VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ŘÍDÍCÍ ELEKTRONIKA MĚNIČE NA BÁZI DSP TEXAS INSTRUMENTS

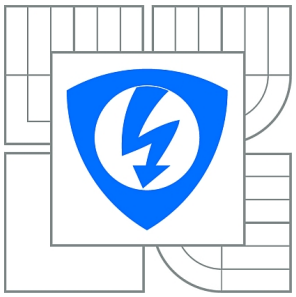
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ADAM VAŠÍČEK
AUTHOR

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ŘÍDÍCÍ ELEKTRONIKA MĚNIČE NA BÁZI DSP TEXAS INSTRUMENTS

A FREQUENCY INVERTER BASED ON TEXAS INSTRUMENTS DSP

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

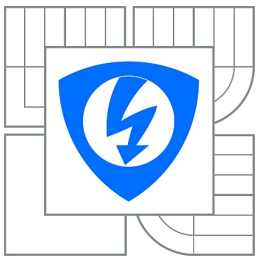
AUTOR PRÁCE — ADAM VAŠÍČEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. SOBĚSLAV VALACH

SUPERVISOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Adam Vašíček	***ID:***	70335
Ročník:	3	***Akademický rok:***	2009/2010

NÁZEV TÉMATU:

Řídící elektronika měniče na bázi DSP Texas Instruments

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte možnosti implementace a optimalizace algoritmů pro řízení výkonového měniče se signálovým procesorem řady 2000 firmy Texas Instruments.

DOPORUČENÁ LITERATURA:

Firemní literatura Texas Instruments.

Termín zadání:	8.2.2010	***Termín odevzdání:***	31.5.2010

Vedoucí práce: Ing. Soběslav Valach

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

ABSTRAKT

Tato bakalářská práce se zabývá optimalizací základních algoritmů řídicích aplikací pro DSC Texas Instruments řady c2000. V první část jsou shrnuty metody řízení a regulace elektrických pohonů s asynchronními motory. Následující část je věnována obecným zásadám optimalizace a metodám výpočtu některých matematických funkcí a operací. Značná část textu se zabývá mj. goniometrickými funkcemi, které jsou pro tuto oblast nepostradatelné. Dále navazuje praktické ověření uvedených postupů na základním algoritmu skalárního řízení frekvenčního měniče. Ten je v několika variantách, včetně asemblerové podoby, odladěn na vývojové desce ezDSP se signálovým procesorem TMS320F2808 od firmy Texas Instruments. Text je doplněn podstatnými částmi zdrojových kódů a stručným popisem vývojového prostředí Code Composer Studio. V závěru práce jsou rozebrány výsledky jednotlivých optimalizací a zhodnocen jejich přínos vzhledem k nárokům na čas programátora.

KLÍČOVÁ SLOVA

optimalizace, C, asembler, kompilace, rychlost, frekvenční měnič, DSP, TMS320F280x, TMS320F2808, c2000, sin, cos

ABSTRACT

This bachelor thesis is focused on control algorithms optimization, especially using the Texas Instruments c2000 digital signal controllers family. The first part roughly describes an AC induction motor controlling and regulation techniques. Later on the control software optimization basics are covered. Particular attention is payed for trigonometric functions which are mandatory for such software. Next part shows practical use of the previously described optimization techniques using an example of the scalar frequency inverter control even going down to the assembly instructions level. At the end the results themselves as well as facing the invested price of optimization are discussed.

KEYWORDS

optimalization, C, assembler, compilation, speed, frequency inverter, DSP, TMS320F280x, TMS320F2808, c2000, sin, cos

VAŠÍČEK, Adam *Řídicí elektronika měniče na bázi DSP Texas Instruments*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2010. 69 s. Vedoucí práce byl Ing. Soběslav Valach

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Řídicí elektronika měniče na bázi DSP Texas Instruments“ 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.

Brno ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

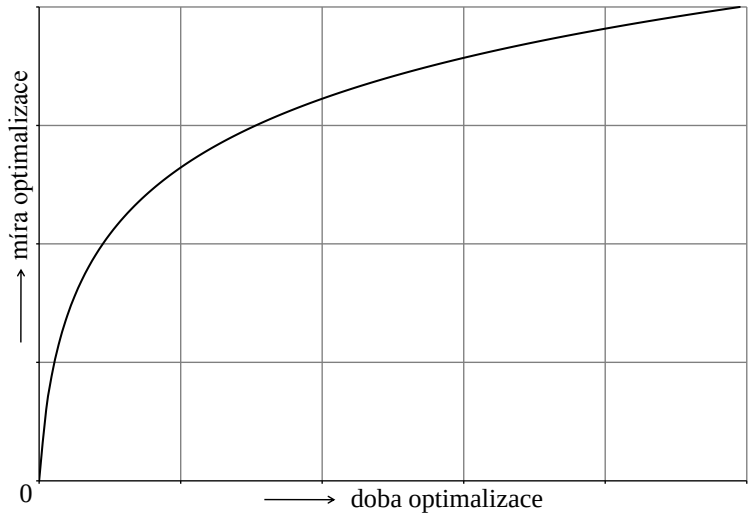
Tato bakalářská práce se zabývá implementací základních algoritmů pro řízení frekvenčních měničů a metodami optimalizace těchto algoritmů, psaných v jazyce C. Optimalizace má za úkol maximálně zefektivnit výsledný strojový kód programu, a to v několika směrech. Především jsou to rychlost programu, jeho velikost a paměťové nároky. Požadavky kladené na program, a tím i jeho optimalizaci, jsou ale značně závislé jak na druhu aplikace jako takové, tak na výpočetních schopnostech procesoru, na kterém program poběží. Je zřejmé, že zcela jinak musíme přistupovat k psaní uživatelské apikace pro PC a programu pro řízení a regulaci v reálném čase. V případě řídicích programů jsou nároky na jejich optimalizaci obvykle značné. Nezřídka potřebujeme například vypočítat hodnoty číslicové regulační smyčky v každé periodě vzorkování, trvající pouze desítky mikrosekund. Algoritmus regulátoru se tedy vykonává mnohotisíc-krát v každé vteřině a musí být co možná nejrychlejší, aby bylo možné procesor využít i k dalším úlohám.

Úroveň optimalizace, které může daný programátor za uritý čas dosáhnout, je do značné míry dána jeho znalostmi jazyka (v našem případě C), procesu kompilace zdrojového kódu a vlastností použitého mikroprocesoru (μP). Zkušený programátor, znající do detailů architekturu cílového μP, bude schopný dosáhnout výsledků, které se na dané architektuře blíží teoretickému maximu. Protože je však optimalizace časově značně náročná, je vždy třeba zvážit, není-li výhodnější použít výkonnější μP a ušetřit tak čas a tím i náklady věnované důsledné optimalizaci. Můžeme říci, že míra optimalizace programu je zhruba úměrná logaritmu času optimalizací stráveného. Tuto závislost vyjadřuje Obr. 1. Záměrně na něm nejsou uvedeny hodnoty, protože ty jsou pro každého programátora i program jiné.

0.1 Členění práce

První část práce, kapitola 1, se zabývá popisem základních metod řízení frekvenčních měničů. Pozornost je věnována skalární regulaci asynchronních pohonů. Její implementaci a zejména optimalizaci se později věnuje praktická část práce.

Druhá kapitola je věnována obecným zásadám optimalizace zdrojového kódu v jazyce C a algoritmům pro matematické výpočty. Nejdříve je popsána metoda vyjadřování desetinných čísel pomocí celočíselných proměnných a vliv základních matematických operací na tato čísla. Následně jsou rozebrány nejdůležitější matematické funkce a možnosti jejich realizace. Pozornost je věnována především goniometrickým funkcím. Poté jsou popsány některé ze zásad pro psaní zdrojových kódu v C s ohledem na optimalizaci jejich rychlosti. U jednotlivých zásad je obecně popsán



Obr. 1: Závislost míry optimalizace kódu na časové náročnosti optimalizace.

efekt, který jejich dodržení (či nedodržení) má na výsledný strojový kód.

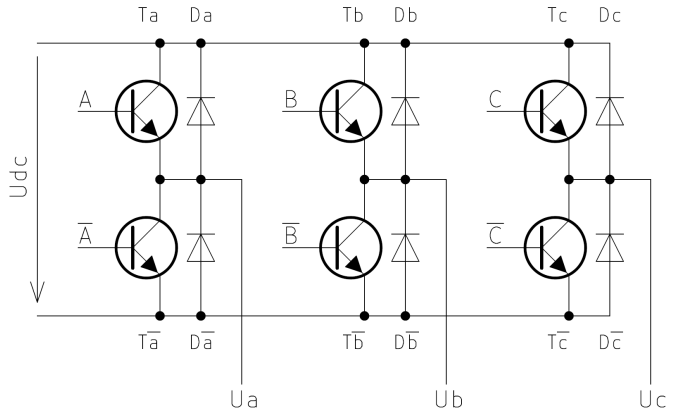
Praktická část práce, kapitola 3, seznamuje nejdříve s detaily architektury digitálního signálového kontroléru (DSC) typu TMS320F2808 firmy Texas Instruments, které jsou pro optimalizaci podstatné. Zhruba popisuje i obecné rysy tohoto DSC a dalších zástupců řady c2000. Podává také základní informace o použitém překladači a vývojovém prostředí. Na úryvcích zdrojových kódů jsou následně popsány použité algoritmy.

Zhodnocení výsledků optimalizací a srovnání několika verzí programu je uvedeno v kapitole 4. Jsou zde tabulky zjištěných hodnot, grafy srovnávající dosažené cíle a změřené průběhy výstupních signálů.

V příloze A je několik ilustračních fotografií vývojového pracoviště i fotografie obrazovky použitého vývojového software. Příloha B popisuje pomocný program, napsaný pro usnadnění zkoušení iteračních algoritmů, využívajících Newtonovu metodu. Na přiloženém CD jsou k dispozici veškeré zdrojové texty, grafy obsažené v práci, volně přístupná literatura, z níž bylo čerpáno a samotný text práce v elektronické podobě. Adresářová struktura CD je popsána v příloze C

1 REGULACE ASYNCHRONNÍCH POHONŮ

K regulaci pohonů s asynchronními motory s kotvou nakrátko se používají frekvenční měniče — střídače. V dnešní době je jejich použití stále více rozšířené od malých pohonů např. v oblasti vzduchotechniky až po trakční motory lokomotiv a lodí s výkony řádu MW. Úkolem frekvenčního měniče je napájet statorová vinutí asynchronního elektrického stroje s kotvou nakrátko tak, abychom docílili plynulé změny jak otáček, tak momentu stroje v co nejširším rozsahu. Toho docílíme vhodnou kombinací sepnutí jednotlivých spínacích prvků střídače. Zjednodušené schéma jeho výkonové části je na obrázku 1.1. (Na schématu jsou nakresleny na místě spínacích prvků bipolární tranzistory, ty však jsou obvykle nahrazeny tranzistory MOS-FET, IGBT nebo tyristory IGCT případně GTO.) Kombinace spínání jsou řízeny řídicím



Obr. 1.1: Zjednodušené schéma silové části střídače

mikroprocesorem, který tak činí podle zvoleného regulačního algoritmu a stavu vstupních a zpětnovazebních a signálů. Algoritmy řízení frekvenčních měničů můžeme rozdělit na několik základních typů, z nichž každý má své výhody a nevýhody. Základní členění je na:

- *Skalární regulace* pracuje pouze se skalárními hodnotami jak žádaných (statorové napětí a kmitočet, případně proud), tak i měřených veličin ve zpětné vazbě (fázové proudy, otáčky hřídele). Jedná se o nejstarší a nejjednodušší metodu regulace. Její hlavní nevýhodou je horší dynamika výsledného pohonu. Výhodou je jednoduchost řídicích algoritmů ve srovnání s pokročilejšími metodami řízení.
- *Vektorová regulace* využívá oproti skalární jak amplitudu, tak fázi jednotlivých veličin. Pracuje na různých metodách řízení magnetického toku a vnitřního elektromagnetického momentu asynchronního stroje. Tyto veličiny nejsou přímo

měřitelné, proto je k jejich výpočtu využíván matematický model stroje. Výsledkem je pohon s lepší dynamikou a širším rozsahem regulace rychlosti i momentu.

- *Přímá regulace* přináší ve srovnání s vektorovou regulací podobné nebo ještě o něco lepší výsledky. Je založena na přímé regulaci momentu motoru v určitém tolerančním pásmu (moment motoru roste nebo klesá podle zvolené kombinace sepnutí střídače). Moment motoru je i zde počítán z fázových proudů a napětí meziobvodu pomocí matematického modelu asynchronního motoru.

Následující část je věnována shrnutí základních informací o skalární regulaci, na jejímž příkladě bude v dalším textu předvedena implementace základních algoritmů a zejména jejich optimalizace.

1.1 Skalární regulace

Skalární regulace bývá založena na konstantním poměru amplitudy a kmitočtu napájecího napětí asynchronního stroje, což je vyjádřeno rovnicí 1.1.

$$\frac{U}{f} = konst. \tag{1.1}$$

Je-li tento poměr konstantní, je konstantní i sycení magnetického obvodu motoru. To však neplatí v oblasti nízkých kmitočtů, kdy se projeví vliv stejnosměrného odporu vinutí statoru motoru. Opačná strana charakteristiky je omezena maximálním výstupním napětím střídače, případně maximálním napájecím napětím stroje. Mužeme sice dále zvyšovat kmitočet, ne však napětí. Snižuje se tedy moment zvratu motoru, s čímž musíme při dimenzování pohonu počítat.

Moment, jakým je motor zatížen, způsobí snížení rychlosti otáčení motoru o skluz a odpovídající zvýšení statorového proudu. To platí až do momentu zvratu, kdy se motor zastaví a chová se jako ve stavu nakrátko. Tomu musí zabánit nadproudová ochrana, hlídající jednotlivé fázové proudy. Měření statorového proudu lze tedy využít k odhadu zatížení motoru a tím i korekci žádaného kmitočtu o kmitočet skluzový. Tím se uzavře zpětnovazební smyčka a můžeme mluvit o regulaci pohonu i v případě, že nebudeme měřit (a regulovat) skutečnou rychlost otáčení rotoru.

1.2 Modulace střídače

Modulací se rozumí metoda spínání výkonových prvků střídače. Pro jednoduchost bude popsáno pouze několik základních typů modulací dvouhladinového střídače (viz zjednodušené schéma na obr. 1.1). V závislosti na výkonové úrovni střídače, která

určuje výběr spínacích prvků a tím i jejich maximální spínací kmitočet[1], volíme i modulační algoritmy. Více na toto téma pojednává např. [8, kap. 6].

Existují dva základní typy modulačních algoritmů a to synchronní a asynchronní. První z nich má pevně daný počet sepnutí výkonových prvků na periodu základní harmonické výstupního napětí, u druhého je pevně dána perioda spínání prvků, nezávisle na této periodě. Aby nedocházelo k nerovnoměrnému rozložení pulzů asynchronní modulace v jednotlivých periodách generovaného napětí, a tím k nebezpečí vzniku interferencí, jsou asynchronní modulace používány pokud můžeme dosáhnout alespoň o řád vyššího kmitočtu spínání výkonových prvků ve srovnání s požadovaným výstupním kmitočtem.

U střídačů nejvyšších výkonů (řádově megawatty), osazených IGCT nebo GTO (viz [9]), je dosažitelný spínací kmitočet jen stovky Hz až jednotky kHz. Ten už může být srovnatelný s nejvyšším napájecím kmitočtem daného stroje. V tomto případě zvolíme pravděpodobně obdélníkové řízení. Jeho výhodou je nejnižší možný počet komutací (spínání a vypínání). Zřejmými nevýhodami je značné harmonické zkreslení fázových proudů i sdruženého napětí. Proto se tato modulace využívá jen v oblasti nejvyšších kmitočtů a při nižších kmitočtech se přechází na jiný modulační algoritmus, například čtyřpulzní modulaci [8, str. 63].

U pohonů nižších výkonů (od stovek W do desítek kW) s tranzistory IGBT nebo MOS-FET se používá pulzně–šířková modulace (PWM). Ta může být buď symetrická nebo asymetrická podle toho, zda čítač modulátoru čítá jen jedním směrem nebo směry oběma. Obvykle se používají symetrické PWM, protože střídač pak produkuje méně rušení. Jednotlivé větve můstku (na Obr. 1.1 značeny a, b a c) totiž nekomutují zároveň. Zadávací signál pro PWM může mít sinusový průběh, potom se jedná o tzv. sinusovou modulaci. Průběh fázových napětí motoru (vzhledem ke středu naptí U_{DC}, který je v případě dvouhladinových střídačů pouze myšlenou hodnotou) je pak popsán rovnicemi

$$\begin{aligned} u_U &= \tfrac{U_{DC}}{2} K \sin(\omega t) \\ u_V &= \tfrac{U_{DC}}{2} K \sin\left(\left(\omega + \tfrac{2\pi}{3}\right) t\right) \\ u_W &= \tfrac{U_{DC}}{2} K \sin\left(\left(\omega + \tfrac{4\pi}{3}\right) t\right) \end{aligned} \tag{1.2}$$

kde K je modulační konstanta, udávající úhel otevření výkonových prvků střídače v rozsahu <0;1>, což odpovídá střídě <0;50)%. U_{DC} je napětí stejnosměrného meziobvodu, ω je úhlová rychlost rotace magnetického pole statoru napájeného motoru a t je čas.

Průběh fázových napětí ale může být jiný, něž sinusový. Často používaná je metoda zvaná *Space Vector Modulation* (SVM, česky modulace prostorového vektoru, popsána například česky v [8] nebo anglicky v [5]). Její výhodou oproti sinusové

[1]Při rozumné velikosti přepínacích ztrát, které jsou přímo úměrné kmitočtu spínání prvků.

modulaci je především dosažení vyšší hodnoty sdružených napětí napájeného stroje při zachování jejich harmonckého průběhu.

2 OPTIMALIZACE ALGORITMŮ

Druhy optimalizace můžeme rozdělit podle úrovně, na jaké je prováděna. Ta může být následující:

1. Úroveň *návrhu algoritmu*, tedy vůbec přístupu k řešení daného problému. V této fázi je optimalizace nejvíce ovlivněna znalostmi programátora a jeho schopností analytického myšlení a invence.
2. Úroveň *zdrojového kódu* programu, tedy zápisu zvoleného algoritmu v daném jazyce. Zde záleží nejvíce na „umění“ programátora nakládat s daným jazykem a do určité míry také na optimalizátoru překladače. Ten totiž může některé navhodné zápisy zjednodušit a zefektivnit.
3. Úroveň *překladu* zdrojového kódu, tedy jeho automatizovaného převodu do kódu strojového. Na této úrovni je míra optimalizace dána nastavením optimalizátoru i překladače a jejich kvalitou.
4. Úroveň *asembleru*, kde záleží opět na „umu“ programátora a hloubce jeho znalostí daného mikroprocesoru, píše-li ovšem přímo v asembleru. Pokud píšeme jen ve vyšším jazyce, jako je C, nemáme téměř žádné možnosti tento stupeň optimalizace ovlivnit. (Výjimkou jsou tzv. předdefinované funkce — intrinsics, popsané v kapitole 2.2.13.)

Platí, že první dva stupně jsou prakticky nezávislé na použité platformě, použijeme-li např. jazyk C. Třetí již může být na různých platformách mírně odlišný. Čtvrtá, nejnižší úroveň optimalizace je úzce vázána na použitý hardware a není tedy přenositelná. Pro dosažení v požadovaném směru (viz dále) maximálně efektivního kódu je třeba věnovat optimalizaci každé úrovně náležitou pozornost. Tato kapitola je dále zhruba členěna právě podle úrovně optimalizace.

Cíle optimalizace algoritmu mohou být různé v závislosti na vlastnostech použitého mikroprocesoru, velikostí pamětí, a hlavně požadavcích na výsledný kód. Můžeme je rozdělit na několik základních cílů:

- *Vysoká rychlost algoritmu* je zřejmě nejčastějším cílem optimalizace. Zvláště v případě algoritmů, jež jsou opakovaně vykonávány.
- *Malá velikost algoritmu* je požadována v případě, že jsme omezeni velikostí paměti programu. Malý program je však často i rychlejší než rozsáhlá aplikace.
- *Malé nároky na paměť dat* jsou nejen v případě řídicí elektroniky vítány. I u moderních stolních počítačů s operační pamětí řádu GB, je žádoucí s pamětí šetřit, protože algoritmus, který s pamětí plýtvá, je často i pomalejší, než algoritmus, který s ní šetří.
- *Velká přesnost výpočtů* je specifickým požadavkem, který patří spíše do oblasti matematických knihoven a SW pro osobní počítače. Určitá přesnost výpočtů matematických funkcí (goniometrické funkce, odmocnina, logaritmus, ale i dě-

lení a podobně) je ale zapotřebí v každé aplikaci. Můžeme říci, že přesnější algoritmy jsou výpočetně, tedy i časově nebo paměťově náročnější.

