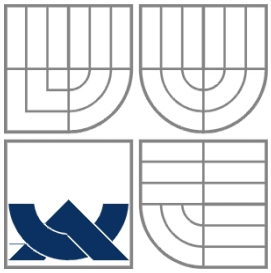
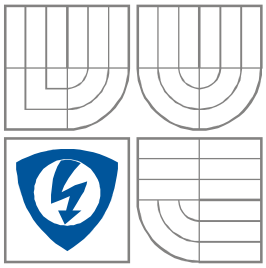


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV RADIOELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF RADIO ELECTRONICS

DIGITÁLNÍ FILTRACE NA 8-BITOVÝCH PROCESORECH

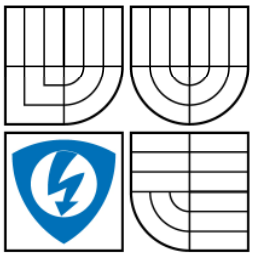
DIGITAL FILTRATION WITH 8-BIT PROCESSORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Filip Záplata
AUTHOR

VEDOUCÍ PRÁCE — Ing. Zbyněk Fedra, PhD.
SUPERVISOR

BRNO, 2009



VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav radioelektroniky

Bakalářská práce

bakalářský studijní obor
Elektronika a sdělovací technika

Student:	Filip Záplata	***ID:***	72765
Ročník:	3	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Digitální filtrace na 8-bitových procesorech

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte možnosti realizace digitální filtrace na osmibitovém procesoru. Prostudujte algoritmy používané pro realizaci digitální filtrace a jejich náročnost. Srovnejte možnost využití vybraných algoritmů v jazyce C a v assembleru.

Otestujte vybrané algoritmy na procesorech Atmel AVR a to jak v jazyce C tak v assembleru, případně otestujte smíšený kód. Zaměřte se na rychlost algoritmu a na omezení jeho možností. Vypracujte zhodnocení a doporučení pro použití digitální filtrace na platformě Atmel AVR a připravte odpovídající knihovnu funkcí a návod na ukázkovou laboratorní úlohu.

DOPORUČENÁ LITERATURA:

[1] MATOUŠEK, D. Práce s mikrokontroléry Atmel AVR. BEN - technická literatura, Praha 2003

[2] MANN, B. C pro mikrokontroléry. BEN - technická literatura, Praha 2003

Termín zadání:	9.2.2009	***Termín odevzdání:***	5.6.2009

Vedoucí práce: Ing. Zbyněk Fedra, Ph.D.

prof. Dr. Ing. Zbyněk Raida
Předseda oborové rady

UPOZORNĚNÍ:

Licenční smlouva
poskytovaná k výkonu práva užít školní dílo

uzavřená mezi smluvními stranami:

1. Pan/paní

Jméno a příjmení:	Filip Záplata
Bytem:	Miřetice 135, Miřetice u Hlinska, 539 55
Narozen/a (datum a místo):	28. března 1987 v Chrudimi

(dále jen „autor“)

a

2. Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií
se sídlem Údolní 53, Brno, 602 00
jejímž jménem jedná na základě písemného pověření děkanem fakulty:
prof. Dr. Ing. Zbyněk Raida, předseda rady oboru Elektronika a sdělovací technika
(dále jen „nabyvatel“)

Čl. 1

Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):

   ☐ disertační práce
   ☐ diplomová práce
   ☒ bakalářská práce
   ☐ jiná práce, jejíž druh je specifikován jako .....
   (dále jen VŠKP nebo dílo)

Název VŠKP:	Digitální filtrace na 8-bitových procesorech
Vedoucí/ školitel VŠKP:	Ing. Zbyněk Fedra, PhD.
Ústav:	Ústav radioelektroniky
Datum obhajoby VŠKP:	_____

   VŠKP odevzdal autor nabyvateli*:

   ☒ v tištěné formě – počet exemplářů: 2
   ☒ v elektronické formě – počet exemplářů: 2

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.

3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.

4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

* hodící se zaškrtněte

Článek 2

Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.

2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.

3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti

   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)

4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3

Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.

2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.

3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.

4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 5. června 2009

……..	……
Nabyvatel	Autor

Anotace

Cílem mé bakalářské práce je digitální filtrace pomocí 8-bitových procesorů. Číslicové filtry jsou hojně využívané a postupně vytlačují jejich analogové protějšky. Jejich výhody jsou nesporné, ale větší rozvoj stále brzdí výpočetní náročnost. Obecně používané mikrokontroléry neoplývají příliš výkonným jádrem, je proto při snaze aplikovat digitální filtr nutno využít co nejlepších algoritmů. Programovat lze v jazyce C a v JSA, případně kombinovat oba jazyky. Úkolem je vytvoření knihovny funkcí podle několika výkonných algoritmů, které budou zajišťovat filtraci. Otestování těchto knihoven a porovnat jejich použitelnost se zaměřením na rychlost a univerzálnost použití a zhodnocení způsobů programování.

Anotation

The aim of my bachelor's thesis is a digital filtration using the 8-bit processors. Digital filters are in use a lot and gradually outdoing analog equivalents. Their advantages are indisputable, but computing still inhibits bigger progress. Generally used microcontrollers do not have very powerfull core, so there is effort to apply as bright as possible algorithms. Program code can be written in C language or assembler eventually is possible to combine both languages. The task is to make a library of functions based on some powerfull algorithms for filtering. Testing those libraries and comparing their useability focused on maximal speed and universality of use and review wals of programming.

Klíčová slova

Digitální filtrace, FIR, IIR, impulsní charakteristika, mikrokontroléry, AVR, GCC, asembler, algoritmus násobení, fourierova transformace, DFFT.

Keywords

Digital filtration, FIR, IIR, impulse response, microcontrollers, AVR, GCC, assembler, multiplication algorithm, fourier transform, DFFT.

ZÁPLATA, F. *Digitální filtrace na 8-bitových procesorech: bakalářská práce.* Brno: FEKT VUT v Brně, 2009. 35 s., 6 příl.

Prohlášení

Prohlašuji, že svou bakalářskou práci na téma Digitální filtrace na 8-bitových procesorech jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne 5. června 2009 .....

podpis autora

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Zbyňku Fedrovi, PhD. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne 5. června 2009 .....

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Číslicové filtry jsou v dnešní době hojně využívanou technikou. Ve srovnání s filtry analogovými mají mnohé výhody. Lze jimi realizovat teoreticky libovolnou frekvenční charakteristiku, a to jak modulovou, tak i fázovou. Jednou z předností je snadná korekce parametrů vytvořeného filtru, pouhou změnou několika číselných koeficientů je možné změnit významně charakter filtru. Jeden filtr se potom používá hned v několika odlišných aplikacích s pouhým rozdílem těchto filtračních koeficientů, které se dají změnit buď jednoduchým přeprogramováním, nebo vlastním obslužným programem.

Číslicová filtrace je pouze sled matematických operací. Tyto operace musejí být prováděny s určitou rychlostí v závislosti na maximálním požadovaném kmitočtu filtrem ještě zpracovatelným. Z toho vyplývá jistá horní kmitočtová hranice, která je závislá na výpočetní výkonnosti hardwaru.

Filtrační algoritmy mohou být spouštěny na jakýchkoli tomu uzpůsobených číslicových zařízeních. Zpracování osobním počítačem je dnes už běžnou záležitostí. Mírně odlišná je snaha aplikovat tyto algoritmy na součásti mnohem menších rozměrů. Ty potom zařadit do obvodů jiných zařízení. Příkladem mohou být signální procesory, jež jsou hojně využívány například v audiotechnice. Zajímavé je naprogramovat obslužný program s digitální filtrací do mikroprocesoru.

Mikroprocesory jsou běžně dostupné a jejich cena není nijak vysoká. Variací je nespočet a práce s nimi dnes již není nic neobvyklého. Nevýhodou ovšem může být nevelká výpočetní schopnost oproti speciálním obvodům. Cílem této práce je, zjistit, jak na tom mikroprocesory s implementovanými digitálními filtry jsou.

Zaměření je hlavně na výkon a univerzálnost filtru, možnosti programování v jazyce C a JSA a výběr algoritmů. Další částí je rozbor možných problémů, které se mohou vyskytnout při práci s vytvořenými filtry, případně pak možnosti jejich odstranění. Poslední rozsáhlá část se věnuje odvození a možné implementaci diskrétní fourierovy transformace na 8-bitové procesory. Výsledkem je srovnání jednotlivých algoritmů, vytvořené knihovny funkcí a jejich dokumentace, výsledky testů a doporučení těchto knihoven.

2 Číslicové lineární filtry

Číslicové filtry jsou zpravidla popisovány matematickými rovnicemi různých tvarů. Základní rovnicí je přenosová funkce. Je vyjádřena v rovině Z pomocí jednotkové kružnice, nulových bodů a pólů a vyjadřuje přenos filtru. Jednoduchou úpravou přenosové funkce můžeme získat frekvenční charakteristiku filtru a zpětnou Z transformací pak diferenční rovnici. Z diferenční rovnice se pak vychází při vlastní realizaci filtru.

Dalším důležitým parametrem je impulsní charakteristika (ICH). Získáme ji jako odezvu filtru na jednotkový impuls. Podle tvaru ICH pak můžeme číslicové filtry rozdělit na dva základní typy:

- FIR (konečná ICH)
- IIR (nekonečná ICH)

2.1 Filtry s konečnou impulsní charakteristikou

FIR filtry jsou přesně určeny konečným počtem své ICH. Z jejich diferenční rovnice

$$y_n = \sum_{i=0}^{N-1} h_i x_{n-i} \tag{2.1}$$

plyne přímá realizace. Koeficienty filtru jsou dány přímo hodnotami ICH h_i. Přenosová funkce má tvar

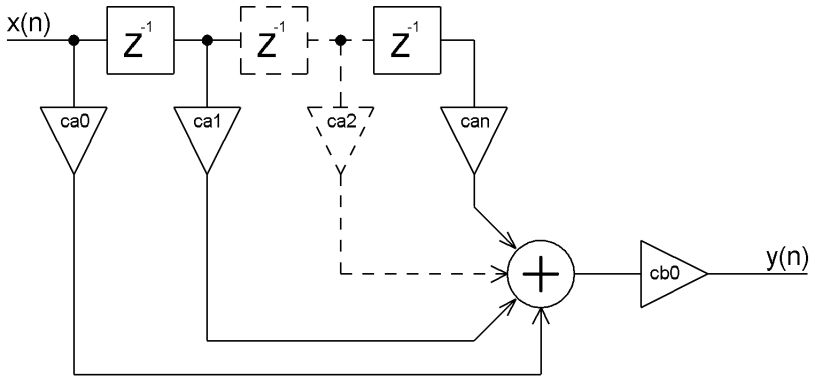
$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i} \ . \tag{2.2}$$

Je charakterizována pouze nulovými body, následně jsou filtry typu FIR absolutně stabilní.

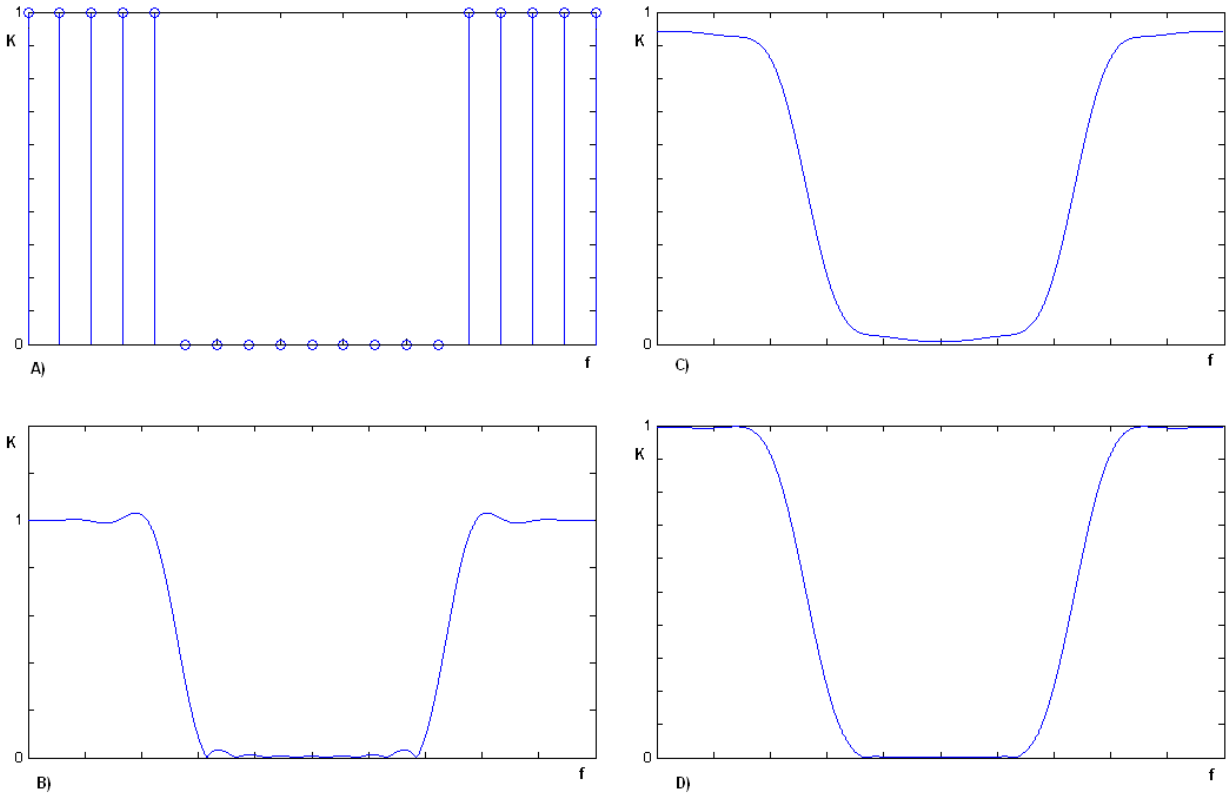
Při návrhu lze ICH získat jednoduše vzorkováním zvolené frekvenční charakteristiky a následnou diskrétní fourierovou transformací (DFT). Takto vzniklá charakteristika se musí váhovat.

Nejjednodušší je váhování obdélníkovým oknem, kdy se vezme v úvahu celý neupravený výsledek DFT, což ovšem nemusí být celá ICH. Kmitočtová charakteristika takto vzniklého filtru je pak silně zvlněná, bylo proto nalezeno množství váhovacích oken podávajících lepší výsledky. Porovnání charakteristik různých oken je na Obr. 2.

Blokové schéma přímé realizace FIR filtru ukazuje Obr. 1.



Obr. 1 Blokové schéma FIR



Obr. 2 Váhovací okna
A) zvolená charakteristika, navzorkovaná
B) obdélníkové okno
C) Bartlettovo okno
D) Hannovo okno

U většiny požadovaných frekvenčních charakteristik vyjde ICH s určitou symetrií. Vycházejí z toho dva zvláštní případy, a to se symetrickou a antisymetrickou ICH.

Matematicky lze pro každý typ odvodit diferenční rovnici pro lichý a sudý počet koeficientů zvlášť. Diferenční rovnice pro lichou ICH má tvar

$$y_n = h_{\frac{N-1}{2}} x_{n-\frac{N-1}{2}} + \sum_{i=0}^{\frac{N-1}{2}-1} h_i \left(x_{n-i} \pm x_{n-(N-1-i)} \right), \tag{2.3}$$

pro sudou pak

$$y_n = \sum_{n=0}^{\frac{N}{2}-1} h_i \left(x_{n-i} \pm x_{n-(N-1-i)} \right). \tag{2.4}$$

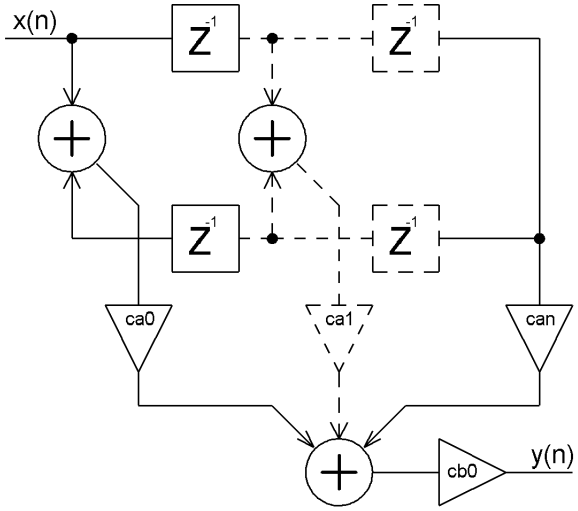
Součet v sumě potom platí pro symetrickou a rozdíl pro antisymetrickou ICH. *Z* transformací dostaneme přenosové funkce. Pro lichou (2.5) a pro sudou ICH (2.6).

$$H(z) = h_{\frac{N-1}{2}} z^{-\frac{N-1}{2}} + \sum_{i=0}^{\frac{N-1}{2}-1} h_i \left(z^{-i} \pm z^{-(N-1-i)} \right) \tag{2.5}$$

$$H(z) = \sum_{i=0}^{\frac{N}{2}-1} h_i \left(z^{-i} \pm z^{-(N-1-i)} \right) \qquad \textbf{(2.6)}$$

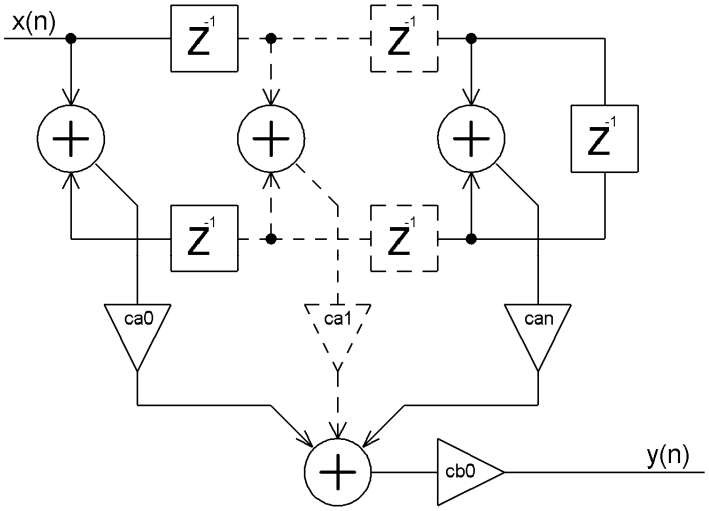
Z rovnic vyplývá, že počet sumačních členů je poloviční, čímž se zredukuje počet zesilovacích prvků (součinů), ostatní prvky zůstávají nezměněny. U filtrů vyšších řádů může tato úprava způsobit citelné urychlení výpočtů. Algoritmus násobení patří mezi časově nejnáročnější části programu, proto lze takto výrazně zvýšit výkon filtru.

