu v klasickom simulátore. Predkompilovaná simulácia má obmedzené možnosti. Pri práci s vývojovým modulom nie je tento mechanizmus dostupný.

[7] Simulácia však môže byť zdĺhavá, čo bude demonštrované v nasledujúcom príklade.

a DA prevodníkov. Vývojový modul **ADSP BF533 EZ-KIT Lite**, ktorý je osadený multikanálovým 96-kHz kodekom AD1836. AD1836 je jednočipový kodek, ktorý podporuje 3 stereo DA kanály a 2 stereo AD kanály, pričom využíva multibitovú sigma-delta technológiu prevodu. Kodek využíva SPI rozhranie na konfiguráciu interných registrov AD1836 (napr. nastavenie frekvencie vzorkovania, módu činnosti, zisk, a pod.). Kodek je pripojený k DSP pomocou rozhrania SPORT0 procesora BF533. Procesor môže komunikovať s audio kodekom v časovom multiplexe (**TDM** – Time Division Multiplex) alebo v dvoj-vodičovom móde (**TWI** – Two Wire Interface, tiež označovaný ako **IS mód**).

TWI mód umožňuje kodeku pracovať so vzorkovacou frekvenciou až 96 kHz, umožňuje však využitie len dvoch DA kanálov. TDM mód podporuje činnosť kodeku len do 48 kHz, umožňuje však využitie všetkých troch DA výstupov.

8.3.1 DEMONŠTRAČNÝ PRÍKLAD PRE PRÍSTUP K AD A DA KODEKOM

Inicializácia kodeku AD1836 vyžaduje detailnú znalosť registrov samotného kodeku ako aj procesora ADSP BF533. Tieto informácie presahujú rámec programov preberaných na cvičeniach a je ich možné nájsť v príslušných manuáloch.

V distribúcii prostredia VisualDSP++ je možné nájsť pripravené projekty pre prácu s perifémymi obvodmi modulu ADSP BF533 EZ-KIT Lite. Príklady pre prácu s audio kodekom AD1836 sú v adresári

..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Audio Codec Talkthrough...

ktorý obsahuje projekty pre TDM a IS módy v ASM aj jazyku C.

Uvedené projekty v jazyku C najskôr inicializujú použité periférie:

EEBIU	- pripojenie externých pamätí
FLASH	- konfigurácia IO vývodov (pre všeobecné použitie) FLASH pamäte
SPORT0	- sériový port procesora
DMA	- použité DMA kanály

následne konfigurujú prerušovací systém procesora a povolia prenos z/do rozhrania SPORT0 pomocou DMA kanálov, čo je realizované v tele funkcie main():

```
//--------------------------------------------------------------------------------------------------//
// 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)
{

        sysreg_write(reg_SYSCFG, 0x32);       //Initialize System Configuration Register
        Init_EBIU();
        Init_Flash();
        Init1836();
        Init_Sport0();
        Init_DMA();
        Init_Sport_Interrupts();
        Enable_DMA_Sport0();

        while(1);
}
```

Následne hlavný program čaká v nekonečnej slučke a systém prerušení realizuje spracovanie vstupných a výstupných dát pomocou funkcie (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 = iChannel0LeftIn;
        iChannel0RightOut = iChannel0RightIn;
        iChannel1LeftOut = iChannel1LeftIn;
        iChannel1RightOut = iChannel1RightIn;
}
```

Najjednoduchšie projekty[8] môžu využiť uvedenú štruktúru projektu a jednoduchou modifikáciou slučky funkcie *Process_Data()* realizovať spracovanie reálnych audio dát (napr. pomocou FIT filtra z minulých cvičení).

Príklad

Pomocou osciloskopu, generátora a vývojového modulu ADSP BF533 EZ-Kit Lite overte funkčnosť funkcie Process_Data().

[8] Uvedená koncepcia spracovania vzoriek však má viaceré praktické obmedzenia. Spracovanie aktuálnej vzorky musí byť ukončené do príchodu nasledujúcej vzorky. Pri zložitejších algoritmoch tento prístup nie je výhodný a je potrebné realizovať napr. blokové spracovanie. Toto je možné realizovať napr. tak, že v prerušení sú dáta zapisované/čítané do/z vyrovnávacích pamätí. Po ich naplnení/vyprázdnení je ich naplnenie/vyprázdnenie signalizované hlavnému programu. V hlavnom programe je potom možné realizovať príslušnú blokovú operáciu. Počas jej realizácie je možné zachytávať dáta do sekundárnej vyrovnávacej pamäte (tzv. **ping-pong technika**). Využitie tejto techniky bude demonštrované v nasledujúcom pri blokovom spracovaní vzoriek AD a DA prevodníkov pomocou IIR filtra.

8.4 LADENE S VYUŽITÍM SPÄTNÉHO TELEMETRICÉHO KANÁLU

Spätný telemetrický kanál (BTC - Background Telemetric Channel) umožňuje realizovať **výmenu údajov** medzi cieľovou aplikáciou (program vykonávaný vo vývojovom module) a nadradeným počítačom **bez zastavenia procesora**. BTC tak poskytuje možnosť **monitorovania** vykonávaného programu v **reálnom čase**.

BTC využíva možnosti rozhrania JTAG a umožňuje:

- monitorovanie stavu programu bez zastavenia procesora,
- sledovanie výstupov implementovaných algoritmov v reálnom čase,
- definovanie vstupných dát pre vykonávaný program,
- záznam (streaming) dát do súboru v reálnom čase[9].

BTC je tak výkonným hardvérovým ladiacim prostriedkom, ktorý umožňuje výrazne zlepšiť ladiace možnosti pri ladení reálnych aplikácií.

8.4.1 ZAČLENENIE BTC DO DSP APLIKÁCIE

Začlenenie BTC vyžaduje nasledujúcu modifikáciu cieľovej aplikácie:

a) pridanie hlavičkového súboru **btc.h** do zdrojového kódu,
b) definovanie jedného alebo viacerých BTC kanálov v zdrojovom kóde,
c) definovanie testovacej slučky (tzv. BTC polling loop) pomocou funkcie *btc_poll()*,
d) inicializácia BTC pomocou funkcie btc_init(),
e) pridanie BTC knižnice *libbtc532.dlb* alebo *libbtc535.dlb* do projektu.

Súčasťou prostredia VisualDSP++ sú projekty demonštrujúce možnosti ladenia pomocou BTC. BTC projekty sú v adresári

..\VisualDSP 5.1.2\Blackfin\Examples\ADSP_BF533 …\Background…\

Detailný opis dvoch projektov je v uvedený v úvodnom manuále prostredia VisualDSP++. Začlenenie BTC kanálov do aplikácie vyžaduje už určité podrobnejšie informácie o hardverových nastaveniach cieľovej dosky. Na druhej strane BTC umožňuje pri vývoji analyzovať údaje priamo z testovanej aplikácie, čo bolo v minulosti nerealizovateľné.

Príklad
Analyzujte začlenenie BTC kanálu v demonštračných aplikáciách v systéme VisualDSP++.

[9] Samozrejme rýchlosť prenosu je závislá na technických možnostiach BTC. V prípade požiadaviek na väčšie prenosové rýchlosti je potrebné využiť rýchly externý JTAG adaptér.

9 IMPLEMENTÁCIA IIR FILTRA POMOCOU PROCESORA ADSP BLACKFIN

IIR filter je, vzhľadom na prítomnosť spätnej väzby, **zložitejšia štruktúra** ako FIR filter. Z pohľadu implementácie je výhodné prenosovú funkciu IIR filtra **rozložiť** na jednoduchšie sekcie druhého rádu - **bikvady**. Všeobecný bikvad je charakterizovaný pomocou **5 koeficientov** b_0, b_1, b_2, a_1, a_2. Prakticky všetky dostupné DSP sú optimalizované aj pre implementáciu IIR bikvadu a procesory s jadrom Blackfin firmy Analog Devices nie sú v tomto smere výnimkou. Podobne ako kódy pre implementáciu FIR filtra aj kódy pre IIR filter efektívne využívajú základné vlastnosti duálnej harvardskej architektúry, modulo adresovanie a využitie MAC inštrukcie.

Na knižničných funkciách, ktoré sú súčasťou VisualDSP++ knižnice bude demonštrované, že dostupné štandardné knižničné funkcie nemusia vyhovovať pre mnohé praktické aplikácie.

Bude tiež demonštrovaná vysoko optimalizovaná externá knižničná funkcia, ktorá umožňuje pri blokovom spracovaní dosiahnuť rýchlosť spracovania blížiacu sa 2,5 cyklom/bikvad, čo je teoretické minimum pre DSP s duálnymi MAC jednotkami (a teda aj DSP s jadrom Blackfin). Uvedená knižničná funkcia naviac eliminuje obmedzenia IIR funkcií knižnice VisualDSP++.

9.1 KNIŽNIČNÉ FUNKCIE PROSTREDIA VISUALDSP++ PRE IIR FILTRÁCIU

Existuje niekoľko verzií implementácie IIR filtrov, ktoré sa líšia štruktúrou zapojenia a počtom koeficientov. Z teórie ČSS je známe, že v praktických implementáciách IIR filtrov je výhodné využívať realizácie IIR filtrov pomocou kaskádného zapojenia sekcií 2 rádu – bikvadov. Prenosové funkcie bikvadov získame z prenosovej funkcie IIR filtra

$$H_{IIR}(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} \tag{9.1}$$

rozkladom do tvaru (pre jednoduchosť predpokladáme, že N je párne)

$$H_{IIR}(z) = \prod_{k=1}^{N/2} \left(\frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}} \right) = \prod_{k=1}^{N/2} H_k(z) \tag{9.2}$$

pričom na implementáciu kompletnej prenosovej funkcie $H_{IIR}(z)$ je využité kaskádne zapojenie $N/2$ bikvadov. Takýto rozklad veľmi často realizujú automaticky aj programy pre praktický návrh filtrov. Rozklad prenosovej funkcie na bikvady je veľmi

výhodný práve pre DSP, pretože umožňuje jednoduchým spôsobom eliminovať problémy reprezentácie koeficientov filtra v zlomkovom formáte. Z predchádzajúcich častí vieme, že prakticky využívané koeficienty bikvadu a_{jk} môžu byť z intervalu

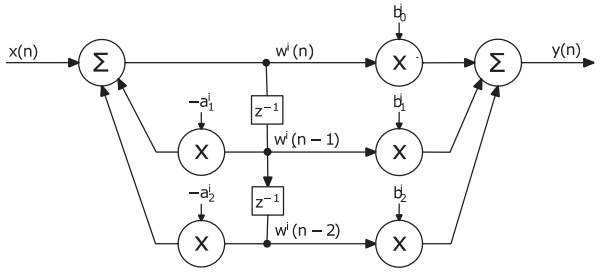
$$\left|a_{jk}\right| < 2.0 \qquad j = 1,2 \quad k = 1,2,\ldots,N/2 \tag{9.3}$$

a tak tieto koeficienty musia byť v prípade DSP s pevnou rádovou čiarkou (pretože interval čísel je u typických DSP obmedzený na interval $\langle -1,1)$) uložené so zmenenou mierkou. V prípade bikvadov stačí všetky koeficienty a_{jk} vydeliť mierkovou konštantou 2. Po zmene mierky koeficientov je samozrejme potrebné zmeniť zodpovedajúcim spôsobom aj zdrojový kód bikvadu.

V DSP knižnici prostredia VisualDSP++ existujú dve funkcie na implementáciu IIR filtra:

```
void iir_fr16( const fract16 x[], fract16 y[], int n, iir_state_fr16 *s)
```

(VisualDSP++ 4.5 C/C++ Compiler and Library Manual for Blackfin Processors. Analog Devices, Inc., April 2006, str. 4-127), ktorá realizuje implementáciu pomocou bikvadov (9.2)[1]. Táto realizácia využíva priamu formu II (DF II) realizácie bikvadu[2], ktorá je znázornená na obr.9.1.



Obr.9.1 Priama forma II realizácie bikvadu využívaná vo funkcii iir_fr16()

Funkcia je optimalizovaná z pohľadu rýchlosti tak, že spracováva paralelne pomocou duálnej MAC architektúry 2 vstupné vzorky a dosahuje rýchlosť ~ 3 cykly/vzoru, čo demonštruje pomerne efektívnu implementáciu[3] tejto funkcie. Významným praktickým obmedzením tejto knižničnej funkcie je skutočnosť, že **koeficienty prenosovej funkcie sú neškálované a teda musia byť z intervalu** $\langle -1,1)$! Množina realizovateľných IIR filtrov je tak výrazne obmedzená a nezahrňuje mnoho praktických IIR filtrov.

Druhou knižničnou funkciou je

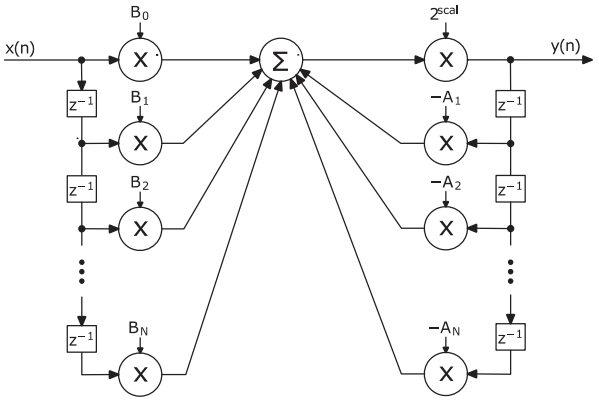
[1] Podľa manuálu funkcia implementuje bikvady so zápornými znamienkami koeficientov v menovateli prenosovej funkcie bikvadov. V komentároch zdrojového kódu sú znamienka kladné ...

[2] Táto realizácie je tzv. kanonickou realizáciou, pretože obsahuje minimálny počet stavebných blokov (predovšetkým oneskorovacích liniek).

[3] Zdrojové kódy všetkých knižničných funkcií sú v **...\Blackfin\lib\src\libdsp**

```
void iirdf1_fr16(const fract16 x[],  frac16 y[],  int n, iirdf1_fr16_state *s)
```

(C/C++ Compiler and Library Manual ..., str. 4-129), ktorá realizuje IIR na základe prenosovej funkcie (9.1). Táto realizácia využíva priamu formu I (DF I) realizácie IIR filtra, ktorá je znázornená na obr.9.2.



Obr.9.2 Priama forma I realizácie IIR filtra využívaná vo funkcii iirdf1_fr16()

Funkcia využíva koeficienty so zmenenou mierkou v tvare $A_1 = a_1 / S$, $\ldots B_N = b_N / S$, pričom vhodne zvolený škálovací faktor S zabezpečuje, že žiadny koeficient nemá hodnotu mimo zlomkového intervalu $\langle -1,1)$. Konverziu koeficientov do vhodnej formy realizuje funkcia

```
void coeff_iirdf1_fr16( const float acoeff[], const float bcoef[], fract16 coeff[], int nstages)
```

(C/C++ Compiler and Library Manual ..., str.4-79).

Funkcia dosahuje rýchlosť ~ 2 cykly/vzorku. Aj keď uvedená hodnota naznačuje veľkú efektivitu, je všeobecne známe, že praktické realizácie IIR filtrov pomocou priamej formy bez rozkladu **sú v aritmetike s konečnou presnosťou náchylné k nestabilite**. V praktických aplikáciách je tak ich využitie značne problematické[4]. Nasledujúci m-súbor **stabdemo.m** demonštruje uvedené vlastnosti. Na jeho spustenie je potrebné mať nainštalované potrebné toolboxy, ktoré sú nainštalované na počítačoch v laboratóriu.

[4] Simulácia **stabdemo.m**, ktorá je súčasťou podkladov k cvičeniu obsahuje demonštráciu nestabilitu priamej realizácie IIR filtra 20-teho rádu aj napriek 64-bitovej presnosti prostredia Matlab. Zároveň ukazuje, že realizácia uvedeného filtra pomocou 10 sekcií bikvadov realizuje filtráciu bez problémov.

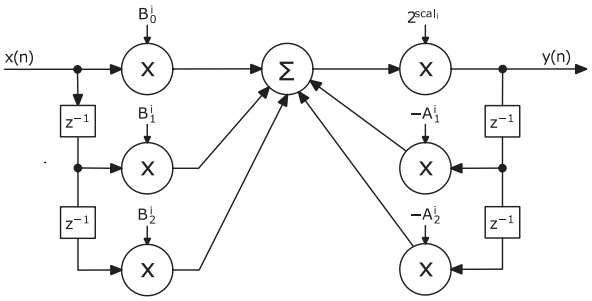
```
% This m-file demonstrates high sensitivity of direct IIR realization (of
% relatively complex IIR filter). The direct realization is instable also
% in double-precission Matlab environment.
% IIR filter realization in the form of casacde of biquad sections is
% stable.

