KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Vaida Lešinskytė

Dvigubo tikslumo slankaus kablelio daugybos realizavimas ir tyrimas

Magistro darbas

Darbo vadovas:
prof. Vacius Jusas

Kaunas, 2011

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Vaida Lešinskytė

Dvigubo tikslumo slankaus kablelio daugybos realizavimas ir tyrimas

Magistro darbas

Vadovas
prof. Vacius Jusas
2011-05-30

Recenzentas
Doc. Giedrius Ziberkas
2011-05-30

Atliko
IFM-9/5 gr. stud.
Vaida Lešinskytė
2011-05-31

Kaunas, 2011

Summary

The paper analyzes the calculation of floating-point multiplication problems. The first chapter analyzes the translation of the floating-point decimal system to binary system and vice versa. It is necessary to make certain that the issue is clearly brought problem. High-precision decimal numbers translating into binary and back to decimal numbers. Even after these actions we have a loss of accuracy, and even more so come on in action. It is also the first chapter of the last section I presented some of the historical catastrophe of the facts that have arisen precisely because of the floating point precision problems.

The second chapter analyzes the hardware and software problems for the realization of floating-point multiplication. This chapter examined the influence of algorithms hardware speed, as well as a software power and accuracy of the results obtained in calculating the rates. It also includes a description and a few floating-point realization algorithms.

The third section of the Algorithm Implementation Requirements, described the main problems encountered in the realization of the algorithm. And summarize the results.

Work at the end of the conclusions and annexes of the algorithm implementation - programming code snippets and comments.

Santrauka

Darbe analizuojamos slankaus kablelio daugybos apskaičiavimo problemos. Pirmame darbo skyriuje analizuojamas slankaus kablelio vertimas iš dešimtainės sistemos į dvejetainę sistemą ir atvirkščiai. Tai reikalinga atlikti tam, kad išryškėtų nagrinėjamos problemos aktualumas. Jau verčiant dešimtainį didelio tikslumo skaičių į dvejetainį ir atgal į dešimtainį, nukenčia jo tikslumas, o dar labiau tai išryškėja vykdant veiksmus. Taip pat pirmame skyriuje paskutiniame poskyryje pateikiau keletą istorinių katastrofų faktų, kurie kilo būtent dėl slankiojo kablelio tikslumo problemų.

Antrame darbo skyriuje analizuojamos aparatūrinės ir programinės įrangos problemos, kylančios realizuojant slankiojo kablelio daugybą. Šiame skyriuje gvildenta aparatūrinės įrangos įtaka algoritmų spartai, taip pat programinės įrangos įtaka gautam rezultato tikslumui ir apskaičiavimo spartai. Šiame skyriuje taip pat aprašyti ir keli slankiojo kablelio realizacijos algoritmai.

Trečiame skyriuje pateikti algoritmo realizacijos reikalavimai, aprašytos pagrindinės problemos, su kuriomis buvo susidurta vykdant algoritmo realizaciją. Apibendrinti pateikiami ir rezultatai.

Darbo gale pateikiamos darbo išvados, o prieduose pateikta algoritmo realizacija – programinio kodo fragmentai ir jų komentarai.

Turinys

SUMMARY 3
SANTRAUKA 4
TURINYS 5
ĮVADAS 6
1 SLANKAUS KABLELIO ARITMETIKA 9
1.1 APIBRĖŽIMAS IR STANDARTAS 9
1.2 TIPAI: KOKIE JIE IR KUO SKIRIASI 9
1.3 APSKAIČIAVIMO PROBLEMOS 11
1.4 VERTIMO IŠ DEŠIMTAINIO FORMATO Į DVEJETAINĮ FORMATĄ IR ATVIRKŠČIAI PAVYZDŽIAI 12
1.5 KELETAS ISTORINIŲ FAKTŲ IR ĮDOMYBIŲ 14
2 DVIGUBO TIKSLUMO SLANKIOJO KABLELIO DAUGYBA 16
2.1 SKAIČIAVIMUS ĮTAKOJANTYS PARAMETRAI 16
2.2 APARATŪRINIAI DAUGYBOS YPATUMAI 17
2.3 PROGRAMINIAI DAUGYBOS APSKAIČIAVIMO ALGORITMAI 20
2.4 TECHNINIAI ALGORITMŲ APSKAIČIAVIMO ASPEKTAI 25
2.5 REKOMENDACIJOS NAUJAM ALGORITMUI 28
3 DVIGUBO TIKSLUMO SLANKAUS KABLELIO DAUGYBOS REALIZAVIMAS 30
3.1 REIKALAVIMAI TESTINEI PROGRAMAI 30
3.2 REALIZACIJOS PROBLEMOS 30
3.3 REALIZACIJOS REZULTATAI 31
IŠVADOS 33
NAUDOTA LITERATŪRA 34
PRIEDAI 35
PROGRAMOS KODAS 35
Failas dfmul.c *35*
Failas softfloat.c *36*
Failas softfloat.h *40*
Failas milieu.h *40*
Failas SPARC-GCC.h *40*

ĮVADAS

Kai žmonijai nebepakako veiksmų su sveikaisiais skaičiais, atsirado realieji. Vystantis technologijoms, daugėja įvairių sudėtingų skaičiavimų kiekis. Daugumą šių skaičiavimų mes jau patikėjome kompiuteriams. Taip pat didėja ir tikslių skaičiavimų svarba. Mažėjant įvairiems technikos stebuklams (pvz.: mobiliesiems telefonams, kompiuteriams ir t.t.) ir augant jų galimybėms, elementai, kurie vykdo tam tikras funkcijas sumažėjo tūkstančius kartų. Tai lėmė išaugusį tikslumo poreikį, nes „milimetras į kitą pusę nei reikia“ reiškia didžiulius nuostolius.

