UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Kristijan Boček

Aritmetična knjižnica vgrajenega sistema za vodenje zaščitnega releja

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: prof. dr. Branko Šter

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00122/2011
Datum: 05.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **KRISTIJAN BOČEK**

Naslov: **ARITMETIČNA KNJIŽNICA VGRAJENEGA SISTEMA ZA VODENJE ZAŠČITNEGA RELEJA**

ARITHMETIC LIBRARY OF EMBEDDED SYSTEM FOR CONTROL OF A FEEDER PROTECTION RELAY

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Zaščitni rele je naprava, namenjena ščitenju srednjenapetostnih vodov. Rele vodi vgrajen mikroprocesorski sistem, za katerega predpostavimo, da ne podpira operacij s plavajočo vejico. Napišite aritmetično knjižnico, ki z uporabo operacij nad celimi števili realizira naslednje funkcionalnosti nad števili, zapisanimi s plavajočo vejico: množenje, deljenje, seštevanje in odštevanje, kvadratni koren, sinus, kosinus, arkus tangens, dvojiški in desetiški logaritem. Pokažite tudi pravilnost delovanja knjižnice.

Mentor:

prof. dr. Branko Šter

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Kristijan Boček,

z vpisno številko 63000142,

sem avtor diplomskega dela z naslovom:

Aritmetična knjižnica vgrajenega sistema za vodenje zaščitnega releja

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Branka Štera
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 05.07.2011

Zahvala

Na tem mestu, bi se rad zahvalil svojemu mentorju, prof. dr. Branku Šteru, za njegovo prijazno pomoč in nasvete pri izdelavi diplomske naloge. Velika zahvala gre moji Nini, ki me je spodbujala in mi vedno stoji ob strani. Zahvala gre tudi mojim staršem in tašči, ki so predolgo čakali na zaključek mojega študija.

Kazalo

Seznam uporabljenih kratic in simbolov

FPC	Feeder protection control - zaščitni rele, je naprava namenjena ščitenju srednjenapetostnih vodov, lahko pa se uporablja tudi kot rezervna zaščita pri transformatorjih ali visokonapetostnih vodih.
FP	Floating point - plavajoča vejica. V računalništvu s tem izrazom opisujemo način za predstavitev realnih števil.
FPU	Floating point unit, je del računalniškega sistema, namenjen opravljanju matematičnih operacij s števili s plavajočo vejico. V uporabi je tudi izraz matematični ko-procesor.
FPE	Floating point emulation - posnemanje enote za delo s plavajočo vejico
CPU	Central processing unit - centralna procesna enota
RISC	Reduced instruction set computer - računalnik z omejenim naborom ukazov
RTAI	Real Time Application Interface - v mesnik za aplikacije v realnem času za Linux
IEEE	Institute of electrical and electronics engineers - neprofitno stanovsko združenje, namenjeno pospeševanju tehnoloških inovacij.
IEEE754	Standard za števila s plavajočo vejico, ki je nastal pod okriljem IEEE

Povzetek

Zaščitni rele oziroma FPR (feeder protection relay) je v splošnem naprava namenjena ščitenju srednjenapetostnih vodov, lahko pa se uporablja tudi kot rezervna zaščita pri transformatorjih ali visokonapetostnih vodih. Praviloma obsega celoten nabor zaščitnih funkcij, funkcij vodenja in lokalne avtomatike. V večini primerov lahko zaščitni rele opredelimo kot vgrajeni (embedded) sistem, ki izvaja določeno (zaščitno) funkcijo ali več funkcij, v realnem času.

Vgrajeni sistemi so računalniški sistemi, namenjeni izvajanju ene ali več namenskih funkcij. Le-te se velikokrat izvajajo v realnem času. Primer sodobnega vgrajenega sistema je npr. ADSL modem. Takšni sistemi so iz strojnega vidika relativno nezmogljivi, zato naletimo na velike težave, ko hočemo funkcionalnost razširiti izven okvirja, določenega za dotično strojno opremo. Glavni tak problem, ki sem ga skušal odpraviti v diplomskem delu, je odsotnost enote za delo s števili s plavajočo vejico (FPU). Problem je še toliko večji, ker mora zaščitni rele za pravilno delovanje v zelo kratkem času (nekaj ms) opraviti kopico kompleksnih izračunov (zajem podatkov, AD/DA pretvorba, algoritmi zaščit ...) v domeni realnih števil. Na tem mestu se nam postavlja vprašanje, ki je rdeča nit diplomskega dela, kako takšne probleme rešimo v domeni celih števil.

V diplomskem delu se bom omejil na sisteme, ki temeljijo na procesorju StrongARM (SA1110) in operacijskem sistemu Linux z RTAI.

Ključne besede:

zaščitni rele, StrongARM, Linux, vgrajen sistem, plavajoča vejica, realni čas

Abstract

Feeder protection relay or FPR is a device designed to protect medium-voltage lines. It can also be used as a backup protection for transformers or high voltage lines. Generally it covers a full range of protective functions, control functions and local automation. In most cases, the protection relay can be defined as embedded system, which carries out certain (protective) function or more of them, in real time.

An embedded system is a computer system designed to perform one or a few dedicated and/or specific functions, often with real-time computing constraints. ADSL modem is a modern example of an embedded system. Low cost combined with low power consumption puts restrictions on the type of CPU that an embedded system can have, normally it is low frequency (50 – 500 MHz) CPU without hardware FPU. The Main topic in my diploma work is how to deal with real numbers on such platforms without FPU. The problem is even larger because the FPR, to function properly, must in a very short time (few ms) perform thousands of complex calculations (data acquisition, AD/DA conversion, protections ...).

The diploma work is based on computer systems with processor StrongArm (SA1110) and operating system Linux with RTAI.

Key words:

feeder protection relay, StrongARM, Linux, embedded system, floating pint, real time

Poglavje 1

Uvod

1.1 Zaščitni rele

Zaščitni rele je naprava, ki je namenjena ščitenju in upravljanju distribucijskih in industrijskih elektroenergetskih omrežij. Lahko se uporablja tudi kot rezervna zaščita pri transformatorjih ali visokonapetostnih vodih. Obsega celoten nabor zaščitnih funkcij, funkcij vodenja in lokalne avtomatike.

1.1.1 Strojna oprema

S strojnega vidika lahko strojna oprema teh naprav temelji na poljubnih arhitekturah kot so ARM, Alpha, SH, PowerPC, x86 in druge, vendar mora izpolnjevati naslednje pogoje:

- majhna (minimalna) poraba električne energije ($\leq$1W)
- zanesljivo delovanje ($>$ 50 let)
- optimalna zmogljivost v različnih okoljih oz. pogojih (mraz, vročina, vlaga, prah)

Praviloma je z majhno porabo pogojena zmogljivost strojne opreme, predvsem centralne procesne enote (CPE). Ponavadi je ta matematično in tudi na splošno relativno nezmogljiva (50-500 Mhz) in najpomembneje, takšne CPE so brez strojne podpore za delo s števili s plavajočo vejico oz. so brez FP enote (FPE).

StrongARM je družina mikroprocesorjev, ki imajo vse zgoraj opisane lastnosti. Njihova uporaba je zelo razširjena ne samo v elektroenergetiki, pač pa tudi v ostalih segmentih vgrajenih sistemov, kot so npr: prenosni telefoni, PDA, kamere, usmerjevalniki, LAN stikala itd. Vsi procesorji iz družine StrongARM temeljijo na arhitekturi ARMv4. Družino sestavljajo spodaj našteti modeli:

- SA-110, 2.5 milijona tranzistorjev (100, 160, 200 MHz)
- SA-1100, 2.5 milijona tranzistorjev
- SA-1110, Intel (133, 206 MHz)
- SA-1500,

V diplomskem delu se bom omejil na Intelov mikroprocesor StrongARM SA-1110, ki je optimiziran za prenosne in vgradne aplikacije. Procesor je 32-bitni Strong ARM RISC, katerega jedro deluje pri 206 MHz. Hitrost zunanjega vodila je 103 MHz. Procesor ima obsežen inštrukcijski in podatkovni predpomnilnik, memory management unit (MMU) in bralno pisalne bufferje. Sposoben je komunicirati z mnogimi perifernimi napravami, kot so: sinhroni DRAM, sinhroni mask ROM (SMROM) in I/O napravami, podobnimi SRAM-u.