% M.Drutarovsky., KEMT FEI TU Kosice, 03-04-2006

% design of a complex IIR filter (A,B are polynomials of 20-th order)
[b,a,v,u,C]=iirdes('ell','p',[0.19 0.2 0.25 0.26]*pi,0.01,0.00001);

t=(1:10000);                          % discrete time
fsig=0.22*pi;                         % frequency from the passband of IIR
x=1.0*sin(2*pi*fsig*t/(2*pi));        % input signal with fsig frequency

% output of direct IIR filter realization
y=filter(b,a,x);                      % output of direct IIR filter realization
figure(1)
plot(y);
title('Instability of direct IIR filter realization');


% output of (biquads) cascade realization
[nsec,dsec]=pairpz(v,u);
yc=cascade(C,nsec,dsec,x);
figure(2);
plot(yc);
title('Stable output of (biquad) cascade realization');
```

9.2 OPTIMALIZOVANÁ KNIŽNIČNÁ FUNKCIA PRE IIR FILTRÁCIU

Uvedené nedostatky odstraňuje knižničná funkcia pre IIR filtráciu pomocou procesorov Blackfin

```
void iir2_fr16( const fract16 x[], fract16 y[], int n, iir_state_fr16 *s)
```

vytvorená autorom tohto dokumentu na KEMT FEI TU v Košiciach. Táto funkcia realizuje implementáciu pomocou bikvadov (9.2) a využíva nekanonickú formu[5] realizácie bikvadu, ktorá je znázornená na obr.9.3.



Obr.9.3 Nekanonická realizácia bikvadu využívaná vo funkcii iir2_fr16()

Funkcia využíva koeficienty so zmenenou mierkou v tvare $A_1 = a_1 / S$, $\ldots B_N = b_N / S$, pričom pevne zvolený škálovací faktor $S = 2$ zabezpečuje, že žiadny koeficient nemá hodnotu mimo zlomkového intervalu $\langle -1,1)$.

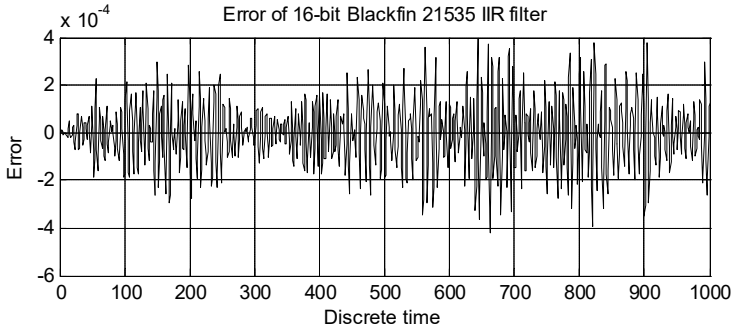
[5] Z pohľadu reálnych implementácií väčší počet oneskorovacích liniek oproti kanonickým realizáciám nie je praktickým obmedzením. Oneskorovacie linky na výstupe bikvadu môžu byť zdieľané ako vstupné oneskorovacie linky nasledujúceho bikvadu, čím sa rozdiel v počte stavebných prvkov pri komplikovanejších IIR filtroch výrazne zmenšuje.

Funkcia je optimalizovaná z pohľadu rýchlosti tak, že spracováva paralelne pomocou duálnej MAC architektúry 2 vstupné vzorky a dosahuje rýchlosť[6] ~ 2,5 cyklu/vzorku, čo demonštruje mimoriadnu efektivitu zdrojového kódu, takto optimalizovaným jadrom knižničnej funkcie:

step	accum. A0	accum. A1	register contents				pointer position
0.		b_2*x_{-2} +b_1*x_{-1}	**R0:** b_1 **R2:** x_1 x_0	**R1:** b_2 -a_2 **R3:** x_{-1} y_{-2}	**R4:** **R6:**	**R5:** **R7:**	I1 → x_{-1} I2 → b_0
1.	b_2*x_{-1}	-a_2*y_{-2}	**R0:** b_1 b_0 **R2:** [x_1 x_0]$_{->x-1\ x-2}$	**R1:** b_2 -a_2 **R3:** x_{-1} y_{-2}	**R4:** **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes incr. I1 by 4 bytes
2.	+b_1*x_0	+b_0*x_0	**R0:** b_1 b_0 **R2:** x_1 y_{-1}	**R1:** -a_1 -a_2 **R3:** x_{-1} y_{-2}	**R4:** **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes
3.	-a_2*y_{-1}	[-a_1*y_{-1}]$_{s2rnd}$	**R0:** b_{1n} b_0 **R2:** x_1 y_{-1}	**R1:** -a_1 -a_2 **R3:** x_{-1} y_{-2}	**R4:** y_{-2} [y_0] **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes
4.	+b_0*x_1	b_{1n}*x_{-1n}	**R0:** b_{1n} b_0 **R2:** x_1 y_{-1}	**R1:** -a_1 b_{2n} **R3:** y_{-1n} y_{-2n}	**R4:** y_{-2n} y_0 **R6:**	**R5:** **R7:**	decr. I1 by 4 bytes incr. I2 by 2 bytes
5.	[-a_1*y_0]$_{s2rnd}$	+b_{2n}*x_{-2n}	**R0:** b_{1n} b_0 **R2:** x_1 y_{-1}	**R1:** -a_{2n} b_{2n} **R3:** x_{-1n} y_{-2n}	**R4:** [y_1] y_0 **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes I1 → x_{-1n} I2 → b_{0n}
6.	b_2*x_{-1}	-a_2*y_{-2}	**R0:** b_1 b_0 **R2:**	**R1:** -a_2 b_2 **R3:** x_{-1} y_{-2}	**R4:** [x_1 x_0]$_{->x-1\ x-2}$ **R5:** **R6:** **R7:**		incr. I2 by 2 bytes incr. I1 by 4 bytes
7.	+b_1*x_0	+b_0*x_0	**R0:** b_1 b_0 **R2:**	**R1:** -a_2 -a_1 **R3:** x_{-1} y_{-2}	**R4:** x_1 y_{-1} **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 byets
8.	-a_2*y_{-1}	[-a_1*y_{-1}]$_{s2rnd}$	**R0:** b_{1n} b_0 **R2:** y_{-2} [y_0]	**R1:** -a_2 -a_1 **R3:** x_{-1} y_{-2}	**R4:** x_1 y_{-1} **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes-
9.	+b_0*x_1	b_{1n}*x_{-1n}	**R0:** b_{1n} b_0 **R2:** y_{-2} y_0	**R1:** b_{2n} -a_1 **R3:** y_{-1n} y_{-2n}	**R4:** **R6:**	**R5:** **R7:**	decr. I1 by 4 bytes incr. I2 by 2 bytes
10.	[-a_1*y_0]$_{s2rnd}$	+b_{2n}*x_{-2n}	**R0:** b_{1n} b_0 **R2:** [y_1] y_0	**R1:** b_{2n} -a_{2n} **R3:** x_{-1n} y_{-2n}	**R4:** **R6:**	**R5:** **R7:**	incr. I2 by 2 bytes I1 → x_{-1nn} I2 → b_{0nn}

Knižničná funkcia bola testovaná na vývojových doskách procesorov Blackfin BF535 ako aj BF533 (ktoré využívajú odlišné zreťazenie v riadiacej a dátových jednotkách) a v oboch prípadoch boli dosiahnuté identické výsledky. Na obr.9.4 sú znázornené výsledky testovania presnosti eliptického IIR filtra 8-rádu pri spracovaní 16-bitových vstupných údajov pomocou procesora ADSP Blackfin BF535. Ako referenčná implementácia bola použitá implementácia kaskády bikvadov pomocou Matlabu.

[6] Funkcia dosahuje tieto hodnoty asymptoticky, t.j. len pre IIR filtre veľkého rádu a veľké bloky spracovávaných vzoriek. Je to priamy dôsledok blokového charakteru všetkých analyzovaných knižničných funkcií pre IIR filtráciu. Keďže uvedená IIR funkcia využíva extrémne optimalizované jadro, ktoré v jednom prechode spracováva 2 vstupné vzorky vo 2 za sebou nasledujúcich bikvadoch, musí byť rád IIR filtra párny. Pri zložitejších IIR filtroch nie je táto podmienka žiadnym obmedzením, pretože nepárny rád môže byť vždy doplnený posledným bikvadom s jednotkovým prenosom.



Obr.9.4 Chyba výpočtu funkcie iir2_fr16() v porovnaní s referenčným výpočtom v Matlabe)

9.3 BLOKOVÉ SPRACOVANIE VZORIEK Z AD A DA PREVODNÍKOV

V jednom z demonštračných príkladov VisualDSP++ pre spracovanie vzoriek z AD a DA prevodníkov vývojovej dosky ADSP BF533 EZ-KIT Lite bolo realizované spracovanie vzoriek z prevodníkov tzv. **rekurzívnym spôsobom** (t.j. spracovanie vstupnej vzorky muselo byť ukončené do príchodu nasledujúcej vzorky toho istého kanálu). Spracovanie bolo realizované v prerušení pomocou funkcie Proces_Data():

```
#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 = iChannel0LeftIn;
        iChannel0RightOut = iChannel0RightIn;
        iChannel1LeftOut = iChannel1LeftIn;
        iChannel1RightOut = iChannel1RightIn;
}
```

Nevýhoda rekurzívneho spracovania sa plne prejaví pri pokuse využiť dostupné knižničné funkcie pre IIR filtráciu, ktoré sú optimalizované pre blokové spracovanie (t.j. dĺžka bloku by mala byť značne väčšia ako 1). Naviac pre dosiahnutie efektívnosti je výhodné, aby blok spracovávaných vzoriek obsahoval párny počet vzoriek.

Jedným zo spôsobov, ako je možné uvedený problém pomerne jednoducho vyriešiť[7] je využitie tzv. **„ping-pong"** metódy, ktorá pre každý kanál[8] využíva 2 bufre. Pre smer AD -> bufre sú to nasledujúce polia short (int16) čísel:

[7] Pri uvedenej metóde je možné využiť HW konfiguráciu kodeku AD1836 a procesora z rekurzívneho spracovania AD a DA vzoriek. Blokové spracovanie je dosiahnuté len pridaním vhodných bufrov, a ich „softvérovo" riadeným ovládaním. Efektívnejším, avšak podstatne komplikovanejším riešením, by bolo priame preprogramovanie AD1836, prerušovacieho systému BF533 a DMA kanálov.