Taip jau nutiko, kad kompiuterijos moksle naudojamos trys pagrindinės skaičiavimo sistemos: dvejetainė, dešimtainė ir šešioliktainė. Žmonės, turintys šiek tiek daugiau žinių apie kompiuterius, puikiai žino, kad galutiniame etape į kompiuterį įrašoma dvejetaine sistema užkoduota informacija, o mūsų įprasti žodžių simboliai, kurie yra koduojami šešioliktaine sistema taip pat paverčiami į dvejetainį kodą ir tik tada saugomi. Apskritai, bet kokia informacija, kuri yra įvedama į kompiuterį, verčiama į dvejetainį kodą ir tada saugoma, siunčiama ir t.t. Tam naudojamos keturios pagrindinės aritmetinės operacijos: daugyba, dalyba, sudėtis ir atimtis. Savo darbe gilinuos į vieną iš šių aritmetinių veiksmų – daugybą.

Kai dirbama vien su sveikaisiais skaičiais arba objektais, kurie yra koduojami sveikaisiais skaičiais – su problemomis nesusiduriame. Visos bėdos prasideda, kai reikia atlikti veiksmus su realiaisiais skaičiais. Vykdant veiksmus su ne ypač didelio tikslumo reikalaujančiais skaičiais (pvz.: 2-3 skaitmenys po kablelio) gaunami rezultatų netikslumai neturi mums lemiamos įtakos, tačiau jei kalbėsime apie skaičius kurių tikslumui reikia 20 skaitmenų po kablelio, tai atmestinas suapvalinimas ar klaidingas, pvz.: penkto skaičiaus po kablelio apskaičiavimas verčiant iš dešimtainės sistemos į dvejetainę sistemą, gali sukelti daugybę problemų. Apsisaugojimui nuo galimų klaidų tie patys skaičiavimai yra tikslinami. Tuomet gaunamas dvigubo tikslumo slankaus kablelio skaičius.

Kai susiduriama su ne standartinėmis priemonėmis sprendžiama situacija, jai ieškoma sprendimo būdų. Yra sukurta metodų, jau dabar pakankamai tiksliai apskaičiuojančių mums buityje reikalingus skaičius, tačiau mokslo srityje, planuojant, projektuojant ir braižant ypatingai didelius objektus (pvz.: 2 metrų spindulio apskritą anteną, kuri geriau, kokybiškiau, tiksliau perduotų signalus), ypatingai mažus objektus (pvz.: paprastą mobiliojo telefono mikroschemą su gerokai daugiau funkcijų nei sukurta anksčiau) ar įvairias sudėtingas sistemas (pvz.: kurios gebėtų tiksliau nuspėti orus, žemės drebėjimus, ugnikalnių išsiveržimus) tokio tikslumo nepakanka. Dėl minėtų priežasčių atsiranda būtinybė ieškoti naujesnių ir tikslesnių tokių skaičių apskaičiavimo būdų.

Viena pagrindinių problemų, su kuria esu susidūrus ir taip pat susidursiu rašydama šį darbą yra darbo atitikimas dabartinės lietuvių kalbos taisyklėms. Su šia problema susiduria visi tiksliųjų mokslų atstovai. Tam, kad išvengtume nesusipratimų dėl dviprasmiškų vertimų į lietuvių kalbą ir padėti suprasti kas parašyta šios srities atstovams, kai kuriuos terminus, išverstus į anglų kalbą rašysiu skliausteliuose prie lietuviško pavadinimo.

Darbo tikslai:

1. Naudodama mokslinę literatūrą pristatysiu, kas yra slankus kablelis, trumpai pateiksiu kaip jis moksliškai aprašomas ir koks jis būna.
2. Slankiojo kablelio daugybos panaudojimo sričių paieška, analizė ir apibendrinimas (atsakysiu į tokius klausimus):
    a. Kokiose srityse naudojamas slankusis kablelio daugybos skaičiavimai.
    b. Kokia slankiojo kablelio daugybos skaičiavimų įtaka galutiniam rezultatui šiose srityse (kokios pasekmės, jei blogai apskaičiuojami realieji skaičiai).
    c. Kokia slankiojo kablelio daugybos skaičiavimų įtaka man, kaip vartotojui.
3. Remdamasi moksline literatūra analizuosiu, kokiu būdu užtikrinamas dvigubas slankiojo kablelio daugybos tikslumas.
4. Esamų slankiojo kablelio daugybos apskaičiavimo metodų analizė (atsakysiu į tokius klausimus):
    a. Kokie dabar naudojami slankiojo kablelio daugybos apskaičiavimo metodai
    b. Kokie slankiojo kablelio daugybos apskaičiavimo metodai daugiausia naudojami
    c. Kokie slankiojo kablelio daugybos apskaičiavimo metodai yra patikimiausi
    d. Kokie slankiojo kablelio daugybos apskaičiavimo metodai yra sparčiausi
    e. Kokie tyrimai šioje srityje daromi šiuo metu.
5. Pagal gautus mokslinės literatūros analizės rezultatus modeliuosiu optimaliausią slankiojo kablelio daugybos apskaičiavimo algoritmą.
6. Realizuosiu sumodeliuotą slankiojo kablelio daugybos apskaičiavimo algoritmą.
7. Gautus slankiojo kablelio daugybos realizacijos rezultatus apibendrinsiu paskutinėje savo darbo dalyje.