1.1.2 Programska oprema

Za programski del bomo uporabili operacijski sistem Linux. Linux je zelo priljubljen operacijski sistem v svetu vgrajenih sistemov. Razlogov za to je več:

- na voljo je pod pogoji GNU (General Public License) in kot tak v osnovi brezplačen
- je odprtokoden in kot tak relativno preprost za implementacijo raznih sprememb
- prenesen (ang. ported) je na vrsto različnih arhitektur kot so: Alpha, MIPS, PowerPC, SH, SPARC, ARM in ostali

Za programiranje aplikacij, ki se izvajajo v realnem času, bom uporabil **RTAI** (RealTime Application Interface) za Linux. RTAI je dodatek za Linuxovo jedro, ki omogoča pisanje aplikacij s strogimi časovnimi omejitvami. Prav tako kot Linux je tudi RTAI odprtokoden in podpira več različnih arhitektur:

- x86 (z in brez FPU)
- x86-64
- PowerPC
- ARM (StrongARM, ARM7: družina clps711x, Cirrus Logic EP7xxx, CS89712, PXA25x)
- MIPS

Zagotavlja deterministično odziv na prekinitve za POSIX-skladne naloge v realnem času. Vmesnik je sestavljen iz dveh glavnih delov:

- Popravek (patch) za Linux-ovo jedro, ki omogoči abstrakcijo strojne opreme oz. HUL (hardware abstraction layer).
- Široka paleta orodij, ki poenostavljajo programiranje aplikacij v realnem času.

Poglavje 2

Optimizacija vgrajenih Linux sistemov brez FPU

2.1 FPU

FPU (floating point unit), enota za računanje s plavajočo vejico ali, kot ga drugače imenujemo računski so(ko)procesor. Namenjen je računanju z realnimi števili. Običajne operacije, ki se izvajajo na FPU, so seštevanje, odštevanje, množenje, deljenje in kvadratni koren. Izjemoma pa različne transcendentne funkcije, kot so eksponentna in trigonometrične funkcije. Te so večinoma realizirane programsko.

Iz procesorskega vidika izgleda koprocesor kot več pomnilniških lokacij, kamor CPU nalaga operande in operacijsko kodo matematične opearacije (ukaz) in jemlje rezultate. Po navadi glavni procesor preneha delati tako dolgo, dokler ne dobi signala iz koprocesorja za uporabo rezultatov. To dosežemo s signalom BUSY ali WAIT, ki jih generira koprocesor.

2.2 Emulacija FPU

Aritmetika v plavajoči vejici se je v računalniških sistemih vedno obravnavala ločeno od aritmetike v fiksni vejici. Vendar lahko z uporabo slednje programsko realiziramo prvo, kar imenujemo FPU emulacija. Emulacija (posnemanje) je računalniška metoda, s pomočjo katere lahko nek računalnik (procesor) izvaja naloge, ki so bile napisane za nek drug računalnik (procesor).

V primeru, ko imamo v sistemu na razpolago FPU in ustrezen prevajalnik, je manipulacija s plavajočo vejico iz uporabniškega vidika preprosta. Prevajalnik preda operacijsko kodo (opcode), namenjeno izvajanju na FPU. FPU ima običajno določene dodatne registre za svojo uporabo, kamor shranjuje končne rezultate določenih operacij. Preko teh registrov prevajalnik predaja funkcijam vrednosti FP argumentov. To je seveda najboljša rešitev, v primeru ko uporabljamo strojni FPU.

Če strojne FPU nimamo oz. če le-te ne želimo uporabiti, se poslužimo FPU emulacije. Prevajalnik v tem primeru nadomesti FP operacijo z množico raznih celoštevilskih operacij. Za takšen pristop načeloma potrebujemo poseben prevajalnik in posebno **matematično knjižnico**, v kateri so realizirani postopki, ki prevedejo FP operacije v celoštevilske. To pa ni nujno pogoj, saj večina operacijskih sistemov Linux vsebuje jedro (kernel), ki ima vključeno FPE podporo. In sicer to izgleda tako, da ko pride do izvedbe FPU ukaza, le-ta na sistemih brez FPE povzroči napako, saj CPU prevzame in poskusi izvesti neveljaven ukaz (unknown instruction exception). Linux-ovo jedro, ki ima vključeno podporo za FPE, takšno izjemo prestreže in ustrezno reagira, tako da jo nadomesti z množico celoštevilskih operacij (emulacija). Očitno je, da je takšna rešitev mnogo slabša (počasnejša) od prejšnje, saj mora jedro v tem primeru opraviti vse FP izračune in emulirati celotno FPU.

Iz performančnega vidika je za emulacijo FPU precej bolj smotrna uporaba posebne matematične knjižnice.

Poglavje 3

Matematična knjižnica

Knjižnica je v računalniški terminologiji izraz za skupek funkcij, konstant, razredov, objektov in predlog, ki jih uporablja nek programski jezik.

V našem primeru bom v knjižnici implementiral algoritme za osnovno aritmetiko nad realnimi števili, predstavljenimi v obliki mantisa-eksponent.

Za realizacijo različnih funkcionalnosti bomo v matematični knjižnici uporabili ti. preslikovalno tabelo (lookup table - LUT). V računalništvu ta izraz označuje podatkovno strukturo, ponavadi polje, kamor so shranjeni predhodno izračunani rezultati za določeno operacijo. Osnovna ideja je, da predhodno izračunamo rezultate zapletenih operacij, ki so ali se lahko izrazijo kot funkcija celoštevilčne vrednosti (indeks je celo število). Rezultati, ki so shranjeni v tabeli, so na voljo kadarkoli med izvajanjem določene aplikacije, ne da bi morali medtem opraviti celoten kompleksen izračun. Prednost takega pristopa je predvsem hiter dostop do rezultatov, ki je hitrejši kot pa če bi vedno znova računali rezultat določene operacije. Poglejmo si generiranje tabele in nato uporabo na preprostem primeru kvadratnega korena celega števila

Algoritem 1 Kvadratni koren - LUT

Vhod: /
Izhod: /

```
1: int Tabela[256]
2: for i = 0 → 255 do
3:     Tabela[i] = sqrt(i) << 4
4: end for
```

Ko imamo zgrajeno tabelo, nam rezultatov ni potrebno vedno znova računati s funkcijo sqrt, ampak le-te enostavno poiščemo v tabeli (izračunamo indeks). V tem primeru je računanje indeksa nepotrebno, saj je vrednost vhodnega parametra x hkrati indeks rezultata v tabeli.

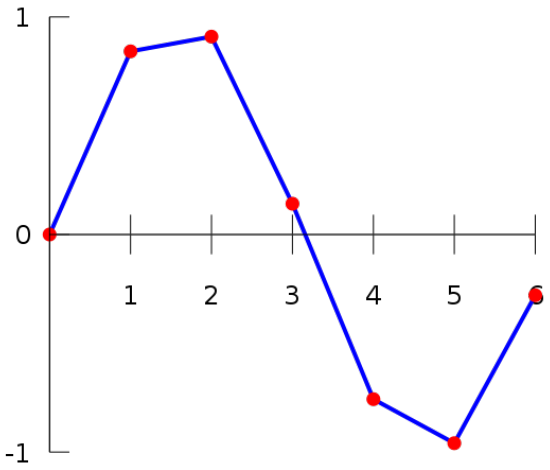
$$\sqrt{x} = \text{Tabela } [x]$$

Pri načrtovanju tabele moramo biti pozorni na:

- **Perioda vzorčenja** oz. razdalja med sosednjima elementoma (vzorcema) posredno določa točnost končnega rezultata. Ker funkcijo vzorčimo na zaprtem intervalu večja oz. manjša perioda vzorčenja pomeni manjše oz. večje število elementov tabele (vzorcev) in posledično bolj oz. manj točen končni rezultat.