```
#pragma align 4
short Left_In0_A[BLOCK_SIZE];
#pragma align 4
short Left_In1_A[BLOCK_SIZE];
#pragma align 4
short Left_In0_B[BLOCK_SIZE];
#pragma align 4
short Left_In1_B[BLOCK_SIZE];
#pragma align 4
short Right_In0_A[BLOCK_SIZE];
#pragma align 4
short Right_In1_A[BLOCK_SIZE];
#pragma align 4
short Right_In0_B[BLOCK_SIZE];
#pragma align 4
short Right_In1_B[BLOCK_SIZE];
```

Pre smer bufre->DA to sú polia[9]:

```
#pragma align 4
short Left_Out0_A[BLOCK_SIZE];
#pragma align 4
short Left_Out1_A[BLOCK_SIZE];
#pragma align 4
short Left_Out0_B[BLOCK_SIZE];
#pragma align 4
short Left_Out1_B[BLOCK_SIZE];
#pragma align 4
short Right_Out0_A[BLOCK_SIZE];
#pragma align 4
short Right_Out1_A[BLOCK_SIZE];
#pragma align 4
short Right_Out0_B[BLOCK_SIZE];
#pragma align 4
short Right_Out1_B[BLOCK_SIZE];
```

Interné riadenie počítadla pre zápis resp. vyčítavanie bufrov v prerušení je odvodené od premenných (kedže sú typu static, nie sú viditeľné mimo zdrojového kódu codec_buffers.c):

```
static int cnt=0;                   // pocitadlo prijatych vzoriek
static int Ping_Pong = 0;           // 0  => from AD to A, from B to DA
                                    // !0 => from AD to B, from A to DA
```

Signalizácia naplnenia AD bufrov resp. vyprázdnenie DA bufrov je signalizovaná hlavnému programu pomocou premennej (jedinej premennej, ktorá musí byť typu volatile):

```
volatile int New_Blocks_Received;   // 1 => AD to A_Blocks,
                                    // 2 => AD to B_Blocks
```

Spracovanie AD vzoriek z ľavého a pravého kanálu vstupu 0 je realizované v nekonečnej slučke hlavného programu v závislosti na tom, ktorý vstupný bufer je signalizovaný ako plný:

[8] Dvojica bufrov je použitá pre každý AD kanál, ako aj pre každý DA kanál. Vhodnejším kódovaním by bolo možné vysielacie a prijímacie bufre zlúčiť, z dôvodu zachovania väčšej prehľadnosti kódu zlúčenie nebolo využité.

[9] Vzorky z AD prichádzajú ako 24-bitové a ako 24-bitové sú vysielané do DA prevodníkov. Keďže použitá IIR funkcia je optimalizovaná pre spracovanie 16-bitových vzoriek, sú vzorky v bufroch uložené ako 16-bitové (short integer).

```
while(1) {
    if(New_Blocks_Received==1) {    // FINISHED ->from AD to A, from A to DA
        iir2_fr16(Left_In0_A, Left_Out0_A, BLOCK_SIZE, &state_left);
        iir2_fr16(Right_In0_A, Right_Out0_A, BLOCK_SIZE, &state_right);
        memcpy(Left_Out1_A, Left_In1_A, 2*BLOCK_SIZE);
        memcpy(Right_Out1_A, Right_In1_A, 2*BLOCK_SIZE);
        New_Blocks_Received = 0;    // signalizacia spracovania bloku
    }
    if(New_Blocks_Received==2) {    // FINISHED ->from AD to B, from A to DA
        iir2_fr16(Left_In0_B, Left_Out0_B,BLOCK_SIZE, &state_left);
        iir2_fr16(Right_In0_B,Right_Out0_B, BLOCK_SIZE, &state_right);
        memcpy(Left_Out1_B,Left_In1_B,2*BLOCK_SIZE);
        memcpy(Right_Out1_B,Right_In1_B,2*BLOCK_SIZE);
        New_Blocks_Received = 0;    // signalizacia spracovania bloku
    }
}
```

Príklad

Analyzujte kompletné zdrojové kódy priloženého projektu blokovej IIR filtrácie a identifikujte

- *umiestnenie začiatku bufrov na adresy s násobkom 4,*
- *kompresiu 24-bitových vzoriek na 16-bitové pri čítaní dát z AD prevodníkov,*
- *dekompresiu 16-bitových vzoriek na 24-bitové pri zápise dát do DA prevodníkov,*
- *umiestnenie (+ inicializáciu) koeficientov a stavových premenných do dátových pamätí L1 z dôvodu dosiahnutia maximálnej rýchlosti,*
- *filtráciu v main.c,*
- *ďalšie zmeny oproti projektu z predchádzajúceho cvičenia úpravou ktorého vznikol projekt blokového spracovania,*
- *spôsob využitia[10] externej knižnice* ***iir2lib.dlb****.*

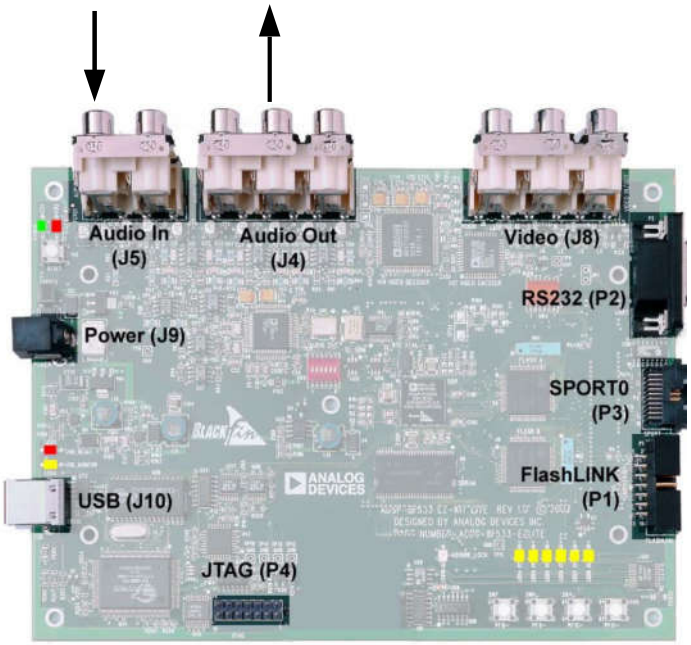
Príklad

Pomocou osciloskopu a preladiteľného generátora odmerajte pásmo priepustnosti IIR filtra v priloženom projekte. Zapojenie audio konektorov J5 a J4 pre tento príklad je znázornené na obr.9.5. Generátor je potrebné zapojiť do druhého vstupu konektora J5 a vstup osciloskopu je potrebné prepojiť s prvým výstupom konektora J4.

Príklad

Navrhnite vlastný IIR filter a upravte zdrojový kód priloženého projektu tak, aby realizoval navrhnutý filter. Overte správnosť pomocou osciloskopu a preladiteľného generátora.

[10] Názov funkcie v externej knižnici **bol zámerne zmenený na iir2_fr16()**, ktorý je odlišný od názvu originálnej funkcie iir_fr16(). Názov by však mohol byť aj zhodný s názvom pôvodnej funkcie iir_fr16() v štandardnej DSP knižnici VisualDSP++. Zahrnutím zdrojového kódu (resp. knižnice) s rovnakým menom ako má funkcia v štandardnej knižnici priamo do projektu je možné nahradiť štandardné funkcie novými s identickým menom.



Obr.9.5 Zapojenie vstupých a výstupných konektorov mdulu ADSP BF533 EZ-KIT Lite využitých v kódoch predpripravených testovacích aplikácií

Dostupné knižničné funkcie pre IIR filtráciu pre procesor Blackfin, ktoré sú súčasťou prostredia VisualDSP++ sú jasným príkladom, že tvorbu knižničných funkcií je potrebné realizovať aj so zohľadnením teórie ČSS a čisto „programátorska" implementácia môže mať vážne praktické nedostatky.

10 ALGORITMUS FFT

Diskrétna Fourierova transformácia (DFT) je jednou z najdôležitejších operácií v číslicovom spracovaní signálov. Základným predpokladom využívania DFT je znalosť jej vlastností a efektívnych algoritmov pre jej výpočet. DFT je pre komplexnú[1] postupnosť $x[n]$ konečnej dĺžky N (t.j. $x[n]=0$ pre $n<0$ a $n\geq N$) definovaná vzťahom

$$X[k]=\sum_{n=0}^{N-1}x[n]e^{-j\frac{2\pi kn}{N}} \qquad k=0,1,\ldots,N-1 \tag{10.1}$$

Pre inverznú DFT (IDFT) platí vzťah

$$x[n]=\frac{1}{N}\sum_{k=0}^{N-1}X[k]e^{j\frac{2\pi kn}{N}} \qquad n=0,1,\ldots,N-1 \tag{10.2}$$

Rýchla Fourierova transformácia (FFT – Fast Fourier Transform) je typickým predstaviteľom rýchleho algoritmu pre výpočet DFT. Algoritmus FFT sa využíva predovšetkým na výpočet spektra signálu, prípadne na rýchly výpočet konvolúcie vo frekvenčnej oblasti. Keďže spektrum signálu je takmer vždy komplexné, je často potrebné využívať na zobrazenie absolútne hodnoty komplexného čísla (funkcia **abs**). Na výpočet FFT v Matlabe sa využíva príkaz **fft**. Na výpočet inverznej FFT (IFFT) je možné využiť príkaz **ifft**.

Príklad
Zobrazte spektrum signálu, ktorý dostaneme ako súčet dvoch sínusoviek s frekvenciami 1500 *Hz a* 4000 *Hz a amplitúdami 3.3, pričom frekvencia vzorkovania je* 48000 *Hz. Vysvetlite, prečo majú amplitúdy sínusoviek vo frekvenčnej oblasti po výpočte FFT odlišnú veľkosť.*

Riešenie