Bloková schémata jsou následující



Obr. 3 Blokové schéma FIR lichá

pro lichou a



Obr. 4 Blokové schéma FIR sudá

pro sudou ICH.

2.2 Filtry s nekonečnou impulsní charakteristikou

Druhou velkou skupinou jsou filtry IIR. Prakticky se jedná o doplnění filtru FIR zpětnovazební (rekurzivní) částí. Neznamená to však rovnost mezi pojmy rekurzivní filtr a nekonečná ICH. Zvláštní případy rekurzivních filtrů mohou být také typu FIR. Tyto filtry

mají vždy nelineární fázovou charakteristiku, k lineární se lze pouze přiblížit. Je možné touto cestou vytvořit i fázovací články, tj. filtry s definovanou fázovou charakteristikou a konstantním přenosem v celém pásmu. IIR filtry mají obecně větší možnosti než FIR právě kvůli rekurzivní části.

Základní diferenční rovnice má tvar

$$y_n = \sum_{i=0}^{k} A_i x_{n-i} + \sum_{i=1}^{l} B_i y_{n-i} \,. \tag{2.7}$$

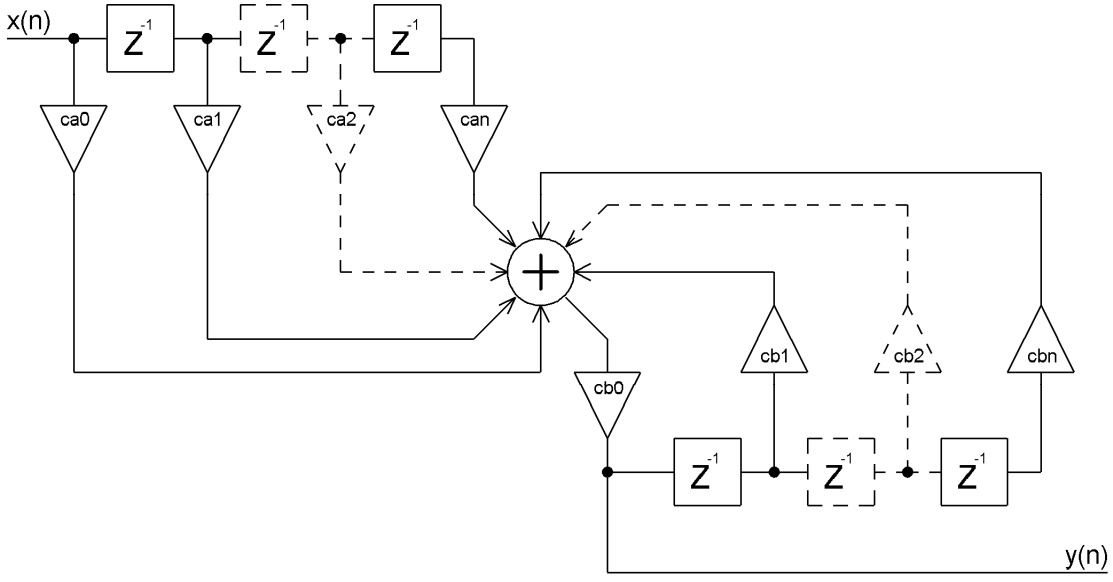
Původ koeficientů filtru A pro přímou větev a B pro zpětnou větev je zřejmý z přenosové funkce

$$H(z) = \frac{\sum_{i=0}^{k} A_i z^{-i}}{\sum_{i=1}^{l} B_i z^{-i}} \,. \tag{2.8}$$

Koeficienty tedy vyjadřují činitele v polynomech po roznásobení, A plyne z nulových bodů a B z pólů.

U FIR filtrů byl systém průchozí pouze přímo, byla tedy zaručena stabilita v celém pásmu pro jakoukoli realizaci. U IIR tomu tak není. Nestabilitu systému může způsobit právě zpětnovazební část. Při návrhu je nutno tento fakt zohlednit. Návrh probíhá nejčastěji v rovině Z, existuje několik pravidel pro tento způsob, která je nutno dodržet.

Stejně jako u filtrů FIR i u filtrů s nekonečnou ICH bylo vypracováno více realizačních postupů. První je jistě přímá realizace, jež vychází z diferenční rovnice popsané výše. Blokové schéma je na Obr. 5.



Obr. 5 Blokové schéma IIR

Úpravami diferenční rovnice lze dospět i k takzvaným kanonickým realizacím. Pro názornost je rozepsána diferenční rovnice do tvaru

$$y_n = \sum_{i=0}^{m} A_i x_{n-i} + \sum_{i=1}^{m} B_i y_{n-i}$$
$$\Downarrow \qquad . \qquad \textbf{(2.9)}$$
$$0 = A_0 x_n + A_1 x_{n-1} + \ldots + A_m x_{n-m} + B_0 y_n + B_1 y_{n-1} \ldots B_m y_{n-m}$$

Následně je rozložena na části se stejným časovým posunem (2.10).

$$\begin{aligned} w_0(n) &= A_0 x_n + B_0 y_n \\ w_1(n) &= A_0 x_n + B_0 y_n + A_1 x_{n-1} + B_1 y_{n-1} \\ &\vdots \\ w_m(n) &= A_0 x_n + B_0 y_n + A_1 x_{n-1} + B_1 y_{n-1} + \cdots + A_l x_{n-m} + B_l y_{n-m} \end{aligned} \qquad \textbf{(2.10)}$$

Z první rovnice je vyjádřen výstupní signál a dosazením do soustavy rovnic a několika úpravami získána soustava diferenčních rovnic pro 1. kanonickou formu [3].

$$\begin{aligned} w_0(n) &= A_1 x_{n-1} + B_1 y_{n-1} + w_1(n-1) \\ w_1(n) &= A_2 x_{n-1} + B_2 y_{n-1} + w_2(n-1) \\ &\vdots \\ w_{m-2}(n) &= A_{m-1} x_{n-1} + B_{m-1} y_{n-1} + w_{m-1}(n-1) \\ w_{m-1}(n) &= A_m x_{n-1} + B_m y_{n-1} \\ y_n &= \frac{1}{B_0}\left[w_1(n) + A_0 x_n\right] \end{aligned} \qquad \textbf{(2.11)}$$

Změnou pořadí výpočtu filtru tak, že se nejprve provede zpětnovazební část, a potom teprve část přímá, bude soustava diferenčních rovnic vypadat takto

$$\begin{aligned} w_0(n) &= \frac{1}{B_0}\left[B_1 w_1(n) + B_2 w_2(n) + x_n\right] \\ w_1(n) &= w_0(n-1) \\ &\vdots \\ w_m(n) &= w_{m-1}(n-1) \\ y_n &= A_0 w_0(n) + A_1 w_1(n) + \cdots + A_m w_m(n) \end{aligned} \qquad \textbf{(2.12)}$$

a bývá nazývána jako 2. kanonická forma [3].

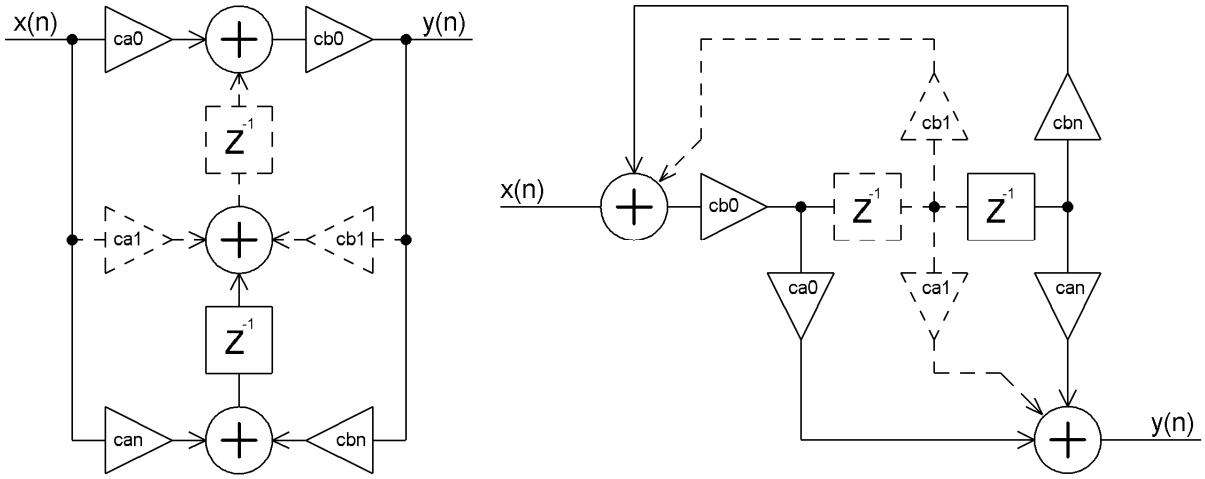
Tyto realizace jsou výhodné především pokud počet členů rekurzivní části odpovídá počtu členů části přímé, tj. $l = k = m$. V rekurzivní části byl přidán jeden koeficient navíc, B_0, není bezpodmínečně nutný, ovšem lze jím výhodně usměrnit přetékání registrů, nebo zaokrouhlovací chybu, tento koeficient je obsažen ve všech uvedených blokových schématech, nikoli však v rovnicích. Odvozené rovnice nerespektují změny znamének, předpokládá se jejich zahrnutí do koeficientů. Bloková schémata filtrů 1. a 2. kanonické formy jsou na Obr. 6.

Filtry lze realizovat také řazením několika filtrů nižšího řádu do série, pak vzniká 3. kanonická forma, nebo paralelně, označovaná jako 4. kanonická forma. Existuje mnoho dalších kanonických realizací.

Návrh koeficientů u IIR filtrů už bohužel není tak jednoznačný, jako u filtrů FIR. Využívá se několik technik.

Nejjednodušší je intuitivní rozdělení nulových bodů a pólů do roviny *Z* a následné odvození přenosové funkce a diferenční rovnice. Tento postup není ovšem příliš vhodný pro složitější frekvenční charakteristiky požadovaných filtrů, kdy při ručním řešení roste složitost nad únosnou míru. Lze jimi realizovat například jednoduché úzkopásmové zádrže, jež vystačí i s 2. řádem.

Jako velice silného nástroje se nabízí využití počítače. Optimalizačními přístupy lze řešit i složitější zadání, pokud je k dispozici postačující programové vybavení. Mezi tyto metody patří například metoda nejmenších čtverců v časové a kmitočtové oblasti. Tyto metody se používají tam, kde jsou kladeny vysoké nároky na filtry, vyšší jak na analogové systémy.



Obr. 6 Kanonické filtry
A) 1. kanonická forma
B) 2. kanonická forma

Zajímavým a hojně využívaným způsobem návrhu jsou postupy založené na podobnosti s analogovými systémy, tj. souvislostí mezi rovinou *Z* a rovinou *p* u Laplaceovy transformace. Patří mezi ně bilineární transformace nebo signální invariance. Jsou využívány Butterworthovy, Besselovy, Čebyševovy a eliptické aproximace. Tyto postupy jsou jednoduché a nenáročné na výpočetní techniku.

3 Filtry na AVR

Moderní mikrokontroléry jsou vybaveny výkonnou aritmeticko-logickou jednotkou, která dokáže vykonat matematické operace během několika málo cyklů. Příkladem je násobící jednotka v mikroprocesorech AVR. Ta umožňuje zpracovat množství výpočtů, např. digitální filtraci, v mnohem kratším čase.

Dále je uvažována pouze možnost realizace programů na mikroprocesorech AVR fy. ATMEL. Konkrétně pak budou všechny požadavky a srovnávací hodnoty vztahovány k procesoru ATmega16.

3.1 ATmega16

Před tvořením programu je vhodné znát základní vlastnosti procesoru a vyhodnotit jejich efektivní využití. V Tab. 1 jsou shrnuty důležité parametry ATmega16.

Tab. 1 Vlastnosti ATmega16

ATmega16	
pracovní kmitočet	16MHz (až 16MIPS)
velikost programové paměti	16k bytů
paměť RAM	1k byte

Výběr pracovního kmitočtu určuje, pro jak náročné aplikace může být digitální filtrace využívána, neboť to je hlavní faktor ovlivňující rychlost zpracování vzorku. Bude velice obtížné vytvořit filtr zpracovávající celé audiopásmo v rozlišení 16 bitů při pracovním kmitočtu 16MHz. Naopak pro datový tok okolo 100Sa/s při rozlišení 8 bitů nemusí být problém vytvořit filtry vyšších řádů, z pozdějšího rozboru bude zřejmé, jaké hodnoty jsou kritické.

Je-li nutné filtr zahrnout do nějaké rozsáhlejší aplikace, může hrát velkou roli velikost programové paměti, kterou bude filtr v procesoru zabírat. Významné úspory lze dosáhnout využitím rozložení do podprogramů.

Rychlost programu ovlivní také paměť RAM, tedy místo pro proměnné a odkládání dat. Přístup k datové paměti může být v některých případech zdlouhavý a je lepší přistoupit k použití většího množství volně použitelných registrů, jejichž množství je limitované.

Tyto aspekty jsou protichůdné a je nutné volit vhodný kompromis. Zde je návrh zaměřen na maximální rychlost zpracování při zachování dobré univerzálnosti a možnosti modifikace filtru.

K programování mikrokontrolérů se využívá převážně jazyka C a JSA. C je z rodiny vyšších jazyků. Kompilátor tedy musí přeložit program napsaný v jazyce C do JSA. Jednotlivé příkazy jsou v knihovnách C psány také v JSA, ale musí být dostatečně univerzální, což většinou zavádí do programu přebytečné instrukce. Kritickou bývá hlavně práce s datovou pamětí, kdy každý příkaz zapíše použité proměnné zpět z pracovních registrů do datové paměti a následující příkaz je tak musí znovu a zbytečně načítat. Na druhou stranu programování v C je mnohem pohodlnější a zpravidla odladění programu bývá jednodušší jak v JSA.

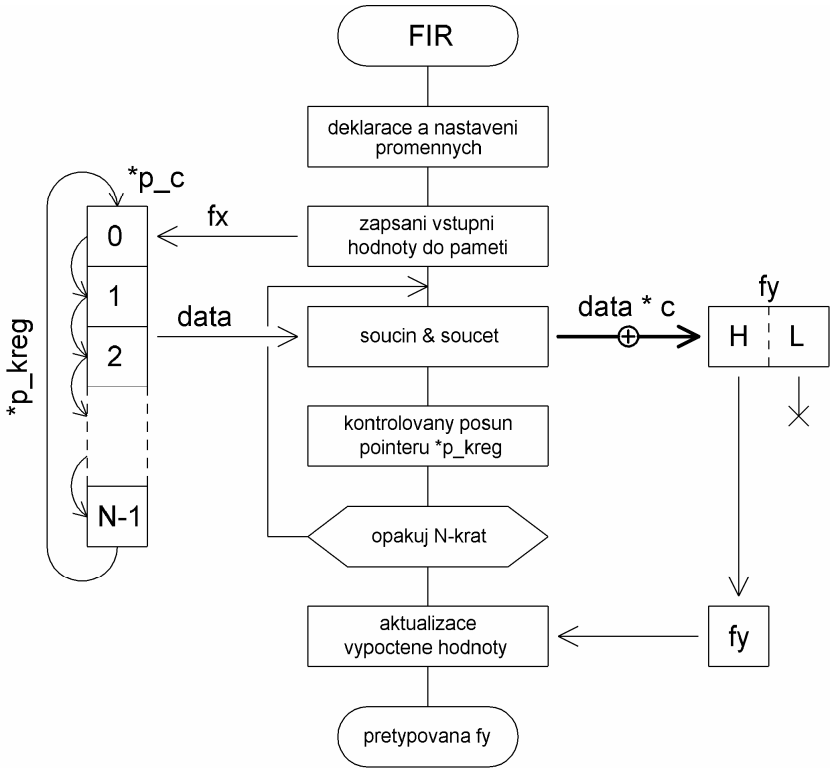
JSA je základní programovací jazyk, pro AVR existuje specifická instrukční sada typu RISC. Každá instrukce má přiřazeno 16 bitové číslo, kompilátor přeloží pouze symbolické adresy přímo na kód v hexadecimálním tvaru. Pro vykonání jedné operace je nutno více

instrukcí, ovšem lze tak více experimentovat. Práce s pamětí v JSA je věcí čistě programátorskou, tzn. že je nutno mít přehled o organizaci celé datové paměti. Práce s JSA bývá dosti obtížná a časově náročná, ale pro nejnáročnější aplikace není prakticky jiného východiska.

3.2 Algoritmy

Před tvorbou programu je dobré pro přehlednost odvozená bloková schémata promyslet a vytvořit na jejich základě vývojové diagramy, které budou ve zjednodušené formě vyjadřovat běh programu.

Jako příklad je nejprve rozebrána přímá realizace filtru FIR. Jeho vývojový diagram je na Obr. 7.