- **Ločljivost** pomeni število bitov, ki jih porabimo za predstavitev vrednosti posameznega elementa tabele (vzorec). Moramo se zavedati, da operiramo v množici celih števil, zato v tabelo ne moremo shranjevati realnih vrednosti. Posledično moramo te realne vrednosti pretvoriti v celoštevilske. V naših primerih bomo to storili tako, da realne vrednosti pomnožimo z nekim od 0 različnim faktorjem, ki mora biti dovolj velik, da obdržimo želeno natančnost vzorcev. Uporabnik oz. programer, ki uporablja takšne algoritme, se mora ob interpretaciji končnih rezultatov tega nujno zavedati. Vzemimo funkcijo kvadratnega korena, ki jo vzorčimo na intervalu [0,255] s periodo 1 in ločljivostjo 8 bitov. Rezultat postopka je tabela velikosti 255 elementov, katerih vrednosti so pomnožene s 16 (4 biti) in ne z 256 (8 bitov), kot bi sprva mislili. Razlog je preprost, in sicer, ločljivost podaja bitno globino celotnega vzorca in ne samo faktorja. Največja vrednost funkcije na intervalu je 15,96871. Za predstavitev celega dela porabimo 4 bite, torej nam preostanejo za decimalni del le še 4 biti. Rezultat, ki ga vpišemo v tabelo, je potemtakem **round(sqrt(x)*16)= 255**. Napaka, ki smo jo pri tem zagrešili, je $\approx 0,2\%$

- **Interpolacija**, nanjo naletimo, kadar moramo vrednost funkcije, ki ima vrednosti znane le v posameznih točkah (pravimo jim interpolacijske točke), izračunati v kakšni točki, različni od interpolacijskih točk. Interpolacija je uporabna predvsem, kadar imamo funkcijo podano v obliki tabele, zanima pa nas vrednost funkcije v točki, ki leži med tabeliranimi vrednostmi. Na sliki 3.1 je predstavljena najhitrejša, linearna interpolacija.



Slika 3.1: Linearna interpolacija na delu sinus funkcije

3.1 Predstavitev števil

Realno število predstavimo s parom celih števil (m, e). Prvo komponento imenujemo mantisa, drugo pa eksponent. Absolutna vrednost mantise se nahaja v mejah $10^{l-1} \leq |m| \leq 10^l - 1$, kjer je naravno število l dolžina mantise. Mantisa je v desetiškem številskem sistemu predstavljena z l-mestnim zapisom. Eksponent je celo število, ki se nahaja na intervalu $0 \leq |e| \leq 10^d - 1$, kjer naravno število d določa območje predstavljivosti števil v tej aritmetiki. Eksponent je predstavljen v desetiškem zapisu z največ d števkami. S parom (m, e) je predstavljeno realno število $x = m * 10^{e-l}$. Na primer, par (100, l), kjer je $l = 3$, predstavlja število $x = 100 * 10^{1-3} = 1.0$. Vsako realno število ni predstavljivo v tej obliki. Če ni, vzamemo najbližje predstavljivo število in pri tem nujno zagrešimo napako. Če upoštevamo, da je mantisa v predpisanih mejah, potem lahko vsako predstavljivo število predstavimo na en sam način, taki predstavitvi pa pravimo normirana. Število nič je izjema, ker je mantisa enaka nič in ni v predpisanih mejah. Predstavimo ga s parom (0, 0).

3.2 Računske operacije

Aritmetika s plavajočo vejico ni točna. Pri predstavitvi, pri računskih operacijah in na koncu pri normiranju, to je, ko izberemo par števil v predpisanih mejah, zagrešimo računske napake.

V spodnjih algoritmih bomo uporabljali celoštevilski konstatnti **MAX_VAL** in **MIN_VAL**, ki predstavljata največjo in najmanjšo vrednost določenega številskega podatkovnega tipa. Če katerikoli številčni operand preseže ti dve vrednosti, pride do preliva (owerflow) ali podliva (underflow) in posledično do napake. Zaradi časovne optimizacije algoritmov, v teh ni realiziranih mehanizmov za detekcijo prelivov in podlivov, čeprav je realizacija sila preprosta.

3.2.1 Vsota in razlika

V tem razdelku je opisana realizacija aritmetične operacije seštevanja. Seveda imamo na tem mestu v mislih tudi operacijo odštevanja, ki pa jo bomo obravnavali kot poseben primer seštevanja, saj je odštevanje v bistvu prištevanje nasprotne vrednosti.

$$a - b = a + (-b)$$

Najprej si bomo, za lažjo predstavo, ogledali, kako izgleda seštevanje/odštevanje števil v plavajoči vejici po standardu IEEE 754 . To bomo storili na primeru dveh števil a_1 in a_2, kjer z m označimo mantiso, z e pa eksponent posameznega števila.

1. če je $e_1 < e_2$, zamenjamo operanda tako, da je a_1 vedno število z večjim eksponenom. Če je $e_1 = e_2$, preverimo mantisi in po potrebi zamenjamo operanda, da velja $a_1 >= a_2$. Izračunamo razliko eksponentov $d = e_1 - e_2 >= 0$.

2. mantiso m_2 pomaknemo v desno za d mest. To je v bistvu operacija binarnega pomika (shift) v desno, kot ga poznamo v nekaterih programskih jezikih kot sta C in C++...

3. seštejemo mantisi m_1 in m_2. Na tem mestu imamo rezultat, ki je sestavljen iz vsote mantis in eksponenta e_1

Z zgornjimi koraki opisan postopek je le kratek, zelo površen oris, ki nam podaja postopke, ki morajo biti realizirani v tem razdelku. Izpustil sem npr. manipulacijo s predznakom, zaokroževanje končnega rezultata in pa navsezadnje reakcijo ob morebitnih prelivih. Razlogi za to so preprosti in sicer v našem primeru je mantisa predznačeno celo število. Zaokroževanje ni potrebno ker, je

seštevanje mantis celoštevilska operacija. Problematični so le morebitni prelivi, ki pa jih bo naš algoritem uspešno napovedoval in temu primerno tudi ukrepal.

Naj bosta z $x_i = (m_i, e_i),\ i = 1, 2$ predstavljeni dve števili. Poiščimo predstavitev (približne) vsote teh dveh števil. Določimo mantiso in eksponent vsote $x = (m_r, e_r)$:

$$e = \max(e_1, e_2), \quad m = [m_1 * 10^{e_1 - e} + m_2 * 10^{e_2 - e}].$$

Z oglatimi oklepaji označimo funkcijo zaokroževanja. Mantiso prilagodimo, da je v predpisanih mejah, in dobimo normirano predstavitev (m_r, e_r). Če je $|m| \geq 10^l$, potem vzamemo $m_r = [m/10]$ in $e_r = e + 1$. Če pa je $|m| \leq 10^{l-1}$, potem pa mantiso pomnožimo s faktorjem 10^s, kjer s izberemo tako, da je mantisa m v predpisanih mejah, torej $m_r = m * 10^s$ in popravimo eksponent $e_r = e - s$. Pri tem lahko prekoračimo obseg eksponenta. Če je vrednost eksponenta premajhna, potem govorimo o podkoračitvi obsega (underflow), če pa je prevelika pa govorimo o prekoračitvi obsega (overflow). Razliko izračunamo tako, da prištejemo nasprotno predznačeno število:

$$e = \max(e_1, e_2), \quad m = [m_1 * 10^{e_1 - e} - m_2 * 10^{e_2 - e}].$$

Pri tem m_r in e_r izračunamo enako kot pri seštevanju.

Algoritem 2 Seštevanje

Vhod: $m1, e1, m2, e2$
Izhod: m, e

1: **if** $m1 < 0$ **then**
2: $predznak1 = -1; m1 = -m1;$
3: **else**
4: $predznak1 = 1;$
5: **end if**
6: **if** $m2 < 0$ **then**
7: $predznak2 = -1; m2 = -m2;$
8: **else**
9: $predznak2 = 1;$
10: **end if**
11: **while** $e1 > e2$ **do**
12: **if** $m1 < (MAX\_VAL/10)$ **then**
13: $m1* = 10; e1 - -;$
14: **else**
15: $m2/ = 10; e2 + +;$
16: **end if**
17: **end while**
18: **while** $e2 > e1$ **do**
19: **if** $m2 < (MAX\_VAL/10)$ **then**
20: $m2* = 10; e2 - -;$
21: **else**
22: $m1/ = 10; e2 + +;$
23: **end if**
24: **end while**
25: **if** $predznak1 == predznak2$ **then**
26: **if** $m1 + m2 < 0$ **then**
27: $temp = m1 \ mod \ 10 + m2 \ mod \ 10 + 5;$
28: $m1 \ / = 10; m1 + = temp;$
29: $m2 \ / = 10; e1 + +;$
30: **end if**
31: **end if**
32: $m = predznak1 * m1 + predznak2 * m2;$
33: $e = e1;$
34: **return** $(m, e);$

Konstanta MAX_VAL je največje število podatkovnega tipa INT32, torej $2^{31} - 1 = 2147483647$. Na tem mestu jo uporabimo kot mehanizem za preprečevanje prelivov.