```
n=0:1023;                                                    % diskretny cas
x=3.3*sin(2*pi*n*1500/48000)+3.3*sin(2*pi*n*4000/48000);
y=fft(x);                                                    % urcenie spektra
plot(abs(y))                                                 % absolutna hodnota komplexneho vektora
```

Príklad
Zobrazte spektrum signálu, na výstupe filtrov navrhnutých v predchádzajúcich príkladoch pre vstupný signál ktorý dostaneme ako súčet dvoch sínusoviek

[1] Samozrejme toto je najvšeobecnejší prípad, vstupná postupnosť môže byť aj rýdzo reálna.

s frekvenciami 1500 *Hz a* 4000 *Hz a amplitúdami 1.0, pričom frekvencia vzorkovania je* 48000 *Hz. Na výpočet výstupu filtra využite funkciu* ***filter****.*

Riešenie

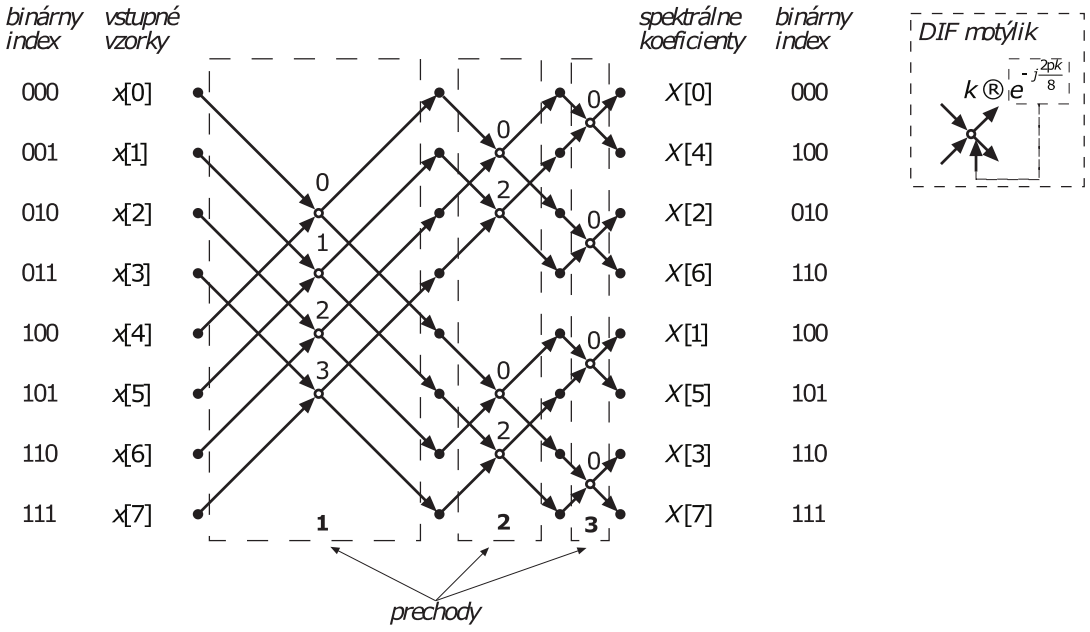
```
n=0:1023;                                                    % diskretny cas
x=1.0*sin(2*pi*n*1500/48000)+1.0*sin(2*pi*n*4000/48000);     % generovanie vstupneho signalu
y=filter(b,a,x);                                             % filtracia navrhnutym IIR filtrom
% y=filter(h,1,x);                                           % filtracia navrhnutym FIR filtrom
plot(abs(fft(x)));                                           % zobrazenie spectra vstupneho
                                                             % signalu
plot(y);                                                     % zobrzenie vystupneho signalu
plot(abs(fft(y)));                                           % zobrazenie spectra vystupneho
                                                             % signalu
```

10.1 ZÁKLADNÁ ŠTRUKTÚRA ALGORITMU FFT

Algoritmy FFT a IFFT sú rýchlostne optimalizované algoritmy výpočtu DFT a IDFT. Existuje množstvo algoritmov FFT, z ktorých sa v oblasti DSP využívajú (z dôvodu podpory špeciálnych adresových režimov adresových aritmetických jednotiek) predovšetkým FFT a IFFT pre

$$N = 2^z \qquad z \in \{1, 2, 3\ldots\} \tag{10.3}$$

ktoré využívajú známy rozklad FFT rozmeru N na dve FFT rozmeru $N/2$ a jeho rekurzívne opakovanie až do $N = 4$ alebo $N = 2$ a realizáciu FFT s rozmerom 4 resp. 2 pomocou tzv. RADIX 4 resp. RADIX 2 motýlikov. Výber vhodnej formy motýlika (RADIX 2 alebo RADIX 4) závisí predovšetkým na štruktúre dátových ciest konkrétneho DSP a tiež na polohe motýlika v procese výpočtu. Na obr. 10.1 je znázornený rozklad FFT pre $N = 8$, ktorý sa skladá z troch *prechodov*.

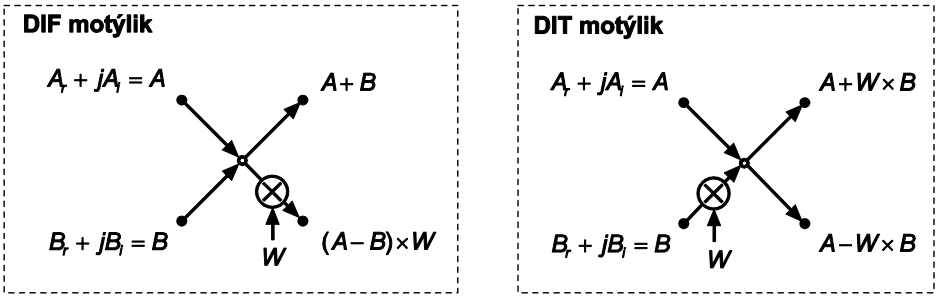


Obr. 10.1 Rozklad FFT s $N = 8$ *na jednotlivé prechody*

Veľmi často napr. rýchlostne optimalizované verzie výpočtu FFT využívajú v prvom prechode RADIX 4 motýliky a v ďalších prechodoch RADIX 2 motýliky.

Výpočty FFT a IFFT sa z hľadiska štruktúry algoritmu líšia len nepatrne a pri všeobecnej analýze algoritmov sa zvyčajne uvažuje len jeden z nich (FFT). Tento prístup je využitý aj v ďalšom opise, niektoré špecifické vlastnosti IFFT sú uvedené v nasledujúcej podkapitole.

V procese výpočtu FFT pomocou uvedených algoritmov dochádza k preusporiadaniu výstupných hodnôt do bitovo-reverzného usporiadania (tzv. DIF – decimácia vo frekvenčnej oblasti), čo je možné kompenzovať preusporiadaním v časovej oblasti (tzv. DIT – decimácia v časovej oblasti). Tieto všeobecne známe formy výpočtu FFT využívajú odlišné motýliky zobrazené na obr. 10.2. Vzhľadom na inštrukčnú sadu DSP nie sú tieto motýliky rovnocenné a napr. DSP5600x umožňuje realizovať jadro DIT motýlika pomocou 6 inštrukcií a jadro DIF motýlika pomocou 7 inštrukcií. Táto vlastnosť naznačuje, že vhodnosť špecifickej formy implementácie FFT pre konkrétny typ DSP je potrebné analyzovať individuálne.



Obr. 10.2 DIF a DIT motýliky

10.2 OPTIMALIZÁCIE VÝPOČTU FFT A IFFT

V tejto časti budú uvedené menej známe spôsoby výpočtu FFT (IFFT), ktoré je možné využiť v reálnych aplikáciách v prípadoch keď priama implementácie GGT (IFFT) nie je dostatočne efektívna. Typickým prípadom je napr. obmedzenie veľkosti rýchlych interných dátových pamätí cieľového DSP. V takomto prípade je výhodné realizovať FFT (IFFT) len do určitej veľkosti. Tento problém bol dominantný v začiatkoch vývoja DSP, použitý princíp však môže byť využitý aj v iných špecifických aplikáciách.