Obr. 7 Vývojový diagram filtru FIR

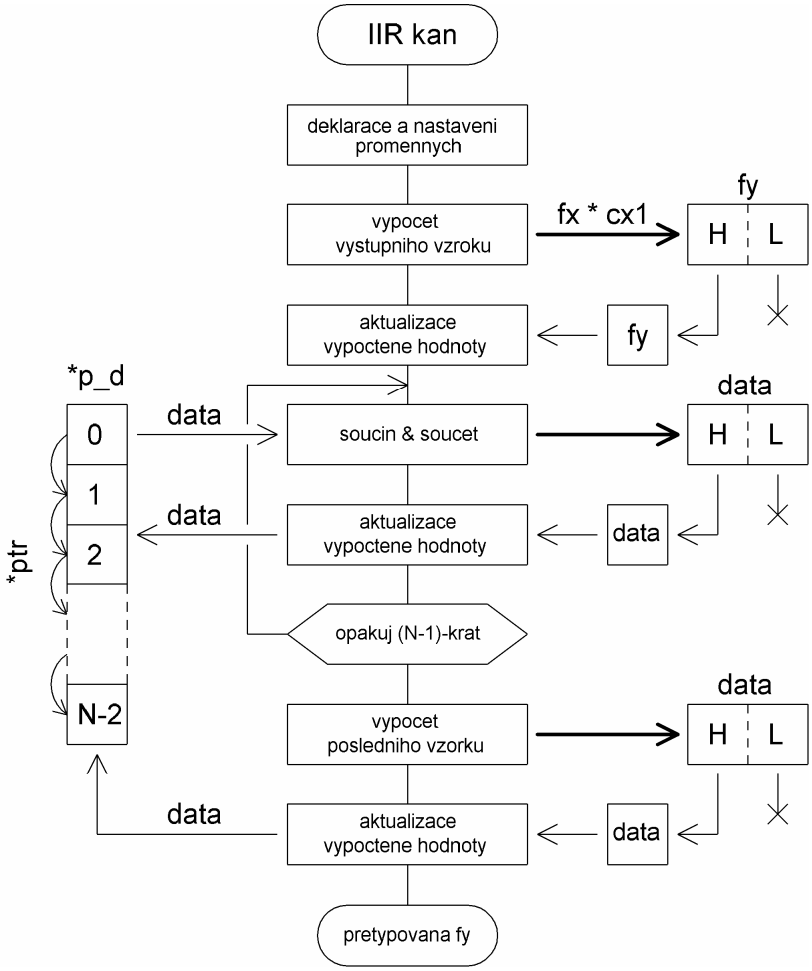
Zpoždění vzorků je zde vytvořeno pomocí kruhového registru indexovaným ukazatelem *p_kreg*. Registr je deklarován jako pole předem dané velikosti, tím je zajištěno, že všechny prvky budou pohromadě za sebou. Do tohoto pole je při každém spuštění zapsána hodnota příchozího vzorku. V průběhu programu se pointer postupně posouvá ve směru šipek a načítá data. Před ukončením funkce je ukazatel nastaven na nejstarší vzorek, ten je tak po dalším spuštění přepsán novou hodnotou. Pro správnou funkci je nutno hodnotu pointeru *p_kreg* zachovávat i po ukončení funkce a vykonávání obslužného programu. Je proto uchovávána jako globální proměnná a nesmí k ní být přístup z vnějšku.

Po zavolání funkce se předají vstupní parametry, deklarují se pomocné proměnné a nastaví se do potřebného stavu. Poté se vstupní vzorek zapíše do kruhového registru a přejde se k cyklu výpočtu. Zde jsou postupně načítány vzorky směrem k nejstaršímu jako *data*. Vzorek je vždy vynásoben příslušným koeficientem a přičten do proměnné *fy*, je tak realizován výpočet dle (3.1). Po vynásobení je velikost potřebné paměti zdvojena, jak je naznačeno silnou šipkou a zdvojeným paměťovým místem *fy*. Dále je inkrementován ukazatel kruhového registru. Kontrolovaným posunem je v tomto případě myšlen skok na hodnotu *p_c*,

tj. na začátek kruhového registru, z konce pole, to je zajištěno jednoduchým podmíněným příkazem. Celý cyklus se opakuje pro každý uložený vzorek, tj. v závislosti na řádu filtru.

$$\begin{aligned} y &= 0 \\ y &= y + x_n \cdot A_0 \\ &\vdots \\ y &= y + x_{n-N+1} \cdot A_{N-1} \\ y &= \frac{1}{k \cdot B_0} \cdot y \end{aligned} \qquad \textbf{(3.1)}$$

Vypočtenou hodnotu *fy* je nakonec ještě důležité upravit do správného tvaru, aktualizovat. Aktualizace nemůže proběhnout tak jednoduše, jak je naznačeno v diagramu přenesením pouze horní poloviny (bytu, slova). Spočívá ve vydělení konstantou, většinou realizovatelným bitovým posunem, a v zaokrouhlení. Také je zde možné aplikovat konstantu upravující kvalitu přenosu B_0, která je ale využita pouze u filtrů IIR. Tuto část vystihuje poslední rovnice v (3.1). Takto je proměnná přetypována a uložena na výstup.



Obr. 8 Vývojový diagram 1. kanonické formy

Identický postup výpočtu je použit pro přímou realizaci filtru IIR, zpětnovazební větev je téměř identická s přímou větví FIR. Filtry se symetrickou ICH využívají také podobného postupu. Cyklus se vykonává po poloviční dobu a před každým součinem je nutno příslušné vzorky nejprve sečíst. Navíc je tento algoritmus doplněn ještě o podmíněný příkaz, který

rozděluje výpočet pro sudou a pro lichou ICH. Vývojové diagramy těchto filtrů jsou uvedeny v příloze.

Kanonické filtry IIR jsou celkově dosti odlišné a vyžadují podrobnější rozbor. Vývojový diagram 1. kanonické formy je na Obr. 8.

Při výpočtu již není využit kruhový registr, do pole dat se ukládají pomocné hodnoty. Tyto hodnoty se při každém výpočtu všechny změní. V cyklu se na základě vstupní a výstupní hodnoty a hodnoty uložené na následující pozici přepočítá příslušná buňka datového pole.

Na začátku se identicky deklarují a nastaví pomocné proměnné. Poté se vypočte výstupní hodnota podle první rovnice v (3.2). Vypočtená hodnota se aktualizuje a přejde se k cyklu, kde se přepočítá datové pole. Poslední hodnota v poli se počítá již mimo cyklus, neboť nelze přičíst předchozí hodnotu. Všechny výpočty jsou zřejmé z (3.2).

$$\begin{aligned} y_n &= B_0\left[A_0 x_n + data_0\right] \\ data_0 &= A_1 x_n + B_1 y_n + data_1 \\ &\vdots \\ data_{N-3} &= A_{N-2} x_n + B_{N-2} y_n + data_{N-2} \\ data_{N-2} &= A_{N-1} x_n + B_{N-1} y_n \end{aligned} \qquad \textbf{(3.2)}$$

V diagramu je vidět, že každý součin je před dalším postupem aktualizován, to je jedna z nevýhod tohoto algoritmu, která může proces zpomalovat a tvořit velké zaokrouhlovací chyby. Přesto má 1. kanonická forma značné výhody. Výpočet je rovnoměrně rozložen do několika operací, výstupní hodnota je dostupná hned po vypočtení první rovnice. Všechny rovnice obsahují maximálně dva součty, náchylnost k přetečení je tedy velice nízká a případně neovlivní výsledky dlouhodobě. Z těchto důvodů je smysluplné zabývat se pouze 1. kanonickou formou IIR filtru. 2. kanonická forma tyto výhody nemá a 3. a 4. formu lze realizovat kaskádním řazením těchto filtrů nižších řádů obslužným programem.

V Tab. 2 jsou podle několika kritérií srovnány jednotlivé algoritmy. Proměnná *R* vyjadřuje řád filtru. Proměnná *N* uváděná při výpočtech, v diagramech, nebo i v kompletních programech udává řád filtru zvýšený o 1. Je zavedena pro zjednodušení zadávání vstupních parametrů funkcí, jak je zřejmé z dokumentace ke knihovnám.

Tab. 2 Náročnost filtračních algoritmů

Filtr	počet aktualizací	součin	součet	velikost datového pole	počet koeficientů
FIR	1	$R+1$	$R+1$	$R+1$	$R+1$
FIR (lichá)	1	$\frac{R}{2}+1$	$R+1$	$R+1$	$\frac{R}{2}+1$
FIR (sudá)	1	$\frac{R+1}{2}$	$R+1$	$R+1$	$\frac{R+1}{2}$
IIR	1	$2(R+1)$	$2(R+1)$	$2R+1$	$2(R+1)$
IIR (kanonická)	$R+1$	$2(R+1)$	$3R+1$	R	$2(R+1)$

3.3 Programování

V těchto podkapitolách jsou uvedeny způsoby, jak z popsaných algoritmů vytvořit funkční program, který je již možno implementovat do systému s mikroprocesorem.

3.3.1 GCC

Složitější programy pro mikroprocesory je jednodušší tvořit v některém z vyšších jazyků. GCC je jedním z vhodných a je k němu dostupné softwarové vybavení. Pro hojné využívání byly knihovny vytvořeny právě v GCC.

Filtr je volán z knihovny jako funkce s návratovou hodnotou. Vstupní hodnoty zpracuje a na výstupu předá jeden vzorek filtrovaného signálu. Pro filtrování určité sekvence vzorků je nutné tuto funkci vložit do cyklu a předávat jí vzorek po vzorku. Filtry využívají několik globálních proměnných, je tak učiněno z důvodu jednoduchého přístupu k datům a zadávání vstupních parametrů. Uživatel je povinen vytvořit pole pro koeficienty a pole pro odkládání dat, uvolnit část paměti, kterou filtry potřebují pro svoji funkci. Velikosti a počet paměťových polí se filtr od filtru liší, podrobnější informace jsou uvedeny v dokumentaci. Před prvním voláním funkce je nutno ještě nastavit některé parametry pomocí funkce *f_init*. Jde pouze o přiřazení adres globálním proměnným, které deklaruje sama knihovna. Tyto proměnné jsou důležité pro funkci kruhových registrů.

Filtr FIR je nejjednodušší, ale obsahuje všechny části, ze kterých sestávají ostatní filtry. Funkci je nutno předat informaci o řádu filtru, který se bude počítat, vstupní vzorek a ukazatele na pole koeficientů a dat.

```
char FIR(char N, char fx, char *p_c, char *p_d)
{
          int fy=0;
          char i=0;

          N--;
          *p_kregx=fx;

          while(i<=N)
          {
              fy+=*(p_kregx--)**(p_c+i);
              if (p_kregx<p_d) p_kregx=p_d+N;
              i++;
          }

          p_kregx++;
          if (p_kregx>p_d+N) p_kregx=p_d;

          fy>>=6;
          fy+=1;
          fy>>=1;

          return (char)fy;
}
```

Zdrojový kód – Filtr FIR v C

Na začátku jsou vytvořeny potřebné pomocné proměnné a upravena hodnota řádu filtru. Poté je vstupní vzorek uložen do pole dat na příslušnou pozici, přepíše tak nejstarší vzorek v kruhovém registru. Následuje cyklus, ve kterém se postupně do pomocné proměnné přičte *N* posledních vstupních vzorků násobených příslušným koeficientem. Součin je přičítán přímo,

bez přetypování, proměnná *fy* je vždy typu s dvojnásobnou délkou, je tak schopna pojmout celý součin bez přetečení. Po výpočtu je ukazatel kruhového registru nastaven na nejstarší hodnotu. Nakonec je výstupní hodnota vydělena a zaokrouhlena, přetypována a předána na výstup.

Všechny funkce jsou uloženy v samostatném souboru *xxxfiltry.c*. K tomuto souboru je vytvořen ještě tzv. hlavičkový soubor, který obsahuje seznam všech funkcí, které bude možné volat externě. Vznikla nekompilovaná knihovna filtrů, ke které je možný přímý přístup a lze ji i editovat. Taková knihovna musí být do hlavního programu připojena pomocí direktivy *include* a zároveň připojena k projektu, neboť není samostatně kompilována.

3.3.2 Asembler

Tam, kde je třeba vykonat příkaz co nejrychleji, je vhodné napsat část programu v asembleru. V rozebírané funkci je první pomalou částí celý cyklus vykonávající výpočet.

Přepsání cyklu může v asembleru vypadat takto:

```
    clr   I                          I=0;
skok:                                do
    ...                              {
    inc   I                              ...
    cp    I, N                           I++;
    brcs  skok                       }
    ...                              while(I<N);
                                     ...
```

Zdrojový kód – Cyklus s podmínkou na konci

Instrukce *cp* od sebe odčítá dvě proměnné, ale výsledek nikam neukládá, pouze nastaví status registr. Zde rozhoduje nastavení carry bitu, podle jeho hodnoty dojde k větvení programu instrukcí *brcs*. Část programu tak bude vykonávána dokud $I < N$, poté se pokračuje dál. Naznačený cyklus je cyklus s podmínkou na konci, nepředpokládá se totiž nekorektní zadání. Doplněním o jednu instrukci se změní na cyklus s podmínkou na začátku. Pokud je část uvnitř dostatečně krátká, stačí užití instrukce *rjmp*, jinak je nutné použít instrukci *jmp*. Instrukce *jmp* dovolí skok delší jak 4k v programové paměti, avšak její vykonání trvá delší dobu. Takový cyklus i s náhradou v C vypadá následovně:

```
    rjmp  skok1                      I=0;
skok2:                               while(I<N)
    ...                              {
    inc   I                              ...
skok1:                               I++;
    cp    I, N                       }
    brcs  skok2                      ...
    ...
```

Zdrojový kód – Cyklus s podmínkou na začátku

Vytvoření podmíněného příkazu je podobné předchozímu příkladu, pouze skok je realizován opačným směrem. Někdy se naskytne potřeba porovnávat mezi sebou více-bytová slova, k tomu se využije instrukce *cpc*, jak je patrno z příkladu podmíněného příkazu.

Nejdůležitější částí je výpočet, který lze nazvat jako součin a akumulace. Díky hardwarové násobičce, kterou AVR obsahují, není žádný problém realizovat takový výpočet velice jednoduše.

```
    cp    IL, NL                    if I==N
    cpi   IH, NH                    {
    brne  skok                          ...
    ...                             }
skok:                               ...
    ...
```

Zdrojový kód – Podmíněný příkaz

```
    muls  A, B                      muls  A, B
    movw  CH:CL, r0:r1              add   CL, r0
                                    adc   CH, r1
```

Zdrojový kód – 8-bitová MAC

Menší komplikace nastane při výpočtech s více-bytovými čísly. Tam jsou k dispozici speciální algoritmy, které jsou podrobně popsány v [7]. Pro správné vynásobení dvou 16-bitových slov je nutné roznásobovat po částech každý byte s každým. Pokud se jedná o znaménkové násobení, horní byty nesou znaménko, proto je nutno použít pro ně určenou instrukci, jak je patrné z příkladu. Více o těchto instrukcích je uvedeno např. v [8].

```
    clr   r2                        clr   r2
    muls  AH, BH                    muls  AH, BH
    movw  CVH:CVL, r1:r0            add   CVL, r0
    mul   AL, BL                    adc   CVH, r1
    movw  CH:CL, r1:r0              mul   AL, BL
    mulsu AH, BL                    add   CL, r0
    sbc   CVH, r2                   adc   CH, r1
    add   CH, r0                    adc   CVL, r2
    adc   CVL, r1                   adc   CVH, r2
    adc   CVH, r2                   mulsu AH, BL
    mulsu BH, AL                    sbc   CVH, r2
    sbc   CVH, r2                   add   CH, r0
    add   CH, r0                    adc   CVL, r1
    adc   CVL, r1                   adc   CVH, r2
    adc   CVH, r2                   mulsu BH, AL
                                    sbc   CVH, r2
                                    add   CH, r0
                                    adc   CVL, r1
                                    adc   CVH, r2
```

Zdrojový kód – 16-bitová MAC

Jsou zde použity proměnné *A* (1. činitel), *B* (2. činitel), *C* (součin), a jejich indexy *H* (horní byte), *L* (dolní byte), *V* (vyšší word). Oproti jazyku C tento algoritmus neznamená žádnou úsporu, v překladači jsou totiž tyto algoritmy již implementovány.

V mnoha aplikacích není nutné rozlišení 16 bitů. Řada AD převodníků je pouze 10 nebo 12-bitových. Pro tyto případy je možné algoritmus mírně zjednodušit a ušetřit několik instrukcí. Příkladem může být násobení s 24-bitovým výsledkem, činitelé tedy nesmí přesahovat hodnoty vyjádřitelné 12 bity. Při nedodržení této podmínky nejsou výsledky správné.

Podobně lze podle potřeb algoritmy upravit i pro 16-bitový výsledek, nebo např. pro součin znaménkového čísla s neznaménkovým.

Nakonec je nutno výsledek převést do stejného tvaru jako je vstupní vzorek, jde v podstatě o několika-násobný posun a zaokrouhlení. V asembleru vždy stačí registry posunout maximálně jen o 4 bity a zbytek realizovat kopírováním nebo přesunutím jednotlivých 8-bitových registrů. Konkrétní aktualizační algoritmus pro 8-bitový výsledek

posune vstupní hodnotu o jeden bit doleva a uvažuje se pak horní byte. Znaménko zůstane zachováno v případě, že je správná hodnota vyjádřitelná 8 bity v dvojkovém doplňku.

```
muls  AH, BH
mov   CVL, r0
mul   AL, BL
movw  CH:CL, r1:r0
mulsu AH, BL
add   CH, r0
adc   CVL, r1
mulsu BH, AL
add   CH, r0
adc   CVL, r1
```

```
clr   r2
muls  AH, BH
add   CVL, r0
mul   AL, BL
add   CL, r0
adc   CH, r1
adc   CVL, r2
mulsu AH, BL
add   CH, r0
adc   CVL, r1
mulsu BH, AL
add   CH, r0
adc   CVL, r1
```

Zdrojový kód – 16-bitová MAC s 24-bitovým výsledkem

Zaokrouhlování je realizováno přičtením 0,5 a ořízntutím desetinné části, resp. přičtením 1 k dvojnásobné hodnotě a celočíselným vydělením 2. V asembleru je rychlejší otestovat konkrétní bit v desetinné části a případně zvýšit výsledek o 1.

```
lsl   AL
rol   AH
sbrc  AL, 7
inc   AH
mov   B, AH
```

Zdrojový kód – Aktualizace a zaokrouhlení

3.3.3 Inline assembler

Když je třeba v GCC využít části programu psané v asembleru, je nutné tak učinit pomocí tzv. inline assembleru. Je to funkce GCC, která má mimo jiné jako vstupní parametry právě asemblerovské instrukce. Každá tato funkce sestává ze 4 základních částí oddělených dvojtečkou.