3.2.2 Množenje

Množenje je ena od osnovnih aritmetičnih dvočlenih operacij. Rezultat množenja je zmnožek ali produkt, števili, ki ju množimo, pa sta množenec in množitelj oziroma faktorja. Operacija množenja števil predstavljenih v obliki *mantisa-eksponent* je sestavljena iz množenja mantis in seštevanja eksponentov.

Na bosta z $x_i = (m_i, e_i), i = 1, 2$ predstavljeni dve števili. Poiščimo predstavitev (približnega) produkta

$$a = m1 * 10^{e1}$$

$$b = m2 * 10^{e2}$$

$$a * b = (m1 * m2) * 10^{(e1-e2)}$$

Na koncu še prilagodimo mantiso tako, da je ta v predpisanih mejah, in dobimo normirano predstavitev (m, e), kjer m in e izračunamo enako kot pri seštevanju.

Algoritem 3 Množenje

Vhod: $m1, e1, m2, e2$
Izhod: m, e

1: $e1 += e2$;
2: $predznak = 1$;
3: **if** $m1 < 0$ **then**
4: $\quad m1 = -m1; predznak = -1$;
5: **end if**
6: **if** $m2 < 0$ **then**
7: $\quad m2 = -m2; predznak = -1$;
8: **end if**
9: $temp = m1 * m2$;
10: $m = predznak * temp$;
11: **if** $m == 0$ **then**
12: $\quad e = 0$;
13: **else**
14: $\quad e = e1$;
15: **end if**
16: **return** (m, e);

3.2.3 Deljenje

Deljenje je aritmetična dvočlena operacija, ki je nasprotna množenju. Naj bosta z $x_i = (m_i, e_i),\ i = 1, 2$ predstavljeni dve števili. Poiščimo predstavitev (približnega) kvocienta;

$$e = e_1 - e_2, \quad m = [m_1/m_2 * 10^l].$$

Operacijo deljenja bomo z uporabo **inverza realnega števila** prevedli na operacijo množenja, ki smo jo realizirali v prejšnjem poglavju. **Inverzni element** ali **inverz** je v algebri element, ki v povezavi z določeno računsko operacijo deluje obratno kot dani elemet a. Inverz elementa a na splošno označimo a^{-1}. Denimo, da je * operacija na množici A in e nevtralni element za to operacijo. Tedaj je inverz elementa $x \in A$ glede na operacijo * tak element $y \in A$, da velja

$$x * y = y * x = e \tag{3.1}$$

če inverz obstaja, je enoličen in, kot smo že omenili, ga označimo z x^{-1} oz $\frac{1}{x}$. Upoštevajoč zgornje trditve lahko zapišemo:

$$\frac{x}{y} = x * y^{-1} \tag{3.2}$$

Za realizacijo potrebujemo funkcijo za izračun inverza realnega števila glede na množenje in pa funkcijo za množenje, ki pa smo jo že realizirali.

Algoritem 4 Inverz

Vhod: $m1, e1$
Izhod: m, e

```
1: if m1 < 0 then
2:     predznak = 1; m = −m1;
3: else
4:     predznak = 1;
5: end if
6: e = −e1
7: while m < (MAX_VAL/10) do
8:     m* = 10; e + +;
9: end while
10: e − = 18;
11: m = (INT64)(MAX_VAL ∗ MAX_VAL)/m;
12: while m > MAX_VAL do
13:     m / = 10; e + +;
14: end while
15: return m ∗ predznak;
```

Rezultat funkcije je inverz realnega števila v obliki mantisa-eksponent. Za izračun končnega rezultata uporabimo algoritem, opisan v prejšnjem razdelku Množenje.

3.2.4 Sinus

Je ena izmed šestih osnovnih trigonometričnih oz. kotnih funkcij. V pravokotnem trikotniku sinus podaja razmerje med kotu nasprotno kateto in hipotenuzo

$$\sin(x) = \frac{\text{nasprotna kateta}}{\text{hipotenuza}}$$

Večina računalnikov, ki je sposobna opravljanja le osnovnih aritmetičnih operacij, ne more neposredno izračunati sinusa. Namesto tega uporabljajo algoritme kot je CORDIC ali kompleksne formule, kot je to na primer Taylorjeva vrsta, za izračun vrednosti sinusa na visoko stopnjo natančnosti:

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}... = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \tag{3.3}$$

Tak pristop je praviloma časovno zelo potraten, še zlati na počasnejših procesorjih. Zato je tak postopek popolnoma neuporaben v aplikacijah, ki potrebujejo več sto izračunov sinusa na sekundo oziroma, kot je to v našem primeru, na milisekundo [ms]. Ena od skupnih rešitev takšnih problemov je uporaba ***preslikovalne tabele***. Najprej izračunamo vrednost funkcije sinus za enako porazdeljene vrednosti kotov na nekem zaprtem intervalu definicijskega območja te funkcije. Nato te vrednosti shranimo v tabelo, kjer so nam na razpolago, kadarkoli je to potrebno. Upoštevajoč **periodičnost** funkcije sinus in pravil za **prehod na ostri kot** (simetrije)

$$\sin(\alpha + 2k\pi) = \sin(\alpha); \quad k = Z \tag{3.4}$$

$$\sin(\pi - \alpha) = \sin(\alpha) \tag{3.5}$$

$$\sin(\pi + \alpha) = -\sin(\alpha) \tag{3.6}$$

je dovolj, da izračunamo vrednosti funkcije le na zaprtem intervalu $[0, 90^0]$.Preden se lotimo gradnje tabele, moramo določiti velikost tabele in ločljivost njenih elementov.

Na kakovost (natančnost) končnega rezultata vplivajo tako število bitov za posamezen vzorec (ločljivost) kot tudi število vzorcev, ki jih zajamemo na intervalu (frekvenca vzorčenja). Več bitov kot porabimo za posamezen vzorec

in več kot je le-teh, boljši bo končni rezultat. Moramo se pa zavedati, da se z večanjem števila vzorcev in bitne globine le-teh sorazmerno povečuje tudi prostorska zahtevnost algoritma. Postopek gradnje preslikovalne tabele:

- Glede na želeno končno natančnost rezultata najprej določimo vzorčno periodo (razmik med sosednima vzorcema) in ločljivost (število bitov za predstavitev posameznega vzorca). V našem primeru bomo uporabili periodo 0,016 in 16 bitno ločljivost.
- Vsak dobljen vzorec binarno pomaknemo v levo za bitno ločljivost in rezultat po potrebi zaokrožimo (rezutat mora biti celo število). V našem primeru postopek izgleda tako:

$$\text{tabela}[0] = \text{round}(\sin(0) * 2^{16}) = 0$$

$$\text{tabela}[1] = \text{round}(\sin(0,016) * 2^{16}) = 18$$

$$\text{tabela}[2] = \text{round}(\sin(0,032) * 2^{16}) = 36$$

$$\ldots$$

$$\text{tabela}[\text{round}\tfrac{90}{0,016}] = \text{round}(\sin(90) * 2^{16}) = 65536$$

Funkcijo, v kateri bomo realizirali funkcionalnost, bomo imenovali ***Sinus***. Delovanje lete je opisano v spodnjih korakih:

- Ker operiramo v domeni celih števil, se nam pojavlja problem, kako funkciji podati vhodni parameter kot, ko le-ta ni celo število. V ta namen vpeljemo dodaten vhodni parameter, ki nam podaja razmerje med kotnimi stopinjami in neko navidezno enoto, ki jo bomo imenovali *delec*. Torej, če hočemo podati funkciji kot $33,567^0$ to storimo tako, da ga pretvorimo v delce. Izmislimo si nek pretvorbeni faktor npr. $1000\frac{delec}{stopinjo}$ in kot pomnožimo. Faktor mora biti dovolj velik, da se ob množenju znebimo vseh decimalk. Rezultat množenja je kot v delcih, ki ga zraven pretvorbenega faktorja podamo funkciji.
- Funkcija sinus je periodična funkcija s periodo 2π, zato kot, ki je izven prve periode, preslikamo na interval $[0, 2\pi]$ z uporabo formule

$$\alpha = \alpha \bmod (360 * \text{pretvorbeni faktor})$$

- Funkcijo smo vzorčili samo v I. kvadrantu, zato moramo kot z uporabo simetrij prevesti iz II, III in IV kvadranta v I.
- Funkcija iz vrednosti kota izračuna indeks rezultata v tabeli. Za to je potrebna perioda vzorčenja, ki mora biti celo število. Če le-ta to ni, jo moramo pomnožiti z ustreznim faktorjem. Rezultat bomo imenovali korak tabele.

$$\text{perioda} * \text{faktor periode} > 1;$$

$$\text{round}(\tfrac{\alpha * \text{faktor periode}}{\text{perioda}}) = indeks$$

Če je le možno, naj bo faktor periode potenca z osnovo 2, saj lahko v tem primeru operaciji množenja in deljenja zamenjamo z binarnim pomikom v levo ali desno.