ROZKLAD NA MENŠIE TRANSFORMÁCIE

Rozklad na menšie transformácie je možné realizovať aj pomocou metód, ktoré rozkladajú (jednorozmernú) transformáciu DFT s rozmerom $N = N_1 N_2$ na postupné počítanie DFT rozmerov N_1 a N_2. Na základe substitúcií

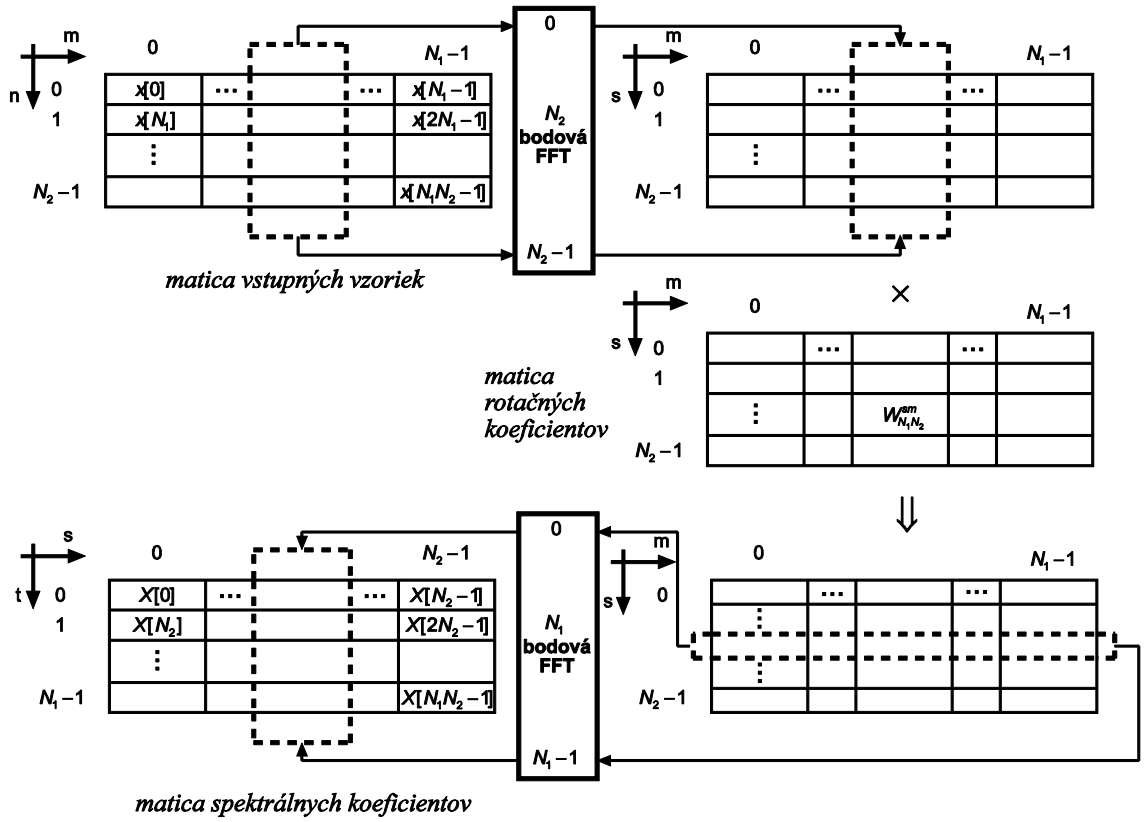
$$k = m + nN_1 \qquad m,t = 0,1,\ldots,N_1 - 1 \tag{10.4}$$

$$l = s + tN_2 \qquad n,s = 0,1,\ldots,N_2 - 1 \tag{10.5}$$

je možné prepísať výpočet DFT do tvaru

$$X[s+tN_2] = \sum_{m=0}^{N_1-1}\sum_{n=0}^{N_2-1} x[m+nN_1]W_{N_1N_2}^{(m+nN_1)(s+tN_2)} = \sum_{m=0}^{N_1-1} W_{N_1}^{tm} W_{N_1N_2}^{sm} \sum_{n=0}^{N_2-1} x[m+nN_1]W_{N_2}^{sn} \tag{10.6}$$

Výpočet DFT s rozmerom $N = N_1 N_2$ je možné podľa rovnice (10.6) interpretovať ako postupné počítanie DFT s dĺžkami N_1 a N_2, čo je znázornené na obr. 10.3 . Vstupná postupnosť $x[n]$ je uložená v matici $N_1 \times N_2$. V prvom kroku sa počíta N_1 DFT zo stĺpcov dĺžky N_2. Výsledkom je matica rovnakého rozmeru, ktorá sa v druhom kroku násobí *maticou rotačných koeficientov* $W_{N_1 N_2}^{sm}$. V treťom kroku sa vypočíta N_2 DFT dĺžky N_1 z riadkov matice z predchádzajúceho kroku. Výsledná matica obsahuje spektrálne koeficienty DFT s rozmerom $N = N_1 N_2$. Násobenie maticou rotačných koeficientov je možné presunúť aj pred výpočet DFT **Error! Reference source not found.**. V praxi je výpočet DFT rozmerov N_1 a N_2 realizovaný algoritmami FFT, pričom pri dostatočne malých rozmeroch je možné pri ich výpočte využiť interné dátové pamäte DSP. Nevýhodou je nutnosť uchovávať veľkú maticu rotačných koeficientov a zvýšené nároky na presun dát.



Obr. 10.3 Princíp využitia N_1 a N_2 rozmerných DFT na výpočet $N = N_1 N_2$ rozmernej DFT

Vhodnosť konkrétnej metódy opäť závisí na vlastnostiach architektúry cieľového DSP, efektivite práce s externou pamäťou a veľkostiach dostupných externých pamätí.

VÝPOČET REÁLNEJ FFT

Reálna FFT (RFFT) je výpočet FFT pre postupnosť $x[n]$, ktorá je čisto reálna. V literatúre je možné nájsť celý rad optimalizovaných algoritmov pre výpočet RFFT. Ich cieľom je znížiť počet niektorých operácií (napr. násobení), počet presunov, prípadne zvýšiť rýchlosť na zreťazených procesoroch. Tieto formy optimalizácie sú však z pohľadu architektúr DSP na báze harvardskej architektúry nevýhodné, v prípade VLIW architektúr by však tieto formy mohli byť zaujímavé.

Pre modifikované harvardské architektúry je výhodné využiť linearitu FFT a Hermitovskú symetriu spektra reálnej postupnosti $x[n]$ v tvare

$$X[k] = X^*[-k] = X^*[N-k] \tag{10.7}$$

pričom pre výpočet RFFT sa využívajú predovšetkým optimalizované algoritmy FFT:

a) Výpočet dvoch *N* rozmerných RFFT pomocou jednej *N* rozmernej FFT
Na základe linearity DFT platí pre reálne postupnosti $x[n]$ a $y[n]$ vzťah

$$DFT\big(z[n]\big) = Z[k] = DFT\big(x[n] + jy[n]\big) = Z_r[k] + jZ_i[k] =$$

$$= \big(X_r[k] - Y_i[k]\big) + j\big(X_i[k] + Y_r[k]\big) \tag{10.8}$$

pričom indexy *r* a *i* označujú reálnu a imaginárnu zložku. Aplikovaním vzťahu (10.7) na $Z[k]$ je možné určiť DFT pôvodných reálnych postupností

$$X[k] = X_r[k] + jX_i[k] = \frac{1}{2}\big(Z_r[k] + Z_r[N-k]\big) + \frac{1}{2}\big(Z_i[k] - Z_i[N-k]\big) \tag{10.9}$$

$$Y[k] = Y_r[k] + jY_i[k] = \frac{1}{2}\big(Z_i[N-k] + Z_i[k]\big) + j\frac{1}{2}\big(Z_r[N-k] - Z_r[k]\big) \tag{10.10}$$

pre $k = 0,1,\ldots,N/2$ [2]. Takto je možné priamo využiť optimalizované algoritmy FFT s následnou jednoduchou modifikáciou výsledkov podľa vzťahov (10.9) a (10.10).

b) Výpočet *N* rozmernej RFFT pomocou *N*/2 rozmernej FFT
Tento spôsob vychádza zo štandardného DIT rozkladu FFT algoritmu dĺžky *N* v tvare

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi nk}{N}} = \sum_{n=0}^{N/2-1} x[2n] e^{-j\frac{2\pi nk}{N/2}} + e^{-j\frac{2\pi k}{N}} \sum_{n=0}^{N/2-1} x[2n+1] e^{-j\frac{2\pi nk}{N/2}} \tag{10.11}$$