Tyrimo metodika

Problemos sudėtingumo ir įvairiapusiškumas nustatymui ir išgryninimui bus naudojamasi visa žinoma literatūra – tiek moksline, kurioje jau aptariamos kylančios problemos, tiek ne moksline, kur gali būti užsiminta apie tam tikrų sistemų galimus netikslius apskaičiavimus. Visa informacija bus analizuojama, o analizės rezultatai atsispindės šiame magistriniame darbe.

Tam kad būtų lengviau suprasti, kokia dvigubo tikslumo slankiojo kablelio svarba, jį lyginsiu su viengubo tikslumo slankiojo kablelio stormati skaičiais. Pavyzdžiuose visur, kur įmanoma pateiksiu tiek viengubo, tiek dvigubo tikslumo slankiojo kablelio pavyzdžius. Eksperimentai bus daromi tik su dvigubo tikslumo slankiuoju kableliu.

Likusioje analitinėje darbo dalyje - bus remiamasi vien tik moksline literatūra (knygos, straipsniai iš žurnalų, mokslinių konferencijų medžiaga). Ieškosiu įvairių esamų slankiojo kablelio daugybos algoritmų analizių ir jų problemų sprendimų. Taip pat peržiūrėsiu šių sprendimų realizaciją ir komentarus, tų kurie su šiais algoritmais yra susidūrę (juos taiko savo sistemose).

Toks konstruktyvus tyrimas padės susirinkti medžiagą savo eksperimentui – slankiojo kablelio daugybos apskaičiavimo algoritmo projektavimui ir realizacijai. Darbo gale viską apibendrinsiu ir pateiksiu savo išvadas ir rekomendacijas, kaip dar būtų galima tobulinti slankiojo kablelio daugybos algoritmą.

1 Slankaus kablelio aritmetika

1.1 Apibrėžimas ir standartas

Su slankiojo kablelio, dar vadinamais realiaisiais skaičiais susiduriame kiekvieną dieną, pavyzdžiui perkame pieną už 2,79 lito. Mokslinių terminų tokiems skaičiams apibrėžti, nenaudojame, nes tokias sumas galime susiskaičiuoti ir „ant pirštų". Realizuojant didelio sudėtingumo sistemas, kur didelis būtinas tikslumas, naudojami ypač didelio tikslumo skaičiai. Taip pat slankiojo kablelio skaičiais paprasta užrašyti ilgus skaičius, kurie pavyzdžiui susideda iš 10 skaitmenų. „Tipiškas skaičius gali būti aprašytas tokia forma:

Reikšminiai skaičiai × bazė$^{\text{eksponentė}}$" [3].

Pvz.: 6,02214179 × 10^{23}

Kai kur dar galima pamatyti ir tokią formą:

6,02214179 E 1023

Slankiojo formatą apibrėžia IEEE-754 standartas. Dvigubo tikslumo slankiojo kablelio skaičiams buvo bandomas kurti atskiras IEEE-854 standartas, bet jis niekada nebuvo baigtas iki galo. Galiausiai viskas buvo apjungta į 2008 metais atnaujintą IEEE-754 slankiojo kablelio skaičių formatą [4, 5].

1.2 Tipai: kokie jie ir kuo skiriasi

Tam, kad būtų galima suvokti į ką bus gilinamasi mano darbe, reikia žinoti, kas yra aplink mano tiriamojo darbo sritį. Geriausiai tai atspindi pirma lentelė, kuri apibrėžia visus slankaus kablelio formatus ir yra pateikta kitame puslapyje.

Iš lentelės matome, kad yra 6 pagrindiniai slankiojo kablelio aritmetikos tipai: viengubo tikslumo (anglų kalba: single precision), viengubo išplėstinio tikslumo (anglų kalba: single extended precision), dvigubas tikslumas (anglų kalba: double precision), dvigubas išplėstinis tikslumas (anglų kalba: double extended precision), keturgubas tikslumas (anglų kalba: quadruple precision), keturgubas išplėstinis tikslumas (anglų kalba: quadruple extended precision). Lentelę galima nagrinėti įvairiais aspektais, priklausomai nuo poreikių. Mano darbe bus gilinamasi į dvigubo tikslumo sritį, kurią lentelėje paryškinau.

Yra keturi pagrindiniai aritmetiniai veiksmai: sudėtis, atimtis dalyba ir daugyba. Savo darbe gilinsiuos į pastarąjį veiksmą – daugybą. Tikslumui palyginti imsiu viengubo tikslumo formatą.

1 lentelė. Slankiojo kablelio formatų ir jų skirtumų santrauka [6].