1. instrukce asembleru
2. výstupní proměnné
3. vstupní proměnné
4. clobber list

Jednotlivé instrukce jsou identické s AVR instrukční sadou. Každá instrukce je uzavřena se svými operandy v uvozovkách. Pokud následuje další instrukce, musí na předchozím řádku být sled symbolů nového řádku *\n\t* opět uzavřený v uvozovkách. Každý řádek příslušející k této funkci musí být ukončen zpětným lomítkem, jinak překladač hlásí chybu a kód nepřeloží.

Operandy mohou být buď přímo použité registry značené standardně *r0, r1 ... r31*, nebo vstupní a výstupní proměnné. Tyto proměnné jsou označovány symbolem *%*, po němž bezprostředně následuje pořadí proměnné z 2. a 3. části číslované od 0. Pokud je některá proměnná vícebytová, pak se k jednotlivým bytům přistupuje pomocí písmene předřazenému číslici, *A* přitom značí nejnižší byte. Při překladu jsou tyto znaky nahrazeny registry, které byly při volání *asm volatile* přiřazeny jednotlivým proměnným.

Tab. 3 Identifikátory proměnných

Identifikátor	Použití	Rozsah
a	jednoduché horní registry	r16 až r23
b	základní ukazatelové páry	Y, Z
d	horní registry	r16 až r31
e	ukazatelové páry	X, Y, Z
G	konstanta s plovoucí desetinnou čárkou	0,0
I	6-bitová kladná celá konstanta	0 až 63
J	6-bitová záporná celá konstanta	-63 až 0
K	celá konstanta	2
L	celá konstanta	0
l	spodní registry	r0 až r15
M	8-bitová celá konstanta	0 až 255
n	16-bitová celá konstanta	
N	celá konstanta	-1
O	celá konstanta	8, 16, 24
P	celá konstanta	1
q	pár ukazatele zásobníku	SPH:SPL
r	jakýkoliv registr	r0 až r31
t	odkládací registr	r0
v	32-bitová celá konstanta	
w	vybrané horní registrové páry	r24, r26, r28, r30
x	ukazatelový pár X	X
y	ukazatelový pár Y	Y
z	ukazatelový pár Z	Z

Veškeré proměnné musí být před výkonem funkce deklarovány i přes to, že jsou to proměnné výstupní. Funkce *asm volatile* sama proměnné nedeklaruje. Před vykonáním jednotlivých instrukcí jsou proměnným přiřazeny registry, u vstupních proměnných jsou do nich z paměti zapsány jejich hodnoty, u výstupních nikoli, jejich obsah je tedy nepředvídatelný. Na konci jsou registry vstupních proměnných zapomenuty a do výstupních proměnných je zapsán obsah příslušných registrů. Některé instrukce neumí pracovat se všemi pracovními registry, je proto nutné překladači předem oznámit, z jaké oblasti má pro každou proměnnou registr vybírat. Každé proměnné se musí do uvozovek přiřadit typ registru označený identifikátorem podle Tab. 3. Za toto označení se pak do závorky uvádí proměnná registru příslušející. Některé identifikátory mohou být doplněny modifikátorem podle Tab. 4, který určuje možnost čtení z registru a zápisu do něj. Výstupním proměnným je přímo nutné nastavit možnost pouze zápisu. Pokud je nutné některou proměnnou použít jako vstupní a zároveň výstupní, je třeba ji uvést jak v části 2, tak v části 3 s tím rozdílem, že ve vstupní části se jí nedefinuje typ registru, ale odkáže se pořadovým číslem na proměnnou výstupní.

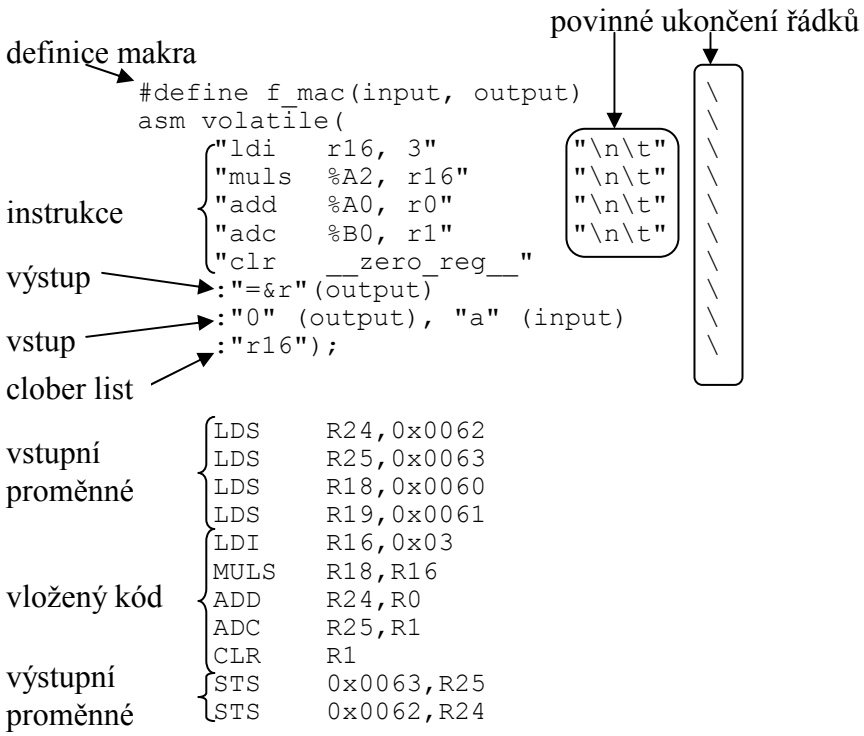
Tab. 4 Modifikátory proměnných

Modifikátor	Popis
=	operand pouze pro zápis, užito u každé výstupní proměnné
+	operand pouze pro čtení, není podporováno v inline asm
&	registr smí být použit pouze pro výstup

Clobber list umožňuje překladač informovat o důležitých událostech. Je to nepovinná součást a nejčastěji se využívá jako seznam využitých registrů. Informuje tedy překladač, že před vykonáním instrukcí si musí zálohovat data v uvedených registrech, neboť jsou v následujícím kódu přepsány. Výrazné využívání clobber listu se nedoporučuje, neboť to omezuje překladač v jeho možnostech optimalizace a může zpomalovat program, nebo dokonce způsobit jeho nekompilovatelnost. Registry *r0* a *r1* jsou výjimkou, tyto registry není třeba uvolňovat, jsou totiž všeobecně využívány jako odkládací registr *__tmp_reg__* a nulový registr *__zero_reg__*. Nicméně právě tyto registry mohou způsobovat některé nepříjemnosti, zvláště nulový registr. Jak napovídá jeho název, je v něm permanentně uložena nula, která je využívána při některých operacích. Kolize nastává při součinu, kdy jsou *r0* a *r1* výstupními registry, nulový registr se tedy může přepsat nenulovou hodnotou. Před ukončením vložené inline funkce je bezpodmínečně nutné tento registr vynulovat, jinak s velkou pravděpodobností při běhu programu způsobí nepředvídatelné chyby, které se jen těžko identifikují.

Důležité při programování v asembleru je využití podmíněných skoků na místa označená labely. Tvorba labelů v inline assembleru je identická se standardním asemblerem, s malými doplňky. Za každý label je nutno připsat znaky %=, ukončit dvojtečkou a celý jej uzavřít do uvozovek. Znaky %= budou při překladu nahrazeny konkrétní číselnou hodnotou, je tak zaručena nezaměnitelnost. Dále je vhodné celou funkci postavit mimo program a předřadit jí direktivu *define* s názvem a seznamem proměnných, vznikne makro, které se v programu už jen zavolá. Program se tím výrazně zpřehlední a je jednoduše umožněno stejnou funkci použít na více místech.

Jednotlivá pravidla jsou dobře patrná z příkladu. Je zde uveden i překlad z disassembleru, kde jsou zřetelné jednotlivé části.



Zdrojový kód – Inline assembler a výpis z disassembleru

4 Vlastnosti implementovaných funkcí

V této kapitole jsou uvedeny nejdůležitější vlastnosti filtračních funkcí z hlediska programové realizace i jejich vlastností, převážně nedostatků, z důvodu omezení výpočetní kapacity. V některých případech jsou uvedena možná vylepšení, kterými je možné tyto funkce dodatečně doplnit.

4.1 Formáty vstupních a výstupních dat

Filtrační funkce je v detailu pouhý výpočetní algoritmus realizovaný za pomoci jednoho, nebo několika cyklů pracujících s číselnými hodnotami. Je tedy nutno předem určit, co a jak budou které hodnoty reprezentovat.

Pro vyjádření celých čísel se v informačních technologiích využívá převážně dvojkového doplňku. Znamená to, že poslední bit má funkci znaménka. Rozsah kladných hodnot se zřejmě zmenší na polovinu, druhá polovina se stane zápornými čísly. Sčítání (odčítání) takto reprezentovaných čísel není z bitového pohledu žádný problém, přetečení registru znamená změnu znaménka. Opravdové přetečení hodnot ve dvojkovém doplňku není indikováno příznakem carry, je nutno jej rozpoznat jiným způsobem.

Předpokladem je, že vstupní signál, který je určen k filtraci pochází z AD převodníku nebo má stejný nekódovaný formát. Vstupní vzorky musí být ve dvojkovém doplňku s rozlišením nejvíce 8 případně 16 bitů. Tato data jsou přepočítána podle vybraného algoritmu a na výstup je uložena hodnota stejného tvaru, jaký je požadován na vstupu.

Desetinná čísla v číslicovém zpracování signálů DSP se vyjadřují ve formátu s pevnou desetinnou čárkou. Celé slovo je rozděleno na celou a desetinnou část, předem je dáno, jak velké tyto části budou. Ve většině případů se používají čísla, jejichž hodnoty se pohybují v rozmezí od -1 do 1. Takové hodnoty se vyjadřují ve zlomkovém tvaru ve dvojkovém doplňku, což je formát s pevnou desetinnou čárkou, kdy MSB vyjadřuje znaménko a zbytek je desetinná část. Zlomkový tvar ve dvojkovém doplňku obsáhne hodnoty z intervalu $\langle -1,1)$. V tomto tvaru jsou udávány hodnoty filtračních koeficientů.

V GCC lze desetinná čísla zapsat pouze do proměnné typu *float* nebo *double* případně *extended double*. Tyto typy jsou s plovoucí desetinnou čárkou a jejich délka je 32 a 64, případně 80 bitů, což je pro DSP použití až zbytečně přesné a především výpočty s nimi zdlouhavé. Tato situace je vyřešena převedením desetinného čísla na číslo celé a to vynásobením 2^{n-1}, kde n je délka slova v bitech. Tato změna ve výsledku znamená pouze změnu v zacházení překladače s proměnnými, dále s nimi zachází přesně jako s celými čísly, avšak binárně je hodnota identická.

$$\begin{aligned} -0{,}2265625_D &\rightarrow -0{,}2265625 \cdot 128 = -29_D \\ 11100011_B &\rightarrow 11100011_B \end{aligned} \qquad \textbf{(4.1)}$$

Při násobení dochází ke zdvojnásobení bitové hloubky výsledku, činitel 2^{n-1} se tak umocní na $2^{2(n-1)}$, je tedy nutno součin pokaždé zpětně vydělit hodnotou 2^{n-1}. Dělení je nejrychleji realizováno bitovým posuvem, viz operace aktualizace a zokrouhlení. Instrukční sada AVR obsahuje skupinu instrukcí, která je určena právě pro násobení čísel ve zlomkovém tvaru. Jsou to instrukce *fmul*, *fmuls*, *fmulsu*. Rozdílem oproti standardním instrukcím pro

násobení je pouze přidaný bitový posuv o jeden bit doleva. Tyto instrukce lze v jistých případech použít pro větší komplexnost výpočtů, avšak zde bylo použito standardních instrukcí *mul*, *muls* a *mulsu*. Při převodu na správnou hodnotu je tedy nutno výsledek součinu vydělit 2^{n-1}, za použití instrukcí pro násobení zlomkového tvaru by bylo nutno dělit 2^n.

4.2 Délka výpočtu

Pro zpracování běžných signálů, jako například zvuk, je třeba aby, výpočty probíhaly co nejrychleji. Program je nutno optimalizovat a vybrat nejrychlejší možné způsoby. Nejprve je vhodné zjistit, které části algoritmu proces výpočtu nejvíce zpomalují. Na následujícím schématu jsou rozepsány počty cyklů k jednotlivým operacím.

```
                                                    počet cyklů

char FIR(char N, char fx, char *p_c, char *p_d)        47
{
          int fy=0;                                     4
          char i=0;                                     2

          N--;                                          5
          *p_kregx=fx;                                  8
-------------------------------------------------------------------
          while(i<=N)                                  10
          {                                                    }
              fy+=*(p_kregx--)**(p_c+i);               55      } 105 x (N-1)
              if (p_kregx<p_d) p_kregx=p_d+N;          27
              i++;                                     13
          }                                                    }
-------------------------------------------------------------------
          p_kregx++;                                   10
          if (p_kregx>p_d+N) p_kregx=p_d;              10
-------------------------------------------------------------------
          fy>>=6;                                      16      }
          fy+=1;                                       10      } 36
          fy>>=1;                                      10      }
-------------------------------------------------------------------
          return (char)fy;                             32
```

Zdrojový kód – Rozpis náročnosti příkazů FIR filtru

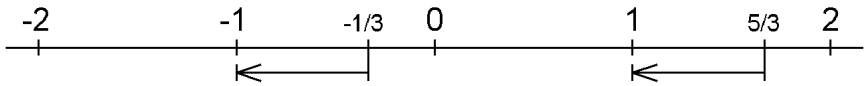
Pokud již není cesty jak zrychlit program v GCC, je ještě možnost nahradit pomalé části makrem napsaným v asembleru. Jak je vidět ze schématu, nejvíce času zabere numerický výpočet, celý cyklus se navíc opakuje podle řádu filtru. Nahrazením tohoto cyklu makrem v inline assembleru lze počet cyklů snížit více jak 4x. Další část, kterou lze výrazně urychlit je aktualizace výstupní hodnoty. Tato část je u filtru FIR nevýrazná, optimalizace nabývá účinku až u kanonické formy IIR filtru, kde je aktualizace zahrnuta i v hlavním cyklu.

I celou funkci lze napsat v asembleru, uspoří se tak nejvíce času. Ovšem oproti kombinovanému jazyku taková úspora už není rapidní. Předávání parametrů při zavolání nebo předávání výstupních parametrů zůstává.

V některých případech z návrhu koeficientů vyjdou některé nulové. Při filtraci střídavých signálů jsou i vstupní vzorky občas nulové. Z těchto předpokladů je zřejmé, že situace, kdy je jeden součinitel nulový, vůbec nemusí být ojedinělá. Lze tedy ošetřením této situace přeskočit celý výpočet a vše zrychlit. Pokud nejsou použity nulové koeficienty, tato podmínka výpočet naopak zpomalí.

4.3 Zaokrouhlování

Číselné hodnoty je možné uchovávat a předávat s určitou omezenou bitovou hloubkou, jsou tak kvantovány. Tyto nepřesnosti mohou způsobovat větší nebo menší odchylky od ideálního stavu.



Obr. 9 Číselná osa

Jak již bylo řečeno, určitou minimalizací těchto chyb je aktualizování výsledku až po provedení celého výpočtu. Součiny se přičítají v plné přesnosti do proměnné, která má dostatečnou velikost a na konci se celá tato hodnota aktualizuje vydělením 2^{n-1}. Toto dělení se provádí výhodně bitovým posuvem, to ovšem s sebou nese některá úskalí. Jak je zřejmé z číselné osy, jsou hodnoty zaokrouhleny jedním směrem. Kladná čísla dolů a záporná nahoru.

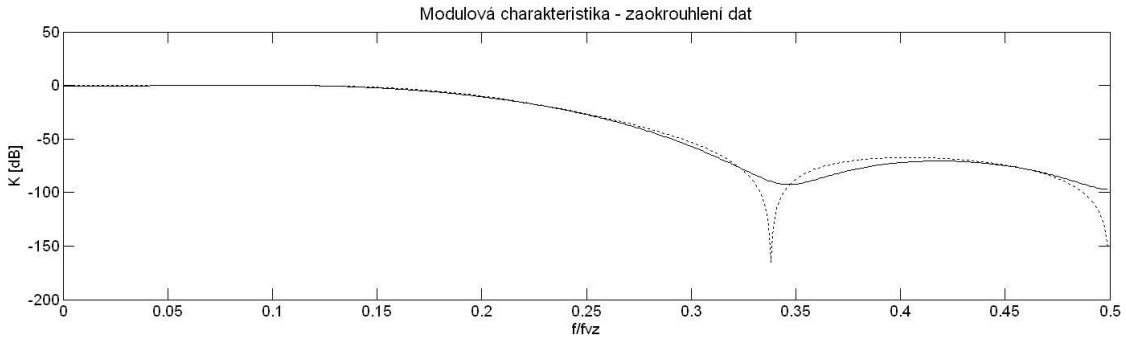
```
y>>=6;                                  lsl   AL
y+=1;                                   rol   AH
y>>=1;                                  sbrc  AL, 7
                                        inc   AH
                                        mov   B, AH
```

Zdrojový kód – Realizace zaokrouhlení

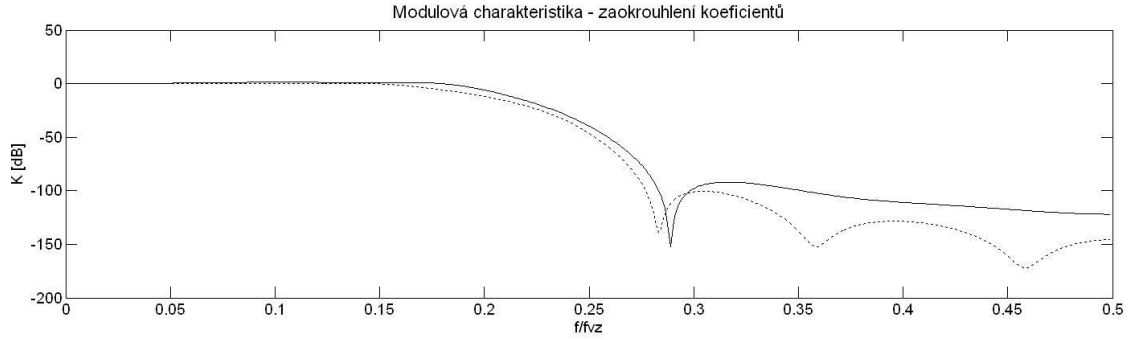
$$
\begin{aligned}
y = 16587 \rightarrow \frac{y}{2^6} = 259 \rightarrow 259 + 1 = 260 \rightarrow \frac{260}{2} = 130 \\
\frac{y}{2^7} = 129{,}58... \cong 130 \\
y = -16587 \rightarrow \frac{y}{2^6} = -260 \rightarrow -260 + 1 = -259 \rightarrow \frac{-259}{2} = -130 \\
\frac{y}{2^7} = -129{,}58... \cong -130
\end{aligned}
\qquad \textbf{(4.2)}
$$

Tento problém je vyřešen jednoduchým algoritmem zaokrouhlování, kde se testuje nejvyšší zanedbávaný bit a případně se výsledek zvýší o 1. Způsob je to jednoduchý, ale podává přesné výsledky. IIR filtry pak podávají výrazně přesnější výsledky. Zaokrouhlování výsledků má velký vliv především na zpětnovazební filtry, kde jsou výstupní hodnoty ještě několikrát použity při výpočtu. Kanonická forma využívá průběžného ukládání výsledků, je proto nutno podělit a zaokrouhlit každý součin zvlášť. Nelze tak využít minimalizaci zaokrouhlovací chyby přičítáním do větší pomocné proměnné jako u přímých realizací. Kanonická forma proto podává horší výsledky způsobené právě častým zaokrouhlováním mezivýsledků. Na Obr. 10 je znázorněno ovlivňování zaokrouhlování u filtrů IIR.