pre $k = 0,1,\ldots,N-1$, čo je možné realizovať pomocou dvoch RFFT dĺžky *N*/2 pre reálne postupnosti $\{x[2n]\}$ a $\{x[2n+1]\}$ a následnou kombináciou výsledkov podľa vzťahu (10.11). Výpočet RFFT dvoch reálnych postupností dĺžky *N*/2 je možné realizovať pomocou FFT komplexnej postupnosti

$$z[n] = x[2n] + jx[2n+1] \tag{10.12}$$

s dĺžkou N/2 pomocou vzťahov (10.9) a (10.10).

[2] Pre k=0 a k=N/2 sú hodnoty $X[k]$ (a samozrejme aj $Y[k]$) čisto reálne, čo sa často v praktických implementáciach využíva na uloženie len N/2 (pre k=*0,1,…,N/2-1*) komplexných spektrálnych hodnôt pre každú postupnosť, pričom je uložená modifikovaná komplexná hodnota $X'[0]=X[0]+jX[N/2]$ pre $X[k]$ (podobne aj pre $Y[k]$).

VÝPOČET IFFT

Rovnice pre IFFT a FFT sú veľmi podobné. Faktor $1/N$ je v praktických implementáciach nepodstatný a je ho možné zahrnúť do zmeny mierky spracovávaných signálov. Rozdiel v rotačných faktoroch

$$e^{-j\frac{2\pi kn}{N}} \quad pre\ FFT \qquad a \qquad e^{j\frac{2\pi kn}{N}} \quad pre\ IFFT \tag{10.13}$$

je možné riešiť v praktických implementáciách predovšetkým nasledujúcimi spôsobmi:

a) FFT kód s odlišnými tabuľkami
Pri tejto metóde je programový kód pre FFT a IFFT *rovnaký*, každý z nich však využíva *odlišnú tabuľku rotačných faktorov*. V prípade, že je potrebné súčasne využívať FFT aj IFFT je potrebné mať v pamäti uložené súčasne dve tabuľky. Táto metóda je výhodná, pokiaľ je programový kód FFT väčší ako veľkosť tabuľky pre rotačné faktory. Naviac tabuľky je možné generovať aj počas vykonávania programu, čo v prípade programového kódu nie je prakticky realizovateľné.

b) Modifikovaný kód s jednou tabuľkou
Táto metóda je duálnou k predchádzajúcej metóde a umožňuje využívať *spoločnú tabuľku rotačných faktorov* a rozdielne programové kódy pre FFT a IFFT. Metóda je výhodná predovšetkým v prípade, že sa využíva univerzálny programový kód pre všetky rozmery FFT a IFFT a je potrebné využívať súčasne rôzne rozmery FFT a IFFT.

c) Výmena reálnej a imaginárnej časti
Táto metóda realizuje výpočet IFFT výmenou reálnej a imaginárnej časti vstupného vektora, výpočtom FFT a opätovnou výmenou reálnej a imaginárnej časti výstupného vektora, čo je zrejmé z nasledujúcich rovníc, ktoré opisujú výpočet vo FFT motýliku (W_r a W_i sú reálna a imaginárna časť rotačných faktorov, A_r a A_i sú reálny a imaginárny vstup FFT motýlika)

$$(A_r + jA_i)(W_r + jW_i) = (A_rW_r - A_iW_i) + j(A_iW_r + A_rW_i) \tag{10.14}$$

$$(A_i + jA_r)(W_r - jW_i) = j(A_rW_r - A_iW_i) + (A_iW_r + A_rW_i) \tag{10.15}$$

Nevýhodou tejto metódy je nutnosť realizácie dvojnásobnej výmeny reálnych a imaginárnych častí N-prvkových komplexných vektorov.

d) Súvislosťou medzi FFT a IFFT
Táto metóda využíva menej známu vlastnosť FFT transformácie, ktorú je možné zapísať v tvare

$$\begin{bmatrix} x[0] \\ x[N-1] \\ x[N-2] \\ \vdots \\ x[2] \\ x[1] \end{bmatrix} = \frac{1}{N} FFT_N \left(FFT_N \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-2] \\ x[N-1] \end{bmatrix} \right) \tag{10.16}$$

čo umožňuje využiť jeden programový kód a jednu tabuľku rotačných faktorov v aplikáciách, ktoré sú z hľadiska veľkostí dostupných pamätí kritické. Nevýhodou uvedenej metódy je nutnosť preusporiadania výstupného vektora, čo môže výrazne skomplikovať vysokoúrovňové programovanie DSP. Oproti predchádzajúcej metóde stačí preusporiadanie len výstupného vektora.

11 KNIŽNIČNÁ FUNKCIA PRE VÝPOČET FFT

Algoritmus výpočtu rýchlej Fourierovej transformácie (**FFT** - Fast Fourier Transformation) patrí základné algoritmy ČSS. Knižnice pre ČSS v prostredí VisualDSP++ poskytujú pre programátora viacero verzií knižničných C funkcií pre výpočet FFT (napr. funkcie pre výpočet pre priame a spätné transformácie, FFT pre rýdzo reálne vstupné postupnosti, FFT pre komplexné postupnosti, FFT využívajúce rôzne základy – Radix 2, Radix 4, a pod.).

Napr. aj jeden z príkladov pre demonštráciu spätného telemetrického kanálu využíval výpočet reálnej FFT v nekonečnej slučke funkcie main():