Jak je vidět, jsou spolu jednotlivé cíle do velké míry svázány. Dobrou optimalizací často zpočátku dosáhneme zlepšení více, případně i všech, parametrů. To však platí jen do určité meze, od které dosáhneme zlepšení jednoho parametru na úkor jiného nebo i všech zbývajících. Například většího zrychlení programu někdy dosáhneme pouze za cenu podstatného zvětšení jeho velikosti a snížení přesnosti.

Algoritmy pro řízení výkonové elektroniky, jejichž optimalizací se tato práce zabývá, musejí být v první řadě co nejrychlejší. Jakékoliv zpoždění může mít za následek havárii celého zařízení, často je proto i snahou psát takové algoritmy, jejichž doba trvání je předem definovaná a nejlépe konstantní. Z omezených zdrojů používaných výpočetních systémů vyplývají omezení možné velikosti programu i dat. Nejčastěji bývají tyto systémy schopné výpočtů pouze s pevnou řádovou čárkou z čehož vyplývají možné problémy s přesností, rozsahem proměnných a tedy i výpočtů.

V závěru kapitoly jsou krátce shrnuty nepříznivé jevy, které s sebou optimalizace přináší.

2.1 Matematické operace a funkce

Volba správného algoritmu řešení daného výpočtu je prvním krokem v optimalizaci. Tato část práce se pokusí popsat část základních algoritmů a jejich vlastnosti. Některé z algoritmů jsou porovnány s jejich ekvivalenty z volně dostupné matematické knihovny IQmath od Texas Instruments. Ta je určena pro digitální signálové kontroléry (DSC) řady c2000 od téže společnosti.

Algoritmy, používané v oblasti řídicí elektroniky jsou často založené na nepříliš triviálních výpočtech. Jak bylo uvedeno, je třeba, aby tyto výpočty byly maximálně optimalizované ve smyslu jejich rychlosti. V podstatě jakýkoliv výpočet by bylo možné realizovat pomocí tabulky hodnot, což je prakticky vyždy nejrychlejší. Její paměťová náročnost ale neúměrně roste s požadovanou přesností a rozsahem vstupní proměnné či proměnných. Proto se používá jen u těch funkcí, které jsou velmi často používány a její využití je výrazně rychlejší v porovnání s jinými metodami, například aproximací funkce Taylorovou řadou nebo Newtonovou iterační metodou.

V následujících částech budou rozebrány některé metody vhodné pro výpočet obvykle používaných funkcí i základních operací. Pozornost bude věnována především algoritmům vhodným pro 16- a 32-bitové systémy s pevnou řádovou čárkou (fixed point).

2.1.1 Desetiná čísla a jejich vyjádření

DSC řady c280x pracují s 32-bitovými a 16-bitovými celými čísly, několik instrukcí pracuje s proměnnými o rozsahu 64 bitů. Záporná čísla jsou vyjádřena pomocí druhého doplňku. Rozsah čísel, které takto můžeme vyjádřit, je zřejmý z tabulky 2.1 Desetiná čísla můžeme vyjádřit pomocí pevně umístěné desetinné čárky, kterou

	rozsah datového typu	
datový typ	od	do
`unsigned int`	0	$2^{16} - 1 = 65\ 535$
`signed int`	$-2^{15} =$ -32 768	$2^{15} - 1 = 32\ 767$
`unsigned long`	0	$2^{32} - 1 = 4\ 294\ 967\ 296$
`signed long`	$-2^{31} =$ -2 147 483 648	$2^{31} - 1 = 2\ 147\ 483\ 647$
`unsigned long long`	0	$2^{64} - 1 \doteq 1,8447 \cdot 10^{19}$
`signed long long`	$-2^{63} \doteq 9,2234 \cdot 10^{18}$	$2^{63} - 1 \doteq 9,2234 \cdot 10^{18}$

Tab. 2.1: Rozsah základních datových typů

pomyslně umístíme na takové místo v proměnné, abychom zaručili, že se proměnná v celém svém rozsahu vejde do nově určeného rozsahu a zároveň zajistili dostatečnou přesnost vyjádření čísla. Například u osmibitové proměnné s pomyslnou desetinnou čárkou mezi 3. a 4. bitem (bity číslovány zprava od 0 v souladu s mocninou 2 dané cifry) vyjadřují jednotlivé bity následující hodnoty.

$$\begin{aligned} \texttt{0xFF}_{\texttt{Q4}} &= 2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = \\ &= 8 + 4 + 2 + 1 + 0,5 + 0,25 + 0,125 + 0,0625 = \\ &= 15 + 15/16 = 15,9375 \end{aligned} \tag{2.1}$$

Z tohoto vidíme, že číslo má rozsah 0 až 15,9375 s krokem 0,0625. Vliv umístění desetinné čárky na rozsah proměnné a krok mezi dvěma sousedními hodnotami vyjadřuje tabulka 2.1.1. V celé práci je pro vyjádření zvoleného umístění myšlené desetinné čárky použito tzv. „Q“ notace, která je používaná mj. i společností TI v jejich dokumentech. Např. označení proměnné jako `Q4` ve výpočtu 2.1 znamená, že poslední 4 bity proměnné jsou její desetinnou částí. Pro nejmenší krok čísla ve formátu `Q`n tedy platí:

$$krok = 2^{-n} \tag{2.2}$$

Tabulka 2.1.1 je pro 16-bitové proměnné, krok však platí i pro proměnné delší — často 32-bitové. Například tabulka hodnot funkce sinus (respektive i kosinus) v paměti ROM DSC je uložena právě v `signed` formátu `Q30`. Uložená čísla tedy

mohou nabývat hodnot

$$\left\langle -2; 1 + \frac{2^{30}-1}{2^{30}} \right\rangle \doteq \langle -2; 2) \tag{2.3}$$

		`signed`		`unsigned`
	krok	od	do[1]	do[1]
`Q1`	0,5	-16 384	16 383	32 767
`Q2`	0,25	-8 192	8 191	16 383
`Q3`	0,125	-4 096	4 095	8 191
`Q4`	0,0625	-2 048	2 047	4 095
`Q5`	0,03125	-1 024	1 023	2 047
`Q6`	0,015625	-512	511	1 023
`Q7`	0,0078125	-256	255	511
`Q8`	0,00390625	-128	127	255
`Q9`	0,001953125	-64	63	127
`Q10`	0,0009765625	-32	31	63
`Q11`	0,00048828125	-16	15	31
`Q12`	0,000244140625	-8	7	15
`Q13`	0,0001220703125	-4	3	7
`Q14`	0,00006103515625	-2	1	3
`Q15`	0,000030517578125	-1	0	1

Tab. 2.2: Rozsah a přesnost 16-bitových čísel s pevnou řádovou čárkou

Sčítání/odečítání desetinných čísel

Sčítání nebo odečítání čísel, které mají shodně umístěnou pomyslnou desetinnou čárku, provádíme stejně jako sčítání přirozených čísel. Pokud však čísla nemají stejný počet desetinných cifer, musíme je pomocí bitových posunů převést na stejný formát. Máme tři možnosti, jak postupovat:

1. Převést oba sčítance na formát s **nižším počtem desetinných cifer**, čímž zachováme číselný rozsah výsledku a tím i omezíme možnost přetečení. Nevýhodou je oříznutí přebývajících desetinných míst přesnějšího sčítance a tedy snížení přesnosti výsledku. Zlepšení můžeme dosáhnout zaokrouhlením výsledku podle hodnoty oříznuté části.

[1] Jsou uvedeny pouze celočíselné části. Skutečné nejvyšší číslo bude rovno součtu uvedeného a čísla 1-*krok*.

2. Nebo sčítance převést na formát s **vyšším počtem desetinných cifer**, čímž zachováme přesnost, avšak za cenu snížení rozsahu proměnné a tím nebezpečí přetečení, kterého si musíme být vědomi.
3. Použít kombinaci uvedených možností a částečně zachovat přesnost za určitého snížení rozsahu výsledku oproti sčítanci s vyšším rozsahem.

Pro úplnost budiž řečeno, že ALU DSC řady c2000 umožňuje namísto přetečení výsledku a nastavení bit přenosu (carry) i saturaci proměnné a nastavení odpovídajícího bitu. Tím se můžeme vyhnout některým problémům spojeným s přetečením, ale tento stav by přesto u dobře napsaného programu buďto neměl nastat, nebo může být i využit, jak je tomu v případě realizovaného programu (viz kapitolu 3.3.2).

Násobení desetinných čísel

Při vynásobení čísla A formátu $\mathtt{Q}a$ a B formátu $\mathtt{Q}b$ dostaneme výsledek C formátu $\mathtt{Q}c$. Platí, že

$$c = a + b \tag{2.4}$$

Toho se dá využít (jak je ukázáno na realizovaném programu v kapitole 3.3.2) tak, že zvolíme formát 16-bitových čísel A a B tak, aby součet 2.4 byl například 16. Potom máme ve vyšším slově výsledku přímo jeho celočíselnou část a v nižším část desetinnou.

Dělení desetinných čísel

Pro výše uvedené proměnné platí při $A/B = C$ pro formát výsledku $\mathtt{Q}c$

$$c = b - a. \tag{2.5}$$

Bude-li $a > b$, vyjde nám $c < 0$, což znamená, výsledek dělení je vlastně roven $C \cdot 2^{-c}$.

Mocnina, odmocnina

Čtverec proměnné A je vlastně $A \cdot A$, tedy formát výsledku bude $\mathtt{Q}(a+a)$. Pro n-tou mocninu A tedy $\mathtt{Q}(n \cdot a)$. Podobně pro n-tou odmocninu platí

$$\sqrt[n]{A \cdot 2^{-a}} = \sqrt[n]{A} \cdot 2^{-a/n}. \tag{2.6}$$

2.1.2 Základní operace

Základní matematické operace s výjimkou dělení (a někdy i násobení) je schopná rychle vykonávat aritmeticko-logická jednotka (ALU) daného mikroprocesoru. V případě, že nám například nestačí rozsah čísel se kterými ALU počítá, jsme i v jejich

případě nuceni realizovat výpočet programově. ALU DSC řady c2000 pracuje se `signed` i `unsigned` čísly v rozsahu 32 bitů, přičemž saturaci a bitové posuvy je možné provádět i s proměnnými délky 64 bitů (typ `long long`).

V případě dělení jsme obvykle odkázáni na softwarové řešení. Díky tomu je značná snaha se dělení vyhnout pomocí jiných operací.

Sčítání, odečítání

Sčítání je v ALU realizováno binárními sčítačkami. Rodina c2000 disponuje 32-bitovou sčítačkou. V případě požadavku sčítání čísel většího rozsahu, musíme sečíst nejdříve nejnižších 32 bitů sčítanců a dále sečítat vyšší 32-bitové řády plus případný přenos z řádů nižších — v asembleru instrukcí `ADDCL`, v C pomocí předdefinované funkce `long __addcu(long, unsigned int);` nebo použít pro výsledek typ `long long`. Pro aplikace, kde potřebujeme sčítat nebo odečítat větší množství čísel malého rozsahu (např. 4 8-bitová čísla), můžeme uložit tyto čísla do jednoho 32-bitového dvouslova a sčítání/odečítání provádět se všemi čísly najednou. Je však zapotřebí zajistit, aby nemohl nastat přenos z kteréhokoli bajtu do bajtu vyššího. Podle [7, kap. 2-17] můžeme napsat kód pro součet $s = x+y$ s ošetřením přetečení následovně:

$$\begin{aligned} s &= (x \wedge \texttt{0x7F7F7F7F}) + (y \wedge \texttt{0x7F7F7F7F}) \\ s &= ((x \oplus y) \wedge \texttt{0x80808080}) \oplus s \end{aligned} \tag{2.7}$$

A pro odečítání

$$\begin{aligned} s &= (x \vee \texttt{0x80808080}) - (y \wedge \texttt{0x7F7F7F7F}) \\ s &= ((x \oplus y) \vee \texttt{0x7F7F7F7F}) \equiv s \end{aligned} \tag{2.8}$$

V případě, že nemůže dojít k přetečení výsledku do vyššího bajtu, stačí samozřejmě pouhé sečtení sčítanců.

Násobení

Rodina c2000 disponuje 32-bitovou násobičkou, která pracuje jak se signed, tak unsigned čísly a dává 64-bitový výsledek, uložený v registrech ACC (vyšších 32 bitů) a P (nižších 32 bitů). Při násobení má totiž výsledek násobení čísela A o délce n_A a B o délce n_B bitů nélku $n = n_A+n_B$. V případě, že potřebujeme vynásobit čísla delší než 32 bitů, budeme postupovat podobně, jako při násobení dvou vícecifferných čísel na základní škole. Pouze nezapisujeme výsledky násobení jednotlivými řády násobitele, které bychom posléze sečetli, ale výsledky dílčích násobení přičítáme ke konečnému výsledku rovnou s patřičným posunem. Pro násobení dvou 64-bitových

čísel $A \cdot B = C$ by postup vypadal následovně: (indexy značí bity proměnné)

$$\begin{aligned} C_{63-0} &= A_{31-0} * B_{31-0} \\ C_{95-0} &= C_{63-0} + (A_{63-32} * B_{31-0}) << 32 \\ C_{96-0} &= C_{95-0} + (A_{31-0} * B_{63-32}) << 32 \\ C_{127-0} &= C_{96-0} + (A_{63-32} * B_{63-32}) << 64 \end{aligned} \tag{2.9}$$

To však platí v případě násobení čísel typu `unsigned`, pro čísla se znaménkem je třeba brát v úvahu znaménka obou proměnných. Konkrétně bychom uvedený postup změnili na vyhodnocení znamének (jsou-li různá, je výsledek záporný), vynásobení absolutních hodnot obou čísel a následnou korekci znaménka výsledku.

Poměrně často využívaným algoritmem při výpočtu polynomů druhého a vyššího řádu, je Hornerovo schéma. Máme polynom n-tého stupně zadán v následujícím tvaru

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \tag{2.10}$$

který můžeme postupným vytýkáním x převést na tvar

$$P(x) = a_0 + x \cdot \Big(a_1 + x \cdot \Big(a_2 + \ldots x \cdot (a_{n-1} + a_n \cdot x)\Big)\Big). \tag{2.11}$$

Při samotném výpočtu potom postupujeme od vnitřní závorky. Pro výpočet polynomu v tomoto tvaru je tedy zapotřebí provést n násobení a n sčítání, kdežto při výpočtu polynomu ve tvaru výrazu 2.10 až $(n^2 + n)/2$ násobení a n sčítání.

DSC řady c2000 mají, stejně jako digitální signálové procesory DSP, navíc oproti běžným μP hardwarovou násobičku rozšířenou o tzv. MAC (*Multiply and ACcumulate*) jednotku. Ta kromě násobení dvou čísel ještě přičte výsledek k další proměnné, to vše během jednoho instrukčního cyklu. MAC řady c2000 podporuje i dvě 16-bitové MAC operace během jednoho instrukčního cyklu prostřednictvím instrukce `DMAC`. Instrukce MAC jsou velmi často používány například v algoritmech číslicové filtrace (IIR a FIR filtry, ...). Lze je využít také při výpočtu polynomů ve tvaru výrazu 2.11.

Na platformách, kde je násobení pomalejší než šćítání a bitové posuny, se často realizuje rychlé násobení konstantou právě pomocí bitových posunů a sčítání. V případě DSC řady c2000 bychom však kýženého efektu nedosáhli, protože instrukce násobení i MAC trvá pouze jeden instrukční cyklus. Více o těchto metodách najdeme například v [7].

Dělení

Dělení je prakticky vždy pomalejší než jiné základní perace, a to i v případě, že je implementováno hardwarově. DSC řady c280x však instrukcí dělení nedisponují a je proto třeba se dělení vyhnout použitím jiného postupu výpočtu, převést jej na jinou

opearaci nebo, v krajním případě, jej implementovat softwarově. V této práci bude rozebrána zejména druhá možnost. První totiž je plně na důvtipu programátora, zatímco softwarová implementace je natolik výpočetně náročná, že se pro použití v rychlých řídicích algoritmech prakticky nehodí. V knihovně IQmath (viz [3]) je definována funkce `long __IQdiv(long, long);`, která využívá Newtonovu iterační metodu dělení s počáteční hodnotou danou tabulkou, uloženou v paměti ROM. Tato funkce trvá 63 instrukcí (podle [3, kap. 5.3]), což je o dost více, než níže popsané postupy.

Budeme se tedy snažit převést dělení na jinou operaci. Nejjednodušším případem je **dělení konstantou**, která je navíc **mocninou 2**. Potřebujeme-li dělit kladnou konstantou A (pro kterou platí $A = 2^n$), potom stačí dělenec posunout o n bitů doprava.

$$n = \log_2 A = \frac{\ln A}{\ln 2} \tag{2.12}$$

Výsledek odpovídá celočíselnému dělení, zbytek je zahozen při bitovém posunu. V případě, že nás zajímá i desetinná část výsledku, je elegantním řešením pouze pohlížet během dalších výpočtů na výsledek jakoby byl ve formátu `Q`n a nijak jej neupravovat. Nebo můžeme předchozí přístupy spojit, dělenec posunout o m bitů doprava (platí $m < n$) a číslo dále brát jako formát `Q`$(n-m)$. Pro zvýšení přesnosti výsledku při jeho posunování doprava jej můžeme zaokrouhlit. To nejlépe provedeme přičtením čísla 2^{m-1} k výsledku před jeho posunutím o m míst doprava.

Pokud potřebujeme dělit prakticky libovolnou konstantou (jinou než mocnina 2), převedeme dělení na násobení přepočítanou konstantou a následný bitový posun vpravo. Současně můžeme využít metod, popsaných o odstavec výše k získání výsledku požadované přesnosti. Potřebujeme-li například vydělit číslo A ve tvaru `Q`a celým číslem $B = 100$, můžeme postupovat takto:

$$\begin{aligned} C &= A \cdot \lfloor 2^{16}/B \rceil = a \cdot \lfloor 2^{16}/100 \rceil \\ C &= C >> 16 \end{aligned} \tag{2.13}$$

V souladu s předchozím textem můžeme vynechat bitový posuv a s výsledkem C nakládat jako s číslem ve formátu `Q`c =`Q`$(a+16)$. Přesnost výsledku ale nemusí být vždy dostačující — vlastně jsme násobili číslem

$$k = \left\lfloor \frac{2^{16}}{B} \right\rceil = \left\lfloor \frac{65536}{100} \right\rceil = 655 = \texttt{0x028F}, \tag{2.14}$$

které je zaokrouhlené. Pro zvýšení přesnosti výsledku stačí, když upravíme konstantu k tak, aby v binární podobě neobsahovala počáteční nuly a výsledek C bude přenější o takový počet míst, kolik nul jsme odstranili z k. Například použitá konstana `0x028F` má prvních 6 bitů rovno 0. Můžeme tedy přepočítat k na

$$k' = \left\lfloor \frac{2^{16+6}}{100} \right\rceil = \left\lfloor \frac{4194304}{100} \right\rceil = 41943 = \texttt{0xA3D7}. \tag{2.15}$$

Výsledek násobení $C = A \cdot k'$ je potom roven číslu $C = A/B$ ve formátu $\mathtt{Q}(a{+}16{+}6)$. Konstantní dělitel B může samozřejmě být i ve tvaru $\mathtt{Q}b$. Výsledný podíl C pak bude ve tvaru $\mathtt{Q}(c+b)$. Díky zaokrouhlování konstanty je i výsledek zaokrouhlený, což může v některých případech (pokud potřebujeme např. zbytek po celočíselném dělení) být problém. Potom stačí k resp. k' místo normálního zaokrouhlování je-li $B > 1$ zaokrouhlit vždy dolů, jeli $B < 1$ zaokrouhlit vždy nahoru.