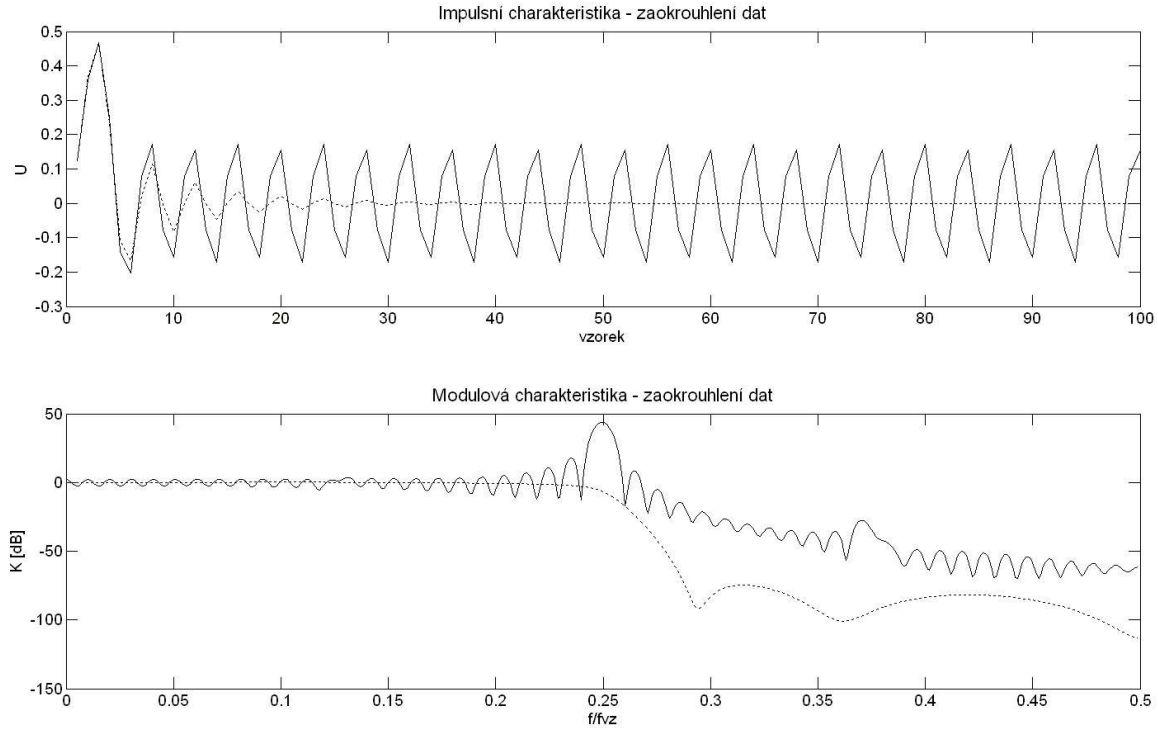
Omezeným kvantováním jsou ovlivněny i vstupní parametry. Filtrační koeficienty je nutno vypočítat externě například pomocí matlabu. Tyto hodnoty jsou vypočteny s velkou přesností a je třeba je zaokrouhlit. Vyjádření koeficientů v 16-bitovém rozlišení nezpůsobí prakticky žádné odchylky. Při použití 8-bitové hloubky je ovšem zaokrouhlení značné a skutečná charakteristika pak může být neúnosně deformována. Příklad, jak zaokrouhlení ovlivní modulovou frekvenční charakteristiku, je na Obr. 11, je naznačena i charakteristika nezaokrouhlených koeficientů.



Obr. 10 Modulová frekvenční charakteristika – zaokrouhlení dat (čárkovaně ideální filtr)



Obr. 11 Modulová frekvenční charakteristika – zaokrouhlení koeficientů (čárkovaně ideální filtr)



Obr. 12 ICH a modulová frekvenční charakteristika nestabilního filtru (čárkovaně ideální filtr)

Toto zaokrouhlování může u IIR filtrů způsobit špatnou konvergenci k nule a možné rozkmitání na jistých kmitočtech, jinými slovy vznikne nestabilní filtr. V takovém stavu se na výstupu objeví až příliš odlišné hodnoty, ty pak významně ovlivňují další počínání filtru. Příklad impulsní a modulové charakteristiky takto navrženého filtru je na Obr. 12.

V charakteristice jsou zřetelné špičky s kladným zesílením. Na těchto kmitočtech se filtr po vybuzení rozkmitá a je nepoužitelný. Tyto problémy jsou fatální při 8-bitovém rozlišení a především u vyšších řádů filtru. Filtry nižších řádů si pamatují méně výstupních hodnot a pravděpodobnost nestability je tak menší. Každý návrh koeficientů je nutno náležitě ověřit a je lepší se vyhnout filtrům vyšších řádů. Tyto filtry mívají větší možnosti co do tvaru frekvenční charakteristiky, nicméně za použití kvalitních návrhových systémů lze vytvořit uspokojivé filtry i velmi nízkých řádů. 16-bitové rozlišení trpí těmito problémy o poznání méně, přesnost vyjádření je dostatečná i pro vyšší řády filtrů, na úkor výpočetní rychlosti.

4.4 Přetékání registrů

Součin dvou čísel je bitové velikosti rovné maximálně součtu bitových velikostí jednotlivých činitelů, tj. součin dvou 8-bitových čísel dá výsledek maximálně 16-bitový. Tato skutečnost je ošetřena právě pomocnými proměnnými typu dvojnásobné velikosti, není tedy v tomto případě za žádných okolností možné, aby došlo k přetečení. Při součtu dvou čísel se bitová velikost může zvýšit maximálně o jeden bit nad největší bitovou velikost sčítanců, tj. součtem dvou 8-bitových čísel je možné dostat maximálně hodnotu vyjádřitelnou 9 bity. Zde může dojít k přetečení, neboť taková situace je považována za krajní a testy potvrdily, že k němu v praxi dochází vážně jen zřídka. Z reálných koeficientů totiž jen málo dosahuje hodnot, kterými se blíží ke svému maximu, navíc jejich znaménka se často střídají. Když už dojde k situaci, kdy by došlo k přetečení, hodnota celkového výsledku je stále příliš vysoká a k přetečení by došlo i na výstupu, jinak řečeno, takový filtr by měl v jisté části přenos větší jak 1. Tato situace je nepřípustná.

Jednotlivé filtrační algoritmy jsou ze své podstaty různě náchylné k přetékání, je to i jejich posláním, mít v různých případech různé vlastnosti. FIR filtry jsou k přetékání méně náchylné než IIR, ale to jen z důvodu, že neobsahují zpětnovazební větev a přičítá se tak menší počet vzorků. Více manipulovat lze s filtry IIR, přímá realizace je k přetékání dosti náchylná, byla tedy vytvořena i 1. kanonická forma. Ta je k přetékání méně náchylná, neboť každý vzorek vznikne součtem pouze tří hodnot. Navíc u IIR filtrů je ve zpětnovazební větvi zabudován jeden koeficient navíc. Tímto speciálním koeficientem lze korigovat poměr možnosti přetečení a přesnosti. Je to kladná celá hodnota, kterou jsou vyděleny všechny uchovávané vzorky, výstupní hodnoty jsou tímto koeficientem opět vynásobeny. Vydělení je aplikováno přímo do koeficientů, tzn. že to program nezpomalí, je to zásah pouze do vstupních hodnot, ne do běhu programu.

```
y+=A*B;                                  y+=(A*B)>>3;

y>>=6;                    ──────────►    y>>=3;
y+=1;                                    y+=1;
y>>=1;                                   y>>=1;
```

Zdrojový kód – Modifikace pro zamezení přetékání

Nicméně pro extrémní využití je možné filtry mírně upravit, aby byly na přetékání méně náchylné. Pomocná proměnná, do které se postupně přičítají součiny, se na konci přetypuje na poloviční bitový rozsah, uchovává tedy hodnoty až se zbytečnou přesností. Je možné

jednotlivé hodnoty přičítat již částečně vydělené a uvolnit tak několik vyšších bitů. Postup je zřejmý z příkladu výše.

Užitečným vylepšením programu je alespoň detekce přetečení, kdy je o přetečení možné informovat okolí a označit vzorek za irelevantní. Již bylo řečeno, že to není možné pomocí příznaku přenosu díky dvojkovému doplňku. Způsob, jakými lze detekovat přetečení, je následující.

```
    clr   ERR
    movw  r16:r17, A
    add   AL, BL
    adc   AH, BH
    eor   r17, BH
    brmi  skok
    eor   BH, AH
    brpl  skok
    ser   ERR
skok:
```

Zdrojový kód – Detekce přetečení

K přetečení může dojít pouze při součtu dvou hodnot se stejnými znaménky. Tato situace je testována exklusivním součtem. Pokud jsou znaménka stejná, projeví se to nulovým znaménkovým bitem, tedy výsledná hodnota je kladná. Pokud je tato podmínka splněna, znaménko výstupu musí být shodné se vstupními hodnotami, otestujeme opětovným provedením exklusivního součtu. Výsledek musí být opět kladný, je-li záporný, došlo k přetečení. Tento stav je možno indikovat nastavením příznakové proměnné. Dále není nutno počítat, neboť výsledek bude nesprávný. U IIR filtrů bude nesprávných i několik následujících hodnot v závislosti na řádu filtru. Je tedy zřejmé, že i méně časté přetékání může způsobit nemalé komplikace.

O přetečení je vhodné informovat obslužný program pomocí nějakého příznaku. V GCC se pro tyto účely často využívá návratová hodnota funkce. Pokud je vše v pořádku, funkce vrátí na výstup nulu, jestliže se vyskytne nějaký problém, na výstupu se objeví nenulová hodnota, často -1. Tento způsob nelze v uvedených funkcích přímo implementovat, neboť návratová hodnota je již obsazena výstupní hodnotou filtru. V případě nutnosti je nejvhodnějším a schůdným způsobem vracet výstupní vzorky jinak. Zavést další vstupní proměnnou, ukazatel na volnou paměťovou pozici či proměnnou, kam si uživatel přeje výstupní hodnotu uložit. Uložení proběhne ještě v použité funkci a návratová hodnota bude uvolněna pro chybové hlášení.

4.5 Shrnutí vlastností jednotlivých filtrů

V této podkapitole jsou stručně srovnány jednotlivé filtrační algoritmy, jsou uvedeny základní výhody a nevýhody, jež podstatně ovlivňují při výběru.

Tab. 5 Srovnání filtračních algoritmů

FIR	X	IIR
+ Výpočet trvá kratší dobu, neboť neobsahuje zpětnovazební větev. + Program je kratší. + Jednoduchý návrh koeficientů. − Pro kvalitní filtraci je nutno použít filtr vyššího řádu.		+ Poskytuje větší možnosti filtrace a různorodé frekvenční charakteristiky. + Pro kvalitní filtraci postačí filtry nízkého řádu. − Výpočet filtračních koeficientů je odkázán na složité algoritmy a je vhodné je získat externě. − Velká zaokrouhlovací chyba může způsobit nestabilitu filtru.

FIR přímá realizace	X	FIR se symetrickou ICH
+ Má neomezenou frekvenční charakteristiku. − Algoritmus není nijak matematicky zjednodušen, výpočet je tedy delší.		+ Výpočet je matematicky zjednodušen a teoreticky se zkrátí. − Součet, který se provádí před násobením, díky uvedenému matematickému zjednodušení, může způsobit přetečení. Je tedy nutno jej ošetřit složitějším postupem a to tento algoritmus naopak zpomaluje.

IIR přímá realizace	X	IIR 1. kanonická forma
+ Je zde zaručena maximální přesnost výpočtu minimalizováním zaokrouhlovacích chyb. − Složitý výpočet. − Větší pravděpodobnost přetečení.		+ Klade menší nároky na paměťový prostor, poloviční oproti přímé realizaci. + Pravděpodobnost přetečení není rovnoměrná, nýbrž roste ke konci řetězce. Případná chyba způsobená přetečením registru nemusí ovlivnit velký počet následujících vzorků. − Zaokrouhlovací chyba ovlivňuje každý mezivýsledek, celková chyba je proto dosti velká.

5 Fourierova transformace

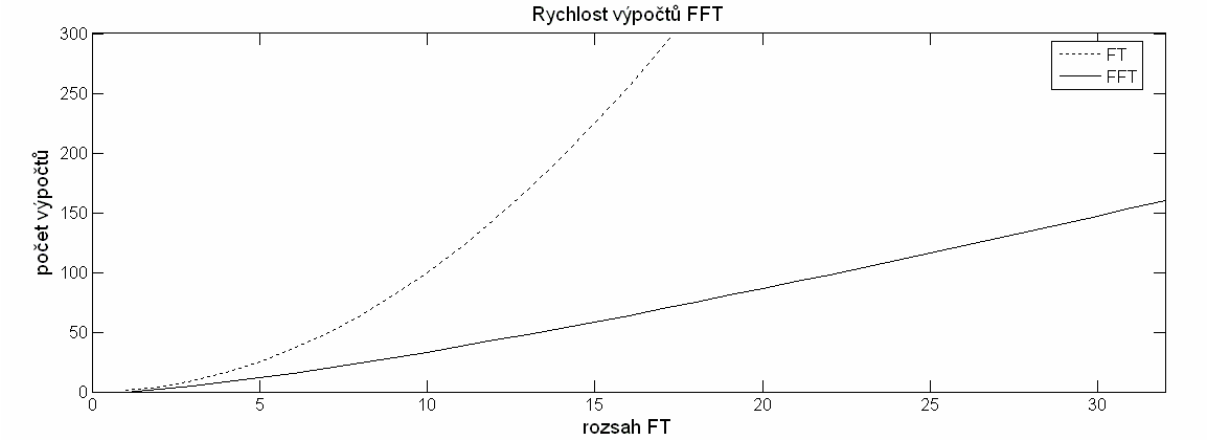
Uvedené matematické postupy využívají i jiné algoritmy užívané k jiným účelům. Takovým algoritmem je i fourierova transformace (FT). Pro číslicové signály se pak používá diskrétní fourierova transformace (DFT). Její výpočet je ovšem velice náročný a pro mnoho systémů by byl nepoužitelný. Bylo tedy vypracováno několik postupů, které výpočet urychlí a do určité míry i zjednoduší, takové algoritmy jsou označovány jako rychlá fourierova transformace (FFT), případně diskrétní rychlá fourierova transformace (DFFT).

Algoritmus DFT přiřazuje číslicovému signálu určité délky jeho spektrum. Základní tvar DFT má tvar (5.1) [12].

$$\begin{aligned} h_a &= DFT(g_b) \\ &= \sum_{b=0}^{N-1} g_b e^{-\frac{2\pi iab}{N}} \quad , 0 \le a \le N-1 \\ &= \sum_{b=0}^{N-1} g_b W_N^{ab} \quad , W_N = e^{-\frac{2\pi i}{N}} \end{aligned} \tag{5.1}$$

Existuje více postupů, jak zjednodušit tuto rovnici pro konkrétnější případy, jedním z nich je tzv. Cooley-Turkey radix-2 algoritmus. Jeho podstatou je rozdělení celé rovnice na malé části, sub-transformace. Radix-2 znamená, že je využíváno takového počtu prvků, vstupních vzorků a tedy i výstupních vzorků, který je vyjádřitelný přirozenou mocninou 2, neboli $N = 2^n, n \in N$. Tento počet umožňuje rozdělit základní rovnici na nejmenší možné části, postup je naznačen v (5.2) [12].

$$\begin{aligned} h_a &= \sum_{b=0}^{N-1} g_b W_N^{ab} \\ &= \sum_{b=0}^{\frac{N}{2}-1} g_{2b} W_{\left(\frac{N}{2}\right)}^{ab} + W_N^a \sum_{b=0}^{\frac{N}{2}-1} g_{2b+1} W_{\left(\frac{N}{2}\right)}^{ab} \\ DFT(h) &= DFT(g_{sude}) + W_N^k \cdot DFT(g_{liche}) \end{aligned} \tag{5.2}$$



Obr. 13 Porovnání rychlosti FT a FFT

Tímto se počet výpočetních cyklů snížil z N^2 na $2\left(\frac{N}{2}\right)^2$. Analogickým rozdělením na nejmenší části se dosáhne snížení až na $N\log_2 N$. Úspora výpočtu je zřejmá z Obr. 13.