Nr.	Parametras	Formatas					
		Viengubas	Viengubas išplėstinis	Dvigubas	Dvigubas išplėstinis	Keturgubas	Keturgubas išplėstinis
1.	*p* (tikslumas, aiškus mantisės plotis bitais)	24	≥ 32	53	≥ 64	113	64
2.	Dešimtainiai tikslumo skaitmenys $p\ l\ \log_2(10)$	7.22	≥ 9.63	15.95	≥ 19.26	34.01	19.26
3.	Mantisės bitas	paslėptas bitas	Neapibrėžtas	paslėptas bitas	Neapibrėžtas	paslėptas bitas	Tikslumo bitas
4.	Tikrasis mantisės plotis bitais	23	≥ 31	52	≥ 63	112	64
5.	E_{max}	+127	≥ +1023	+1023	≥ +16383	+16383	+16383
6.	E_{min}	-126	≤ - 1022	-1022	≤ -16382	-16382	-16382
7.	Eksponentės nuolydis	+127	Neapibrėžtas	+1023	Neapibrėžtas	+16383	+16383
8.	Eksponentės plotis bitais	8	≥ 11	11	≥ 15	15	15
9.	Ženklo plotis bitais	1	1	1	1	1	1
10.	Formato plotis bitais: 9+8+4	32	≥ 43	64	≥ 79	128	80
11.	Maksimalus diapazono dydis 2^{Emax+1}	3.4028E+38	≥ 1.7976E+308	1.7976E+308	≥ 1.1897E+4932	1.1897E+4932	1.1897E+4932
12.	Minimalus diapazono dydis 2^{Emin}	1.1754E-38	≤ 2.2250E-308	2.2250E-308	3.3621E-4932	3.3621E-4932	3.3621E-4932
13.	Minimalus diapazono dydis (nenormalizuotas) $2^{Emin-(4)}$	1.4012E-45	≤ 1.0361E-317	4.9406E-324	≤ 3.6451E-4951	6.4751E-4966	1.8225E-4951
14.	Fortrano kalbos tipas	REAL*4		REAL*8		REAL*16	REAL*10
15.	C kalbos tipas	float		double		long double	long double

1.3 Apskaičiavimo problemos

Kaip nebūna pigaus ir gero daikto, taip nebūna galimybės sukurti tiksliausio ir greičiausio algoritmo. Vadinasi, galimi variantai yra:

- tiksliausias, bet lėtas algoritmas
- greičiausias, bet ne ypatingai tikslus algoritmas
- greičio ir tikslumo santykis

Kiekvienu atveju mes turime tam tikrus poreikius, tad sprendžiame ką aukoti: greitį ar tikslumą. Jei būtini abu – siekiame jų santykio.

Dar viena problema, su kuria susiduriama skaičiuojant slankiojo kablelio skaičius yra apvalinimo klaidos. [7] šaltinyje rašoma: „slankiojo kablelio aritmetika nėra tiksli nuo tada, kai tam tikriems realiesiems skaičiams aprašyti reikia begalinio skaitmenų kiekio, pavyzdžiui e, π ir 1/3." Būtina numatyti galimus begalinius skaičius ir galėti koreguoti reikiamą tikslumą, nes jei tai nepadaroma – skaičiavimų ciklas tampa begalinis. Kitas apvalinimo aspektas – 1.5 turi būt apvalinamas į 1 ar į 2? Iki 0, 5 apvalinama į 0. Tai gerai ar blogai? To svarba kyla priklausomai nuo situacijos.

Sakykime, mums reikia apskaičiuoti mokesčius. Pavyzdžiui, bandelė kainuoja litą. Pridėtinės vertės mokestis – 21%. Kiek litų sumokėsime mokesčių nuo dešimt bandelių? Kompiuterio programos automatiškai skaičiuoja taip:

1*0,21=0Lt (apvalinama iki litų, nes mokesčius mokame litais)

0+0+0+0+0+0+0+0+0+0=0Lt (juk neperkama po dešimt, o tik po vieną)

Tokių situacijų, tik daug sudėtingesnių prekyboje sutinkama nuolat.

Susiduriama ir su kitokiomis apskaičiavimo klaidomis, kurios skirstomos į tris pagrindines grupes:

- Matavimo klaidos (praktinės, dėl netikslumų matuojant ir teorinių netikrumo principų).
- Diskretizavimo klaidos (jos atsiranda net vartojant tikslius skaičius).
- Statistines klaidos.

Daug problemų sukelia ir netikėtas staigus skaičiavimų nutraukimas. Tokios problemos atsiranda, kai reikia apskaičiuoti tam tikras funkcijas. Visame funkcijos skaičiavimo etape skaičiavimo tikslumas yra vienodas. Pavyzdžiui, funkcija artėja į nulį. Kai skaičiai dideli – efekto nesimato ir didelis tikslumas jiems nebūtinas (kuo daugiau skaičiavimų, tuo daugiau vargo kompiuteriams), tačiau prie pat ribinės reikšmės – nulio tikslumas turi būti gerokai didesnis, nes kai jo nėra – prarandama galutinė reikšmė.

Kilusias problemas padeda spręsti skaitinė analizė. O problemos gali būti tokios:

- Stabilumas. Stabilumas bus gerai apgalvotas, jei sprendimų, kurių reikia mažam duomenų kiekiui bus galima tik jam ir pritaikyti. Skaitinis algoritmas numato, kokios galimos algoritmo klaidos, skaitinė analizė padeda jas spręsti. Tikslumas priklauso nuo problemos sąlygų ir algoritmo stabilumo.
- Poveikis. Priklauso nuo algoritmo stabilumo. Gerai apgalvoti ir išspręstos šių teigiamų skaičių veiksmų problemos: sudėties, atimties, kėlimo laipsniu, ir dalybos. Susiduriama su sunkumais sprendžiant teigiamų skaičių atimties problemas.
- Specifinių funkcijų apskaičiavimas.