V případě, že se přesto dělení proměnnou nevyhneme, převedeme dělení na násobení převrácenou hodnotou. To se nejčastěji provádí pomocí Newtonovy metody (také bývá označována jako metoda Newton–Raphsonova nebo metoda tečen). Princip metody je zřejmý z jejího definčního vztahu

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}, \qquad n \geq 0 \tag{2.16}$$

kde y_{n+1} je výsledek iterace $n+1$, y_n je výsledek předchozí iterace, $f(y_n)$ je funkce, jejíž hodnotu chceme v daném bodě x přibližně zjistit a $f'(y_n)$ je její derivace. Dosazením $f(y) = y^{-1} - x$ dostaneme pro n-tou iteraci

$$y_{n+1} = y_n \left(2 - x \cdot y_n\right). \tag{2.17}$$

Rychlost konvergence řady je závislá na volbě počátečního odhadu y_0. Ten se může vypočítat například jako rozdíl logaritmů dělence a dělitele (viz příslušná kapitola o výpočtu logaritmů). Tím totiž získáme alespoň předpoládaný řád výsledku. Případně můžeme využít tabulku počátečních hodnot, kterou tímto mezivýsledkem indexujeme. Pro volbu počátečního odhadu je možné také využít program popsaný v příloze B.

2.1.3 Goniometrické funkce

Výpočet goniometrických funkcí lze opět řešit několika způsoby. Zvolení konkrétního postupu je v případě řídicích algoritmů dáno požadavky na maximální rychlost, přesto však co možná nejmenší velikost a zároveň dostatečnou přesnost. To jsou ale požadavky značně protichůdné a konečné řešení je tedy vždy kompromisem. Pokud bychom nepožadovali co nejvyšší rychlost algoritmu, zvolíme zřejmě aproximaci Taylorovým polynomem dostatečně vysokého stupně. Pro funkci sinus je jeho tvar následující:

$$\sin(x) = \sum_{n\to 0}^{\infty} \frac{(-1)^n}{(2n+1)!} \cdot x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \tag{2.18}$$

Uvedený polynom 7. stupně má na rozsahu $-1 < x < 1$ chybu jen 0,003%. Ta se však dále zvyšuje se čtvercem x, takže pro přesný výpočet bychom potřebovali

buďto zvýšit stupeň polynomu nebo využít periodicity funkce sinus a symetrie jejího průběhu podle osy x. Stačilo by nám tedy znát hodnoty pouze prví čtvrtperiody a pro ostatní úhly využít těchto vlatností. Podobně můžeme využít Taylorova polynomu k výpočtu funkce kosinus:

$$\cos(x) = \sum_{n\to 0}^{\infty} \frac{(-1)^n}{(2n)!} \cdot x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \tag{2.19}$$

V řídicích algoritmech je ale rychlost prioritou a proto se výpočty goniometrických funkcí provádějí pomocí tabulky hodnot. Požadujeme-li vysokou přesnost výsledku, můžeme aproximovat hodnoty dané tabulkou. V nejjednodušším případě lineární aproximací, chceme-li ještě vyšší přesnost, pak pomocí složitějších aproximací na základě více hodnot. Dále bude v této části práce zhodnoceno několik metod výpočtu funkce sinus, která je velmi často používána. Uvedené postupy je možné aplikovat i na mnohé jiné funkce.

Tabulka hodnot

Tabulku hodnot funkce můžeme řešit dvěma způsoby, z nichž každý má své výhody i nevýhody, které budou dále rozvedeny. Počet hodnot tabulky volíme obvykle roven mocnině 2, aby bylo možné tabulku jednoduše indexovat a při vhodné normalizaci indexu tím zároveň zamezíme možným chybám při překročení rozsahu tabulky.

1. Prostou tabulkou hodnot funkce. Graf, odpovídající jejím hodnotám při délce tabulky 32 hodnot, znázorňuje obrázek 2.1.
2. Tabulkou hodnot, které odpovídají aritmetickému průměru hodnoty příslušícímu aktuálnímu a následujícímu indexu tabulky. Její podobu ilustruje graf na obrázku 2.2.

Hodnoty tabulky T_1 tedy budou pro všechny indexy a tabulky funkce sinus o délce n hodnot dány pro první typ tabulky vztahem

$$T_2(a) = \sin\left(a \cdot \frac{2\pi}{n}\right), \tag{2.20}$$

zatímco v případě druhého typu, tabulky T_2, bude vztah následující:

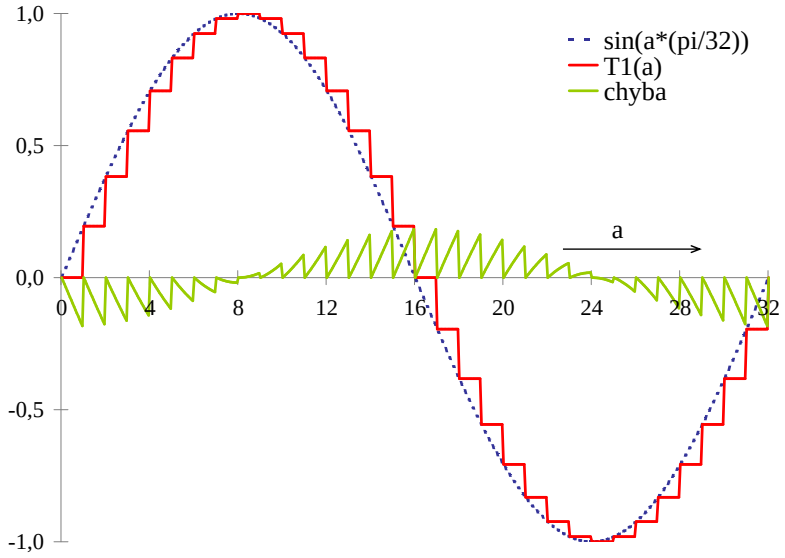
$$T_2(a) = \frac{\sin\left(a \cdot \frac{2\pi}{n}\right) + \sin\left((a+1) \cdot \frac{2\pi}{n}\right)}{2}. \tag{2.21}$$

Na grafech 2.1 a 2.2 jsou vyneseny i chyby Δ použitých tabulek, pro které platí

$$\Delta(a) = T_x(a\%n) - \sin(a) \tag{2.22}$$

kde symbol % vyjadřuje celočíselné dělení. Z průběhu chyb je zřejmé, že tabulka T_2 poskytuje hodnoty s téměř poloviční chybou oproti tabulce T_1. Maximum chyby Δ_{max} se u obou tabulek blíží hodnotě

$$\Delta_{max} = \max\left|T_x(a) - T_x(a \pm 1)\right| \qquad \text{pro } a = 0\ldots n \tag{2.23}$$



Obr. 2.1: Graf funkce sinus a její tabelace prvního typu včetně chyby tabelace.

Dále z grafů vidíme, že chyba se zvětšuje se strmostí funkce, tedy s velikostí její derivace. V místech, kde je zadávací funkce nejstrmější, je nejvyšší i diference odpovídajících hodnot tabulky. Naopak ve vrcholech funkce sinus, kdy se její derivace blíží 0, jsou chyby obou tabulek nejnižší. Můžeme tedy přímo vyčíslit chybu tabulky funkce sinus, například jako

$$\begin{aligned} \Delta_{max} &= T_x\left(\tfrac{n}{2}-1\right) - T_x\left(\tfrac{n}{2}\right) = \\ &= \sin\left(\left(\tfrac{n}{2}-1\right)\cdot\tfrac{2\pi}{n}\right) - \sin\left(\tfrac{n}{2}\cdot\tfrac{2\pi}{n}\right) = \\ &= \sin\tfrac{2\pi}{n} \end{aligned} \tag{2.24}$$

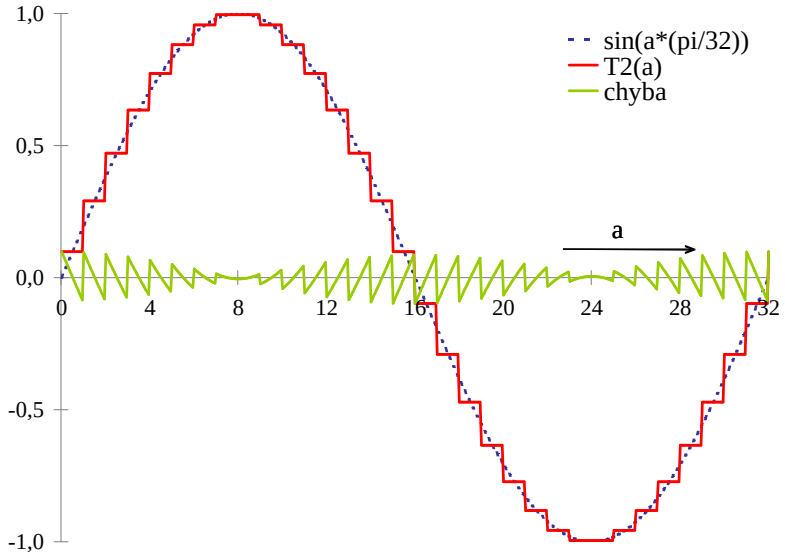
Skutečná maximální chyba se bude limitně blížit Δ_{max} tak, jak se bude reálný úhel α aproximované tabelované funkce $\sin\alpha$ blížit hodnotě $\frac{2\pi}{n}$.

Lineární aproximace

Výrazného zmenšení chyb uvedených tabulek dosáhneme použitím aproximace hodnot, které jsou mezi libovolnými dvěma indexy tabulky. Nejjednodušší je lineární aproximace, která spočívá v prostém proložení sousedních hodnot tabulky přímkou. Její obecný vztah má tvar

$$y_a = y_0 + (x_a - x_0)\cdot\frac{y_1 - y_0}{x_1 - x_0}, \tag{2.25}$$

kde $y_0 = f(x_0)$, $y_1 = f(x_1)$ jsou hodnoty, mezi kterými povede aproximační přímka a y_a je aproximovaná hodnota v bodě x_a. Dělení se naštěstí v našem případě vyhneme, protože čísla x_1 a x_0 jsou indexy tabulky a jejich rozdíl je tedy vždy roven 1 (případně



Obr. 2.2: Graf funkce sinus a její tabelace druhého typu včetně chyby tabelace.

-1, postupujeme-li od konce tabulky). Vztah 2.26 se tedy zjednodušší na

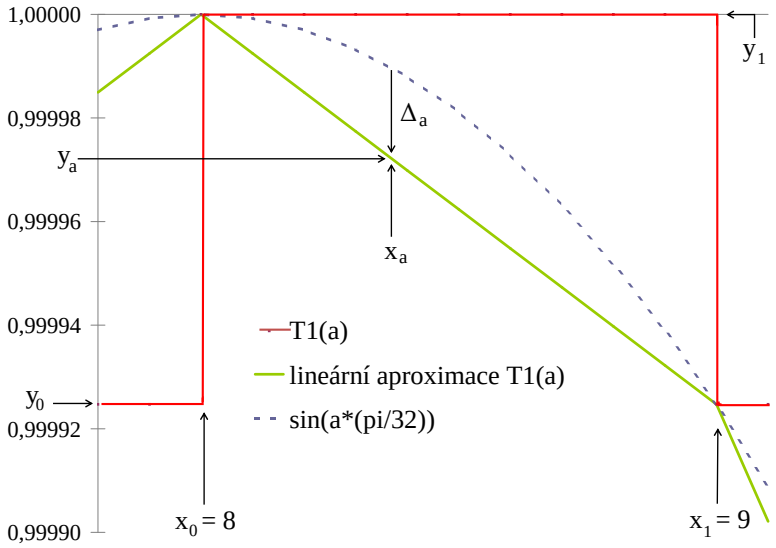
$$y_a = y_0 \cdot (1 - x_a + x_1) + y_1 \cdot (x_a - x_1). \tag{2.26}$$

Je zřejmé, že lineární aproximace nemá smysl pro tabulku T_2. Došlo by totiž k nežádoucímu posunu fáze výsledku o

$$\Delta_\omega = \frac{i}{2} \cdot \frac{2\pi}{n}. \tag{2.27}$$

Použijeme tedy tabulku T_1. Z principu lineární aproximace je vidět, že se bude od hodnot aproximované funkce nejvíce odchylovat v těch částech průběhu, které jsou nejvíce vzdáleny od lineárního. V případě funkce sinus i kosinus jsou to jejich extrémy v bodech $\pi/2 + k\pi$ resp. $k\pi$. Naopak v bodech, kdy je hodnota těchto funkcí limitně blízká nule je můžeme považovat za lineární, a chyba aproximace je tedy minmální. To vidíme i z průběhu chyby na grafu 2.4.

Knihovna IQmath poskytuje k použití goniometrické funkce sinus a kosinus, přičemž obě mají část tabulky hodnot společnou. Je to část mezi indexy 128 až 512, které odpovídají rozsahu $<\pi/2; 2\pi>$ pro funkci sin a $<0;3\pi/2>$ pro kosinus. Jejich vlastnosti shrnuje následující tabulka. Vidíme, že rychlost 46 (resp. 44) instrukčních cyklů není vzhledem k dosažené přesnosti malá. Přesnost 30 bitů je však v řídicích algoritmech zřídkakdy nutná. Vždyť například PWM modulátor, pro který se počítají hodnoty v dále uvedeném programu, pracuje s hodnotami v rozsahu <0;1666>. To není ani celý rozsah 12-bitové proměnné. Je tedy nasnadě, že popsaná lineární aproximace je více než dostačující (je-li vůbec zapotřebí, jak je shrnuto v kapitole 3.3.2).



Obr. 2.3: Detail lineární aproximace $\sin(a)$ mezi indexy $a = 32$ a 33 tabulky T_1

	rychlost	**přesnost**	**velikost**
funkce	instr. cykly	bity	16-bitová slova
`__IQsin`	46	30	49
`__IQcos`	44	30	47

Tab. 2.3: Goniometrické funkce knihovny IQmath. Funkce mají jeden parametr typu `long` a vrací typ `long` ve tvaru daném opět použitou proměnnou (viz [3]).

2.1.4 Cyklometrické funkce

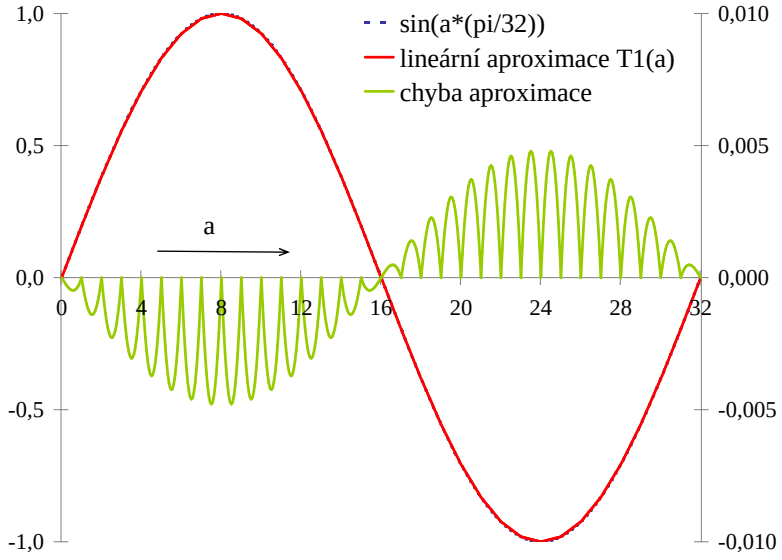
Inverzní funkce k funkcím goniometrickým, funkce cyklometrické, bývají v případě pořeby počítány podobně, jak bylo uvedeno v předcházející části na příkladu funkce sinus. Nejčastěji používanou je z cyklometrických funkcí zřejmě funkce arkustangens.

$$\operatorname{arctg}(x) = \frac{1}{\operatorname{tg}(x)} \tag{2.28}$$

Je využívána například při transformacích souřadnic pro výpočet úhlu vektoru z jeho složek a podobně. Můžeme ji také vyjádřit pomocí Taylorovy řady jako

$$\operatorname{arctg}(x) = \sum_{n\to 0}^{\infty} \frac{(-1)^n x^{2n+1}}{2x+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \tag{2.29}$$

Můžeme se také setkat s funkcí $\operatorname{arctg2}(x, y)$, jejímž výsledkem je úhel v radiánech, které svírá osa x s bodem x, y (vrchol úhlu je v počátku — bodě $0, 0$). Pro $y > 0$ je úhel kladný, pro $y < 0$ záporný. Funkce je definována pomocí běžné funkce $\operatorname{tg}(\alpha)$



Obr. 2.4: Lineární aproximace $\sin(a)$ mezi hodnotami tabulky T_1 s výslednou chybou (vynesena na vedlejší osu y). Průběh funkce $\sin(a)$ (tečkovaně) je překryt aproximovaným průběhem (červená plná čára), od kterého se jen málo liší.

takto

$$\operatorname{arctg} 2(x,y) = \begin{cases} \operatorname{arctg}(y/x) & x > 0 \\ \pi + \operatorname{arctg}(y/x) & y \geq 0, x < 0 \\ -\pi + \operatorname{arctg}(y/x) & y < 0, x < 0 \\ \pi/2 & y > 0, x = 0 \\ -\pi/2 & y < 0, x = 0 \\ \text{nedefinováno} & y = 0, x = 0 \end{cases}. \tag{2.30}$$

Můžeme také postupovat pomocí vzorců pro poloviční úhel funkce tangens

$$\operatorname{arctg} 2(x,y) = 2\operatorname{arctg}\frac{y}{\sqrt{x^2+y^2}+x} = 2\operatorname{arctg}\frac{\sqrt{x^2+y^2}-x}{y}. \tag{2.31}$$

V knihovně IQmath jsou definovány funkce arcsin, arccos, arctg a arctg2 (ta má 2 parametry — délky odvěsen — a vrací délku přepony pomyslného pravoúhlého trojúhelníku). Funkce a jejich vlastnosti jsou shrnuty v následující tabulce.

2.1.5 Odmocnina

Je nejčastěji počítána pomocí Newtonovy iterační metody, popsané v závěru části zabývající se dělením. Pro výpočet odmocniny budeme řešit funkci $y^2 - x = 0$ pro $x \geq 0$. Potom dostaneme pro iteraci $n+1$ vztah

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right). \tag{2.32}$$

	rychlost	**přesnost**	**velikost**
funkce	instr. cykly	bity	16-bitová slova
`__IQasin`	154	—	82
`__IQacos`	170	—	93
`__IQatan`	109	25	123
`__IQatan2`	109	26	123

Tab. 2.4: Cyklometrické funkce knihovny IQmath. Funkce mají jeden parametr typu `long` (resp. 2 u `IQatan2`) a vrací typ `long` ve tvaru daném opět použitou proměnnou (viz [3]).

Pro dosažení konvergence je (zvláště v případě použití celočíselné matematiky) zapotřebí vhodně volit počáteční odhad y_0. K tomu může napomoci program, jehož zdrojový kód je uveden v příloze B, kde jsou i další informace o použití tohoto programu a jeho možnostech. Při vhodné volbě počátečního odhadu stačí i jediná iterace pro získání odmocniny s přesností lepší než 0,2%, ale rozsah počátečních hodnot, pro které toto platí, je tak malý, že se většímu počtu iterací nevyhneme. Hlavní nevýhodou je zde ale nutnost dělení proměnnou y_n.

Další možností výpočtu odmocniny, která nevyžaduje dělení, je použít metodu půlení intervalů. V každé iteraci postupujeme následovně

$$\begin{aligned} a &= a >> 1 \\ y &= \begin{cases} y + a & \text{pro } y^2 < x \\ y - a & \text{pro } y^2 > x \end{cases} , \end{aligned} \tag{2.33}$$

kde a je počáteční odhad, který zvolíme nejčastěji jako polovinu maximálního výsledku, y je výsledný odhad odmocniny po n-té iteraci a x je odmocňované číslo. Je evidentní, že s každou iterací se zvýší přesnost výsledku o jeden řád.

Knihovna IQmath nabízí funkci `long __IQsqrt(long);`, která počítá poměrně přesně (s přesností na 29 bitů podle [3, str. 26]) odmocninu. Tato funkce využívá pro počáteční odhad Newtonovy metody tabulku hodnot v ROM paměti. Podle [3, str. 26] trvá 63 instrukčních cyklů a její velikost je 66 slov.