K dalšímu zjednodušení je vhodné postupovat pomocí binárního vyjádření indexů *a* a *b*, podle (5.3).

$$b = [b_{n-1}\cdots b_1\ b_0] = 2^{n-1}b_{n-1} + \cdots + 2b_1 + b_0 \tag{5.3}$$

Jednotlivé bity mohou nabývat úrovně 0 nebo 1, celá rovnice se pak rozloží na (5.4) [12]. Zde je upřednostněno rozkládání pomocí indexu *b*, což naznačuje rozklad v časové oblasti. Postup proto směřuje tam, kde ve výsledku bude signál v časové oblasti rozdělen na jednotlivé 2-vzorkové části. Je možný i rozklad v kmitočtové oblasti, kde se rovnice rozkládá odlišným způsobem za pomoci indexu *a*. Konkrétní postup je uveden v [12].

$$\begin{aligned}
h(a) &= \sum_{b=0}^{N-1} g_b e^{-\frac{2\pi abi}{N}} \\
h([a_{n-1}\cdots a_1\ a_0]) &= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-1}=0}^{1} g([b_{n-1}\cdots b_1\ b_0])W_N^{ab} \\
&= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-1}=0}^{1} g([b_{n-1}\cdots b_1\ b_0])W_N^{a\left(2^{n-1}b_{n-1}+[b_{n-2}\cdots b_1\ b_0]\right)} \\
&= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-2}=0}^{1} W_N^{a[b_{n-2}\cdots b_1\ b_0]}\sum_{b_{n-1=0}}^{1} g(b)W_N^{a\left(2^{n-1}b_{n-1}\right)}
\end{aligned} \tag{5.4}$$

Výraz $W_N^{a\left(2^{n-1}b_{n-1}\right)}$, známý jako twiddle factor, lze s využitím periodicity komplexní exponenciální funkce zjednodušit podle (5.5).

$$\begin{aligned}
W_N^{a\left(2^{n-1}b_{n-1}\right)} &= \exp\left(-2\pi i\cdot\frac{[a_{n-1}\cdots a_1\ a_0]2^{n-1}b_{n-1}}{2^n}\right) \\
&= \exp\left(-2\pi i\cdot\frac{\left(2^{n-1}a_{n-1}+\cdots+2a_1+a_0\right)b_{n-1}}{2}\right) \\
&= \exp\left(-2\pi i\cdot\frac{a_0 b_{n-1}}{2}\right) \\
&= W_2^{a_0 b_{n-1}}
\end{aligned} \tag{5.5}$$

Dalším analogickým postupem vznikne vyjádření (5.6) [12].

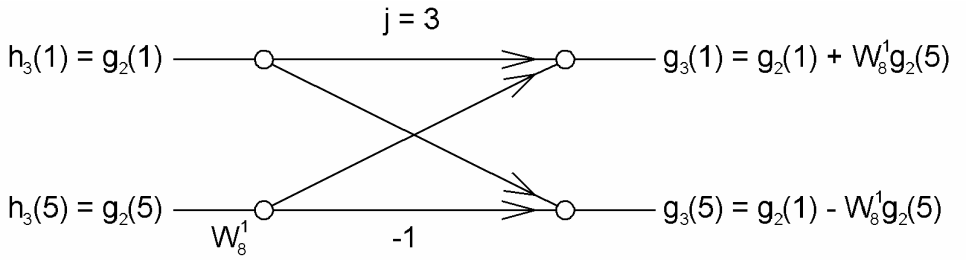
$$h([a_{n-1}\cdots a_1\ a_0]) = \sum_{b_0=0}^{1} W_N^{[a_{n-1}\cdots a_1\ a_0]b_0}\cdots\sum_{b_{n-2}=0}^{1} W_4^{[a_1\ a_0]b_{n-2}}\sum_{b_{n-1}=0}^{1} g(b)W_2^{a_0 b_{n-1}} \tag{5.6}$$

Toto vyjádření ještě neumožňuje vytvořit výpočtový algoritmus, k tomu je ještě nutno rozepsat jednotlivé sumy do více kroků.

$$
\begin{aligned}
g_1(a_0, b_{n-2}, b_{n-3}, \cdots, b_1, b_0) &= \sum_{b_{n-1}=0}^{1} W_2^{a_0 b_{n-1}}\, g([b_{n-1}\ b_{n-2}\ \cdots b_0]) \\
g_2(a_0, a_1, b_{n-3}, \cdots, b_1, b_0) &= \sum_{b_{n-2}=0}^{1} W_4^{[a_1\ a_0] b_{n-2}}\, g_1(a_0, b_{n-2}, b_{n-3}, \cdots, b_1, b_0) \\
&\vdots \\
g_n(a_0, a_1, a_2, \cdots, a_{n-2}, a_{n-1}) &= \sum_{b_0=0}^{1} W_N^{[a_{n-1} \cdots a_1\ a_0] b_0}\, g_{n-1}(a_0, a_1, a_2, \cdots, a_{n-2}, b_0)
\end{aligned}
\qquad \textbf{(5.7)}
$$

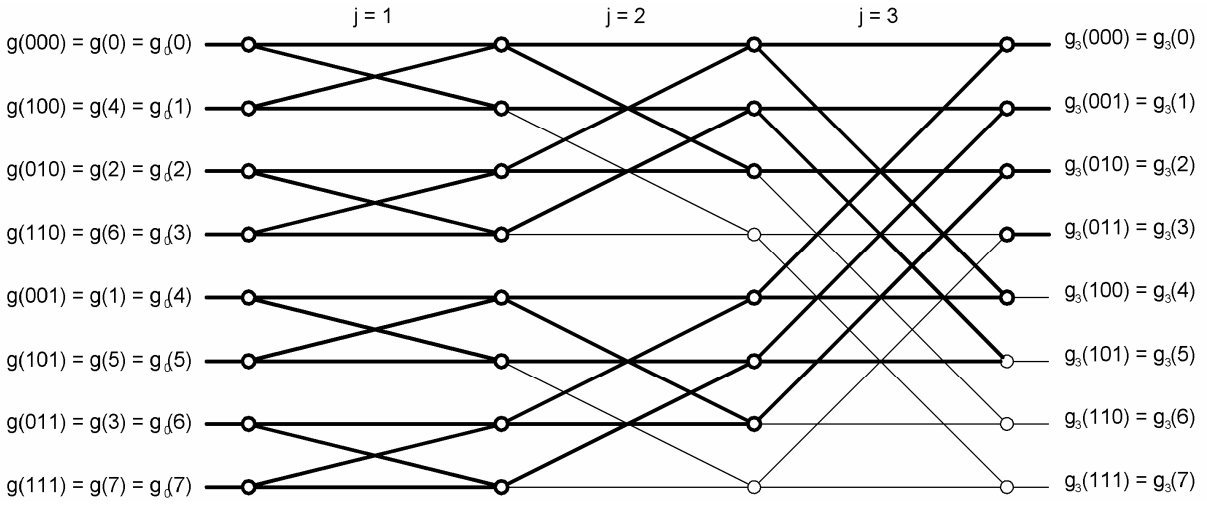
Hodnoty vzorků je zřejmě možné přepisovat přímo, tzn. vyzvednout vzorky, provést jeden krok výpočtu a výsledek uložit zpět na příslušné paměťové pozice. Výsledná posloupnost nebude ještě správná, bude seřazena v bitově reverzním pořadí, jak je zřejmé z (5.7), LSB *a* odpovídá MSB *b*. Tomu lze jednoduše předejít předřazením algoritmu bitově reverzního řazení celému výpočtu, tedy nejprve správně seřadit vstupní vzorky.

Jeden elementární blok výpočtu lze schematicky vyjádřit dle Obr. 14, tzv. motýlkem. Je zde naznačen i postup výpočtu. Vychází se ze dvou hodnot vzorků, tyto vzorky se vynásobí příslušnými koeficienty, posčítají a uloží se zpět na stejná místa. Postup je srozumitelnější z rovnic.



Obr. 14 Motýlek

Z těchto bloků se sestaví celé schéma FFT, Obr. 15 pro $n = 3$, a je přehledně vidět postup výpočtů a pozice v paměti. Toto schéma tvoří *n* sloupců, v každém sloupci je 2^{n-1} motýlků. Motýlci tvoří v každém sloupci určitý počet skupin, 2^{n-j}, kde *j* je pořadí sloupce. V každé skupině je 2^{j-1} postupně překrývajících se motýlků.



Obr. 15 Schéma DFFT pro n=3

Algoritmus výpočtu DFFT je níže, pro jednoduchost a větší srozumitelnost je algoritmus předveden jako část programu v jazyku pro matlab. Tvoří jej několik cyklů. Hlavní cyklus ovládá postup v rámci sloupců. Proměnná *j* poukazuje na právě počítaný sloupec. V každém tomto cyklu se vypočítá základ pro násobící koeficient *W*. V dalších výpočtech je počítáno s přirozenými mocninami *W*. Následuje dvojnásobný cyklus s proměnnými *a* a *b*. Proměnná *b* ve vnitřním cyklu posouvá výpočet po skupinách, její počáteční hodnota je nulová a zvyšuje se ne o 1, ale o dvojnásobek šířky motýlka, značenou proměnnou *d*. Vnější cyklus, ovládaný proměnnou *a*, se stará o jednotkový posun výpočtu v rámci jedné skupiny. Před přechodem na další sloupec je ještě nutno zdvojnásobit *d*, tj. rozšířit motýlka.

```
d=1;
for j=1:1:n                                  → výpočet koeficientů
    W=exp(-2*pi*i/(2*d));
    for a=0:1:d-1
        for b=0:2*d:N-1
            x1=h(a+b+1)+W^a*h(a+b+d+1);
            x2=h(a+b+1)-W^a*h(a+b+d+1);      } výpočet motýlka
            h(a+b+1)=x1;
            h(a+b+d+1)=x2;
        end
    end
    d=2*d;                                   → rozšíření motýlka
end;
```

Zdrojový kód – DFFT matlab

Vstupní data byla při odvozování uvažována jako imaginární, nicméně v drtivé většině je vstupními vzorky signál reálný. Tato skutečnost má zásadní dopad na spektrum takového signálu. Spektrum reálného signálu je vždy symetrické, tedy jeho využitelná část je pouze poloviční. Takto je možné algoritmus výrazně zjednodušit a urychlit výpočet téměř o polovinu. Výpočty, které jsou nutné pro reálné vstupní signály jsou v Obr. 15 naznačeny tučně.

Dále je nutno podotknout, že všechny výpočty jsou prováděny v oboru komplexních čísel, což situaci značně komplikuje. GCC natož pak asembler tyto počty nemá standardně implementovány, je proto nutno je realizovat čistě programově, čímž výrazně naroste počet prováděných matematických operací. Takový program pro matlab ukazuje Zdrojový kód – DFFT pro reálná data.

Koeficienty *W* nejsou tentokrát numericky počítány, ale jejich hodnoty jsou uloženy v paměti odkud se postupně volají. Zjednoduší to výpočet a při tom nejsou velké nároky na paměť. Hodnoty těchto koeficientů pro $n = 4$ uvádí Tab. 6.

Tab. 6 Koeficienty DFFT

koeficient	$W_2 = \exp(-i\pi)$	$W_4 = \exp\left(-i\frac{\pi}{2}\right)$	$W_8 = \exp\left(-i\frac{\pi}{4}\right)$	$W_{16} = \exp\left(-i\frac{\pi}{8}\right)$
komplexní hodnota	-1	$-i$	$0{,}7071 - 0{,}7071i$	$0{,}9239 - 0{,}3827i$

První koeficient je 1, to znamená, že není nutno při výpočtu provádět součin. Druhý koeficient je *–i*, zde se také nemusí provádět součin, pouze přeskládat hodnoty a změnit znaménka. Toto je dosti velké zjednodušení a je vhodné výpočet pro tyto koeficienty provést samostatně. Algoritmus se tak rozdělí dále ještě na 3 části. V prvním cyklu se počítá motýlek pro koeficient $W = 1$, je to jen součet a rozdíl reálných vstupních vzorků. Tento výpočet je

rychlý. Výpočet pro koeficient *–i* je prováděn v posledním cyklu. Zde se díky uvažování reálného vstupu nepočítá kompletní motýlek, ale jen jeho polovina, druhá je nepotřebná pro spektrum. Hlavní cyklus již probíhá podle standardního postupu s jediným rozdílem. Výstupní hodnoty motýlka se neukládají na stejná místa, hodnota, která by spadala do pravé poloviny sub-spektra je kvůli symetrii spektra reálného vstupu uložena na symetrickou pozici v levé polovině. Dále je nutno vypočítat *a*-tou mocninu koeficientů *W*, je tak učiněno postupným násobením v cyklu *a*. Části algoritmu jsou zapojovány postupně, zpočátku není vůbec nutné do výpočtu zahrnovat poslední dva cykly, to je ošetřeno podmínkou.

```
for j=1:1:n
    p=2^j;
    for b=0:p:N-1
        xare=hre(b+1)+hre(b+p/2+1);
        xbre=hre(b+1)-hre(b+p/2+1);
        hre(b+1)=xare;
        hre(b+p/2+1)=xbre;
    end                                   } výpočet kompletního motýlka
    if j>1
        wre=real(W(j-1));
        wim=imag(W(j-1));
        wre1=wre;                         } vyzvednutí koeficientů z paměti
        wim1=wim;
        a=1;
        for a=1:1:p/4-1
            for b=0:p:N-1
                xare=hre(b+a+1)+hre(b+p/2+a+1)*wre-him(b+p/2+a+1)*wim;
                xaim=him(b+a+1)+hre(b+p/2+a+1)*wim+him(b+p/2+a+1)*wre;
                xbre=hre(b+a+1)-hre(b+p/2+a+1)*wre+him(b+p/2+a+1)*wim;
                xbim=-him(b+a+1)+hre(b+p/2+a+1)*wim+him(b+p/2+a+1)*wre;
                hre(b+a+1)=xare;
                him(b+a+1)=xaim;
                hre(b+p/2-a+1)=xbre;
                him(b+p/2-a+1)=xbim;
            end
            xare=wre*wre1-wim*wim1;
            xaim=wre*wim1+wim*wre1;       } výpočet mocniny koeficientů
            wre=xare;
            wim=xaim;
        end
        a=p/4;
        for b=0:p:N-1
            hre(b+a+1)=hre(b+a+1)-him(b+p/2+a+1);
            him(b+a+1)=him(b+a+1)-hre(b+p/2+a+1);   } výpočet polovičního motýlka
        end
    end
end;
```

výpočet kompletního motýlka s uložením na levou polovinu spektra

Zdrojový kód – DFFT pro reálná data

V příkladu jsou použita dvě samostatná datová pole *hre* a *him*, pro reálnou a imaginární složku signálu. Po proběhnutí výpočtu je v nich uložen obraz v kmitočtové oblasti, také ve složkovém tvaru, ovšem jen některé pozice jsou využity. Pro reálné vstupní hodnoty je na vstupu imaginární část nulová, registr je tedy zbytečný. Potřebný obraz reálného signálu zabírá pouze první polovinu registru *hre* a první polovinu registru *him*. Je zřejmé, že polovina uvolněného datového prostoru je vždy nevyužita. Tento nedostatek je v kompletním programu vhodné odstranit.

Výstupní formát dat je datové pole s komplexními hodnotami spektra ve složkovém tvaru, v mnoha aplikacích jsou ale žádoucí spíše hodnoty modulu a fáze. Obraz v časové oblasti by bylo možné určit i pomocí výpočtů s komplexními hodnotami v goniometrickém

tvaru. Součin takových hodnot by byl dokonce výpočetně výhodnější, ovšem k sečtení dvou komplexních čísel v goniometrickém tvaru je zapotřebí výpočet goniometrických funkcí a to mnohonásobně. Tato cesta by nevedla ke zdárnému výsledku s největší úsporou času. Pro převod komplexních čísel ze složkového do goniometrického tvaru je opět nutno spočítat nebo alespoň znát hodnoty goniometrických funkcí. Tyto hodnoty je možné dostatečně jemně kvantovat a uložit je napevno do paměti, třeba programové. Vznikne tak tabulka hodnot goniometrických funkcí, odkud lze velice efektivně příslušné hodnoty odečítat. Tento postup se hojně využívá právě v rychlostně omezených systémech.

Důležitou částí výpočtu DFFT je algoritmus pro bitově reverzní řazení. V literatuře [12] je uveden Gold-Rader algoritmus a vypadá následovně.

```
l=0;
for j=0:1:N-2
    k=N/2;
    if j<l
        pom=h(l+1);
        h(l+1)=h(j+1);
        h(j+1)=pom;
    end
    while k<=l
        l=l-k;
        k=k/2;
    end
    l=l+k;
end
```

Zdrojový kód – Gold-Rader algoritmus

Tento postup je vhodný, ale pro programy ve vyšších jazycích, tj. v GCC, v asembleru je bitově reverzní pořadí zajištěno bitovým přesunem indexu a postupným ukládáním na již určené pozice.

Procesory AVR disponují pro výpočet DFFT dosti slabou výpočetní kapacitou, je tedy předem jasné, že použitelný program pro DFFT lze vytvořit pouze v asembleru. Příklad funkce v asembleru, která je schopna spočítat DFFT pro 16 vzorků s 8-bitovou hloubkou s opakovací frekvencí téměř 10kHz na ATmega16, je uveden v příloze i se stručným popisem jednotlivých funkčních bloků. Dále jsou uvedeny jeho menší části, které je nutno ještě podrobněji popsat.

Komplexní součin je operace, která se v programu často opakuje a její výpočet je zdlouhavý, byly proto vytvořeny dvě makra, která ošetřují tuto operaci. Pro univerzálnost a menší nároky na volné pracovní registry byl součin rozdělen na dvě samostatné části, první dává výsledek reálné části a druhá jeho imaginární část. Vstupní proměnné jsou tedy vždy 4 a výstupní 1. Makra jsou uvedena ve Zdrojový kód – Makra komplexního součinu.

```
.macro mulcpxre
          push  r16
          push  r17
          mov   r16, wre
          mov   r17, datre
          muls  r16, r17
          lsl   r0
          rol   r1
          mov   data, r1
          tst   r0
          brpl  skok10
          inc   data
skok10:   mov   r16, wim
          mov   r17, datim
          muls  r16, r17
          lsl   r0
          rol   r1
          sub   data, r1
          tst   r0
          brpl  skok11
          inc   data
skok11:   pop   r17
          pop   r16
.endmacro
```

Makro komplexního součinu – reálná část.