Algoritem 5 Sinus

Vhod: $kot, faktor$
Izhod: $\sin(x)$

1: **if** $kot < 0$ **then**
2: $kot = -kot; predznak = -1;$
3: **end if**
4: $temp = 360 * faktor;$
5: **if** $kot > temp$ **then**
6: $kot\% = temp;$
7: **end if**
8: **if** $(0 <= temp)$ *and* $(temp <= (90 * faktor))$ **then**
9: **return** $(Tabela[temp] >> KORAK\_TABELE)) * predznak);$
10: **end if**
11: **if** $(90 * faktor < temp)$ *and* $(temp <= 180 * faktor)$ **then**
12: **return** $(Tabela[180 * faktor - temp] >> KORAK\_TABELE)) * predznak);$
13: **end if**
14: **if** $(180 * faktor < temp)$ *and* $(temp <= 270 * faktor)$ **then**
15: **return** $(Tabela[temp - 180 * faktor] >> KORAK\_TABELE)) * predznak);$
16: **end if**
17: **if** $(270 * faktor < temp)$ *and* $(temp <= 360 * faktor)$ **then**
18: **return** $(Tabela[360 * faktor - temp)] >> KORAK\_TABELE)) * predznak);$
19: **end if**

3.2.5 Kosinus

Kotna funkcija kosinus je definirana kot razmerje med kotu priležno kateto in hipotenuzo

$$\cos(x) = \frac{\text{kateta}}{\text{hipotenuza}}$$

Kosinus kota bomo z uporabo ***komplementarnega kota*** in pravil za ***prehod na ostri kot*** izračunali podobno kot sinus iz prejšnjega razdelka.

Ostri kot meri manj kot 90^0. Prehod na ostri kot:

II. Kvadrant $\cos(x) = -\cos(180^0 - x)$

III. Kvadrant $\cos(x) = -\cos(x - 180^0)$

IV. Kvadrant $\cos(x) = \cos(360^0 - x)$

Komplementarna kota sta v geometriji kota, katerih vsota je enaka 90^0. Kotne funkcije komplementarnih kotov:

$$\cos(90^0 - x) = \sin(x)$$

Z uporabo zgornjih pravil lahko napišemo algoritem, ki bo izračunal kosinus danega kota. Spodnji algoritem za izračun kosinusa uporablja isto preslikovalno tabelo kot algoritem za izračun sinusa.

Algoritem 6 Kosinus

Vhod: $kot, faktor$
Izhod: $\cos(x)$

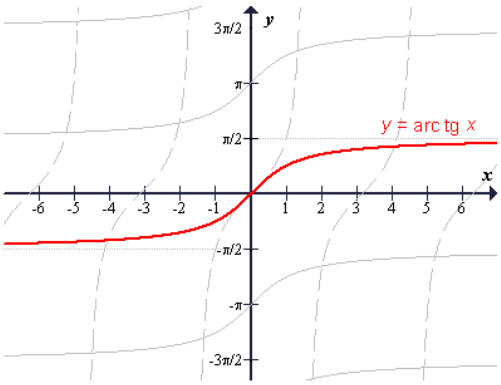
1: **if** $kot < 0$ **then**
2: $kot = -kot$;
3: **end if**
4: $temp = 360 * faktor$;
5: **if** $kot > temp$ **then**
6: $kot\% = temp$;
7: **end if**
8: **if** $(0 <= temp)\ and\ (temp <= (90 * faktor))$ **then**
9: **return** $Tabela[90 * faktor - temp] >> KORAK\_TABELE$;
10: **end if**
11: **if** $(90 * faktor < temp)\ and\ (temp <= 180 * faktor)$ **then**
12: **return** $Tabela[temp - 90 * faktor] >> KORAK\_TABELE$;
13: **end if**
14: **if** $(180 * faktor < temp)\ and\ (temp <= 270 * faktor)$ **then**
15: **return** $Tabela[270 * faktor - temp] >> KORAK\_TABELE$;
16: **end if**
17: **if** $(270 * faktor < temp)\ and\ (temp <= 360 * faktor)$ **then**
18: **return** $Tabela[temp - 270 * faktor] >> KORAK\_TABELE$;
19: **end if**

3.2.6 Inverzni tangens

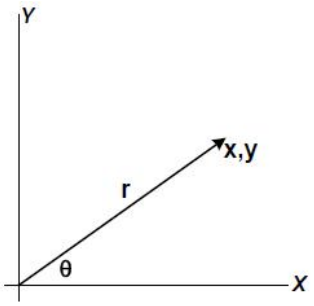
Arkus tangens (arctan ali arctg ali arc tan ali arc tg) je inverz funkcije tangens. Za poljubno realno število a je $\arctan(a)$ enak številu x, za katero velja $\tan(x) = a$:

$$\arctan(a) = x \Leftrightarrow \tan(x) = a$$

Ker ima enačba $\tan(x) = a$ neskončno mnogo realnih rešitev, je po dogovoru rezultat funkcije arkus tangens po absolutni vrednosti najmanjša rešitev te enačbe (glej sliko 3.2).
Za realizacijo funkcionalnosti bomo poskušali rešiti naslednji problem, in sicer iz kartezičnih podatkov neke točke v 2d prostoru bomo poskušali izračunati kotno pozicijo te točke.
Kartezične koordinate določajo oddaljenost točke od središča v x in y smeri. Polarne koordinate definirajo položaj točke, ki je za r oddaljena od izhodišča



Slika 3.2: Graf y = arctan(x)



Slika 3.3: Dvodimenzionalni prikaz kartezičnih in polarnih parametrov

in kot Θ nad oz. pod x-osjo. Vrednosti kartezičnih in polarnih koordinat na različen način definirajo isti edinstven položaj točke (glej sliko 3.3 in 3.4). Njun odnos je prikazan v enačbi spodaj

$$r = \sqrt{x^2 + y^2} \tag{3.7}$$

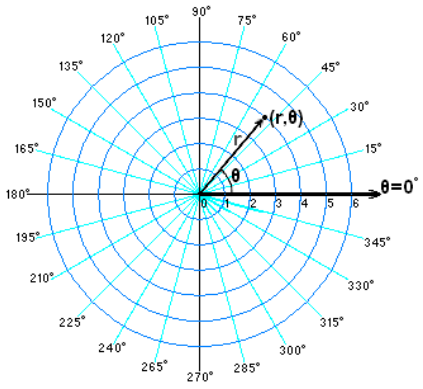
$$\Theta = f(x, y) \tag{3.8}$$

Funkcijo f imenujemo arkus tangens (arctan) in je definirana:

$$f(x, y) = \arctan(\frac{y}{x}) \tag{3.9}$$

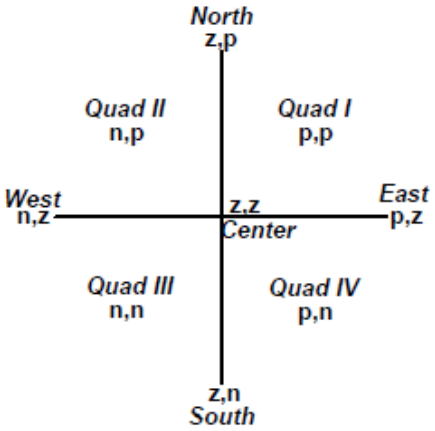
Iz zgornje enačbe dobimo polarni kot Θ - pri tem moramo paziti na pravilno izbiro vrednosti kota, saj upoštevajoč definicijo velja:

$$\arctan(\tfrac{y}{x}) = \arctan(\tfrac{-y}{-x})$$



Slika 3.4: Polarni koordinatni sistem

Vrednosti parametrov, x in y, so lahko pozitivne (P), negativne (N) ali nič (Z), s čimer dobimo 9 različnih kombinacij, ki so prikazane na sliki 3.5



Slika 3.5: 9 kombinacij x-y parametrov

Za realizacijo funkcionalnosti uporabimo, podobno kot za sin in cos, preslikovalno tabelo, katere elementi so vzorci, ki so rezultat vzorčenja funkcije. Če upoštevamo recipročnost parametrov

$$\arctan(\frac{y}{x}) = 90 - \arctan(\frac{x}{y}); \quad x > 0 \tag{3.10}$$

vidimo da je dovolj, če funkcijo vzorčimo na zaprtem intervalu [0,1], saj je $\arctan([0,1]) = [0^0, 45^0]$. V našem primeru smo funkcijo vzorčili z vzorčno periodo 0,016 in 16-bitno globino.