2.1.6 Inverzní odmocnina

Převrácená hodnota odmocniny se nejčastěji počítá Newtonovou iterační metodou. Její tvar v tomto případě vypadá následovně

$$y_{n+1} = \frac{1}{2} y_n \left(3 - x y_n^2\right) . \tag{2.34}$$

Pro dosažení konvergence je (zvláště v případě použití celočíselné matematiky) zapotřebí vhodně volit počáteční odhad y_0. K tomu může napomoci program, jehož zdrojový kód je uveden v příloze B, kde jsou i další informace o použití tohoto programu a jeho možnostech. Při vhodné volbě počátečního odhadu stačí i jediná iterace pro získání odmocniny s přesností lepší než 0,2%.

V knihovně IQmath najdeme funkci `long __IQisqrt(long);`, která počítá inverzní odmocninu s podobnými výsledky, jako předchozí funkce pro výpočet odmocniny. Využívá také tabulku hodnot v ROM paměti pro počáteční odhad, trvá 64 instrukčních cyklů, její přesnost je také 29 bitů a velikost 69 slov [3, str. 26].

2.1.7 Logaritmus

Logaritmus je opět nejčastěji počítá Newtonovou iterační metodou. Pro logaritmus o základu 2 $f(y_n) = 2^x - a$ platí

$$y_{n+1} = y_n + \frac{1}{\ln 2}\left(\frac{x}{2^{y_n}} - 1\right). \tag{2.35}$$

Zajímavé je, že metoda konverguje (podle [7]) i při zokrouhlení $1/\ln 2 \doteq 1,4427$ na 1 či 2. Konvergence se tím ale zpomalí. Logaritmus o základu 2 vrací číslo shodné s počtem bitů logaritmovaného čísla x. DSC rodiny c2000 mají instrukci `CSB ACC`, která, jak uvádí [1, str. 240], zjistí přímo počet stejných úvodních bitů (postupuje od nejvýznamnějších bitů, tedy zleva), odečte 1 a výsledek uloží do registru `T`. Tato instrukce je právě využívána knihovnou IQmath k indexaci tabulek počátečních odhadů některých funkcí, k čemuž se tato instrukce velmi hodí. V případě psaní kódu pro architekturu, která podobnou funkcí nedisponuje, lze nalézt několik jednoduchých a poměrně rychlých algoritmů ($10\tilde{2}0$ základních instrukcí) k jejímu nahrazení v [7, kap. 5].

Kinhovna IQmath žádnou funkci k výpočtu logaritmu neobsahuje.

2.2 Optimalizace zdrojového kódu

Jak již bylo řečeno, úkolem optimalizace je zefektivnění generovaného strojového[1] kódu. V kompilovaných vyšších programovacích jazycích, mezi něž C patří, odpovídá za překlad zdrojového kódu do asembleru[2] daného procesoru kompilátor (překladač).

[1] Strojový kód je binární reprezentací instrukcí daného mikroprocesoru, na základě které mikroprocesor kód vykonává.

[2] Asembler, také nazývaný jazyk symbolických instrukcí, je přehledným zápisem instrukcí procesoru, rozšířený o symbolické zápisy adres, konstant, proměnných, návěští, makra a jiné věci, usnadňující zápis programu. Jedna instrukce v asembleru odpovídá jedné strojové instrukci (ale její trvání může být delší než jeden strojový cyklus). Asembler je i program, zajišťující překlad kódu v asembleru do strojového kódu.

Úroveň kompilátoru a jeho optimalizačních schopností je tedy rozhodující a jeho výběru je dobré věnovat náležitou pozornost. Pro platformy jako x86, x86_64, PowerPC, ARM, SPARC a jiné, patří bezesporu mezi špičku open source[3] překladač GCC. Dále existuje množství komerčních překladačů, jejichž použití se v případě některých mikroprocesorů (jako jsou například právě některé DSC Texas Instruments) nevyhneme.

S optimalizací začínáme u (zatím neoptimalizovaného) kódu, který postupně uravujeme k dosažení zlepšení požadovaného parametru. Pomocí simulátoru, emulátoru nebo prostého sečtení strojových cyklů (respektive odečtení počáteční od koncové adesy při optimalizaci velikosti) vygenerovaného strojového kódu zjistíme jakého zlepšení jsme dosáhli. Je-li výsledek postačující, není další optimalizace nutná a můžeme ji tedy ukončit.

Dále jsou uvedeny různé metody optimalizace v jazyce C. Pořadí jejich aplikace může být v různých případech odlišné, záleží na postupu programátora. Výsledná optimalizace je prakticky vždy kombinací několika nebo i všech uvedených postupů. Samozřejmostí je, že programátor nejprve provede rozbor možných řešení problému a zvolí nejlepší možné řešení.

2.2.1 Datové typy

Datové typy proměnných je třeba volit tak, abychom se vyhnuli nutnosti častého přetypování, které může (ale ne vždy musí, v závislosti na daném případě i architektuře) způsobovat při překladu vkládání dalších instrukcí a tím jak zvětšování velikosti kódu, tak jeho zpomalování. Dobrým pomocníkem zde bývá datový typ `union`, díky kterému můžeme přistupovat jak kupříkladu k proměnné typu `long` jako celku nebo jeho polovinám typu `int` (resp. `unsigned int`) každé zvlášť. Zde je třeba mít na vědomí, je-li typ `signed` či `unsigned` a vyvarovat se tak případných chyb.

Pozornost zaslouží také použití klíčového slova `volatile`, které způsobí, že překladač při optimalizaci nezkopíruje obsah takto definované proměnné, obvykle v rámci zrychlení smyčky, do některého registru procesoru. Vždy se tak vyhodnocuje aktuální hodnota proměnné, tedy i s případnými změnami způsobenými obsluhou přerušení. Použití tohoto klíčového slova v případech, kdy to není nutné, zabrání překladači v optimalizaci a tím může zpomalit výsledný kód.

[3] S otevřeným zdrojovým kódem, dostupný zdarma.

2.2.2 Předávání parametrů

Při volání funkcí máme v C několik možností jak jim předávat parametry. Je jasné, že nejhorší možností je předávat větší množství dat (pole, strukturu, řetězec a pod.) hodnotou. Tím dojde ke zkopírování dat na zásobník, což může trvat poměrně dlouho a navíc tím plýtváme pamětí dat. V takovém případě je lepší možností předávat pouze *ukazatel* na první prvek pole nebo datové struktury. Další možností, která je ve světě osobních počítačů často považována za špatnou s ohledem na modularitu a rozšiřitelnost software, je použití globálních proměnných. Jejich použití nám dává větší přehled o nárocích programu na paměť dat, protože se parametry neukládají na zásobník. Zvýšení rychlosti programu, oproti řešení s předáváním parametrů hodnotou, můžeme zaznamenat pouze u funkcí s větším počtem parametrů (nebo parametry, jejichž velikost přesahuje mez danou konkrétní architekturou a překladačem), u kterých tím ušetříme práci se zásobníkem. Dále si využitím globálních proměnných usnadníme použití inline funkcí (viz 2.2.9).

2.2.3 Přímá vs. nepřímá adresace

Obecně můžeme říci, že algoritmy využívající proměnné na předem známých adresách, tedy přímé adresace, bývají rychlejší. Záleží však na instrukčním souboru dané architektury. Ta totiž může zahrnovat (jako v případě DSC rodiny c2000) režimy adresace, kdy je možné během jediného strojového cyklu provést nejen operaci s operandy dané instrukce, ale i inkrementaci/dekrementaci ukazatele — CPU obsahuje i tzv. aritmetickou jednotku adres (více např. viz [1, kap. 2.1]). Vhodným použitím nepřímé adresace tedy můžeme v některých případech (zpracování pole hodnot ve smyčce a pod.) dosáhnout zefektivnění programu.

2.2.4 Podmínky

Další věcí, která si zaslouží pozornost, je zápis podmínek. Je třeba zvolit výpočetně co možná nejjednodušší porovnávaný výraz a zamyslet se nad koncepcí případných vnořených podmínek. Zvláště u některých moderních μP, disponujících pipeline[4], způsobuje totiž každý skok zpomalení běhu programu. Z pipeline se totiž musí vyhodit již předzpracované instrukce a musí se začít s dekódováním od instrukce na cílové adrese skoku. Zde však opět záleží na nuancích použité architektury. Některé (např. ARM) využívají tzv. delayed branch (spožděné větvení programu), kdy se zpracuje ještě jedna nebo i několik instrukcí po instrukci větvení bez ohledu na to, zda se bude skákat jinam či ne. Díky tomu se nemusí zahazovat tyto již zpracovávané

[4]Pipeline umožňuje paralelní zpracování instrukcí — v případě DSC rodiny c2000 je osmiúrovňová a najednou se tedy může v různé fázi zpracování vyskytovat až osm instrukcí.

instrukce z pipeline. Je zřejmé, že tohle lze provést pouze s instrukcemi, které by byly vykonány v obou situacích (skok/běh bez skoku) a přitom jejich přesunutí neovlivní vyhodnocení větvení. Ani v asemblerových zdrojových kódech ale nemusíme na toto chování brát zřetel, protože o vhodné přeskupení instrukcí se postará asembler sám při překladu do strojového kódu.

2.2.5 Cykly

Chceme-li dosáhnout maximální rychlosti programu, je možné kratší cykly s konstantním počtem opakování rozvinout na prosté opaování těla cyklu. Tím ušetříme instrukce skoku, které patří k časově náročnějším (např. u jádra c28xx jsou 7 resp. 4 strojové cykly dlouhé). Na druhou stranu se tím ale může poměrně značně zvětšit velikost výsledného kódu.

Některé překladače samy umí cykly, u kterých to je možné a v daném případě i výhodné, rozvinout, voláme-li je s odpovídajícím přepínačem.

2.2.6 Rekurzivní funkce

Rekurzivní funkce (funkce volající samy sebe), jsou často dobré z hlediska úspory velikosti programu, avšak rychlost výsledného kódu sníží a to zejména díky velkému počtu volání funkce v poměru k „užitečnému“ kódu a tím i značné režii při práci se zásobníkem. V oblasti řídicích programů se tak s nimi setkáme jen výjimečně.

2.2.7 Tabulky hodnot

Značného urychlení složitějších matematických operací dosáhneme použitím předem vypočtených tabulek hodnot, které indexujeme sečtením počáteční adresy tabulky a vstupní proměnné. Tato metoda je použita pro výpočet funkce sinus v praktické části práce. V tomto případě spočívá celý výpočet pouze ve vhodném škálování vstupní proměnné tak, aby její hodnota 0 až 512 odpovídala úhlu 0 až 2π. Vlastní zjištění odpovídající hodnoty funkce sinus vypadá naříklad takto:

```
// x = {0;512} odpovídá {0;2*PI}
// sin = {-2^30;2^30} odpovídá {-1;1} - formát Q30
const long* const ptr = (long*) 0x3FF000;
long sin = *(ptr + x);
```

Formát ukládání desetinných čísel, jako například uvedený `Q30`, je blíže popsán v kapitole 2.1.1.

Vlastní tabulka je uložena v paměti programu. V uvedeném příkladě byla využita tabulka funkce sinus, uložená v paměti ROM již od výrobce. Pokud chceme vytvořit vlastní tabulku, může použitý kód vypadat následovně:

```
unsigned int tabulka[] = { // převrácené hodnoty ve formátu Q15
                32768,     // 1/2 = 0,5
                21845,     // 1/3 = 0,33...
                16384,     // 1/4 = 0,25
                13107,     // 1/5 = 0,2
                  ...
                1040};     // 1/63 = 0,015873
unsigned int div = tabulka[x]; // div = 1/x
```

2.2.8 Tabulky skoků

Další možné zefektivnění programu spočívá ve využití v předešlém oddíle popsaných tabulek hodnot. Ne však pro zjišťování hodnot, ale pro adresy bloků programu, na které se skočí v případě, že proměnná (často udávající například aktuální stav stavového automatu) má odpovídající hodnotu. Tyto tabulky (tzv. *switchtables*) vytváří automaticky i některé překladače pro konstrukce typu `switch(`*proměnná*`){ case` n`: ... }`.

2.2.9 `inline` funkce

Použitím klíčového slova `inline` u často volaných funkcí dosáhneme vložení jejich těla na všechny místa jejich volání. Tím sice opět značně naroste velikost výsledného kódu, ale eliminují se tím instrukce volání podprogramu, ukládání návratové adresy na zásobník a podobně. Toto řešení se tedy vyplatí především u funkcí, které jsou volány pouze z malého počtu míst. V opačném případě může být nárůst velikosti kódu neúnosný.

Samotný překladač se při překladu s volbou `-o3` (viz kapitolu 2.2.12) může u některých funkcí rozhodnout, že budou inline. Jestli tak učiní, či nikoliv, se rozhodne na základě počtu volání funkce a délky jejího kódu. My můžeme ovlivnit toto rozhodování pomocí přepínače `-oi`*velikost* při volání překladače z příkazového řádku. Je-li parametr *velikost* roven 0, pak je automatické vkládání funkcí vypnuto. Pokud je *velikost* celé kladné číslo, budou vkládány ty funkce, jejichž součin počtu volání a velikosti (ve slovech) je menší než daný práh. Ostatní budou standardně volány. Existuje však několik výjimek, které nebudou nikdy vloženy. Jsou to případy pokud:

1. funkce deklaruje lokální `static` proměnnou nebo proměnnou typu `struct`, `union`, `enum`
2. je parametrem funkce proměnná typu `struct`, `union` nebo `volatile`
3. obsahuje proměnnou typu `static`, `volatile`
4. má proměnný počet argumentů
5. je rekurzivní
6. obsahuje direktivu preprocesoru `#pragma`
7. využívá příliš zásobník (má mnoho lokálních proměnných)

2.2.10 `static inline` funkce

Jejich efekt je podobný, jako u `inline` funkcí, ale překladač je vždy vkládá na požadovaná místa, bez kontroly uvedených pravidel. Funkce se tedy chová do jisté míry jako makro v jazyce C, navíc jsou však kontrolovány typy parametrů. Stejně jako makra se také `static inline` funkce uvádí v hlavičkových souborech. Pokud chceme program přeložit bez optimalizací (pro ladění a pod.), musíme pomocí podmíněného překladu zajistit, že bude dostupný i standardní ekvivalent dané funkce. K tomu se využívá symbol preprocesoru `_INLINE`, který je definován v případě, že kompilátor bude optimalizovat výsledný kód.

2.2.11 Makra v C

Funkce můžeme v jazyce C definovat také jako makra pomocí direktivy preprocesoru `#define`. Výsledný efekt je prakticky stejný jako v případě `static inline` funkcí. Výhodou je, že se vyhneme podmíněnému překladu a v některých případech dosáhneme zvýšení rychlosti kódu (překladač lépe optimalizuje použití jednotlivých registrů — viz [2, kap. 3.3]), nevýhodou to, že není při překladu kontolováno dodržení odpovídajících datových typů u „parametrů funkce“. Jednoduchý příklad makra pro součet dvou vektorů ve složkovém tvaru, může vypadat například takto:

```
#define vadd(ax,ay,bx,by) {\
(ax) += (bx);               \
(ay) += (by);}
```

2.2.12 Optimalizace překladu

Samotné překladače umožňují prostřednictvím různých parametrů příkazové řádky nastavení úrovně optimalizace. Obvykle jsou to především parametry `-o0` až `-o3`, jejichž význam je následující:

- `-o0` povolí optimalizace na úrovni registrů. To zahrnuje následující typy optimalizací:
  - Alokace proměnných do volných registrů procesoru.
  - Odstraňuje kód, který se nikdy nevykoná. (Podmínky typu `if(0){}` a pod.)
  - Kde je to možné, přesune vyhodnocení podmínky ukončení smyčky na její konec, čímž se ušetří skok na následující instrukci při jejím ukončení.
  - Zjednodušuje výrazy. Například výraz `a=(b+2)*2;` upaví na `a=b*2+4;`, je-li to na dané platformě výpočetně jednodušší.
  - Zefektivňuje větvení programu ve smyslu sloučení nadbytečných podmínek a pod. Lepší optimalizace zde můžeme dosáhnout profilováním kódu, kdy se statisticky zjišťuje, které podmínky jsou jak vyhodnocovány a podle výsledku se zpětně zoptimalizuje program tak, abychom dosáhli minimálního možného množství skoků při běhu programu srovnatelném s během při profilaci.
  - Vkládá těla `inline` funkcí na místa jejich volání, je-li to možné (viz kapitola 2.2.9).
- `-o1` provede optimalizace na úrovni bloků kódu. Tedy optimalizace úrovně `-o0` plus:
  - Vytváří lokální kopie konstant `const`.
  - Odstraňuje přiřazení, která nemají v rámci daného bloku význam nebo jsou nadbytečná.
  - Sjednotí společné části výrazů daného bloku kódu a jejich výsledek použije na místo původních částí výrazů.
- `-o2` provede optimalizaci na úrovni funkcí. Zahrnuje kroky `-o0` i `-o1` plus:
  - Optimalizuje smyčky.
  - Sjednotí společné části výrazů (jako u -o1), avšak globálně, ne pouze na úrovni daného bloku.
  - Odstaní přiřazení, která nemají z globálního pohledu význam nebo jsou nadbytečná.
- `-o3` optimalizuje každý soubor jako celek. Zahrnuje všechny předchozí optimalizace plus:
  - Odstraní funkce, které nejsou nikde volány.
  - Zjednodušší funkce, jejichž návratové hodnoty nejsou používány. Tedy upraví je v podstatě na tvar `void` *funkce*`(`*parametry*`);`.
  - Automaticky změní funkce, které jsou dostatečně malé na `inline` (práh porovnání lze měnit přepínačem `-oi`*hodnota* — více viz kapitolu 2.2.9 a nápovědu CCS).
  - Změní pořadí deklarací funkcí tak, aby byly známy jejich parametry před tím, než je optimalizován kód volající tyto funkce.

  - V případě, že je některý z argumentů funkce při každém jejím volání stejný, přesune se jeho hodnota do těla funkce a ta je pak volána bez tohoto argumentu.

V případě, že použijeme přepínač `-o3`, můžeme využít i následující přepínače, které nám umožní další optimalizaci:

- `-on`*n* zakáže nebo povolí vytváření souboru informací o optimalizaci podle hodnoty *n*:
  - $n = 0$: Nevytvářet informační soubor.
  - $n = 1$: Vytvářet informační soubor.
  - $n = 2$: Vytvářet informační soubor se všemi informacemi.
- `-pm` aplikuje optimalizaci stupně `-o3` na celý program jako celek. Tedy ne pouze na každý soubor zvlášť.

Může se zdát, že překladač udělá optimalizaci kódu za nás, to je však mylné. Vždy se vyplatí napsat kód maximálně efektivně a překladač jej ještě více zoptimalizuje. Pokud ale napíšeme kód, který obsahuje zbytečnosti a není efektivní, překladač jej sice značně „učeše“, ale výsledek s největší pravděpodobností nebude tak dobrý.

Popsaný význam jednotlivých přepínačů platí pouze v případě Code Generation Tools verze 4.1.0 (které jsou součástí Code Composer Studio verze 3.1.0). V případě jiných překladačů se může lišit nebo mohou být použity zcela jiné přepínače. Jelikož je ale uvedený překladač v současnosti zřejmě jediným rozšířeným řešením pro DSC Texas Instruments, zabývá se dále tato práce pouze tímto vývojovým softwarem.

2.2.13 Intrinsics — předdefinované funkce

„Intrinsics“ jsou v podstatě krátké funkce, sestávající z jednoho až několika příkazů asembleru daného procesoru. Využívají příkazy daného mikroprocesoru, které by byly v C podstatně složitěji zapsány a vedly by k neefektivnímu výslednému kódu. Nejsou však volány, jsou přímo vkládány na požadovaná místa — jsou tedy obdobou maker nebo `inline` funkcí. Kompilátor architektury c2000 nabízí několik takových funkcí. Například pro omezení rozsahu hodnot proměnné je to funkce `long __IQsat(long` *proměnná*`, long` *maximum*`, long` *minimum*`);`. Ta využívá instrukcí `MINL` (viz [1, str. 312]) a `MAXL` (viz [1, str. 309]), které nejsou z C jinak dostupné. Použití této funkce značně urychlí i zpřehlední program, nemusí se totiž vyhodnocovat podmíněné skoky.