Vstupní hodnoty jsou *wre*, *wim*, *datre* a *datim*.
Vypočtená hodnota je uložena do proměnné *data*.
Výpočet probíhá podle následujícího výrazu.

$$DATA = \mathrm{Re}(W)\cdot\mathrm{Re}(DAT) - \mathrm{Im}(W)\cdot\mathrm{Im}(DAT)$$

```
.macro mulcpxim
          push  r16
          push  r17
          mov   r16, wim
          mov   r17, datre
          muls  r16, r17
          lsl   r0
          rol   r1
          mov   data, r1
          tst   r0
          brpl  skok12
          inc   data
skok12:   mov   r16, wre
          mov   r17, datim
          muls  r16, r17
          lsl   r0
          rol   r1
          add   data, r1
          tst   r0
          brpl  skok13
          inc   data
skok13:   pop   r17
          pop   r16
.endmacro
```

Makro komplexního součinu – imaginární část.

Vstupní hodnoty jsou *wre*, *wim*, *datre* a *datim*.
Vypočtená hodnota je uložena do proměnné *data*.
Výpočet probíhá podle následujícího výrazu.

$$DATA = \mathrm{Re}(W)\cdot\mathrm{Im}(DAT) + \mathrm{Re}(DAT)\cdot\mathrm{Im}(W)$$

Zdrojový kód – Makra komplexního součinu

Další částí je funkce, která realizuje bitově reverzní pořadí. Pomocí této funkce je nutno nejprve postupně uložit vstupní vzorky do vyhrazeného datového pole adresovaného ukazatelem *dadr*. Do proměnné *data* se vloží vstupní vzorek a zavolá se funkce *sdat*, ta si jej uloží na správnou pozici. Data tedy musí být ukládána jedno po druhém, nikoliv náhodně. Proměnné *ptrl* a *ptrh* jsou využity jako pomocné.

Bez těchto částí je funkce *fft* uvedená v příloze nefunkční. Tato funkce je volána po naplnění celého datového pole vstupními daty. Pole je předem připraveno na pojmutí komplexních dat v kmitočtové oblasti. Po proběhnutí funkce jsou hodnoty obrazu uloženy do stejného datového pole v komplexním složkovém tvaru v pořadí reálná a následně imaginární.

```
stdat:    clr   ptrl
          mov   ptrh, i
          ldi   j, n              ;opakuj n-krát
          lsr   ptrh              ;přesuň bit
          rol   ptrl
          dec   j
          brne  PC-3
          lsl   ptrl              ;proložení ukládání (Re a Im)
          clr   ptrh
          ldi   XL, low(dadr)     ;*data
          ldi   XH, high(dadr)
          add   XL, ptrl          ;ir = *data + ptrl
          adc   XH, ptrh
          st    X+, data          ;ulož na místo ir
          st    X, ptrh           ;vymaž Im
          ret
```

Zdrojový kód – Bitově reverzní řazení

Použité proměnné je nutno předem deklarovat, registry lze vybírat podle následujícího rozdělení.

registry r2 ÷r25	**registry r16 ÷ r25**
ptrl, ptrh, wre, wim, datre, datim	*i, j, p, a, b, xare, xaim, xbre, xbim, data*

Tato funkce byla otestována pro několik rozsahů. Výsledky testů udává Tab. 7.

Tab. 7 Rychlost DFFT

rozsah *n* (*N*)	**2 (4)**	**3 (8)**	**4 (16)**	**5 (32)**
počet cyklů	310	734	1715	3944
poměrná rychlost [kSa/s]	59,5	23,1	9,6	4,1

6 Závěrečné zhodnocení

Výsledky testování knihoven v příloze potvrzují, že možnosti filtrace na ATmega16 jsou omezeny. Hranice jsou ovšem dosti vysoko, aby úplně znemožňovaly jejich použití. Pomyslným měřítkem je jistě použitelnost v audio-pásmu, tedy standardně 44kSa/s. Pro tyto kmitočty je možné využít jen filtraci 8-bitových dat, a to do 5. řádu IIR. FIR filtry jsou obecně rychlejší, jejich matematický postup je jednodušší. Verze FIR filtru se symetrickou ICH je z výpočetního pohledu zdánlivě zjednodušena, ovšem součet dvou vzorků před násobením vyžaduje použití z části 16-bitového algoritmu pro váhování koeficienty, což je činí neefektivními a celá výhodnost je tak ztracena. Tyto filtry se tedy hodí spíše pro digitální zpracování na vícebitových procesorech. Přímá realizace FIR filtru je naopak vhodná pro 8-bitové AVR a nepotýká se s většími problémy. IIR filtry mají oproti FIR díky rekurzivní větvi větší filtrační možnosti, ale za cenu větší výpočetní náročnosti. Tato náročnost je až dvojnásobná oproti FIR. Přímá realizace IIR je přesná, ale to je jediná její výhoda. Při výpočtech může snadno dojít k přetečení, které ovlivní velký počet následujících vzorků. Tato nectnost je potlačena v 1. kanonické formě IIR filtru. Tento typ je také nejvhodnějším pro digitální filtraci na 8-bitových procesorech AVR. V některých případech může rychlostí předčít i filtr FIR při stejné kmitočtové charakteristice, a to díky velkým možnostem IIR i malých řádů.

Náročnost číslicových filtrů charakterizuje především počet nutných vykonání operace multiply and accumulate (přičti součin). Na AVR je ovšem implementována hardwarová násobička, která tuto operaci činí téměř maximálně rychlou. Větší zpomalení nastane, jak již bylo zmíněno u FIR se symetrickou ICH, vyskytne-li se potřeba vynásobit vícebitová čísla, pak rychlost výpočtu výrazně klesá. Z tohoto důvodu je také zpracování 16-bitových dat cca dvakrát pomalejší, avšak 16-bitové vzorky je výhodné zpracovávat rychlostí do 20kSa/s, což je poměrně dobrý výsledek.

Rychlost filtrace z programového hlediska ovlivňuje především programovací jazyk. Rozdíl mezi GCC a JSA je zřejmý. Filtry v GCC jsou prakticky nevyužitelné. Naopak kombinací JSA a GCC lze dosáhnout uspokojivých výsledků. Proto kombinace jazyků je vhodným řešením, neboť přepsáním celé funkce do JSA se dosáhne již jen minimálního zlepšení. Podstatného zrychlení se také dosáhne předáváním parametrů pomocí ukazatelů a použitím kruhového registru pro zpožďovaná data.

Rozlišení 8 bitů není příliš jemné, to se také projevilo při testech filtrů IIR. Velká zaokrouhlovací chyba vzorků může snadno způsobit nestabilitu. Na tuto skutečnost je nutno nezapomenout a při návrhu koeficienty otestovat. Správný návrh stabilitu zajistí. Zaokrouhlení koeficientů má neblahý vliv především na zkreslení kmitočtové charakteristiky. Na 16-bitové zpracování má zaokrouhlování mnohonásobně menší vliv.

Fourierovu transformaci lze výrazně zjednodušit zavedením algoritmů radix-2. Uvážením pouze reálných vstupních dat se výpočet také výrazně zkrátí. Výsledný algoritmus je už natolik jednoduchý, že je možné jej výhodně použít i na 8-bitových procesorech AVR. Testy potvrzují, že výpočet může být dosti rychlý a využití tohoto algoritmu je široké, ovšem za předpokladu programování v JSA. Mnohem jednodušší je DFFT aplikovat v GCC, ale pouze pro příležitostné použití, kde nebude záviset na rychlosti. Hloubka 8 bitů je malá, ale v mnoha případech podá dostačující informace o spektru analyzovaného signálu. Takový algoritmus se dá využít například jako orientační spektrální analyzátor pro měřicí přístroje nebo i efektní zobrazovač spektra v audiotechnice.

Literatura

[1] ZÁPLATA, F. *Digitální filtrace na 8-bitových procesorech: semestrální projekt.* Brno: FEKT VUT v Brně, 2009. 19 s., 3 příl.

[2] JAN, Jiří. *Číslicová filtrace, analýza a restaurace signálů*. 2. upravené a rozšířené vydání, brož. Brno: Vysoké učení technické v Brně, VUTIM 2002. 429 stran. ISBN 80-214-2911-9.

[3] VÍCH, Robert; SMÉKAL Zdeněk. *Číslicové filtry*. 1. vydání. Praha: Academia 2000. 220 stran. ISBN 80-200-0761-X.

[4] MATOUŠEK, David. *C pro mikrokontroléry ATMEL AT89S52 6. díl.* 1. vydání. Praha: BEN technická literatura 2007. 240 stran. ISBN 978-80-7300-215-2

[5] PALACHERLA, Amar. *Implementing IIR Digital Filters* [online]. Microchip application note AN540 [cit. 15. srpna 2008]. Dostupný z www: <http://ww1.microchip.com/downloads/en/AppNotes/00540c.pdf>

[6] *Digital filters with AVR* [online]. ATMEL application note AVR223 [cit. 30. října 2008]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc2527.pdf>

[7] *Using the AVR® hardware multiplier* [online]. ATMEL application note AVR201 [cit. 30. října 2008]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/DOC1631.PDF>

[8] *8 bit AVR Instruction set* [online]. ATMEL [cit. 2. října 2008]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf>

[9] *ATmega16* [online]. ATMEL datasheet [cit. 15. srpna 2008]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf>

[10] LAND, Bruce R. *Fixed point mathematical functions in Codevision C and assembler* [online]. Cornell University Electrical Engineering 476 [cit. 15. srpna 2008]. Dostupný z www: <http://instruct1.cit.cornell.edu/courses/ee476/Math/index.html>

[11] *Fixed point DSP functions in Codevision C and assembler* [online]. Cornell University Electrical Engineering 476 [cit. 15. srpna 2008]. Dostupný z www: <http://instruct1.cit.cornell.edu/courses/ee476/Math/avrDSP.htm>

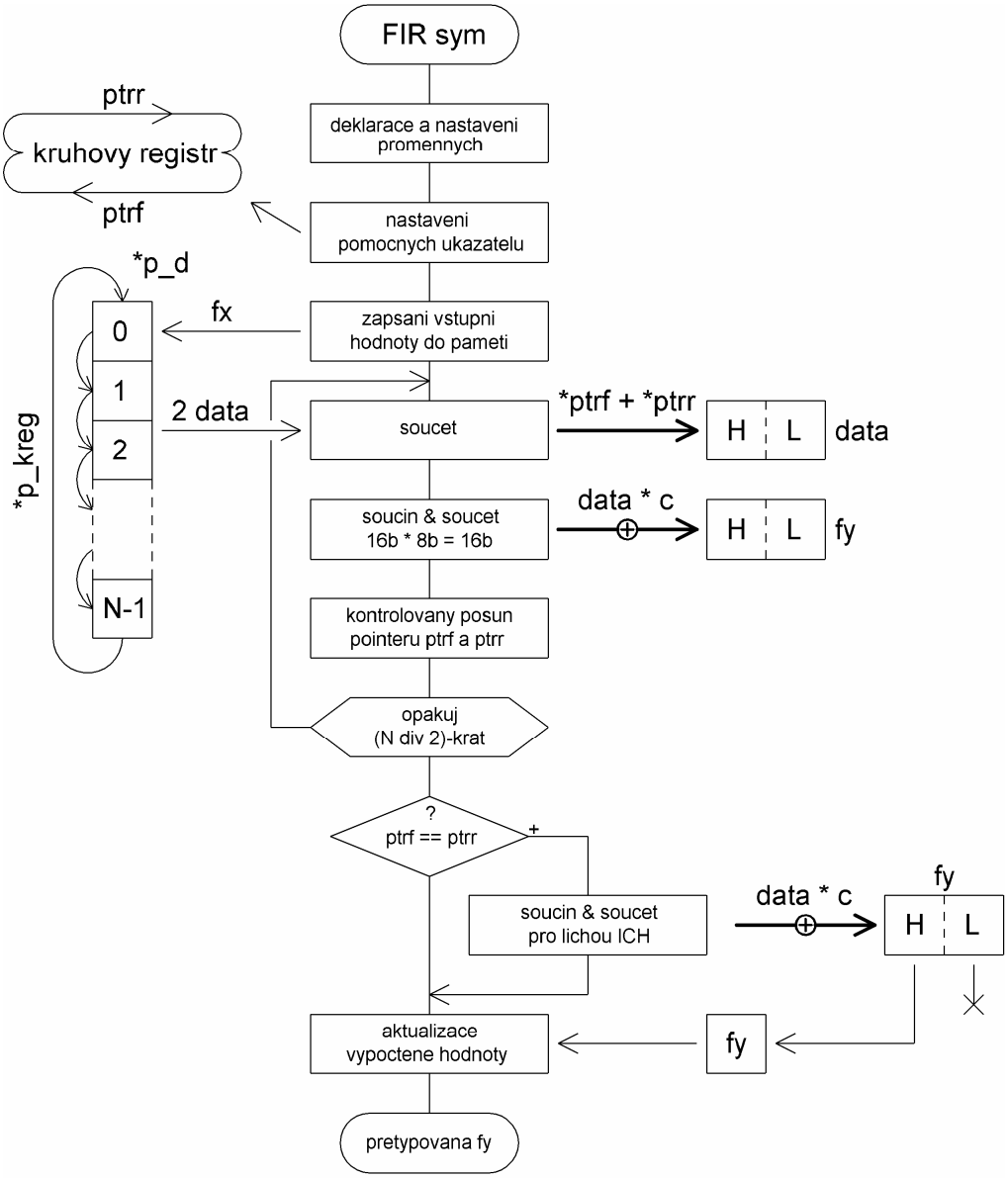
[12] GOUGH, Brian. *FFT Algorithms* [online]. Květen 1997 [cit. 15. srpna 2008]. Dostupný z www: <http://www.cbrc.jp/~tominaga/translations/gsl/fftalgorithms.pdf>

7 Seznam zkratek

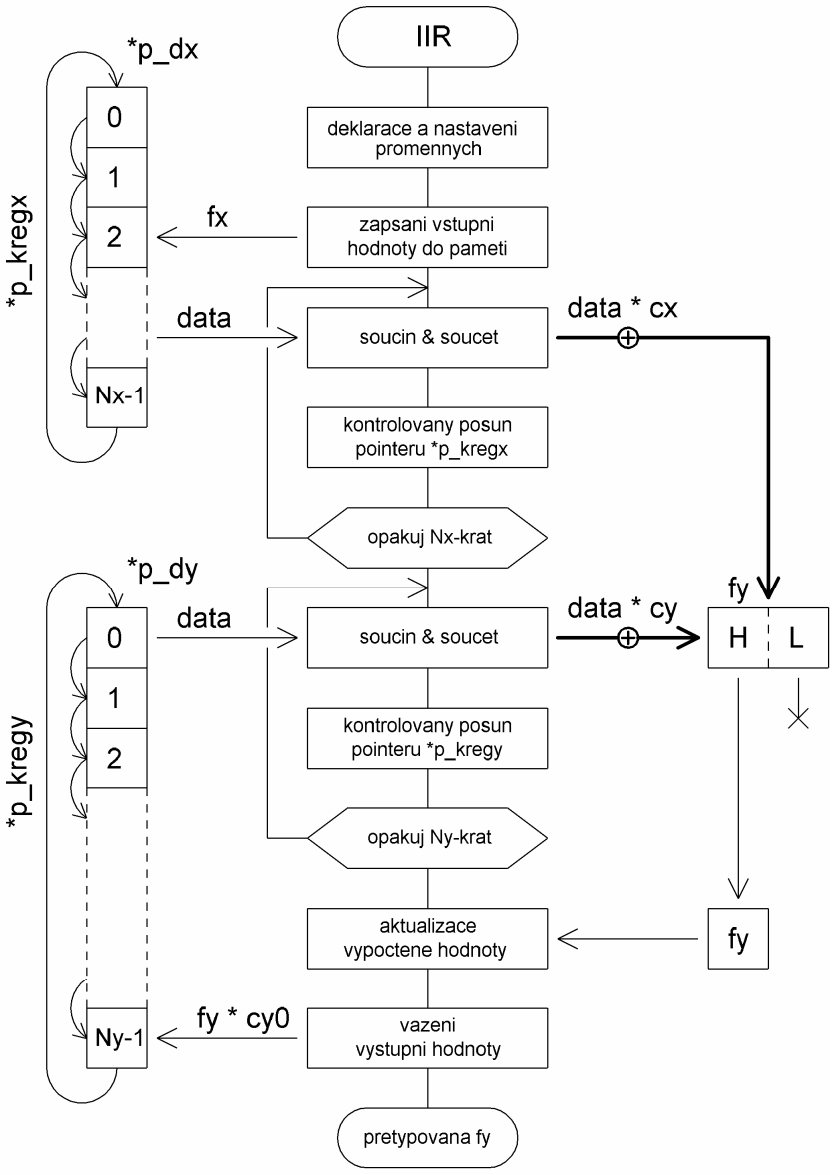
AD	Analogově-digitální
DFT	Diskrétní fourierova transformace (Discrete Fourier Transform)
DFFT	Diskrétní rychlá fourierova transformace (Discrete Fast Fourier Transform)
DSP	Číslicové signální zpracování (Digital Signal Processing)
FT	Fourierova transformace (Fourier Transform)
FFT	Rychlá fourierova transformace (Fast Fourier Transform)
FIR	Konečná impulsní charakteristika (Finite Impulse Response)
GCC	Kompilační knihovny projektu GNU (GNU Compiler Collection)
ICH	Impulsní charakteristika
IIR	Nekonečná impulsní charakteristika (Infinite Impulse Response)
JSA	Jazyk symbolických adres
LSB	Nejméně významný bit (Least Significant Bit)
MAC	Součin a součet (Multiply and ACcumulate)
MSB	Nejvýznamnější bit (Most Significant Bit)