1.4 Vertimo iš dešimtainio formato į dvejetainį formatą ir atvirkščiai pavyzdžiai

Kad suprastume, kaip vyksta vertimo procesas, turėtume mokėti versti slankiojo kablelio skaičius iš dešimtainės skaičiavimo sistemos į dvejetainę ir atvirkščiai. Problema, su kuria susiduriu – tokių skaičių paprastas kompiuterinis skaičiuotuvas neskaičiuoja, tad teks pasikliauti savo kruopščiais skaičiavimais. Jau anksčiau buvau ėmusi tokią žinomą cheminę konstantą – Avogadro skaičių, tad su juo ir dabar paeksperimentuosime. Aprašau du apskaičiavimo variantus (abu jie teisingi).

1. Rankinis/paprastas/žmoniškas būdas – paprasta ir aišku.

Verčiamas skaičius: 6,02214179

Skaičiuodama atskiriu sveikąją ir realiąją skaičiaus dalis ir verčiu jas į dvejetainį formatą. Sveikosios dalies vertimas:

6 mod 2 = 0 (sekančiam etapui: 6 div 2 = 3)

3 mod 2 = 1 (sekančiam etapui: 3 div 2 = 1)

1 mod 2 = 1 (sekančiam etapui: 0 div 2 = 0)

Sveikosios dalies dvejetainis kodas: 110. Realiosios dalies vertimas:

0.02214179 * 2 = **0**.04428358

0.04428358 * 2 = **0**.08856716

0.08856716 * 2 = **0**.17713432

0.17713432 * 2 = **0**.35426864

0.35426864 * 2 = **0**.70853728

0.70853728 * 2 = **1**.41707456

0.41707456 * 2 = **0**.83414912

0.83414912 * 2 = **1**.66829824

0.66829824 * 2 = **1**.33659648

0.33659648 * 2 = **0**.67319269

0.67319269 * 2 = **1**.34638592

0.34638592 * 2 = **0**.69277184

0.69277184 * 2 = **1**.38554368

0.38554368 * 2 = **0**.77108736

0.77108736 * 2 = **1**.54217472

0.54217472 * 2 = **1**.08434944

0.08434944 * 2 = **0**.16869888

0.16869888 * 2 = **0**.33739776

0.33739776 * 2 = **0**.67479552

0.67479552 * 2 = **1**.34959104

0.34959104 * 2 = **0**.69918208

0.69918208 * 2 = **1**.39836416

0.39836416 * 2 = **0**.79672832

0.79672832 * 2 = **1**.59345664

Versdami realiąją dalį, matome, kad tai begalinis dvejetainis skaičius, o tai ką gavome atrodys štai taip (gauta iš paryškintų skaitmenų): 0.000001011010101100010101, visas skaičius atrodytų taip:

6,02214179 ≈ 110.000001011010101100010101

Bandom versti atgal, taip pat dalimis:

$110 \Rightarrow 2^2 + 2^1 = 4 + 2 = 6$

$0.000001011010101100010101 \Rightarrow 2^{-6} + 2^{-8} + 2^{-9} + 2^{-11} + 2^{-13} + 2^{-15} + 2^{-16} + 2^{-20} + 2^{-22} + 2^{-24} = 0.015625 + 0.00390625 + 0.001953125 + 0.00048828125 + 0.0001220703125 + 0.000030517578125 + 0.0000152587890625 + 0.00000095367431640625 + 0.0000002384185791015625 + 0.000000059604644775390625 = 0.022141754627227783203125$.

Atgal verstas skaičius būtų toks: 6. 022141754627227783203125 ir jis yra apytiksliai lygus pradiniam skaičiui 6,02214179. Parengta naudojant [8] medžiagą.

2. Mokslinis formatas naudojant dvigubą tikslumą

Pats vertimas atrodo taip pat, kaip aprašyta pirmame skyrelyje, tik moksliniu formatu rašant mums dar reikia apsiskaičiuoti eksponentę E. Slankaus kablelio dvigubo tikslumo formatas, kaip jau minėta pirmoje lentelėje yra 64 bitų tikslumo (aprašomas 64 dvejetainės sistemos simboliais). Skaičiaus reikšmė apibrėžiama 52 skaitmenimis, eksponentė turi 11

skaitmenų, likęs vienas skaitmuo nurodo skaičiaus ženklą (0 – teigiami skaičiai, 1 – neigiami skaičiai).

Verčiant skaičių 6,02214179 į dvigubo tikslumo kablelio skaičių gaunama tokia jo forma:

1 .1000000101101010110001010110010111111011001100011001 E 22.

E 22 reikšmėje pirmas skaitmuo reiškia dvejetainę sistemą, atras laipsnį, kuriuo keliame, tad šis skaičius gaunamas taip: verčiant prieš kablelį gavosi daugiau nei vienas skaitmuo, todėl perkeliame kablelį ir padauginame iš E laipsnio, kuris apsirašo eksponentėje

Eksponentės E plotis yra 11 bitų, kitaip sakant E_{max}=1023, todėl eksponentės verčiamo skaičiaus dvejetainė reikšmė yra 1023+2=1025. Dvejetainė eksponentės forma atrodo taip:

10000000001

Skaičiaus ženklas yra 0, nes skaičius teigimas.