2.2.14 Asemblerové funkce a jejich volání z C

Zřejmě nejefektivnějším řešením, kdy si u časově nenáročných částí programu zachováme pohodlí jazyka C, zato pro obsluhy přerušení a jiné na rychlost náročné

procedury využijeme v maximální míře možnosti procesoru, je použití asemblerových funkcí uvnitř programu, psaného v jazyce C. Protože však kompliátor nemůže mít ponětí o funkcích psaných přímo v asembleru, je veškerá zodpovědnost za správnost funkce, její volání, práci se zásobníkem a podobně, plně na programátorovi. Ten musí přesně vědět, jak kompilátor postupuje při alokaci registrů a paměti pro uložení parametrů volané funkce a kam má funkce ukládat návratovou hodnotu. V případě použité architektury c2000 a překladače Code Generation Tools verze 4.1.0 je postup následující:

1. Mění-li funkce hodnotu kteréhokoliv z registrů `XAR1`, `XAR2` nebo `XAR3`, musí jeho hodnotu uložit. Volající funkce totiž předpokládá, že tyto hodnoty se nezmění. Jakékoliv jiné registry mohou být měněny bez ukládání.
2. Volaná funkce alokuje na zásobníku místo pro všechny lokální proměnné a argumenty funkcí, které může volat. To provede pouze jednou na svém začátku zvýšením hosnoty ukazatele zásobníku `SP`.
3. Následuje vlastní kód funkce, přičemž argumenty jí předávané jsou umístěny podle následujících kritérií:
   - Je-li argumentem 64-bitová proměnná (typ `long long`), je horních 32 bitů umístěno do akumulátoru `ACC`, spodních 32 bitů do registru `P`. Jakákoli další 64-bitová proměnná je umístěna na zásobník.
   - Je-li argumentem 32-bitová proměnná (typ `long`, případně `float`), je první umístěna do `ACC` (je-li volný), každá následující je uložena na zásobník. Tak se děje posupně počínaje posledním argumentem k prvnímu.
   - Argumenty – ukazatele jsou uloženy do registrů `XAR4` a `XAR5`. Případné další na zásobník.
   - Zbylé 16-bitové argumenty jsou zkopírovány do volných registrů v pořadí `AL` (nižší slovo `ACC`), `AH` (vyšší slovo `ACC`), `XAR4`, `XAR5`.
   - Jakékoliv další argumenty jsou umístěny na zásobník v pořadí od posledního neuloženého argumentu po první argument (nejvíce vlevo).
   - Je-li argumentem struktura, předává a ukazatel na ni, případně se zkopíruje na zásobník. Situace je však v tomto případě složitější, viz nápovědu CCS.
4. Volaná funkce uloží návratovou hodnotu podle jejího datového typu:
   - 16-bitová hodnota (`int`, `short`, `char`, `enum`): `AL`
   - 32-bitová hodnota (`long`): `ACC`
   - 64-bitová hodnota (`long long`): `ACC:P`
   - ukazatel: `XAR4`
5. Volaná funkce odalokuje místo na zásobníku, alokované v kroku 2. Odečte tedy od `SP` daný počet slov.
6. Navrátí hodnoty registrů uložené v kroku 1.

7. Pomocí instrukce `LRETR` se navrátí zpět na instrukci následující po instrukci, která funkci zavolala.

2.3 Cena optimalizace

Je evidentní, že optimalizace může být značně časově náročná. To však není jediná cena, kterou za ni platíme. Nepříznivé důsledky optimalizace jsou následující:

1. Nebezpečí vytvoření chyb v programu, vznikající s každou jeho úpravou. Můžeme jej eliminovat důslednou kontrolou a rozdělením programu na bloky s jasně definovanou jednoduchou strukturou. Již toto nás ale může svým způsobem omezit v možnostech optimalizace.
2. Zhoršení přehlednosti kódu, které s sebou přináší podobná nebezpečí, jako bod předchozí. Navíc ztěžuje případné rozšiřování aplikace nebo její změny.
3. Zhoršená přenositelnost optimalizovaného kódu. V případě psaní v asembleru je kód v původní podobě prakticky nepřenositelný.
4. Horší možnosti ladění kódu, což je částečně způsobeno i bodem 2.
5. Zhoršení některých z parametrů kódu (zvětšení velikosti, snížení přesnosti, zvýšení paměťových nároků nebo zpomalení) při větší míře optimalizace.

Je tedy vždy namístě zvážit, zda není celková cena za optimalizaci vyšší než cena výkonějšího mikroprocesoru (je-li ovšem dostupný).

Na druhou stranu je třeba uvést, že i na optimalizaci software lze uplatnit tzv. *Paretův princip*, který říká že 80% času provádění programu zabírá pouhých 20% algoritmů. Právě těchto 20% (nezřídka i výrazně méně) vyžaduje pečlivou optimalizaci, kdežto zbývajících 80% ji často téměř nepotřebuje. To jsou například algoritmy pro komunikaci s uživatelem, inicializace, zaznamenávání stavu stroje nebo měniče, hlídání veličin s velkou časovou konstantou (teplota) a podobně.

3 PRAKTICKÁ IMPLEMENTACE

Tato část práce se zabývá ukázkou postupů optimalizace, popsaných v předchozí kapitole. Budou ověřeny na implementaci základního algoritmu skalárního třífázového sinusového modulátoru. Ten je základem pro jednoduché aplikace frekvenčních měničů. Pohon napájený takto řízeným střídačem nemá dynamické vlastnosti srovnatelné s vektorovým řízením, ale cílem práce je především popsat optimalizační postupy. Ty jsou platné i pro kterékoliv jiné řídicí programy.

3.1 Architektura DSC rodiny c2000

Aby bylo možné efektivně využít možnosti, které mají DSC řady TMS320x280x, je třeba mít povědomí o architektuře jak jejich CPU, tak o DSC jako celku. Řada c280x je na trhu cca. od roku 2005 a jejím předchůdcem je řada TMS320x240. I ta již byla vyvíjena speciálně pro použití v systémech řízení a regulace elektrických pohonů a silové konverze — DC/DC měniče, korekce účiníku (PFC), frekvenční měniče, regulace bezkartáčových stejnosměrných motorů (označované BLDC) atd. Na rozdíl od popisované řady c2000, která je 32-bitová, ale byla jejich architektura 16-ti bitová a maximální hodinový kmitočet CPU byl 60MHz. Do rodiny DSC, označované jako c2000, dnes patří několik řad, o nichž blíže pojednávají následující odstavce. Všechny mají několik společných znaků. Jsou to zejména CPU (a tedy i instrukční soubor) a část periferií. Liší se jak rychlostí, množstvím periferií, velikostí pamětí, pouzdrem, maximálním hodinovým kmitočtem, tak i ALU (aritmeticko-logickou jednotkou).

3.1.1 Popis TMS320F2808

Hlavními rysy použitého DSC jsou:

- napájecí napětí 1,8 V (CPU) a 3,3V (vstupy/výstupy, flash paměť, periferie)
- hodinový kmitočet až 100MHz (instrukční cyklus 10ns)
- 32-bitové CPU modifikované Harwardské architektury
- osmiúrovňová „pipeline“ s HW ochranou proti čtení z adresy na kterou teprve má být předchozí instrukcí zapisováno
- 32*32 bit jednotka MAC (násobení a přičtení výsledku)
- programová sběrnice — šířka 32 bitů, adresa 22 bitů
- datová sběrnice — dvojnásobná (zvlášť pro čtení a zápis), šířka 32 bitů pro data i adresu
- 128kB flash paměti na čipu (64k 16bitových slov)
- 36kB RAM pro program a data (18k 16bitových slov)

- 8kB boot ROM pro programování v aplikaci (ISP) a tabulky matematických funkcí (viz kapitolu 2.1)
- 128bitový klíč k ochraně paměti proti neoprávněnému čtení
- 16 PWM výstupů, 4 z nich s „vysokým rozlišením“ (150ps!)
- až 6 32bitových nebo 16bitových časovačů
- 2 asynchronní sériové porty (UART) s 16 byte FIFO
- 4 porty SPI, 1 I^2C, 2 CAN
- 12-bitový A/D převodník, multiplexovaný na 2*8 vstupů přes dva vzorkovače, doba trvání A/D převodu 160ns
- až 35 číslicových vstupů/výstupů s filtrováním zákmitů hran vstupních signálů (nastavitelný počet cyklů, kdy se nesmí změnit logická úroveň vstupu aby byla změna hodnoty procesorem vyhodnocena)
- JTAG emulační rozhraní
- 3 nízkopříkonové režimy
- dodáván pouzdrech TQFP100 nebo PBGA100
- rozsah pracovních teplot -40 – 85°C nebo -40 – 125°C

3.1.2 Přehled řad rodiny c2000

Celkem patří do rodiny c2000 k 20. květnu 2010 58 typů z následujících řad.

TMS320*x*280*n* je řadou, do níž patří i použitý DSC TMS320F2808. Písmeno „F“ na místě x znamená, že DSC je vybaven Flash pamětí programu, zatímco řada s písmenem „C“ obsahuje paměť typu ROM programovanou výrobcem. Poslední číslice na místě n označuje velikost paměti dat i programu DSC a množství periferií integrovaných na čipu. Tato řada DSC může pracovat s hodinovým kmitočtem 60 – 100MHz. Má pouze ALU provádějící výpočty v pevné řádové čárce. Všechny typy jsou pinově kompatibilní, v pozdrech TQFP100 nebo PBGA100.

TMS320*x*281*n* je, na rozdíl od předchozí řady, schopná běhu do 150MHz. Její periferie jsou obecnější, méně specializované na řízení pohonů a výkonové elektroniky. Počet pinů je 128-176.

TMS320*x*282*nn* běží, stejně jako u předchozí řada, do 150MHz. Navíc disponuje šestikanálovou jednotkou DMA pro přímý přístup periferií k datové paměti. Má větší počet pinů — 176 až 179.

TMS320*x*283*nn* — „Delfino“ je pinově shodná s předchozí řadou. CPU této řady ale pracuje s čísly s plovoucí desetinnou čárkou — *floating-point*. Běží na kmitočtech od 100 do 300MHz. Dostupné v pouzdrech se 176 až 256 vývody.

TMS320*x*2802*n* — „Piccolo“ má CPU s pevnou řádovou čárkou. Všechny typy této řady běží od 40 do 60MHz. Jsou určeny pro méně náročné aplikace, jak

napovídají i jejich pouzdra se 38 a 48 vývody.

TMS320*x*2803*n* — „Piccolo“ je výkonnější variantou předchozí řady „Piccolo“. Běží na 60MHz, v pouzdrech o 64 a 80 vývodech. Disponuje *floating-point* koprocesorem pro paralelní výpočet rychlých regulačních smyček. Obě řady „Piccolo“ mají výhodu pouze jednoho napájecího napětí 3,3V oproti ostatním řadám, které mají ještě 1,8 nebo 1,9V pro napájení CPU.

3.2 Vývojové prostředí

Algoritmy jsou ověřeny na vývojové desce eZdsp od firmy Spectrum Digital Inc., osazené digitálním signálovým kontrolérem (DSC) typu TMS320F2808 od firmy Texas Instruments Inc. (dále TI). Prostřednictvím této desky je možné naprogramovat DSC a následně program ladit přes emulační rozhraní JTAG. Pro tyto účely je deska USB kabelem propojena s osobním počítačem vybaveným příslušným SW. Tím je integrované vývojové prostředí (IDE) Code Composer Studio (též od TI).

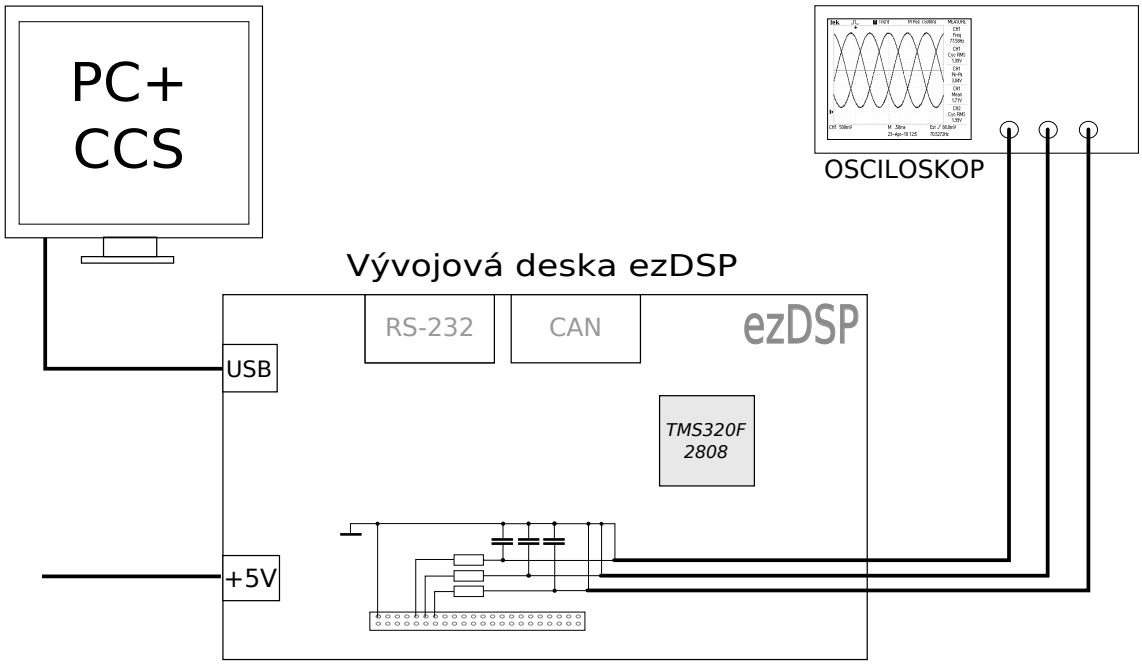
3.2.1 Code Composer Studio

Toto IDE je dostupné v podobě časově omezené plné verze z webových stránek TI[1] nebo placené neomezené verze. Umožňuje kompletní vývoj firmware pro DSC TI od editace zdrojových kódů, přes překlad až po ladění a profilování kódu pomocí simulátoru nebo emulátoru. Editor CCS (viz obrázek A.4) usnadňuje práci programátora zvýrazňováním syntaxe i automatickým doplňováním kódu. Ve stejném okně je možné po přeložení zdrojových kódů přejít k ladění a optimalizaci programu. Velice užitečný pro optimalizaci i ladění je náhled kódu současně v podobě zdrojového kódu v C a jeho přeložené podobě v asembleru. CCS spolupracuje přímo s destičkou ezDSP, takže je možné sledovat a měnit stav proměnných, nastavovat breakpointy a podobně opět přímo v hlavní okně editoru.

3.2.2 Zkušební pracoviště

Zkušební pracoviště bylo zapojeno podle obrázku 3.1. Deska ezDSP je napájena stejnosměrným napětím 5V ze síťového adaptéru. RC články jsou vytvořeny zasunutím součástek do dutinkové lišty přímo na desce.

[1] http://www.ti.com



Obr. 3.1: Zapojení pracoviště s deskou ezDSP. Pro zobrazení signálu na osciloskopu byly generované obdélníkové signály filtrovány RC články R=10kΩ a C=100nF.

3.3 Implementace, optimalizace

Zvolený ukázkový algoritmus trojfázového sinusového modulátoru byl implementován nejdříve čistě v jazyce C. Program byl od počátku koncipován s ohledem na dosažení co možná nejvyšší rychlosti vykonávání. Po odladění funkčnosti základního programu v C byl tento program postupně překládán s různými optimalizačními parametry překladače. Každá přeložená verze programu byla nahrána do DSC na destičce ezDSP, na které byla v profilovacím okně CCS změřena minimální a maximální rychlost vykonávání algoritmu. Z informací uvedených linkerem v souboru s příponou .map, byla zjištěna velikost vygenerovaného kódu. Výsledky byly zaznamenány pro pozdější grafické zpracování.

Následně byl kód dále optimalizován použitím předdefinované funkce `__IQsat`. Opět byly vyzkoušeny a zaznamenány různé stupně optimalizace a zanamenány jak výsledné rychlosti, tak velikosti generovaných strojových kódů.

V poslední fázi optimalizace byl po překladu maximálně optimalizovaného kódu (přepínače `-o3 -pm -op0 -oi32`) pomocí přepínače `-k` uložen soubor obsahující zdrojový kód funkce `sin_mod()` v asembleru. Tento soubor (sin_mod.asm) byl poté ručně upraven tak, aby využíval méně paměti ze zásobníku a naopak více registrů procesoru. Některé bloky instrukcí byly přeorganizovány, aby docházelo méně často ke konfliktům zápisu a čtení stejných dat instrukcemi v pipeline CPU. Byly také změněny některé instrukce.

Jednotlivé fáze optimalizace jsou i s příslušnými komentáři zdrojových kódů podrobněji rozebrány v dalším textu.

3.3.1 Koncepce programu

Po inicializaci hodin a Flash paměti DSC, která je v úvodu funkce `main()` (na přiloženém CD v adresáři src), je provedena inicializace modulu pulzně–šířkové modulace PWM. Ten je nakonfigurován na symetricou PWM (čítá nahoru i dolů) s periodou 33,33 μs. Modulační kmitočet tedy bude roven 30kHz. Protože možnosti emulátoru s použitou vývojovou deskou ezDSP nezahrnují pohodlnou práci s obsluhami přerušení, je funkce `sin_mod()` napsána jako standardní funkce. Po vykonání potřebných inicializací se zavolá funkce `main_loop()`, která zajistí volání funkce modulátoru `sin_mod()` právě každých 33,33 μs. Při normálním běhu (tzn. ne při ladění aplikace) se o volání modulační funkce a zároveň spuštění příslušných A/D převodů postará modul ePWM DSC [6]. Modulační funkce se totiž stane funkcí obsluhy přerušení modulu ePWM. Z tohoto důvodu nesmí mít žádné parametry ani vracet žádnou hodnotu. Veškeré potřebné proměnné jsou tedy řešeny jako globální.

3.3.2 Zdrojové kódy

Dále jsou komentovány zdrojové kódy funkce `sin_mod()` a to ve třech podobách, lišících se v podobě a vůbec přítomnosti kódu, zajišťujícího ořezání výsledku. Ve čtvrté části je vypsána asemblerová podoba funkce. Kompletní zdrojové kódy aplikace jsou uloženy na přiloženém CD v adresáři src.

Sinusová modulace v C

Níže je uveden výpis zdrojového kódu funkce `sin_mod`, která počítá sinusovou PWM modulaci třífázového dvouhladinového frekvenčního měniče. Funkce používá lokální proměnné typu `T_I32`, který je definován následovně:

```
typedef union           // Datový typ T_I32
{
   long all;            // Pro přístup k celé 32-bitové proměnné
   struct
   {
      unsigned int lsw; // Pro přístup k nižšímu
      int msw;          // a vyššímu 16-ti bitovému slovu proměnné
   }half;
}T_I32;
```

Jeho účel je, jak naznačují komentáře, umožnit programátorovi jednoduchý přístup k oběma částem proměnné. Podobně je deklarována i globální proměnná `phase`, která slouží jako sumátor fáze pro modulátor.

```
union T_UI32            // Datový typ union:T_UI32
{
   unsigned long all;   // Pro přístup k celé 32-bitové proměnné
   struct
   {
      unsigned int lsw; // Pro přístup k nižšímu
      unsigned int msw; // a vyššímu 16-ti bitovému slovu proměnné
   }half;
}phase;
```

Dále funkce využívá několik předdefinovaných konstant, definovaných následovně:

```
#define FI120     21845  // fáze 120°
#define FI240     43691L // fáze 240°
#define HALF_AMPL 833*65536
```

Vlastní funkce `sin_mod` sestává z následujících příkazů:

```
01: void sin_mod(void)
02: {
03:   T_I32 s0;
04:   T_I32 s120;
05:   T_I32 s240;
06:   const long* const ptr = (long*) 0x003FF000;

07:   s0.all = *(ptr+(phase.half.msw>>7));
08:   s120.all = *(ptr+((phase.half.msw+FI120)>>7));
09:   s240.all = *(ptr+((unsigned int)(phase.half.msw+FI240)>>7));

10:   s0.all = ((long) s0.half.msw * ampl) + HALF_AMPL;
11:   s120.all = ((long) s120.half.msw * ampl) + HALF_AMPL;
12:   s240.all = ((long) s240.half.msw * ampl) + HALF_AMPL;

13:   phase.all += f_add;

14:   EPwm1Regs.CMPA.half.CMPA = s0.half.msw;
15:   EPwm2Regs.CMPA.half.CMPA = s120.half.msw;
```

```
16:   EPwm3Regs.CMPA.half.CMPA = s240.half.msw;
17: }
```

Na řádcích 3–5 jsou deklarovány pomocné lokální proměnné `s0`,`s120` a `s240`. Jsou typu `union` (viz dříve uvedenou definici datového typu `T_I32`) aby bylo možné snadno přistupovat k jejich jednotlivým částem namísto případných bitových posunů a pod. Jsou dále použity pro ukládání mezivýsledků pro jednotlivé fáze.