8 Seznam příloh

Příloha 1. Vývojový diagram FIR filtru se symetrickou ICH



Příloha 2. Vývojový diagram filtru IIR



Příloha 3. Tabulka srovnání 8-bitových filtrů

filtr	FIR						FIR symetrická ICH						IIR						IIR kanonická					
jazyk	GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA	
řád	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s
2	425	37,6	236	67,8	216	74,1	383	41,8	269	59,5	252	63,5	714	22,4	427	37,5	357	44,8	512	31,3	296	54,1	225	71,1
3	498	32,1	255	62,7	239	66,9	394	40,6	276	58,0	259	61,8	891	18,0	469	34,1	397	40,3	677	23,6	332	48,2	260	61,5
4	578	27,7	274	58,4	258	62,0	516	31,0	316	50,6	298	53,7	1041	15,4	494	32,4	433	37,0	842	19,0	368	43,5	295	54,2
5	658	24,3	293	54,6	277	57,8	519	30,8	323	49,5	305	52,5	1211	13,2	538	29,7	473	33,8	1007	15,9	404	39,6	330	48,5
6	738	21,7	312	51,3	292	54,8	633	25,3	363	44,1	351	45,6	1371	11,7	583	27,4	511	31,3	1172	13,7	440	36,4	365	43,8
7	818	19,6	331	48,3	303	52,8	652	24,5	370	43,2	351	45,6	1531	10,5	614	26,1	549	29,1	1337	12,0	476	33,6	400	40,0
8	898	17,8	350	45,7	322	49,7	766	20,9	410	39,0	390	41,0	1691	9,5	652	24,5	587	27,3	1502	10,7	512	31,3	435	36,8
9	978	16,4	369	43,4	353	45,3	769	20,8	417	38,4	397	40,3	1851	8,6	690	23,2	625	25,6	1667	9,6	548	29,2	470	34,0
10	1058	15,1	388	41,2	376	42,6	891	18,0	457	35,0	436	36,7	2011	8,0	728	22,0	663	24,1	1832	8,7	584	27,4	505	31,7
11	1138	14,1	407	39,3	391	40,9	902	17,7	464	34,5	443	36,1	2154	7,4	764	20,9	701	22,8	1997	8,0	620	25,8	540	29,6
12	1218	13,1	426	37,6	414	38,6	1008	15,9	504	31,7	489	32,7	2331	6,9	804	19,9	739	21,7	2162	7,4	656	24,4	575	27,8
13	1298	12,3	445	36,0	433	37,0	1019	15,7	511	31,3	489	32,7	2491	6,4	842	19,0	777	20,6	2327	6,9	692	23,1	610	26,2
14	1378	11,6	464	34,5	444	36,0	1141	14,0	551	29,0	528	30,3	2651	6,0	880	18,2	815	19,6	2492	6,4	728	22,0	645	24,8
15	1458	11,0	483	33,1	455	35,2	1152	13,9	558	28,7	535	29,9	2811	5,7	918	17,4	853	18,8	2657	6,0	764	20,9	680	23,5

Příloha 4. Tabulka srovnání 16-bitových filtrů

filtr	FIR						FIR symetrická ICH						IIR						IIR kanonická					
jazyk	GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA		GCC		GCC a JSA		JSA	
řád	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s	cykly	kSa/s
2	774	20,7	382	41,9	322	49,7	658	24,3	353	45,3	371	43,1	1245	12,9	634	25,2	549	29,1	1235	13,0	570	28,1	394	40,6
3	909	17,6	423	37,8	365	43,8	667	24,0	417	38,4	437	36,6	1529	10,5	724	22,1	635	25,2	1642	9,7	651	24,6	473	33,8
4	1051	15,2	466	34,3	408	39,2	854	18,7	413	38,7	431	37,1	1813	8,8	797	20,1	717	22,3	2049	7,8	736	21,7	554	28,9
5	1193	13,4	509	31,4	453	35,3	853	18,8	477	33,5	491	32,6	2097	7,6	891	18,0	803	19,9	2456	6,5	815	19,6	637	25,1
6	1335	12,0	552	29,0	494	32,4	1030	15,5	473	33,8	489	32,7	2381	6,7	984	16,3	885	18,1	2863	5,6	896	17,9	718	22,3
7	1477	10,8	597	26,8	537	29,8	1049	15,3	539	29,7	557	28,7	2665	6,0	1061	15,1	969	16,5	3270	4,9	975	16,4	797	20,1
8	1619	9,9	640	25,0	582	27,5	1226	13,1	533	30,0	549	29,1	2949	5,4	1147	13,9	1053	15,2	3677	4,4	1060	15,1	882	18,1
9	1761	9,1	681	23,5	623	25,7	1225	13,1	591	27,1	617	25,9	3233	4,9	1233	13,0	1137	14,1	4084	3,9	1139	14,0	973	16,4
10	1903	8,4	726	22,0	668	24,0	1412	11,3	593	27,0	609	26,3	3517	4,5	1321	12,1	1221	13,1	4491	3,6	1222	13,1	1050	15,2
11	2045	7,8	767	20,9	709	22,6	1421	11,3	657	24,4	677	23,6	3782	4,2	1405	11,4	1305	12,3	4898	3,3	1313	12,2	1152	13,9
12	2187	7,3	810	19,8	752	21,3	1588	10,1	653	24,5	671	23,8	4085	3,9	1493	10,7	1391	11,5	5305	3,0	1384	11,6	1212	13,2
13	2329	6,9	855	18,7	797	20,1	1597	10,0	717	22,3	737	21,7	4369	3,7	1577	10,1	1473	10,9	5712	2,8	1463	10,9	1289	12,4
14	2471	6,5	896	17,9	840	19,0	1784	9,0	713	22,4	729	21,9	4653	3,4	1663	9,6	1557	10,3	6119	2,6	1558	10,3	1380	11,6
15	2613	6,1	939	17,0	881	18,2	1793	8,9	779	20,5	795	20,1	4937	3,2	1749	9,1	1643	9,7	6526	2,5	1641	9,8	1455	11,0

Příloha 5. Dokumentace knihoven

Seznam knihoven

- **knihovna funkcí v GCC**
  hlavičkový soubor: `cfiltry.h`
  soubor s funkcemi: `cfiltry.c`

- **knihovna funkcí v GCC v kombinaci s asemblerem**
  hlavičkový soubor: `cjsafiltry.h`
  soubor s funkcemi: `cjsafiltry.c`

- **knihovna funkcí v asembleru**
  hlavičkový soubor: `jsafiltry.h`
  soubor s funkcemi: `jsafiltry.c`

Všechny knihovny obsahují navenek identické funkce. Jejich použití je naprosto totožné, rozdíl je pouze v rychlosti vykonání.

Seznam funkcí

```
void f_init(char *p_dx, char *p_dy)
void f_init16(int *p_dx, int *p_dy)
char FIR(char N, char fx, char *p_c, char *p_d)
char FIR_sym(char N, char fx, char *p_c, char *p_d)
char IIR(char Nx, char Ny, char fx, char *p_cx, char *p_cy, char *p_dx, char *p_dy)
char IIR_kan(char N, char fx, char *p_cx, char *p_cy, char *p_d)
int FIR16(char N, int fx, int *p_c, int *p_d)
int FIR_sym16(char N, int fx, int *p_c, int *p_d)
int IIR16(char Nx, char Ny, int fx, int *p_cx, int *p_cy, int *p_dx, int *p_dy)
int IIR_kan16(char N, int fx, int *p_cx, int *p_cy, int *p_d)
```

Globální proměnné

```
char *p_kregx, *p_kregy
int *p_kregx16, *p_kregy16
```

Tyto proměnné jsou vytvořeny v rámci knihovny jako globální, slouží k indexování datového pole. Přístup k nim je zakázán.

Popis funkcí

`void f_init(char *p_dx, char *p_dy)`

Zajišťuje přiřazení ukazatele na datové pole do vnitřní globální proměnné. Vstupními hodnotami jsou ukazatele na datová pole vytvořená uživatelem pro přímou větev (`*p_dx`) a zpětnovazební větev filtrace (`*p_dy`).

`void f_init16(int *p_dx, int *p_dy)`

Identická funkce k `f_init16` pro 16-bitové filtry.

`char FIR(char N, char fx, char *p_c, char *p_d)`

Funkce přímé realizace filtru FIR.
Význam proměnných:

- `N` Zadání řádu filtru
  $N = řád\ filtru + 1$
- `fx` Vstupní vzorek filtru.
- `*p_c` Ukazatel na první prvek pole koeficientů.
  $velikost\ pole = N$
- `*p_d` Ukazatel na první pozici datového pole.
  $velikost\ pole = N$

`char FIR_sym(char N, char fx, char *p_c, char *p_d)`

Funkce filtru FIR se symetrickou impulsní charakteristikou.
Význa proměnných:

- `N` Zadání řádu filtru.
  $N = řád\ filtru + 1$
- `fx` Vstupní vzorek filtru.
- `*p_c` Ukazatel na první prvek pole koeficientů.
  $velikost\ pole = (N+1)\mathrm{div}(2)$
- `*p_d` Ukazatel na první pozici datového pole.
  $velikost\ pole = N$

`char IIR(char Nx, char Ny, char fx, char *p_cx, char *p_cy, char *p_dx, char *p_dy)`

Funkce přímé realizace filtru IIR.
Význam proměnných:

- `Nx` Počet datových pozic k ukládání v přímé větvi.
- `Ny` Počet datových pozic k ukládání ve zpětnovazební větvi.
  Řád filtru určuje největší hodnota.
  $řád\ filtru = \max(Nx, Ny) - 1$
  Zde je možné vytvořit nesymetrický filtr, větve nemusí mít stejný počet koeficientů.
- `fx` Vstupní vzorek filtru.
- `*p_cx` Ukazatel na první prvek pole koeficientů pro přímou větev.
  $velikost\ pole = Nx$
- `*p_cy` Ukazatel na první prvek pole koeficientů pro zpětnovazební větev.
  $velikost\ pole = Ny$

`*p_dx` Ukazatel na první pozici datového pole pro přímou větev.
$velikost\ pole = Nx$

`*p_dy` Ukazatel na první pozici datového pole pro zpětnovazební větev.
$velikost\ pole = Ny - 1$

```
char IIR_kan(char N, char fx, char *p_cx, char *p_cy, char *p_d)
```

Funkce 1. kanonické realizace filtru IIR.
význam proměnných:

`N` Počet datových pozic nutných pro filtraci.
$N = řád\ filtru + 1$

`fx` Vstupní vzorek filtru.

`*p_cx` Ukazatel na první prvek pole koeficientů pro přímou větev.
$velikost\ pole = N$

`*p_cy` Ukazatel na první prvek pole koeficientů pro zpětnovazební větev.
$velikost\ pole = N$

`*p_d` Ukazatel na první pozici datového pole.
$velikost\ pole = N - 1$

```
int FIR16(char N, int fx, int *p_c, int *p_d)
int FIR_sym16(char N, int fx, int *p_c, int *p_d)
int IIR16(char Nx, char Ny, int fx, int *p_cx, int *p_cy, int *p_dx, int *p_dy)
int IIR_kan16(char N, int fx, int *p_cx, int *p_cy, int *p_d)
```

Funkce pro 16-bitové zpracování. Význam proměnných je identický s 8-bitovými funkcemi.

Makra v knihovně cjsafiltry

Tato makra nejsou volně přístupná jejich využití je striktně interní.

```
f_aktual(vstup, vystup)
```

Aktualizace a zaokrouhlení na 8-bitové úrovni.

```
f_aktual16(vstup, vystup)
```

Aktualizace a zaokrouhlení na 16-bitové úrovni.

```
f_macFIR(f_y, p_dat, p_koef, p_reg, f_N)
f_macFIR16(f_y, p_dat, p_koef, p_reg, f_N)
```

Cyklus výpočtu přímé větve přímých realizací filtrů.

```
f_macFIR_sym(f_y, p_dat, p_koef, p_reg, f_N)
f_macFIR_sym16(f_y, p_dat, p_koef, p_reg, f_N)
```

Cyklus filtrace se symetrickou impulsní charakteristikou.

```
f_macIIR(f_y, p_dat, p_koef, p_reg, f_N)
f_macIIR16(f_y, p_dat, p_koef, p_reg, f_N)
```

Cyklus výpočtu zpětnovazební větve filtrů.

```
f_macIIR_kan(f_x, f_y, p_dat, p_koefx, p_koefy, f_N)
f_macIIR_kan16(f_x, f_y, p_dat, p_koefx, p_koefy, f_N)
```

Cyklus výpočtu 1. kanonické formy IIR filtru.

Zadání koeficientů

Filtrační koeficienty se zadávají jako pole celých 8-bitových (16-bitových) čísel. Počet koeficientů je dán řádem a typem filtru. FIR se symetrickou impulsní charakteristikou vyžadují zadat pouze polovinu koeficientů (symetrie).

Reálná hodnota koeficientu se převede na koeficient pro filtry podle následujícího postupu.

$$\begin{aligned} c_8 &\cong \text{reálný koeficient} \cdot 128 \\ c_{16} &\cong \text{reálný koeficient} \cdot 32768 \end{aligned} \tag{1}$$

Pole koeficientů je ve formátu

$$\begin{aligned} &\text{char } fcx[N] = \{A_0, A_1, A_2, ..., A_{N\text{-}1}\}; \\ &\text{char } fcy[N] = \{B_0, B_1, B_2, ..., B_{N-1}\}; \end{aligned} \tag{2}$$

Rekurzivní filtry obsahují ještě jeden koeficient navíc, B_0, ten umožňuje řídit pravděpodobnost přetékání na úkor přesnosti výpočtu. Tento koeficient násobí výsledek celou hodnotou, tzn. váží ho reálným koeficientem $|c_y| > 1$. Je jím také možno vykompenzovat nemožnost zadání koeficientů $|c| > 1$. Ovlivňuje ostatní koeficienty podle vztahu (3) a pokud nejsou jeho vlastnosti využity, jeho hodnota je 1.

$$[A_0, A_1, ..., A_{N-1}, B_1, B_2, ..., B_{N-1}] = \frac{[a_0, a_1, ..., a_{N-1}, b_1, b_2, ..., b_{N-1}]}{B_0} \tag{3}$$

Vytvoření datového pole

Datové pole tvoří pole celých čísel, které je vhodné předem naplnit nulovými hodnotami. Tato podmínka není nutná, první výsledky stejně nebudou správné, ovšem předejde to možnému přetečení při nevhodné kombinaci hodnot.

Příloha 6. Příklad funkce DFFT v JSA (není kompletní)

```
fft:    ldi   j, 1              ;j = 1, p = 2
        ldi   p, 2
skok1:  ldi   b, 2*dn           ;b = dn
        clr   ptrh              ;pomocna nula
        ldi   XL, low(dadr)     ;b = *data
        ldi   XH, high(dadr)
        movw  Y, X              ;Y = b + p
        add   YL, p
        adc   YH, ptrh
        ld    xare, X           ;xare = data(b)
        mov   xbre, xare        ;xbre = data(b)
        ld    data, Y
        add   xare, data        ;xare = xare + data(b + p)
        sub   xbre, data        ;xbre = xbre - data(b + p)
        st    X, xare           ;data(b) = xare
        st    Y, xbre           ;data(b + p) = xbre
        movw  X, Y              ;X += 2 * p
        add   XL, p
        adc   XH, ptrh
        sub   b, p              ;b -= 2 * p
        sub   b, p
        brne  PC-15             ;opakuj, dokud b > 0
        cpi   j, 2              ;přeskoč jestliže j<=1
        brcc  skok7
        rjmp  skok
skok7:  mov   i, p              ;ptrl = p/2 - 1
        lsr   i
        subi  i, 2
        brne  skok8
        rjmp  skok3             ;přeskoč jestliže j = 2
skok8:  clr   a                 ;a = 0
        ldi   ZH, high(padr-4)  ;W = LPM(*padr + j - 2)
        ldi   ZL, low(padr-4)
        add   ZL, j
        adc   ZH, ptrh
        add   ZL, j
        adc   ZH, ptrh
        ld    wre, Z+
        ld    wim, Z
skok2:  inc   a                 ;a += 1
        inc   a
        ldi   b, 2*dn           ;b = dn
        ldi   XL, low(dadr)     ;b = *data
        ldi   XH, high(dadr)    ;X = *data + a
        add   XL, a
        adc   XH, ptrh
        mov   ptrl, a           ;ptrl = 2 * a
        add   ptrl, a
skok4:  movw  Y, X              ;Y = a + b + p = X + p
        add   YL, p
        adc   YH, ptrh
        ld    xare, X+          ;xa = data(b + a)
        mov   xbre, xare
        ld    xaim, X           ;xb = conj data(b + a)
        mov   xbim, xaim
        ld    datre, Y+         ;data = W * data(b + a + p)
        ld    datim, Y
        mulcpxre
        add   xare, data
        sub   xbre, data
```

```
          mulcpxim
          add   xaim, data
          sub   data, xbim
          st    X, xaim             ;data(b + a) = xa
          st    -X, xare
          sub   YL, ptrl            ;Y = b - a + p
          sbc   YH, ptrh
          st    Y, data             ;data(b - a + p) = xb
          st    -Y, xbre
          add   XL, p               ;X += 2 * p
          adc   XH, ptrh
          add   XL, p
          adc   XH, ptrh
          sub   b, p                ;b -= 2 * p
          sub   b, p
          breq  skok14              ;opakuj, dokud b > 0
          rjmp  skok4
skok14:   sbiw  Z, 1                ;dat = LPM(*padr + j - 2)
          ld    datre, Z+
          ld    datim, Z
          mulcpxre                  ;W = W * dat
          mov   xare, data
          mulcpxim
          mov   wim, data
          mov   wre, xare
          cp    i, a                ;opakuj, dokud a <> p/4 - 1
          breq  skok3
          rjmp  skok2
skok3:    ldi   a, 2                ;a = p/2
          add   a, i
          ldi   b, 2*dn             ;b = dn
          ldi   XL, low(dadr)       ;X = *data + a
          ldi   XH, high(dadr)
          add   XL, a
          adc   XH, ptrh
skok5:    movw  Y, X                ;Y = a + b + p
          add   YL, p
          adc   YH, ptrh
          ld    xare, X+            ;xa = data(b)
          ld    xaim, X
          ld    data, Y+            ;xaim -= data(b + p).re
          sub   xaim, data
          ld    data, Y             ;xare -= data(b + p).im
          sub   xare, data
          st    X, xaim             ;data(b) = xa
          st    -X, xare
          sbiw  Y, 1                ;X += 2 * p
          movw  X, Y
          add   XL, p
          adc   XH, ptrh
          sub   b, p                ;b -= 2 * p
          sub   b, p
          brne  skok5               ;opakuj, dokud b > 0
skok:     inc   j                   ;j++
          lsl   p                   ;LSL(p)
          cpi   p, 2*dn             ;opakuj, dokud p < dn
          brcc  skok6
          rjmp  skok1
skok6:    ret
```