1.5 Keletas istorinių faktų ir įdomybių

- Smagus internetinis konverteris, verčiantis dešimtainį slankaus kablelio skaičių į dvejetainius: http://babbage.cs.qc.edu/IEEE-754/Decimal.html
- Realios skaitinės katastrofos. Parodo realų 1.3 skyriuje aptartų problemų aktualumą (parengta pagal [7] šaltinį):
    - Ariane 5 reaktyvinis lėktuvas. Skrydis tetruko 40 sekundžių. Nuostolis – 7 milijardai dolerių. Priežastis – valdymo sistemoje verčiant 64 bitų slankiojo kablelio skaičių į 16 bitų sveikąjį skaičių su ženklu, įvyko perpildymas. Dėl neegzistuojančios problemos atsiskyrę varikliai lėmė Ariane 5 sprogimą.
    - Raketos „Patriot“ katastrofa. 1991 vasario 25 amerikiečių raketa turėjo sunaikinti „Iraqi Scud“ raketą, bet pataikė į kareivių stovyklą. Žuvo 26 žmonės. Nelaimę lėmė 1/10 sekundės netikslumas dėl naudojamo 24 bitų slankiojo kablelio formato.
    - Intel FDIV Bug Error Pentium aparatūrinėje įrangoje slankiojo kablelio dalybos grandinėje. Intel kompanijai tai kainavo 300 milijonų dolerių.
    - Nuskendo Sleipner naftos platforma. 700 milijonų dolerių kainavusi platforma nuskendo Šiaurės jūroje 1991 rugsėjį. Lėmė pakankamai neįvertintas baigtinis įtampos elementų skaičius.
    - Vankuverio vertybinių popierių birža. Dėl apvalinimo klaidos skaičiavimuose po 22 mėnesių jos indeksas buvo nuvertintas daugiau kaip 50%
- Skaičiuotuvai dažniausiai rodo 10 skaitmenų, bet skaičiuoja 13 skaitmenų tikslumu [7].

- Bankuose, pinigų cirkuliacijos skaičiavimams keliami aukšti tikslumo skaičiavimai, pavyzdžiui, euro keitimo operacijos turi būti skaičiuojamos su 6 skaitmenimis po kablelio [7].

2 Dvigubo tikslumo slankiojo kablelio daugyba

2.1 Skaičiavimus įtakojantys parametrai

Norėdami sužinoti numanomą pirmaujančių bitų reikšmę, turime turėti galvoje, kad bet koks skaičius gali būti išreikštas moksliniu žymėjimu skirtingais būdais. Pavyzdžiui, skaičius 3 gali būti išreikštas tokiais būdais:

3.00×10^0

0.03×10^2

3000×10^{-3}

Siekdami padidinti atvaizduojamų skaičių kiekį, slankaus kablelio skaičiai saugomi normalizuota forma. Tai iš esmės apibūdina šakninis kablelis po pirmo ne nulinio skaitmens. Normalizuota forma skaičius 3 pateikiamas tokia forma: 3.00×10^0.

Teigiamų slankiojo kablelio skaičių intervalas gali būti padalintas į normalizuotus skaičius (kurių tikslumui reikalingas visas mantisės dydis) ir nenormalizuoti skaičiai, kuriuos naudojame tik daliai trupmeninių skaičiavimų. Kokio dydžio šie intervalai yra viengubo tikslumo skaičiams ir dvigubo tikslumo skaičiams, galime pamatyti iš antros lentelės.

2 lentelė. Slankiojo kablelio intervalai [10].

	Nenormalizuoti	Normalizuoti	Apytiksliai dešimtainis
Viengubas tikslumas	Nuo $\pm 2^{-149}$ iki $(1-2^{-23})\times 2^{-126}$	Nuo $\pm 2^{-126}$ iki $(2-2^{-23})\times 2^{-127}$	Nuo $\pm \sim 10^{-44.85}$ iki $\sim 10^{38.53}$
Dvigubas tikslumas	Nuo $\pm 2^{-1074}$ iki $(1-2^{-52})\times 2^{-1022}$	Nuo $\pm 2^{-1022}$ iki $(2-2^{-52})\times 2^{-1023}$	Nuo $\pm \sim 10^{-323.3}$ iki $\sim 10^{308.3}$

Pateiktoje lentelėje matome tam tikras verčių ribas, kurias peržengdami pasiekiame perpildymą (tiek teigiamoms tiek neigiamoms reikšmėms), tiek reikšmių nepripildymą (tiek neigiamoms, tiek teigiamoms reikšmėms). Tai keturios iš penkių kritinių verčių, penktoji kritinė reikšmė yra nulis. Be minėto, veikiančio intervalo galime išskirti realiai veikiantį efektyvų intervalą. Jis yra apibrėžtas IEEE slankiojo kablelio skaičių standarte ir pateiktas sekančioje trečioje lentelėje.

3 lentelė. Slankiojo kablelio efektyvūs intervalai [10].

	Dvejetainis	Dešimtainis
Viengubas tikslumas	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38.53}$
Dvigubas tikslumas	$\pm (2-2^{-52}) \times 2^{1023}$	$\sim \pm 10^{308.25}$

Kaip ir kiekviena taisyklė, taip ir slankiojo kablelio skaičiai turi savo išimtis, kai gaunama ne konkreti skaitinė reikšmė, tam tikros išimtys. Tokių galimų išimčių apibendrinimas pateiktas 4 lentelėje.

4 lentelė. Slankiojo kablelio specifinės reikšmės [11].