Na řádku 6 je definován konstantní ulazatel na konstantu, určující počátek tabulky funkce sinus v ROM paměti DSC.

Na řádcích 7–9 jsou do pomocných proměných uloženy hodnoty funkce sinus odpovídající úhlu danému sumátorem fáze `phase` (v případě proměnné `s120` zvětšenému o hodnotu odpovídající 120°, resp. 240° u `s240`). Jeho horní slovo je posunuto doprava o 7 bitů, čímž získáme proměnou o rozsahu 9 bitů. To odpovídá délce tabulky funkce sinus, která je právě $2^9 = 512$ hodnot. Je zřejmé, že hodnota bude přesná pouze pokud jsou zahozené bity fáze rovny 0, tedy pokud je fáze násobkem $(360/512)°$. Pro reálné použití by toto zjednodušení nemuselo přinést problémy, jak je rozebráno níže.

Na řádcích 10–12 je hodnota funkce sinus vynásobena požadovanou amplitudou `ampl` a je přičtena hodnota `HALF_AMPL`, odpovídající polovině amplitudy (při požadavku na výstupní amplitudu 0 tedy bude střída spínání všech tranzistorů 50%).

Na řádku 13 je k sumátoru fáze přičtena hodnota úměrná požadovanému výstupnímu kmitočtu. Jedná se o číslicovou obdobu integrace požadované hodnoty kmitočtu, kterou také získáme fázi výstupního napětí vůči počátku. Vhodnou volbou rozsahu proměnné a konstant bylo záměrně dosaženo přetečení proměnné při dosažení hodnoty odpovídající 360°. Proměnná tedy podává informaci o fázi aktuálně generované periody vůči jejímu počátku.

Na řádcích 14–16 jsou vypočené hodnoty uloženy do odpovídajících registrů modulu ePWM digitálního signálového kontroléru.

Jelikož je funkce koncipována jako obsluha přerušení (viz část 3.3.1), nelze ji deklarovat jako `inline`.

Pro výpočet hodnoty funkce sinus byla použita přímo hodnota z tabulky (typu T_1), uložené v paměti ROM. Jak již bylo uvedeno, obsahuje 512 hodnot na celé periodě funkce sinus. Ze vztahu 2.24 můžeme vypočítat maximální chybu zjištěné hodnoty jako

$$\Delta_{max} = \sin\frac{2\pi}{n} = \sin(2\pi/512) \doteq 0,0123 = 1,23\%. \tag{3.1}$$

Tato hodnota byla uvážena jako vyhovující z několika důvodů. První důvod je, že tato hodnota je po dalších úpravách použita jako modulační konstanta PWM modulátoru, který především svou časovou diskretizací vnáší do skutečné výstupní

hodnoty mnohdy větší chybu. Dále je zřejmé, že tato chyba symetrická a nemůže tedy docházet např. ke vznikům nežádoucí stejnosměrné složky. Je ovšem pravda, že žádaný kmitočet může spolu s chybou výpočtu (která je závislá na fázi), vytvářet nežádoucí interference, které se superponují na základní požadovaný průběh výstupního napětí. Pokud by se jejich amplituda, dosahující přibližně velikosti chyby tabulky, ukázala být příliš vysoká, bude nutné přistoupit k lineární aproximaci tabulky, popsané v kapitole . Potm bude zřejmě rychlejší počítat hodnoty funkce sinus pouze pro první dvě fáze. Třetí fázi pak vypočítáme, vyjdeme-li z pravidla, že součet fázových napětí je v každém okamžiku 0.

$$u_U + u_V + u_W = 0 \tag{3.2}$$

Dosazením vztahu 1.2 dostaneme

$$\begin{aligned} \frac{\mathrm{U_{DC}}}{2} K \sin(\omega t) + \frac{\mathrm{U_{DC}}}{2} K \sin\left(\left(\omega + \frac{2\pi}{3}\right) t\right) + \frac{\mathrm{U_{DC}}}{2} K \sin\left(\left(\omega + \frac{4\pi}{3}\right) t\right) &= 0 \\ \sin(\omega t) + \sin\left(\left(\omega + \frac{2\pi}{3}\right) t\right) + \sin\left(\left(\omega + \frac{4\pi}{3}\right) t\right) &= 0, \end{aligned} \tag{3.3}$$

z čehož můžeme vyjádřit sinus fáze W

$$\sin\left(\left(\omega + \frac{4\pi}{3}\right) t\right) = -\sin(\omega t) - \sin\left(\left(\omega + \frac{2\pi}{3}\right) t\right). \tag{3.4}$$

Sinusová modulace s ořezem v C

Kód funkce je shodný s předchozím (funkce `sin_mod()`), pouze mezi řádky č. 13 a 14 je vloženo následujících 12 řádků:

```
01:   if(s0.half.msw > SMAX)
02:      s0.half.msw = SMAX;
03:   else if(s0.half.msw < SMIN)
04:      s0.half.msw = SMIN;
05:   if(s120.half.msw > SMAX)
06:      s120.half.msw = SMAX;
07:   else if(s120.half.msw < SMIN)
08:      s120.half.msw = SMIN;
09:   if(s240.half.msw > SMAX)
10:      s240.half.msw = SMAX;
11:   else if(s240.half.msw < SMIN)
12:      s240.half.msw = SMIN;
```

A před vlastní funkcí jsou definovány použité konstanty jako:

```
#define SMIN      0     // Dolní mez ořezání
#define SMAX      1666  // Horní mez ořezání
```

Z výpisu kódu je evidentní, že pouze zajistí ořezání rozsahu proměnných `s0.half.msw`, `s120.half.msw` a `s240.half.msw` do rozsahu <SMIN;SMAX>. U tohoto kódu získáme různou hodnotu minimálního a maximálního počtu instrukčních cyklů, potřebných k provedení algoritmu, a to díky pomíněným skokům. Tento stav nastane, překročí-li zadaná amplituda `ampl` hodnotu 3332. V opačném případě je čas algoritmu dán optimalizátorem překladače, tedy tím, jestli se bude skákat nebo ne.

Sinusová modulace s ořezem pomocí intrinsics

Základ funkce je opět shodný s funkcí `sin_mod()`, jen místo vložení podmínek podle předchozí části, vložíme mezi řádky č. 13 a 14 následující tři řádky:

```
01:   s0.all = __IQsat(s0.all, SMAX, SMIN);
02:   s120.all = __IQsat(s120.all, SMAX, SMIN);
03:   s240.all = __IQsat(s240.all, SMAX, SMIN);
```

Každý řádek provádí saturaci jedné z fází, a to opět do rozsahu <SMIN;SMAX>, který je definován stejně, jako v případě ořezu podmínkami v předchozí části této kapitoly.

Sinusová modulace s ořezem v asembleru

Jak již bylo uvedeno, optimalizace spočívá nejdříve v úpravách asemblerového kódu generovaného překladačem. Vhodným přemístěním proměnných mezi registry tím můžeme snížit nároky funkce na zásobník a také ušetřit několik instrukčních cyklů.

Někdy můžeme pro dosažení stejného výsledku použít jiné instrukce, než jaké zvolil překladač. Další možností je minimalizace krátkých smyček pomocí instrukce `RPT`, následované tělem smyčky, vyjádreném jednou opakovatelnou instrukcí. Její použití popisuje [1, str. 469].

Výsledný optimalizovaný kód vypadá takto:

```
01:   .global     _phase
02:   .global     _ampl
03:   .global     _f_add
04:   .global     _EPwm1Regs
05:   .global     _EPwm2Regs
06:   .global     _EPwm3Regs
07:   .global     _sin_pwm

08:_sin_pwm:
09:   MOVW     DP,#_phase+1
10:   MOVL     XAR4,#4190208
11:   ADDB     SP,#4
12:   MOV      AL,@_phase+1
13:   MOVL     XAR6,XAR4
34:   MOVL     ACC,*+XAR4[0]
35:   MOVL     *-SP[4],ACC
36:   MOVL     ACC,@_f_add
37:   MOV      AR6,@_ampl
38:   ADDL     @_phase,ACC
39:   MPYXU     ACC,T,@AR6
40:   MOVW     DP,#_EPwm1Regs+9
41:   ADD      ACC,#1666 << 15
42:   MOV      T,*-SP[1]
43:   MOV      AL,#0
44:   MAX      AH,@AL
45:   MOV      AL,#1665
46:   MIN      AH,@AL
47:   MOV      @_EPwm1Regs+9,AH
```

```
14:   LSR     AL,7
15:   MOV     ACC,AL<<1
16:   ADDL    ACC,XAR4
17:   MOVL    XAR4,ACC
18:   MOVL    ACC,*+XAR4[0]
19:   MOV     T,AH
20:   MOV     AL,@_phase+1
21:   ADD     AL,#21845
22:   LSR     AL,7
23:   MOV     ACC,AL<<1
24:   ADDL    ACC,XAR6
25:   MOVL    XAR4,ACC
26:   MOVL    ACC,*+XAR4[0]
27:   MOVL    *-SP[2],ACC
28:   MOV     ACC,@_phase+1
29:   SUB     AL,#21845
30:   LSR     AL,7
31:   MOV     ACC,AL<<1
32:   ADDL    ACC,XAR6
33:   MOVL    XAR4,ACC
48:   MOVW    DP,#_EPwm2Regs+9
49:   MPYXU   ACC,T,@AR6
50:   ADD     ACC,#1666 << 15
51:   MOV     T,*-SP[3]
52:   MOV     AL,#0
53:   MAX     AH,@AL
54:   MOV     AL,#1665
55:   MIN     AH,@AL
56:   MOV     @_EPwm2Regs+9,AH
57:   MOVW    DP,#_EPwm3Regs+9
58:   MPYXU   ACC,T,@AR6
59:   ADD     ACC,#1666 << 15
60:   MOV     AL,#0
61:   MAX     AH,@AL
62:   MOV     AL,#1665
63:   MIN     AH,@AL
64:   MOV     @_EPwm3Regs+9,AH
65:   SUBB    SP,#4
66:   LRETR
```

4 DOSAŽENÉ VÝSLEDKY

Na zkušební desce ezDSP bylo odzkoušeno několik uvedených algoritmů pro sinusovou modulaci. Všechny popisované funkce byly prakticky ověřeny měřením výstupních průběhů osciloskopem a optimalizovány. Pomocí optimalizačních nástrojů Code Composer Studia byla změřena rychlost vykonávání funkcí a jejich velikost v závislosti na stupni optimalizace. Tyto hodnoty jsou uvedeny v příslušných tabulkách a grafech.

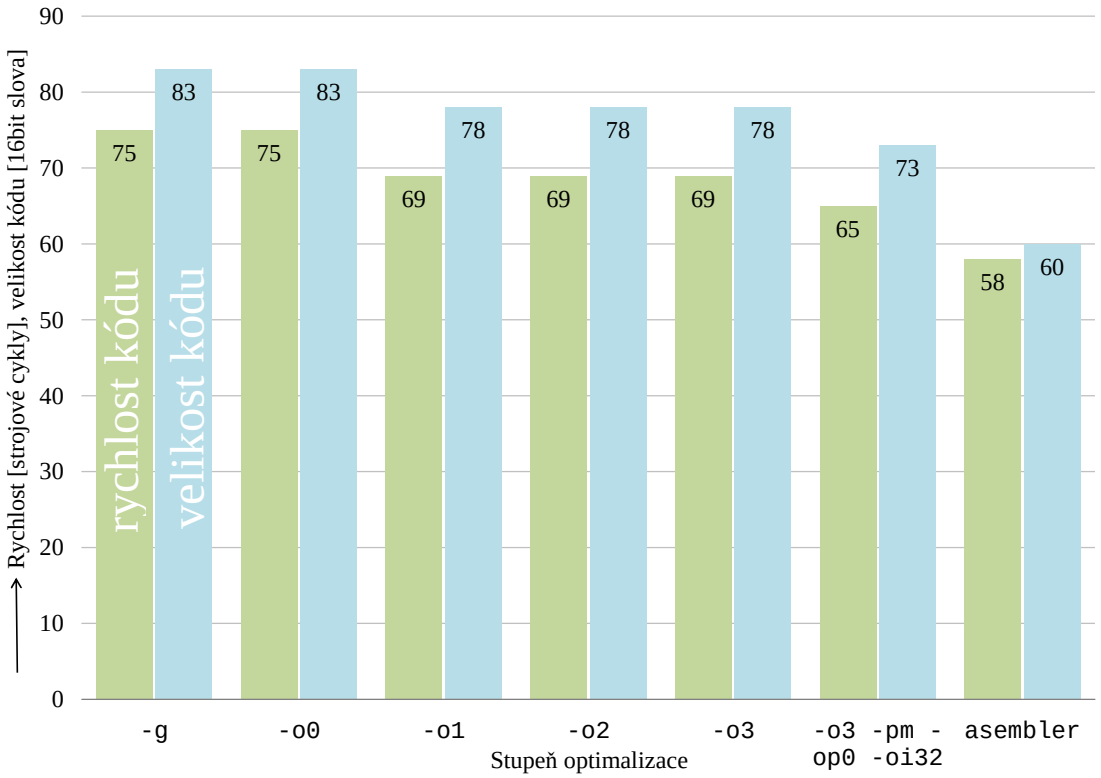
4.1 Optimalizace sinusové modulace

Jako první byl optimalizován algoritmus prosté sinusové modulace bez omezení rozsahu vypočtené modulační konstanty. Algoritmus byl napsán pouze v jazyce C tak, jak je uveden v kapitole 3.3.2. Časy změřené pomocí nástroje Profiler, který je součástí CCS, jsou pro jednotlivé stupně optimaliace shrnuty v následující tabulce. Pro srovnání je na posledním řádku uveden výsledek optimalizace funkce jejím přepsáním v asembleru. Z grafu je patrné, že optimalizace sinusové modu-

	rychlost	**zlepšení rychl.**	**velikost**	**zlepšení vel.**
optimalizace	strojové cykly	%	slova	%
`-g`	75	0,0	83	0,0
`-o0`	75	0,0	83	0,0
`-o1`	69	8,0	78	6,0
`-o2`	69	8,0	78	6,0
`-o3`	69	8,0	78	6,0
`-o3 -pm -op0 -oi32`	65	13,3	73	12,0
asembler	58	22,6	60	29,4

Tab. 4.1: Průběh optimalizace funkce `sin_mod()` a dosažené zlepšení oproti překladu bez optimalizace.

lace nepřinesla až tak razantní efekt, jak je uváděno např. v [2, kapitoly 3.3 až 3.5]. Až převedení do asembleru přineslo zrychlení kódu o cca. 22%. Důvodem je patrně to, že kód sinusové modulace je oproti příkladům v [2] blíže strojovému kódu a tak se příliš neuplatní schopnosti optimalizátoru. V případě, že by byl kód napsán v C neefektivně, došlo by patrně k větším rozdílům u jednotlivých stupňů optimalizace. Funkčnost celého algoritmu byla ověřena měřením výstupních průběhů podle zapojení na obr. 3.1. Měření bylo ve skutečnosti provedeno dvoukanálovým osciloskopem,



Obr. 4.1: Průběh optimalizace funkce `sin_mod()` a porovnání s asemblerovou verzí.

přičemž při skládání průběhů všech tří fází do jednoho obrázku bylo využito vstupu externí synchronizace. Ten byl stále připojen na jednu z fází, čímž bylo dosaženo správného vzájemného posunu fází i na uložených obrázcích. Ty byly potom sloučeny do jednoho obrázku č. 4.2.

Při kmitočtu PWM modulátoru 30 kHz, což odpovídá ve zkoušeném programu zvolené periodě 33,32 μs, je algoritmus volán v každé vteřině n-ktát:

$$n = f_{PWM} = 30 \cdot 10^3. \tag{4.1}$$

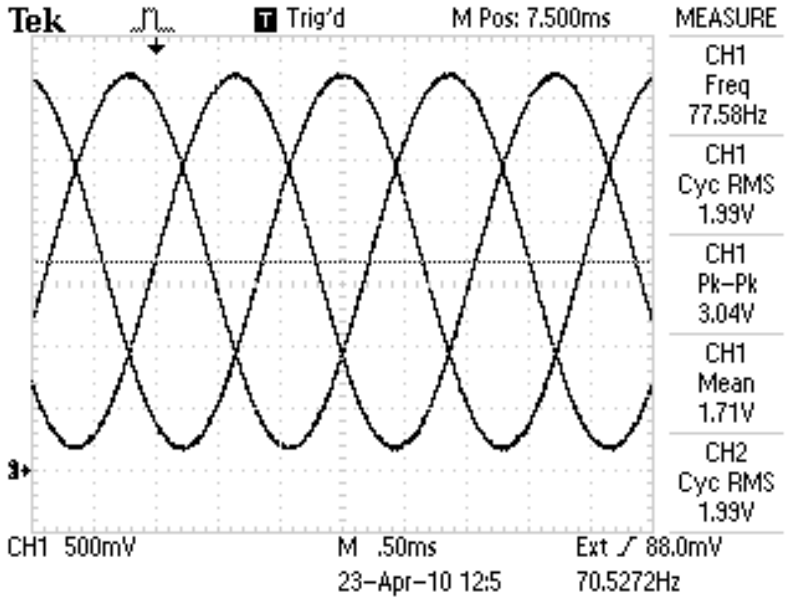
Pokud tedy použijeme nepomalejší variantu (`-g`), zabere modulátor m-procentní část výpočetní kapacity DSC (s dobou trvání instrukčního cyklu T_{instr}=10 ns).

$$m = 100 \cdot n \cdot T_{alg} T_{instr} = 100 \cdot 30 \cdot 10^3 \cdot 75 \cdot 10 \cdot 10^{-9} = 2,25\% \tag{4.2}$$

Naopak pro nejvíce optimalizovanou verzi (`-o3 -pm -op0 -oi32`) to bude

$$m = 100 \cdot 30 \cdot 10^3 \cdot 65 \cdot 10 \cdot 10^{-9} = 1,95\%. \tag{4.3}$$

Je zřejmé, že algoritmus zabírá nepatrnou část výpočetního výkonu DSC. V reálné aplikaci však bude zatížen dalšími priodicky vykonávanými operacemi, jako například vyhodnocením a hlídáním fázových proudů, regulací rychlosti, momentu a pod.



Obr. 4.2: Průběh vyfiltrovaných napětí jednotlivých fází přímo z PWM výstupů DSC.

4.2 Optimalizace sinusové modulace s ořezáním v jazyce C

Kód je od předchozího rozšířen o saturaci výsledných modulačních konstant. Ta je řešena pomocí podmínek, což způsobuje proměnnou délku vykonávání funkce v závislosti na jejich vyhodnocení.

Funkčnost algoritmu byla opět vyzkoušena pomocí připojeného osciloskopu. Tentokrát je zobrazen průběh pouze jedné fáze, ale s postupně se měnící hodnotou žádané amplitudy. Amplituda byla změněna přímo změnou proměnné `ampl`, a to v řadě 1000, 2000, 3000, 4000, 6000. Amplituda 1000 odpovídá na obrázku 4.4 neořezanému průběhu s nejmenším rozkmitem, amplituda 6000 naopak průběhu nejvíce ořezanému. Z průběhu signálu je zřejmé, že při oříznutí vrcholů průběhu není výstupní napětí harmonické. Zvětšuje se sice jeho efektivní hodnota, ale také obsah lichých vyšších harmonických, jak vidíme z frekvenčního spektra na obrázku 4.5. To může být nepřípustné z hlediska vytvářeného rušení i napájeného stroje.

Pro zjištění výpočetní náročnosti algoritmu musíme uvažovat nejhorší případ, tedy maximální možnou délku trvání algoritmu. V případě nejlépe optimalizované verze (`-o3 -pm -op0 -oi32`) tedy bude procentuální zatížení DSC (vztah 4.2)

$$m = 100 \cdot 30 \cdot 10^3 \cdot 92 \cdot 10 \cdot 10^{-9} = 2,76\%. \tag{4.4}$$

	rychlost strojové cykly		**zlepšení** %	
optimalizace	min.	max.	min.	max.
`-g`	108	112	0,0	0,0
`-o0`	90	96	16,7	14,3
`-o1`	96	102	11,1	8,9
`-o2`	93	99	13,9	11,6
`-o3`	90	96	16,7	14,3
`-o3 -pm -op0 -oi32`	83	92	23,1	17,9
asembler	70	70	35,2	37,5

Tab. 4.2: Průběh optimalizace funkce `sin_mod()` se saturací v jazyce C a dosažené zlepšení oproti překladu bez optimalizace.