Algoritem 7 Inverzni tangens

Vhod: y, x
Izhod: $\arctan(x/y)$

```
1: Normalizacija x in y na velikost tabele
2: if y >= 0 then
3:     if x >= 0 then
4:         if x >= y then
5:             rezultat = Tabela[(y + 8) >> 4];
6:         else
7:             rezultat = 90 << 16 − Tabela[(y + 8) >> 4];
8:         end if
9:     else
10:        x = −x
11:        if x >= y then
12:            rezultat = 180 << 16 − Tabela[(y + 8) >> 4];
13:        else
14:            rezultat = 90 << 16 + Tabela[(y + 8) >> 4];
15:        end if
16:    end if
17: else
18:    y = −y;
19:    if x >= 0 then
20:        if x >= y then
21:            rezultat = 360 << 16 − Tabela[(y + 8) >> 4];
22:        else
23:            rezultat = 270 << 16 + Tabela[(y + 8) >> 4];
24:        end if
25:    else
26:        x = −x;
27:        if x >= y then
28:            rezultat = 180 << 16 + Tabela[(y + 8) >> 4];
29:        else
30:            rezultat = 270 << 16 − Tabela[(y + 8) >> 4];
31:        end if
32:    end if
33: end if
34: return rezultat
```

3.2.7 Kvadrati koren

V tem razdelku je opisana implementacija celoštevilske funkcije kvadratnega korena **(int)sqrt(int)**. Kvadratni koren je nenegativno realno število, za katerega velja:

$$\sqrt{b} = a \Rightarrow a^2 = b; \quad a \geq 0 \tag{3.11}$$

Realizacija korenjenja je na današnjih, sodobnih računalniških arhitekturah realizirana bodisi strojno oz. z uporabo programskih knjižnic, ki največkrat za izračune izkoriščajo hitre FPU. Če hočemo hiter algoritem, moramo preslikovalno tabelo zasnovati tako, da za računanje indeksa rezultata uporabimo samo kombinacije levega in desnega binarnega pomika (operaciji množenja/deljenja nista zaželeni). Razlog je preprost in sicer na StrongARM je operacija binarnega pomika v povprečju 3x hitrejša kot operacija množenja.
Kvadratni koren je funkcija, definirana na intervalu $[0, \infty)$. Ker nimamo na voljo „neskončno“ velikega podatkovnega tipa, moramo določiti maksimalno vrednost korenjenca, ki jo še lahko obdelamo. Rezultat korenjenja je število, ki je po absolutni vrednosti manjše od korenjenca, zato je maksimalna vrednost korenjenca enaka največjemu številu uporabljenega podatkovnega tipa, npr:

$$\text{UINT16, od 0 do } 65,535\ (2^{16}-1); \quad \sqrt{[0, 65535]} = [0, 256]$$
$$\text{UINT32, od 0 do } 4294967295\ (2^{32}-1); \quad \sqrt{[0, 4294967295]} = [0, 65535]$$

V našem primeru bomo uporabili 32 bitni nepredznačen celoštevilski tip UINT32 (long). Funkcijo tabeliramo na intervalu [0,255]. Tabelirane vrednosti pomnožimo s poljubnim številom, katerega koren je celo število in hkrati potenca števila 2. V našem primeru bomo vrednosti celoštevilsko pomnožili s 16, kar je enako, kot če bi vrednosti binarno premaknili za 4 v levo. Vrednosti, ki padejo izven tabele, normiramo na velikost tabele po spodnjem postopku:

$$2^{2i} \leq x \leq 2^{2i+2} - 1;\ i = 4, 5, 6...$$

$$\sqrt{x} = \sqrt{x} * \frac{\sqrt{2^{2i+2-8}}}{\sqrt{2^{2i+2-8}}} = \frac{\sqrt{x * 2^{2i+2-8}}}{\sqrt{2^{2i+2-8}}} = \sqrt{\frac{x}{2^{2i+2-8}}} * \sqrt{2^{2i+2-8}}$$

Algoritem za izgradnjo preslikovalne tabele smo že uvodoma opisali.

Algoritem 8 Kvadratni koren

Vhod: Celo število (UINT32) $x \geq 0$.
Izhod: $\sqrt{x}$

```
1:  if x ≥ 65536 then
2:      if x ≥ 16777216 then
3:          if x ≥ 268435456 then
4:              if x ≥ 1073741824 then
5:                  if x ≥ 4294836255 then
6:                      return 65535;
7:                  end if
8:                  temp = Tabela[x >> 24] << 8;
9:              else
10:                 temp = Tabela[x >> 22] << 7;
11:             end if
12:         else if x ≥ 67108864 then
13:             temp = Tabela[x >> 20] << 6;
14:         else
15:             temp = Tabela[x >> 18] << 5;
16:         end if
17:     else
18:         if x ≥ 1048576 then
19:             if x ≥ 4194304 then
20:                 temp = Tabela[x >> 16] << 4;
21:             else
22:                 temp = Tabela[x >> 14] << 3;
23:             end if
24:         else
25:             if x ≥ 4194304 then
26:                 temp = Tabela[x >> 12] << 2;
27:             else
28:                 temp = Tabela[x >> 10] << 1;
29:             end if
30:         end if
31:         return temp
32:     end if
33: end if
34: if x ≤ 65536 then
35:     if x ≥ 256 then
36:         if x ≥ 4096 then
37:             if x ≥ 16384 then
38:                 temp = Tabela[x >> 8] + 1;
39:             else
40:                 temp = (Tabela[x >> 6] >> 1) + 1;
41:             end if
42:         else
43:             if x ≥ 1024 then
44:                 temp = (Tabela[x >> 4] >> 2) + 1;
45:             else
46:                 temp = (Tabela[x >> 2] >> 3) + 1;
47:             end if
48:         end if
49:         return temp;
50:     else
51:         return Tabela[x] >> 4;
52:     end if
53:     temp = (temp + 1 + x/temp)/2;
54:     temp = (temp + 1 + x/temp)/2;
55:     if temp ∗ temp ≥ x then
56:         return temp − 1;
57:     end if
58:     return temp;
59: end if
```

3.2.8 Logaritem z osnovo 2

Logaritem oziroma logaritemska funkcija je v matematiki funkcija, ki je definirana:

$$\log_a x = y \Leftrightarrow a^y = x \tag{3.12}$$

Levi del enačbe preberemo logaritem x z osnovo a. x imenujemo logaritmand ali argument. Logaritemska funkcija je definirana le za pozitivna števila, njena zaloga vrednosti pa so vsa realna števila:

$$Df = R^+$$
$$Zf = R$$

V tem razdelku bomo obravnavali realizacijo algoritma za računanje dvojiškega (osnova 2) algoritma. Takšen logaritem imenujemo tudi binarni oz. dvojiški logaritem. Funkcija narašča povsod, kjer je definirana, ima ničlo pri x=1 in ima navpično asimptoto pri x=0.