Viengubas tikslumas		Dvigubas tikslumas		Atvaizduojamas objektas
Eksponentė	Mantisė	Eksponentė	Mantisė	
0	0	0	0	Nulis
0	Ne nulis	0	Ne nulis	± nenormalizuotas skaičius
1 - 254	Bet kas	1 - 2046	Bet kas	± normalizuotas skaičius
255	0	2047	0	± begalybė
255	Ne nulis	2047	Ne nulis	NaN (ne skaičius)

2.2 Aparatūriniai daugybos ypatumai

Dauginant du slankiojo kablelio dešimtainius didelio tikslumo skaičius kompiuteriui kyla pakankama daug problemų, nes šie skaičiai turi būti paversti į dvejetainius ir tik tuomet gali būti atlikta daugybos operacija. Nuo tikslumo poreikio priklauso ir operacijos sudėtingumas. Kai reikia ypač didelio tikslumo skaičiai turi pakankamai daug skaitmenų po kablelio, kuo daugiau skaitmenų po kablelio, tuo sudėtingiau apskaičiuoti tikslai ir greitai. To svarba ypač išryškėja kuriant sudėtingus aparatus, tam kad būtų išvengta tokių katastrofų, kaip minėtos 1.5 skyrelyje.

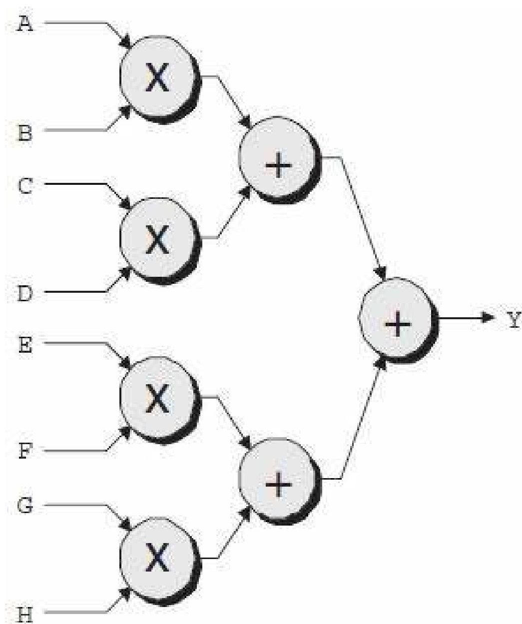
Kuriant programinę įrangą, slankiojo kablelio aritmetikos problemoms spręsti skiriamas didelis dėmesys. Kompiliatoriaus pagalba padaromi kai kurie „optimizavimai“, kurie veikia pagal gerai sukurtos programinės įrangos uždavinius. Dėl to, dešimtainiai slankiojo kablelio algoritmai yra suprojektuoti taip, kad būtų gautas laukiamas rezultatas, kai skaičiai atvaizduojami dešimtaine sistema.

Nedidelės slankiojo kablelio aritmetikos klaidos gali atsirasti vykdant tam tikrus matematinius algoritmus daug kartų. Taip gali nutikti perskaičiuojant matricą ar sprendžiant diferencialines lygtis. Tokių algoritmų projektavimui skiriama daug dėmesio. Projektuojant tokius algoritmus stengiamasi numatyti galimas klaidas ir jų taisymo būdus.

Kompiuterio aparatūrinė įranga dvejetainę sistemą supranta kaip signalo buvimą arba jo nebuvimą. Kompiuterio aparatūrinė, tinkamai suprojektuota įranga gali atlikti iki 1000 kartų greičiau tuos pačius veiksmus, kurie gali būti realizuoti ir programavimo kalbomis. Įvairūs dvejetainiai skaičiavimai atliekami skaitmeninių signalų pagalba, pasitelkiant

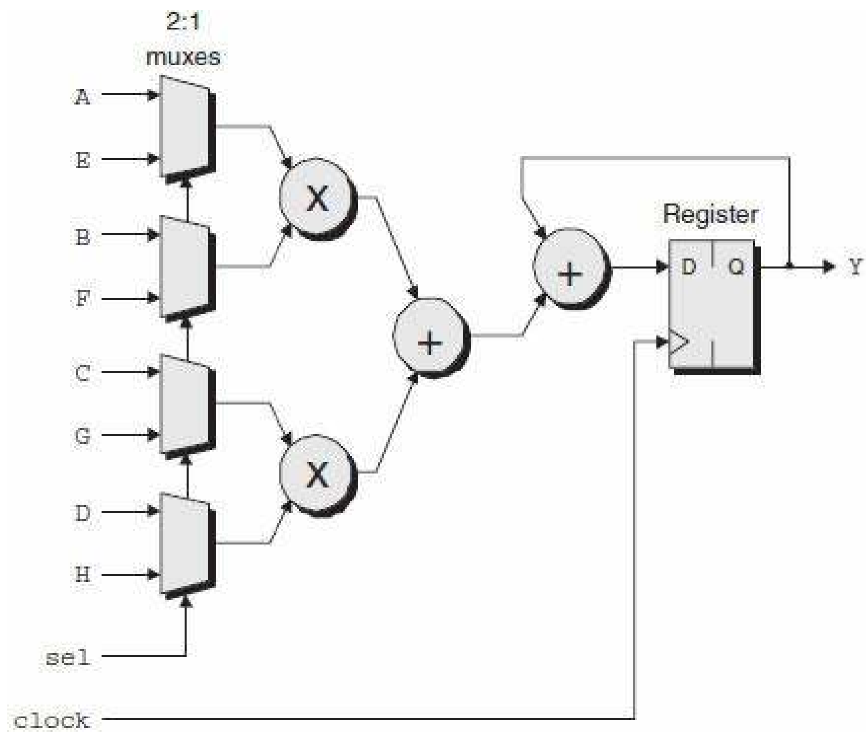
skaitmeninių signalų apdorojimo algoritmus. Toliau bus trumpai aptariami trys pagrindiniai kompiuterių aparatūrinės įrangos realizacijos tipai, kurie turi įtakos skaitmeninių signalų apdorojimo algoritmų realizacijai.