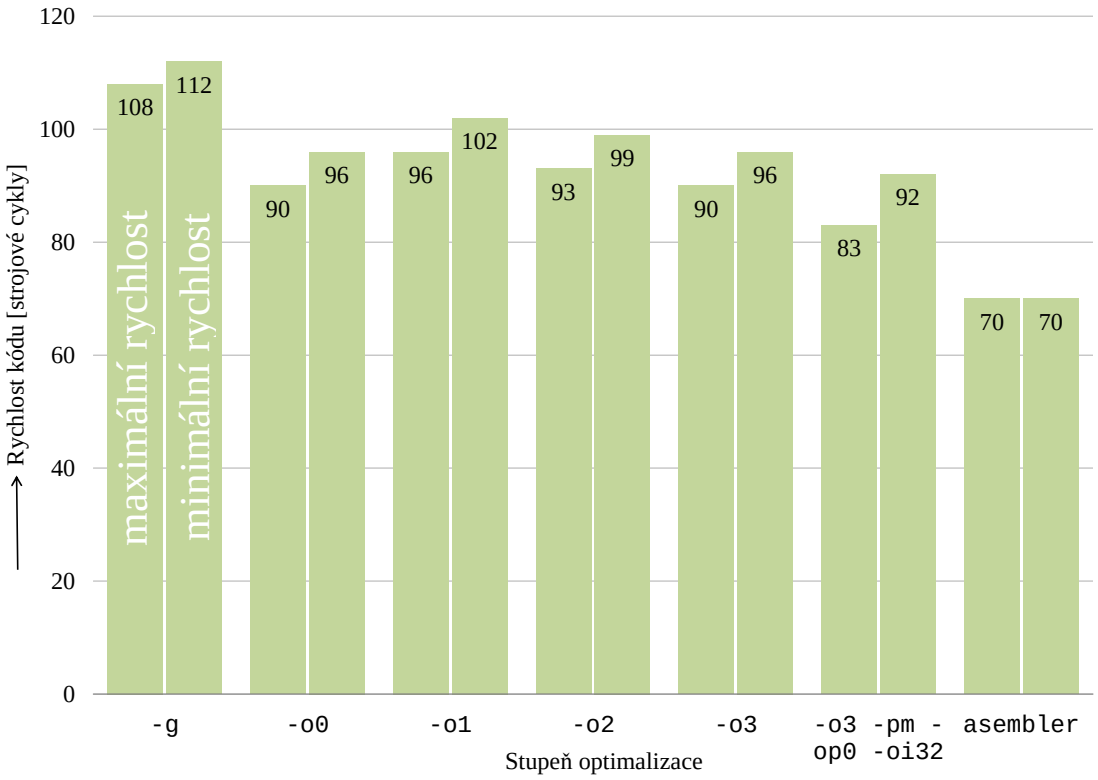
4.3 Optimalizace sinusové modulace s ořezáním pomocí intrinsics

Namísto předchozího ořezání pomocí podmínek je nyní použito intrinsics (předdefinovaných funkcí). Tím bylo dosaženo odstranění podmínek a tedy nezávislosti rychlosti funkce na vstupních datech i celkového zrychlení.

	rychlost	**zlepšení**	**velikost**	**zlepšení**
optimalizace	strojové cykly	%	slova	%
`-g`	89	0,0	92	0,0
`-o0`	79	11,2	89	3,3
`-o1`	82	7,9	90	2,2
`-o2`	75	15,7	84	8,9
`-o3`	75	15,7	84	8,9
`-o3 -pm -op0 -oi32`	75	15,7	84	8,9
asembler	70	21,3	81	12,0

Tab. 4.3: Průběh optimalizace funkce `sin_mod()` se saturací pomocí intrinsics a dosažené zlepšení oproti překladu bez optimalizace.

Průběhy výstupních napětí byly podle očekávání shodné s předchozími, algoritmus tedy je plně funkční. Nejlépe optimalizovaná verze tohoto algoritmu zabere



Obr. 4.3: Průběh optimalizace funkce `sin_mod()` se saturací v jazyce C a porovnání s asemblerovou verzí.

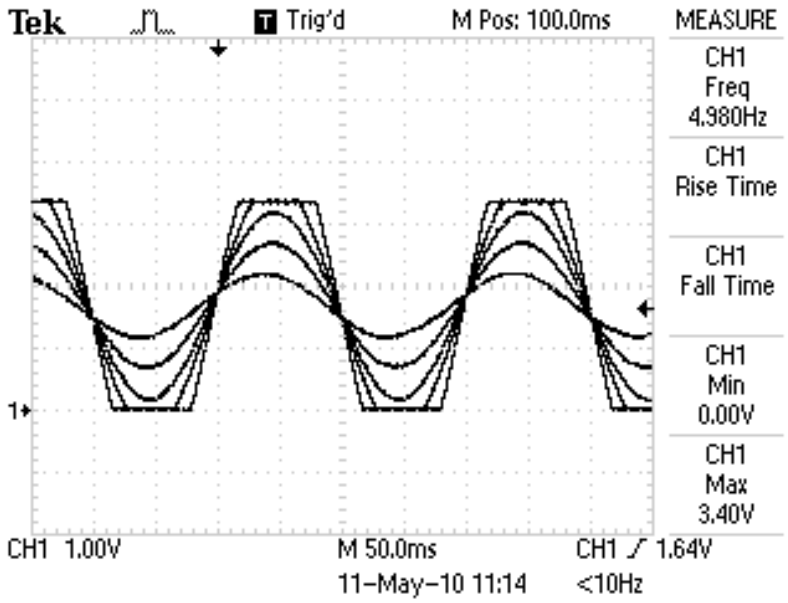
podle vztahu 4.2

$$m = 100 \cdot 30 \cdot 10^3 \cdot 75 \cdot 10 \cdot 10^{-9} = 2,25\% \tag{4.5}$$

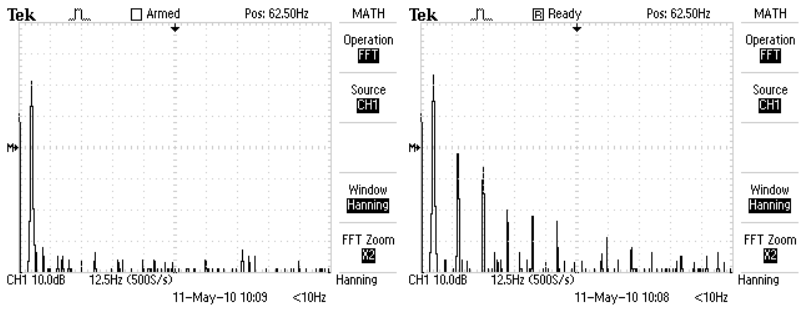
výpočetního výkonu DSC, což je stejně, jako nejméně optimalizovaná verze algoritmu bez saturace výsledku.

4.4 Výsledky implementace v asembleru

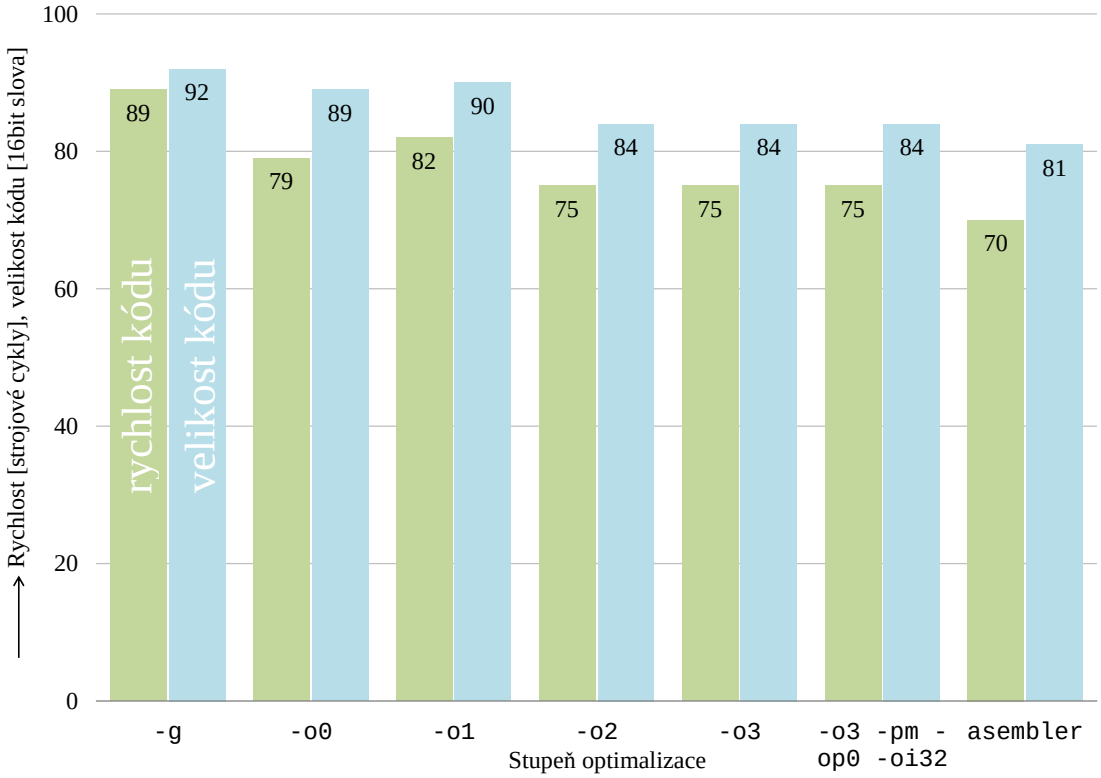
Výsledky asemblerové verze jak algoritmu s ořezáním výsledku, tak bez něj, jsou pro srovnání uvedeny v grafech a tabulkách pro tyto výše uvedené verze funkce `sin_mod()`, psané v jazyce C. Vidíme, že asemblerová verze byla vždy znatelně lepší. Běh verze bez saturace výsledků zabírá podle vztahu 4.2 1,74% výpočetní kapacity DSC. Se saturací je to 2,1%.



Obr. 4.4: Průběh vyfiltrovaného napětí jedné z fází v závislosti na zadané amplitudě `ampl`.



Obr. 4.5: V levé části je spektrum filtorvaného výstupního napětí z PWM těsně pod hranicí saturace (`ampl` = 3000), v pravé části saturovaného napětí (`ampl` = 4000). Vidíme, že nesaturovaný průběh je sinusovka s minimálním harmonickým zkreslením. Naproti tomu ořezaný průběh je již značně neharmonický s vysokým obsahel lichých harmonických.



Obr. 4.6: Průběh optimalizace funkce `sin_mod()` se saturací pomocí intrinsics, dosažené zlepšení oproti překladu bez optimalizace a srovnání s výsledky asemblerové verze.

5 ZÁVĚR

Z uvedených grafů a tabulek je zřejmé, že každý stupeň optimalizace, včetně posledního, kterým byla realizace funkce modulátoru v asembleru, přinesla jisté zlepšení a to jak co se týká rychlost kódu, tak jeho velikosti. Z výpočtených hodnot vytížení výpočetního výkonu DSC je zřejmé, že jeho rychlost nám bez problémů, ovšem při alespoň minimálních optimalizacích kódu, umožní realizaci poměrně složitých algoritmů, jako například vektorová či přímá regulace momentu asynchrnonního motoru. Z údajů o rychlosti jednotlivých funkcí matematické knihovny IQmath je zřejmé, že se hodí buďto pro jednodušší algoritmy nebo v případech, kdy vyžadujeme maximální přesnost výpočtů. Jejich rychlost však nemusí být pro reálná zařízení, která navíc musí komunikovat s uživatelem či nadřazeným systémem, dostačující — jsou optimalizovány pro maximální přesnost i za cenu nižší rychlosti. Proto je velká část textu věnována metodám výpočtu některých funkcí, které se mohou stát jejich náhradou. Výhoda knihovny IQmath tak spočívá především v možnosti rychlého ověření postupu výpočtů v počáteční fázi vývoje programu.

Srovnáme-li dosažené hodnoty rychlosti a velikosti funkce, vidíme, že ta, napsaná v jazyce symbolických adres — asembleru, je vždy znatekně rychlejší, než její ekvivalent psaný v C. V případě první varianty programu, bez ořezání výsledných modulačních konstant, je to o 10,7%, v případě ořezání podmínkami v jazyce C o 23,9% a konečně v případě ořezání pomocí intrinsics o 6,7%, tedy pouze pět instrukčních cyklů. Můžeme tedy říct, že ke psaní jednotlivých funkcí přímo v asembleru, se uchýlíme pouze v krajních případech, kdy vyžadujeme skutečně nejvyšší rychlost algoritmů bez kompromisů, a to i za cenu značného zvýšení časové náročnosti nejen psaní vlastního kódu, ale také jeho pozdějšího rozšiřování a údržby. Naopak použití předdefinovaných funkcí, intrinsics, nevyžaduje mnoho času programátora navíc, avšak jejich použití přinese často poměrně značný efekt.

Závěrem bych rád uvedl, že na základě podrobného studia DSC Texas Instruments typu TMS32F2808, jeho periferií i instrukčního souboru, jsem měl možnost se přesvědčit, že jeho nasazení v oblasti výkonové elektroniky je nadmíru opodstatněné. Jeho periferie, architektura, instrukční soubor, ale i poměr cena[1]/výkon a kvalita vývojového software, jsou této oblasti takříkajíc „šity na míru“.

[1]V době psaní práce okolo $ 12 v množství 1000 kusů, v menším množství pochopitelně více.

LITERATURA

[1] *TMS320C28x CPU and Instruction Set Reference Guide* [online]. Texas Instruments Inc., 2001. Poslední aktualizace leden 2009 [cit. 21. 3. 2010]. Dostupné z URL: <http://focus.ti.com/lit/ug/spru430e/spru430e.pdf>. ID: SPRU430E

[2] *Optimizing Digital Motor Control (DMC) Libraries* [online]. Texas Instruments Inc., 2007. Poslední aktualizace 15. 3. 2007 [cit. 17. 3. 2010]. Dostupné z URL: <http://focus.ti.com/lit/an/spraak2/spraak2.pdf>. ID: SPRAAK2

[3] *C28x IQmath Library* [online]. Texas Instruments Inc., 2009. Poslední aktualizace 1. 7. 2009 [cit. 10. 4. 2010]. Dostupné z URL: <http://focus.ti.com/docs/toolsw/folders/print/sprc087.html>. ID: SPRC087

[4] *TMS320F280x Data Manual* [online]. Texas Instruments Inc., 2007. Poslední aktualizace červen 2009 [cit. 28. 4. 2010]. Dostupné z URL: <http://focus.ti.com/lit/ds/symlink/tms320f2808.pdf>. ID: SPRS230K

[5] *AC Induction Motor Control Using Constant V/Hz Principle and Space Vector PWM Technique with TMS320C240* [online]. Texas Instruments Inc., 1998. Poslední aktualizace 1998 [cit. 15. 3. 2010]. Dostupné z URL: <http://focus.ti.com/lit/an/spra284a/spra284a.pdf>. ID: SPRA284A

[6] *TMS320x280x, 2801x, 2804x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide* [online]. Texas Instruments Inc., 2008. Poslední aktualizace červenec 2009 [cit. 2. 4. 2010]. Dostupné z URL: <http://focus.ti.com/lit/ug/spru791f/spru791f.pdf>. ID: SPRU791F

[7] WARREN, Henry S. *Hacker's delight*. 1. vyd. Boston: Pearson education, Inc., 2003. 301 s., ISBN 0-201-91465-4

[8] JAVŮREK, J. *Regulace moderních elektrických pohonů*. 1. vyd. Praha: GRADA, 2003. 264 s., ISBN 80-247-0507-9

[9] *Výkonové spínací prvky: Učební texty* [online]. Ostrava: Vysoká škola báňská — Technická univerzita Ostrava. Fakulta elektrotechniky a informatiky. Katedra elektroniky, 2003. Elektronická skripta. Poslední aktualizace 2003 [cit. 10. 5. 2009]. Dostupné z URL: <http://fei1.vsb.cz/kat448/data/vsp/vsp-predn.pdf>

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

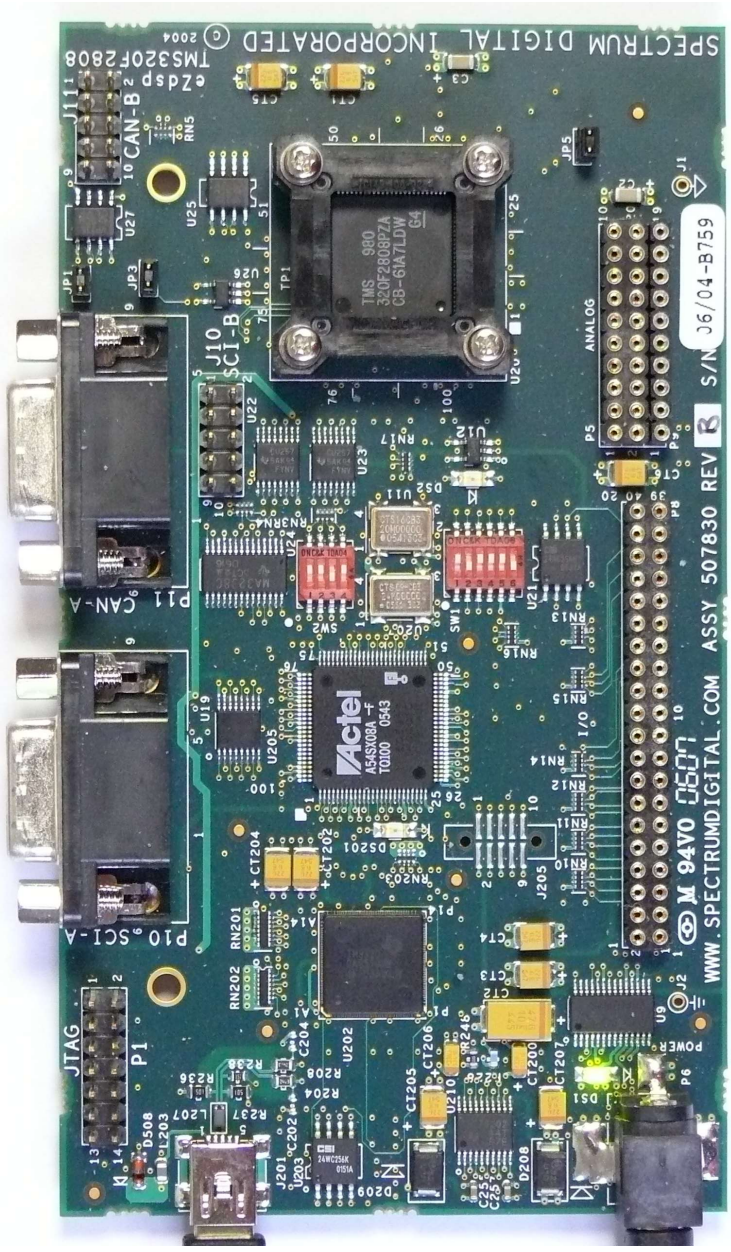
$\wedge$	logický součin, „a“ – AND
$\vee$	logický součet, „nebo“ – OR
$\oplus$	výlučný logický součet, „nerovnost“ – XOR
A/D	*Analog/Digital* – analogově/číslicový
CAN	*Controller Area Network* – sběrnice CAN je sériová komuninikační sběrnice používaná především v automobilovém průmyslu, ale i v jiných odvětvích
CCS	*Code Composer Studio* – komerční vývojové prostředí od firmy Texas Instruments pro μP, DSC i DSP z jejich produkce
CPU	*Central Processing Unit* – centrální procesorová jednotka, výpočetní jádro mikroprocesoru
DSC	*Digital Signal Controller* – digitální signálový kontrolér je moderní řídicí obvod vycházející aritmeticko-logickou jednotkou z DSP, ale integrující v na jednom čipu i flash paměť programu a řadu periferií určených speciálně pro řídicí aplikace (např. A/D převodníky, PWM výstupy, komunikační rozhraní a pod.). Viz str. 41 a [4]
DSP	*Digital Signal Processor* – digitální signálový procesor
ePWM	*Enhanced Pulse–Width Modulator* – je pulzně–šířkový modulátor, periferie DSC řady c2000, více v [6]
GTO	*Gate Turn Off thyristor* – vypinatelný tyristor, více viz [9, kap. 1.4]
IGBT	*Insulated gate bipolar transistor* – bipolární tranzistor s izolovanou bází, více v [9, kap. 1.8]
IGCT	*Integrated Gate Commutated Thyristor* – vypinatelný tyristor s integrovanými budiči řídicí elektrody, více viz [9, kap. 1.5]
MOS-FET	*Metal Oxide Semiconductor Field Effect Transistor* – unipolární výkonový tranzistor s izolovaným hradlem, více viz [9, kap. 1.7]
PWM	*Pulse–Width Modulation* – pulzně–šířková modulace
TQFP	*Thin Quad Flat Pack* – tenké čtvercové pouzdro pro SMT integrované obvody s vývody na všech stranách