```
while (1) {

        //generate input
        create_samples(freq);

        //fft
        rfft_fr16(input_arr, t, out, w, wst, n, block_exponent, scale_method);

        for (i=0; i<NUMPOINTS; i++) {
                mag[i] = sqrt(out[i].re*out[i].re+out[i].im*out[i].im);
        }

        //write to BTC channel
        btc_write_array(0, (unsigned int*)input_arr, sizeof(input_arr));
        btc_write_array(1, (unsigned int*)mag, sizeof(mag));


        freq += BTC_CHAN2;
        if (freq > MAXFREQ) freq = MINFREQ;
}
```

Kompletný projekt je možné nájsť v adresári
..\Blackfin\Examples\ ADSP-BF533 EZ-Kit Lite\Background_Telemetry\ fftDemo

Príklad[1]

*Preštudujte dokumentáciu ku knižničnej funkcii **rfft_fr16()** a na upravte projekt BTC_fft tak, aby demonštroval správnu funkčnosť knižničnej funkcie v simulátore prostredia ViusalDSP++.*

[1] Podobný typ úloh bude realizovaný v zadaniach, ktorých úlohou je demonštrovanie funkčnosti vybraných knižničných funkcií pre DSP.

ZOZNAM POUŽITEJ LITERATÚRY

[1] ADSP-BF533 Blackfin Processor Hardware Reference (Rev.3.1). Analog Devices, Inc., May 2005.

[2] ADSP-BF53x/BF56x Blackfin Processor Programming Reference (Rev.1.1). Analog Devices, Inc., February 2006.

[3] Davari, B. – Dennard, R.H. – Shahidi, G.G.: CMOS Scaling for High Performance and Low Power – The Next Ten Years. *Proceedings of the IEEE*, Vol.83, No.4, April 1995, pp.595-606.

[4] Drutarovský, M.: An Implementation of High Performance IIR Filtration on 2-MAC Blackfin DSP Architecture. Proceedings of DSP-MCOM 2005, September 13-14, 2005, Kosice, Slovakia, pp.110-113.

[5] Drutarovský, M.: Signálové procesory v číslicovom spracovaní signálov. Habilitačná praca, KEMT FEI TU Košice, 2009, pp.1-137.

[6] Duhamel, P.: Implementation of "Split – Radix" FFT Algorithms for Complex, Real, and Real - Symetric Data. *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol.34, No.2, June 1986, pp.285-295.

[7] Hlavička, J.: *Architektura počitačů*. Skriptum ČVUT, Praha 1996.

[8] Magyar, A.: *Číslicové Zpracování Signálu II: Signálové procesory a jejich použití*. Skriptum TU v Košiciach, Košice 1991.

[9] Norsworthy, S.R. – Schreier, R. – Temes, G.C.: *Delta – Sigma Data Converters: Theory, Design, and Simulation*. IEEE Press, New York, 1997.

[10] Ondráček, O.: Signály a sústavy. STU Bratislava, 2008.

[11] Pollard, J.M.: The Fast Fourier Transform in a Finite Field. *Mathematics of Computations*, Vol.25, No.114, April 1971, pp.365-374.

[12] Porat, B.: A Course in Digital Signal Processing. John Wiley & Sons, Inc., New York, 1997.

[13] Proakis, J.G. - Manolakis, D.G.: *Introduction to Digital Signal Processing*. Macmillan Publishing Company, New York, 1988.

[14] Stevens, J.: DSPs in Communications. *IEEE Spectrum*, September 1998, pp.39-46.

[15] White, S.A.: Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review. *IEEE ASSP Magazine*, July 1989, pp.4-19.

[16] VisualDSP++ 4.5 Getting Started Guide. Analog Devices, Inc., 2006.

[17] VisualDSP++ 4.5 C/C++ Compiler and Library Manual for Blackfin Processors. Analog Devices, Inc., 2006.

[18] http://www.analog.com/en/products/processors-dsp.html

[19] http://www.filter-solutions.com/

[20] https://data.kemt.fei.tuke.sk/SignaloveProcesory/_web/

ZOZNAM POUŽITÝCH SKRATIEK

AAU	**A**ddress **A**rithmetic **U**nit, adresová aritmetická jednotka
A/D	**A**nalog to **D**igital, analógovo-číslicový (prevodník)
ALU	**A**rithmetic **L**ogic **U**nit, aritmeticko-logická jednotka
API	**A**pplication **P**rogramming **I**nterface, aplikačné programové rozhranie
ASIC	***A**pplication **S**pecific **I**ntegrated **C**ircuit*, zákaznícky obvod
BDTI	***B**erkeley **D**esign **T**echnology, **I**nc*.
BFU	**B**it **F**ield **U**nit, jednotka pre bitové operácie
BMU	**B**it **M**anipulation **U**nit, jednotka pre bitové operácie
BU	**B**ranch **U**nit, jednotka pre vetvenie
CCOP	**C**yclic Code **CO**-**P**rocessor, koprocesor pre cyklické kódy
CELP	**C**ode **E**xcited **L**inear **P**rediction, súbor metód kompresie (reči) na báze lineárnej predikcie
CMOS	***C**omplementary **M**etal **O**xide **S**emiconductor*, komplementárny MOS
COFF	**C**ommon **O**bject **F**ile **F**ormat, štandardný formát relatívnych súborov
CPU	**C**entral **P**rocessing **U**nit, centrálna procesorová jednotka
CRC	**C**yclic **R**edundancy **C**heck, metóda detekcie chýb s využitím syndrómu cyklických blokových kódov
ČSS	**Č**íslicové **S**pracovanie **S**ignálov
DA	**D**istributed **A**rithmetic, distribuovaná aritmetika
D/A	**D**igital to **A**nalog, číslicovo-analógový (prevodník)
DCT	**D**iscrete **C**osine **T**ransform, diskrétna kosínusová transformácia
DFT	**D**iscrete **F**ourier **T**ransform, diskrétna Fourierova transformácia
DIF	**D**ecimation **I**n **F**requency, decimácia vo frekvenčnej oblasti
DIT	**D**ecimation **I**n **T**ime, decimácia v časovej oblasti
DM	**D**ata **M**emory, dátová pamäť
DMA	**D**irect **M**emory **A**ccess, priamy prístup do pamäte
DP	**D**ata **P**rocessor, dátový procesor
DRAM	**D**ynamic **RAM**, dynamická RAM
DSP	**D**igital **S**ignal **P**rocessor, číslicový signálový procesor
EFCOP	**E**nhanced **F**ilter **CO**-**P**rocessor, rozšírený filtračný koprocesor
ENIAC	***E**lectronic **N**umerical **I**ntegrator **A**nd **C**alculator*
EPROM	***E**rasable **P**rogrammable **R**ead-**O**nly **M**emory*, pamäť na čítanie mazateľná UV žiarením
ETSI	***E**uropean **T**elecommunications **S**tandard **I**nstitute*, Európsky telekomunikačný štandardizačný úrad
FCOP	**F**ilter **CO**-**P**rocessor, filtračný koprocesor
FFA	**F**inite **F**ield **A**rithmetic, aritmetika v konečných (Galoisových) poliach
FFT	**F**ast **F**ourier **T**ransform, rýchla Fourierova transformácia

FIR	**F**inite **I**mpulse **R**esponse, konečná impulzová odpoveď
FLASH	elelektricky prepisovateľná pamäť na čítanie
FPGA	**F**ield **P**rogrammable **G**ate **A**rray, užívateľsky programovateľné hradlové pole
G	**G**iga, miliarda (prípadne 2^{30} =1073741824 pre veľkosť pamäte)
GOPS	**G**iga **O**perations **P**er **S**econd, miliárd operácií za sekundu
GSM	**G**roup **S**pécial **M**obiles, digitálny mobilný systém
IEEE 754	štandard pre reprezentáciu čísel v pohyblivej rádovej čiarke
IDFT	**I**nverse **DFT**, inverzná DFT
IFFT	**I**nverse **FFT**, inverzná FFT
IIR	**I**nfinite **I**mpulse **R**esponse, nekonečná impulzová odpoveď
ILP	**I**nstruction **L**evel **P**arallelism, paralelizmus na úrovni inštrukcií
IM	**I**nstruction **M**emory, inštrukčná pamäť
IP	**I**nstruction **P**rocessor, inštrukčný procesor
I/O	vstupno-výstupný
JTAG	**J**oin European **T**est **A**ction **G**roup, skupina, ktorá vytvorila testovací štandard IEEE 1149.1-1990
K	**K**ilo, tisíc (prípadne 2^{10} =1024 pre veľkosť pamäte)
LIW	**L**ong **I**nstruction **W**ord, dlhé inštrukčné slovo
LSB	**L**east **S**ignificant **B**it, najmenej významový bit
LSI	**L**arge **S**cale **I**ntegration, vysoký stupeň integrácie
M	**M**ega, milión (prípadne 2^{20} =1048576 pre veľkosť pamäte)
MAC	**M**ultiply and **AC**cumulate, operácia násobenia a akumulácie
MCU	**MiC**rocontroler **U**cnit, mikrokontrolér
MF	**M**atched **F**ilter, prispôsobený filter
MFOPS	**M**illion **F**loating **O**perations **P**er **S**econd, milióny operácií v pohyblivej rádovej čiarke za sekundu
MIMD	***M**ultiple **I**nstructions **M**ultiple **D**ata*
MISD	***M**ultiple **I**nstructions **S**ingle **D**ata*
MIPS	**M**illion **I**nstructions **P**er **S**econd, milióny inštrukcií za sekundu
MOPS	**M**illion **O**perations **P**er **S**econd, milióny operácií za sekundu
MOS	**M**etal **O**xide **S**emiconductor, polovodič s kovovým oxidom
MOSFET	**M**etal **O**xide **S**emiconductor **F**ield **E**ffect **T**ransistor, poľom riadený tranzistor s hradlom izolovaným kovovým oxidom
NMOS	**N** channel **MOS**, štruktúra MOS s kanálom typu N
NOP	**N**o **OP**eration, prázdna operácia
OS	**O**peračný **S**ystém
$O(N^x)$	zložitosť úmerná mocnine N^x
PLL	**P**hase **L**ock(ed) **L**oop, slučka fázového závesu
PROM	**P**rogrammable **R**ead **O**nly **M**emory, trvalo programovateľná pamäť
RAM	**R**andom **A**ccess **M**emory, pamäť s ľubovoľným prístupom
RFFT	**R**eal **FFT**, FFT s čisto reálnym vstupom
RISC	**R**educed **I**nstruction **S**et **C**omputer, počítač (procesor) s redukovaným súborom inštrukcií
RNS	**R**esidue **N**umber **S**ystem, zvyškový číselný systém
SA	**S**ystolic **A**rray, synchrónne systolické pole
SCI	**S**erial **C**ommunication **I**nterface, sériové komunikačné rozhranie
SIMD	***S**ingle **I**nstruction **M**ultiple **D**ata*
SMD	**S**urface **M**ount **D**evice, súčiastka pre povrchovú montáž

SRAM	**S**tatic **RAM**, statická RAM
SSI	**S**erial **S**ynchronous **I**nterface, sériové synchrónne rozhranie
TI	***T**exas **I**nstruments*
ULSI	**U**ltra **L**arge **S**cale **I**ntegration, ultravysoký stupeň integrácie
VA	**V**iterbi **A**lgorithm, Viterbiho algoritmus
VCOP	**V**iterbi **CO**-**P**rocessor, Viterbiho koprocesor
VelociTI	VLIW modifikácia firmy Texas Instruments
VLES	**V**ariable **L**ength **E**xecution **S**et, inštrukčná sada s premenlivou dĺžkou
VLIW	**V**ery **L**arge **I**nstruction **W**ord, inštrukčné slovo s veľkou dĺžkou
VLSI	**V**ery **L**arge **S**cale **I**ntegration, veľmi vysoký stupeň integrácie
VMOS	technológia N**MOS** v žliabku **V**
VSELP	*Vector **S**um **E**xcited **L**inear **P**redictive Coding*, variant CELP