Izračun dvojiškega logaritma bomo izvedli v dveh korakih, in sicer najprej izračunamo celi del (C), nato pa z uporabo preslikovalne tabele poiščemo še preostanek rezultata (O).

$$\log_2 x = \log_2(2^C * O) = C + \log_2(O) \tag{3.13}$$

Celi del rezultata dobimo tako, da vhodni podatek (logaritmand) binarno premikamo v levo, da dobimo mantiso. Število premikov v levo nam da celi del, ki ga popravimo glede na faktor (povečavo) vhodne vrednosti x, ki ga podamo kot vhodni parameter. Vhodni parameter, faktor, je namenjen za povečanje natančnosti in računanje z "racionalnimi"števili, in sicer se uporablja kot 2^{faktor}. Osnova 2 je uporabljena, ker je dvojiški logaritem izveden s binarnim premikanjem besede. Preden gremo iskati vrednost ostanka v tabelo, odstranimo vodilno 1 in ostanek velikosti 1 do 2 omejimo na število bitov dolžine tabele.

Velikost tabele je poljubna, velja pa splošno pravilo, večja kot je tabela, večja je natančnost končnega rezultata. Npr: če vzamemo tabelo velikosti 512 (2^9) elementov, lahko v njo shranimo 9-bitne ostanke. Za predstavitev ostankov bomo uporabili 10 bitno ločljivost. To izgleda tako, da realne vrednosti pomnožimo z 2^{10} in jih zaokrožene vpišemo v tabelo. Pri majhnih ostankih bo napaka največja. Na splošno napako izračunamo po naslednji formuli

$$\log_2(1 + O_{max}/2^{velikost\ tabele}) \tag{3.14}$$

Napaka se razpolovi s podvojitvijo velikosti tabele.

Algoritem 9 Logaritem z osnovo 2

Vhod: x,faktor $x \geq 0$.
Izhod: $log_2 x$

1: **if** $x \leq 0$ **then**
2: **return** error;
3: **end if**
4: $LOG\_MAX\_VAL = 1 << (ARCH\_INT - 1)$
5: **while** $x \leq LOG\_MAX\_VAL$ **do**
6: $x = x << 1$;
7: $eksponent = eksponent + 1$;
8: **end while**
9: $x = x$ xor LOG_MAX_VAL;
10: $eksponent = eksponent + 1$;
11: $x = x >> (ARCH\_INT - VELIKOST\_TABELE\_BITS - 1)$;
12: $temp = Tabela[x]$;
13: $temp+ = (ARCH\_INT - eksponent - faktor) << LOG\_TAB\_LSHIFT$;
14: $rezultat = TmpValue >> (LOG\_TAB\_LSHIFT - faktor)$;
15: **return** $rezultat$;

Logaritme z drugimi osnovami izračunamo po naslednji formuli

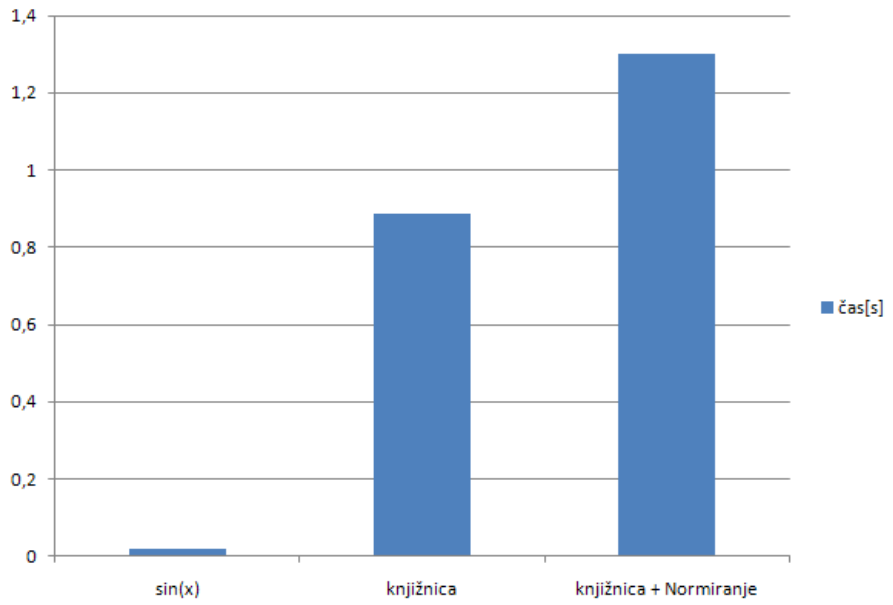
$$\log_a x = \frac{\log_2 x}{\log_2 a} \tag{3.15}$$

Poglavje 4

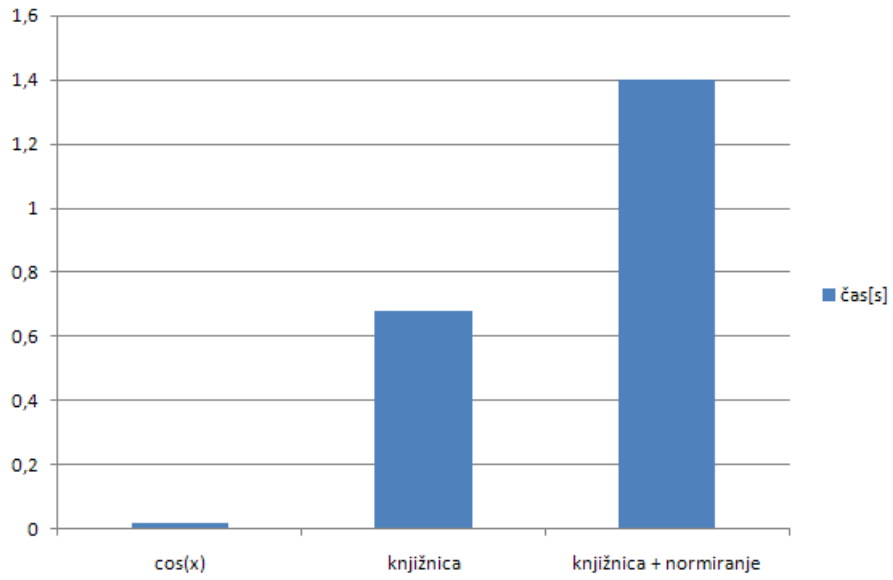
Analiza rezultatov

V tem razdelku bomo predstavili razlike v hitrosti izvajanja funkcij iz naše matematične knjižnice v primerjavi z funkcijami, ki so del standardne matematične knjižnice programskega jezika C/C++ (math.h). Ker na sistemu s procesorjem StrongArm ne moremo izvajati vseh funkcij iz standardne matematične knjižnice programskega jezika C (math.h), sem testiranje izvedel na osebnem računalniku s procesorjem Pentium 4 (2400Mhz). Na spodnjih grafih je prikazan čas, ki ga potrebuje posamezna funkcija za 20000000 izračunov.

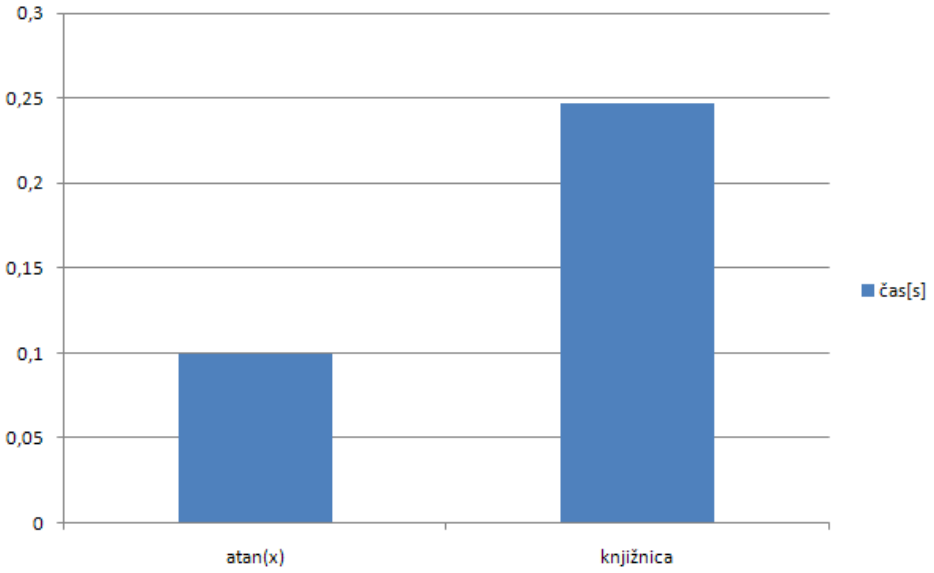
V prvem stolpcu je prikazan izvajalni čas za funkcije, ki so del knjižnice programskega jezika C (math.h). Izvajalni čas funkcij iz knjižnice napisane za zaščitni rele predstavljata drugi in tretji stolpec. V tretjem stoplpcu je prikazan čas, ko mora funkcija zraven osnovnega izračuna opraviti še normalizacijo vhodnega podatka, ker leta pade izven območja delovanja funkcije. Tak primer je npr. funkcija sinus, ki ji podamo kot večji od 360^0.