μP	mikroprocesor, v obecném pojetí souhrný název pro mikrokontroléry, DSC, DSP, mikroprocesory a podobně

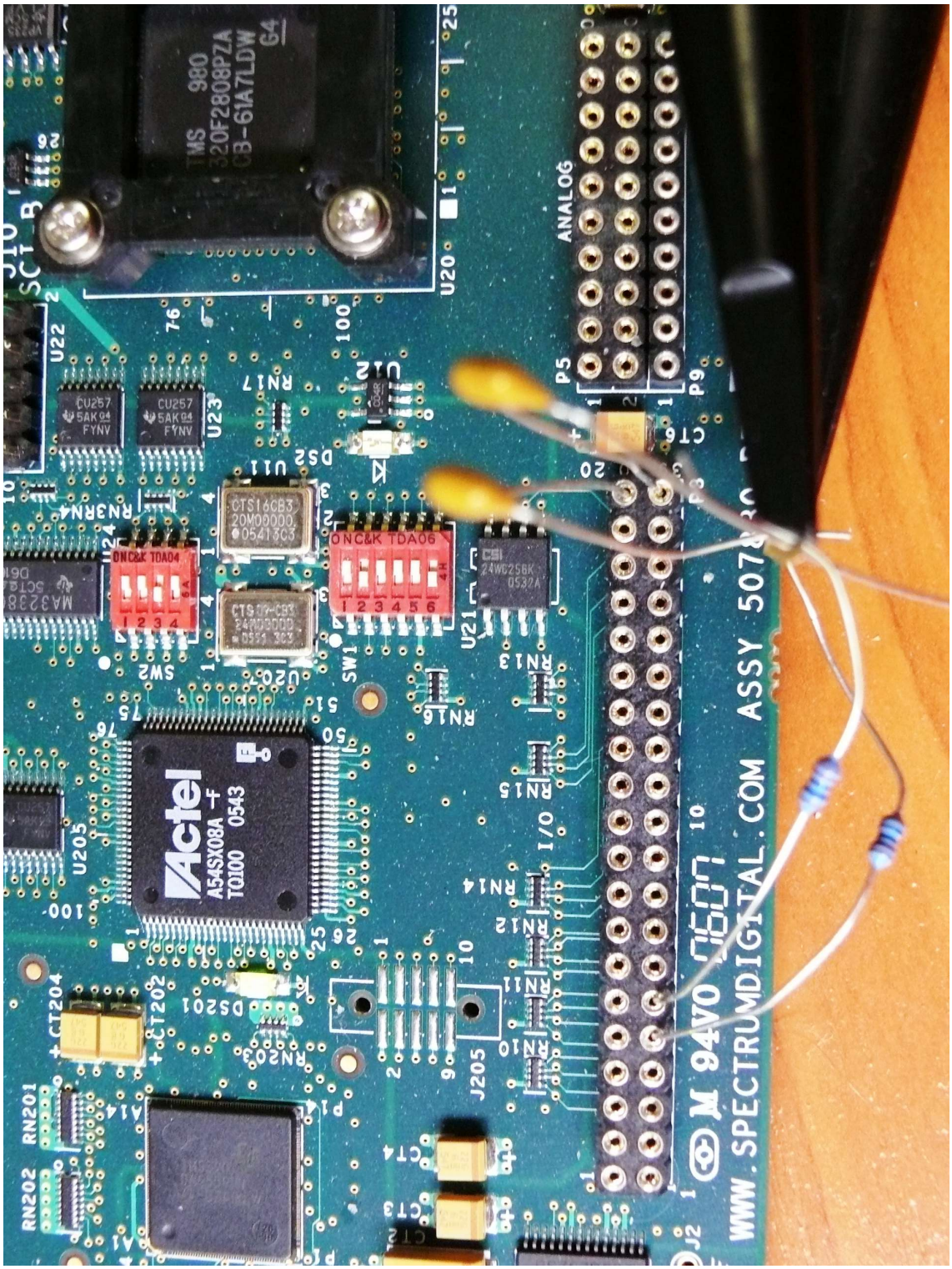
SEZNAM PŘÍLOH

A FOTOGRAFIE PRACOVIŠTĚ

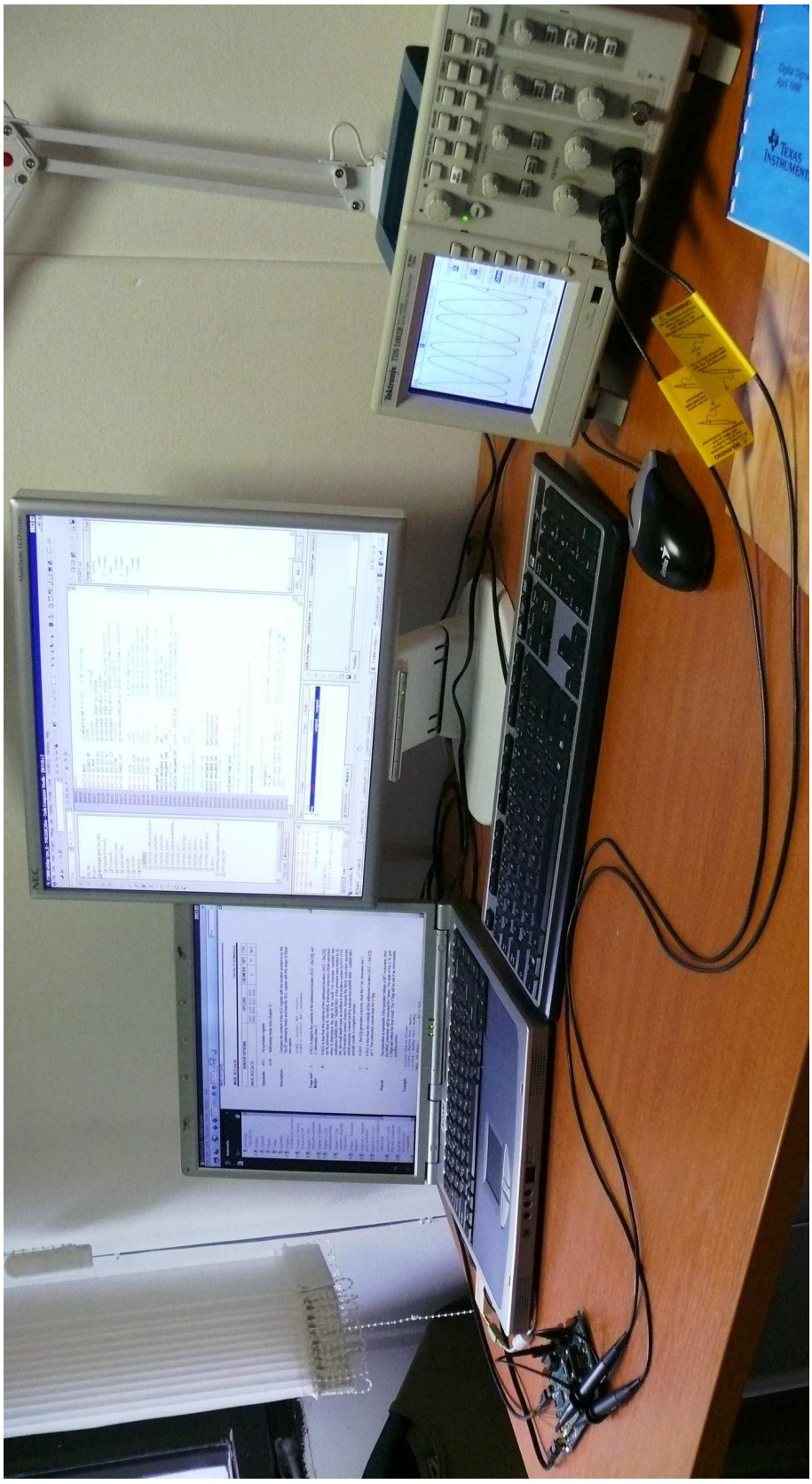
Fotografie jsou pro lepší využití prostoru stránky otočeny o 90°.



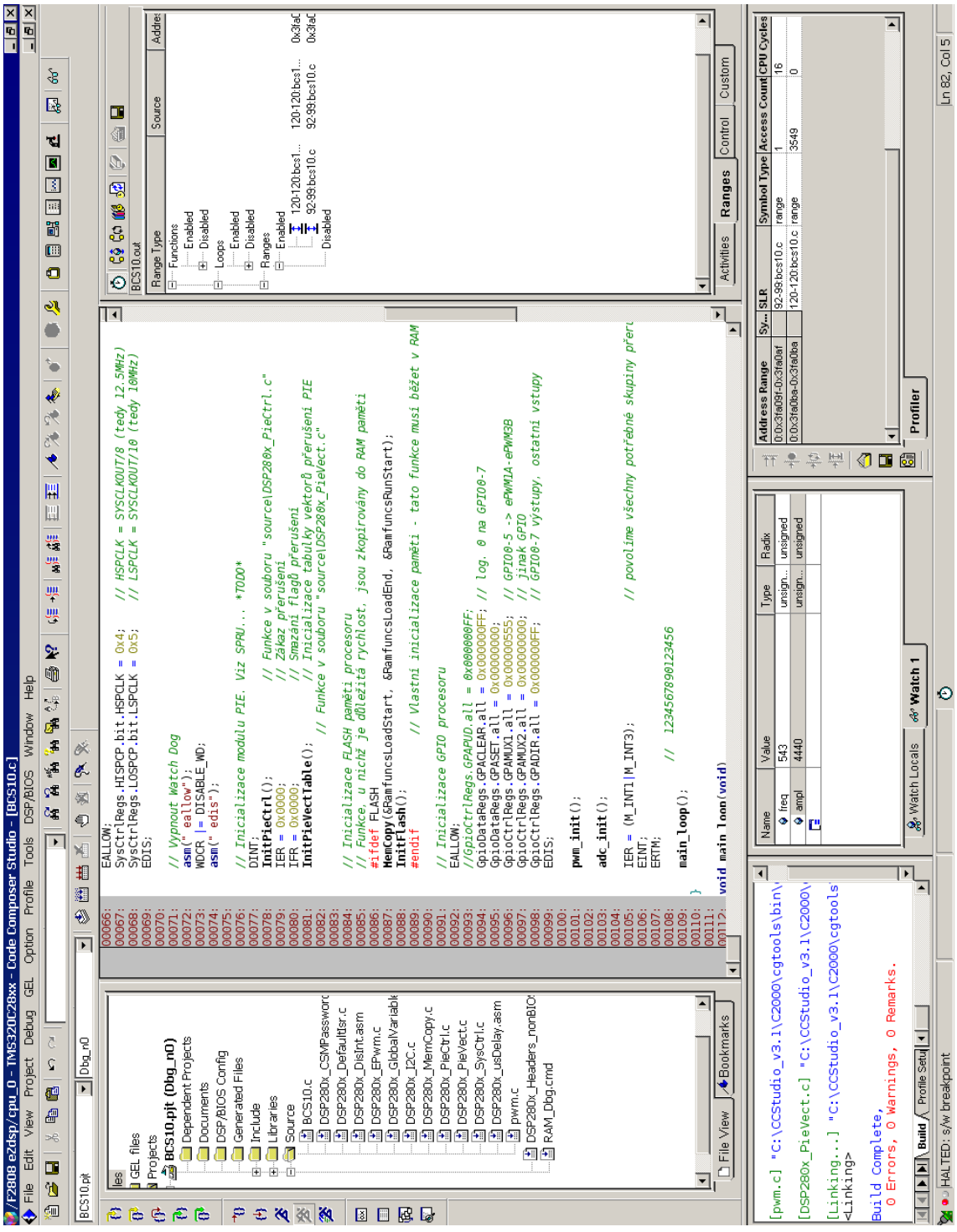
Obr. A.1: Fotografie univerzální vývojové desky ezDSP s digitálním signálovým kontrolérem TMS320F2808 (v patici na pravé části desky).



Obr. A.2: Fofografie připojení osciloskopu k desce ezDSP. Dutinky č. 7 a 8 náleží výstupům PWM1A a PWM1B, 9 a 10 jsou PWM2A/B a 11,12 jsou PWM3A/B. Osciloskop připojen přes filtrační RC články na piny 7 a 9, tedy PWM1A a PWM2A. Dutinky 39 a 40 jsou, stejně jako do nich vsunuté vývody 100nF keramických kondenzátorů RC článků, připojeny na zem.



Obr. A.3: Fotografie celého pracoviště s vývojovou deskou ezDSP, počítačem a osciloskopem Tektronix TDS1002B.



Obr. A.4: Snímek obrazovky IDE Code Composer Studio. V levém sloupci je adresářová struktura projektu, v prostřední části zdrojový kód programu. V pravé části okno Profilování, kde můžeme vybrat části kódu, funkce nebo smyčky, které hodláme optimalizovat. Výsledek profilování (počet a délku vykonávání jednotlivých částí) kódu vidíme v pravé spodní části okna. Uprostřed můžeme sledovat nebo měnit hodnoty jednotlivých proměnných a v levé spodní části vidíme výsledek překladu zdrojových kódů aplikace.

B POČ. ODHAD NEWTONOVY METODY

Tato kapitola popisuje jednoduchý program, který zkouší zadaný interval počátečních hodnot Newtonovy metody. Pro každou hodnotu zjistí, kolik iterací je třeba k dosažení dané přesnosti výsledku. Výstupem programu je, mimo hlavičky a ukončovacího řádku, pole čísel, oddělených středníkem (tzv. CSV formát). Každý řádek platí pro hodnotu počátečního odhadu, vypsanou na první pozici. Jednotlivé sloupce odpovídají vstupním hodnotám x ze zadaného rozsahu. Výsledky vypíše na standardní výstup ve formátu čísel oddělených středníkem.

Program obsahuje iterační algoritmus pro celočíselný výpočet inverzní odmocniny. Je dělaný poměrně univerzálně tak, aby jej bylo možné jakkoli upravit, doplnit algoritmy pro výpočty jiných funkcí nebo změnou prvního `for` cyklu počítat počáteční odhad na základě x. Změny parametrů jsou pro jednoduchost řešeny úpravami zdrojového kódu. Po každé úpravě je tedy třeba program přeložit a spustit. To usnadňuje program GNU make, který na základě souboru makefile zavolá překladač a spustí přeložený program. Výstup programu přesměruje do souboru vystup.csv, který můžeme dále zpracovat tabulkovým editorem (např. MS Excel).

Algoritmus byl v níže uvedené podobě přeložen překladačem GCC 3.4.5 v prostředí MingW32. Spustitelný binární soubor uložený na CD byl odzkoušen pod operačním systémem Windows XP, měl by však snad bez problémů běžet na všech 32-bitových verzích OS Windows.

```
#include <stdio.h>
#include <math.h>

// volbou následujících konstant změníme chování programu
#define FQ      16        // interní reprezentace čísel - Q formát, viz kap. 2.3.1
#define TOLER   0.2/100   // tolerance výsledku Newtonovy metody (0.2/100 = 0,2%)
#define PMIN    2         // minimální počáteční odhad
#define PMAX    20000     // maximální počáteční odhad
#define PDIF    1         // krok počátečního odhadu
#define XMIN    2         // počáteční hodnota vstupní proměnné
#define XMAX    50        // maximální hodnota vstupní proměnné
#define XDIF    1         // krok vstupní proměnné
#define STEPS   50        // maximální počet iterací

#define K1      ((unsigned long)1<<FQ)      // = 2^16 = 1,0 ve formátu Q16
#define K1p5    ((unsigned long)(1.5*(K1))) // 1,5 ve formátu Q16
#define SQRT    1         // výpočet odmocniny
#define ISQRT   2         //         inverzní odmocniny
#define INV     3         //         převráceného čísla

#define FUNKCE  ISQRT     // volba funkce, která se použije

// Následující funkce vrátí počet iterací pro dosažení daného výsledku
// unsigned long ini - počáteční odhad
// long x - vstupní proměnná
// float spravne - hodnota správného výsledku
unsigned short iteraci(unsigned long ini, long x, float spravne)
```

```
{
  long i;
  short j;
  long long pom;
  i = ini;
  for(j=0; j<STEPS; j++)
  {
    // Podobně můžeme definovat i jiné iterační funkce:
    #if (FUNKCE == ISQRT)
      #define FKON  (1.0/sqrt(j)) // kontrolní funkce pro ověření výsledku
      pom = (i*i)>>FQ;            // iterace podle vztahu 2.34
      pom = (pom*(x>>1))>>FQ;
      i = (i*(K1p5 - pom))>>FQ;
    #endif
    if((((float)i/K1) < (1.0+(TOLER))*spravne) && (((float)i/K1) > (1.0-(TOLER))*spravne))
      return j;                   // je-li výsledek v toleranci, vrátíme počet iterací
  }
  return STEPS;                   // v opačném případě vrátíme max. počet
}

int main(void)
{
  unsigned long i, j;
  float res;
  short iter;

  printf(" Program ITERACE  - A. Vasicek 2010\n");
  printf("----------------------------------------\n");
  printf("Sloupec 1: pocatecni odhad iterace\n");
  printf("Dalsi sloupce: pocet iteraci pro\n");
  printf("           toleranci vysledku do 0.2%%\n");
  printf("========================================\n");
  printf("y0; %d; ....(krok %d)....; %d;\n", XMIN, XDIF, XMAX);
  printf("----------------------------------------\n");

  for(i = PMIN; i<PMAX+PDIF; i+=PDIF)
  {
    printf("%d;", i);             // první vypsanou položkou bude počáteční odhad y0
    for(j=XMIN; j<XMAX+XDIF; j+=XDIF)
    {                             // dále pro všechny vstupní hodnoty z rozsahu...
      res = FKON;                 // ...zjistíme počet iterací...
      iter = iteraci(i, (long)j*K1, res);
      printf("%d;", iter);        // ...a vypíšeme jej
    }
    printf("\n");                 // pro další y0 začneme na novém řádku
  }
  printf("========================================\n");
  printf(" Konec!!!");
  return 0;
}
```

C ADRESÁŘOVÁ STRUKTURA CD

Výpis adresářové struktury CD, které je přílohou práce, je tvořen názvy jednotlivých adresářů, psaných bezpatkovým písmem. U jednotlivých adresářů je standardním patkovým písmem popsán jejich obsah.

- Kořenový adresář obsahuje elektronickou podobu práce ve formátu PDF.
- kody obsahuje zdrojové kódy optimalizovaného programu a soubory potřebné CCS.
  - coff obsahuje jednotlivé verze programu, přeložené do objektového formátu COFF.
  - include obsahuje hlavičkové soubory pro práci s periferiemi DSC a pro jednotlivé zdrojové soubory.
  - sources obsahuje zdrojové kódy dalších funkcí, nutných pro úspěšné přeložení programu.
  - vysledky obsahuje výsledky profilování kódu ve formátu CSV.
- literatura obshuje literaturu, která je volně dostupná na internetu, ve formátu PDF. Jedná se o dokumenty platné v květnu 2010, časem se však jejich internetová podoba může měnit.
- newton obsahuje komentovaný zdrojový kód programu pro určení počátečních odhadů Newtonovy iterační metody, přeložený program, soubor makefile pro automatizaci překladu a vytvoření výsledku a ukázkový výstupní soubor va formátu CSV.
- obrazky obsahuje veškeré obrázky, grafy, schémata a průběhy z této práce ve formátech PDF, PNG, JPEG nebo SVG.