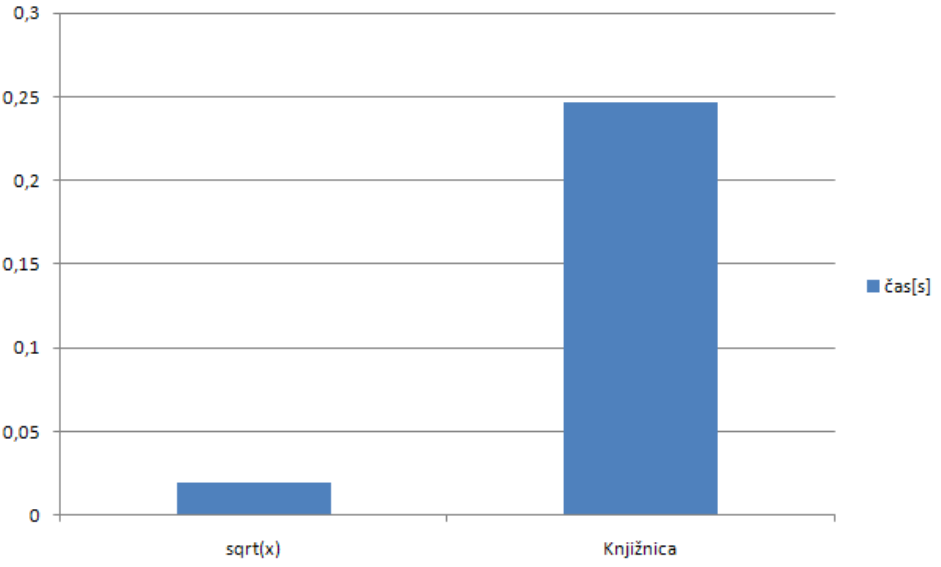
Slika 4.1: Izvajalni časi funkcije, sinus



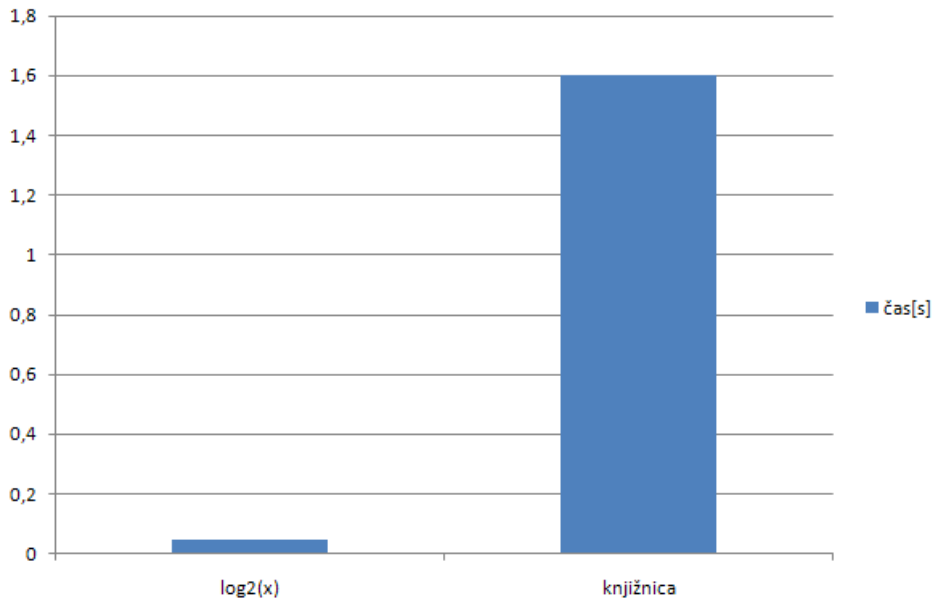
Slika 4.2: Izvajalni časi funkcije, kosinus



Slika 4.3: Izvajalni časi funkcije, inverzni tangens



Slika 4.4: Izvajalni časi funkcije kvadratnega korena



Slika 4.5: Izvajalni časi funkcij logaritem z osnovo 2

Poglavje 5

Sklepne ugotovitve

V diplomski nalogi sem predstavil matematično knjižnico, ki jo za svoje delovanje uporablja vgrajeni sistem za kontrolo zaščitnega releja, ki je eden od pomembnejših gradnikov elektroenergetskih sistemov. Zaščitni rele večino časa takorekoč ne dela nič oziroma le spremlja stanje elektroenergetskega sistema. Vendar, ko mora opraviti določeno nalogo (npr.odklopiti vod), mora le-to opraviti 100-odstotno pravilno in točno takrat, ko je to potrebno. Zato mora celotna programska oprema releja do potankosti ustrezati določenim standardom in normam. Matematična knjižnica, ki je del te programske opreme, ni nobena izjema. Treba se je zavedati, da vsi algoritmi, ki tečejo na napravi, za svoje pravilno delovanje uporabljajo funkcionalnosti, realizirane v tej knjižnici.

Predstavljene rešitve so plod večmesečnega raziskovalnega dela. Ideje za rešitve posameznih funkcionalnosti sem našel na internetnih straneh, le-te sem nato optimiziral in prilagodil za delovanje na našem sistemu. Po končani realizaciji je sledilo obdobje intenzivnega testiranja, kjer se je pojavilo kar nekaj napak zaradi neizkušenosti v programskem jeziku C, saj sem nevede zanemaril probleme sprostitve pomnilniških resursov, odkrivanje tovrstnih napak pa je zaradi neočitnih krivcev nemalokrat težavno.

Najpomembnejši del, iz programskega vidika, vsakega zaščitnega releja so zaščitne funkcije. V našem primeru te funkcije za svoje delovanje uporabljajo funkcionalnosti, ki sem jih realiziral v matematični knjižnici. Torej lahko rečemo, da je pravilno delovanje zaščitnega releja v veliki meri odvisno od matematične knjižnice. Slabo napisana in neoptimizirana knjižnica ima lahko za posledico nepravilno delovanje zaščitnega releja, kar pa je v današnjem času kompleksnih elektroenergetskih sistemov popolnoma nesprejemljivo.

Slike

Literatura

[1] Intel® StrongARM SA-1110 Microprocessor. Dostopno na:
http://http://opensimpad.org/images/8/87/SA-1110_Manual.pdf

[2] Osnove numerične matematike. Dostopno na:
http://www.shrani.si/f/1T/qh/3pORRqa2/knjiga.pdf

[3] Kodek Dušan, *Arhitektura računalniških sistemov*, Ljubljana, 2000

[4] Računalniška arhitektura. Dostopno na:
http://www.fri.uni-lj.si/si/izobrazevanje/8630/class.html

[5] Aritmetika s plavajočo piko. Dostopno na:
http://matematika.fe.unilj.si/people/borut/homepage/numericne_metode/doc/fl.pdf

[6] Floating Point to Fixed Point Conversion of C Code, Dostopno na:
http://www.superkits.net/whitepapers/floating-point-to-fixed.pdf

[7] Square Roots. Dostopno na:
http://www.azillionmonkeys.com/qed/sqroot.html

[8] Sine/Cosine Look-Up Table. Dostopno na:
http://www.xilinx.com/support/documentation/ip_documentation/sincos.pdf

[9] Dave Van Ess, *Algorithm - ArcTan as Fast as You Can*, January 6, 2006
Dostopno na:
http://www.cypress.com/?docID=2279

[10] Optimisation and Implementation of the Arctan Function for the Power
Domain. Dostopno na:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.4931&rep=rep1&type=pdf

[11] Optimization of Embedded Linux systems without FPU. Dostopno na:
http://http://www.ll.mit.edu/HPEC/agendas/proc08/Day1/11-Day1-
PosterDemoA-Spetka-abstract.pdf