Pirmame paveiksle matome, kaip schematiškai atrodo lygiagretus daugybos funkcijos realizavimas, luste, galinčiame sparčiai įvykdyti skaičiavimus. Tokiam daugybos realizavimui reikia daug lusto realizavimo išteklių, nors jis greitas ir tikslaus atsakymo ilgai laukti nereikia.



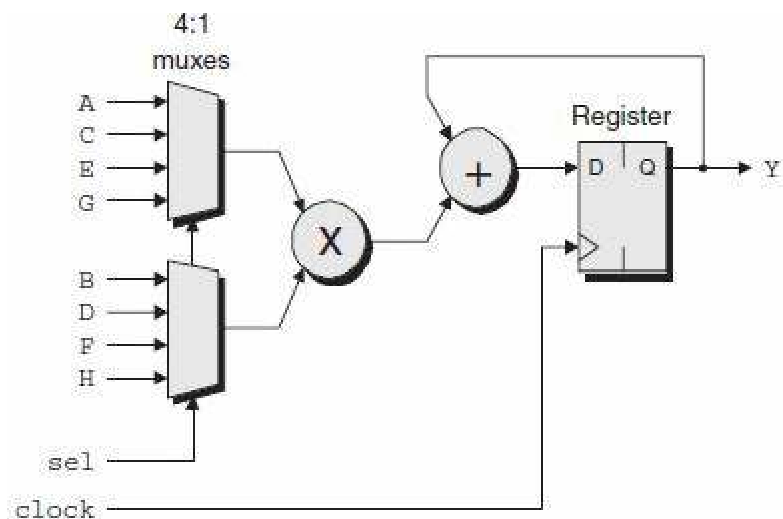
1 pav.: Lygiagretus daugybos funkcijos realizavimas (paveikslėlis paimtas iš 12 šaltinio).

Antru būdau realizuojant algoritmą – pasirenkamas sąlyginai didelis daugiklis, tačiau nuo to nukenčia algoritmo apskaičiavimo sparta, o lusto išteklių reikia sąlyginai daug. Laiko sugaištama dauginant (A*B) ir (C*D), vėliau rezultatai sudedami ir pridedami prie jau esančių duomenų registre ir tik tada saugomi. Tada imami sekantys daugikliai. Geriausias santykis tokiam algoritmui yra 2 multiplekseriai vienam registrui.



2 pav.: Funkcijos realizavimas taupant lusto išteklius ir prarandant vykdomų apskaičiavimų greitį (paveikslėlis paimtas iš 12 šaltinio).

Daugiausia laiko sugaištama naudojant trečią daugybos realizacijos variantą, nors lusto išteklių atžvilgiu tai yra ekonomiškiausias variantas. Skaitmeninių signalų apdorojimo algoritmai (kitaip tariant dvejetainiai skaičiavimai) realizuojami nuosekliai, pirmiausia sudauginant (A*B), pridedant rezultatą prie jau esančio rezultato registre ir tada vykdant sekantį pagal eilę daugybos algoritmą.



3 pav.: Nuoseklus funkcijos įgyvendinimas (paveikslėlis paimtas iš 12 šaltinio).

Kiekvienas algoritmas taikomas tam tikru atveju, pagal poreikius: jei reikia spartos – pasirenkamas lustas su dideliais ištekliais, jei neturime didelių lusto išteklių – pasirenkamas nuoseklus skaitmeninių signalų apdorojimo algoritmas.

2.3 *Programiniai daugybos apskaičiavimo algoritmai*

Kai turime tam tikrą slankaus kablelio daugybos realizavimą luste, jo savybes galime šiek tiek realizuoti algoritmų pagalba. Toliau bus nagrinėjama keletas algoritmų, kurie naudojami slankiojo kablelio daugybai realizuoti. Juos analizuoju tam, kad būtų suprastos realizacijos problemos.

Hornerio algoritmas

Hornerio metodas yra greitas, kodavimo atžvilgiu efektyvus metodas, skirtas dvejetainių skaičių daugybai ir dalybai mikrokontroleryje be aparatūrinio daugiklio. Vienas iš dvejetainių skaičių bus padaugintas kaip pavaizduotas trivialus daugianaris:

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n$$

kur a_i=1 ir x=2. Tuomet x (arba x laipsnis) yra pakartotinai apskaičiuojamas. Dvejetainėje skaičiavimo sistemoje, kurios pagrindas yra 2, x=2, todėl laipsnis 2 yra pakartotinai sudaugintas.

Pavyzdys:

Pavyzdžiui rasime dviejų skaičių (0,15625) ir m reikšmę:

$(0.15625)m=(0.00101_b)m=(2^{-3}+2^{-5})m=(2^{-3})m+(2^{-5})m=2^{-3}(m+(2^{-2})m)= 2^{-3}(m+2^{-2}(m))$.

Metodas

Tam, kad rastume dviejų skaičių, „d“ ir „m“ rezultatą, turime:

1. Saugoti tarpinius rezultatus, apibrėžtus kaip (d)
2. Pradėti vykdyti (m) su mažiausią reikšmę turinčiu ne nuliniu bitu
   a. Skaičiuojamas iš dešinės bitų pozicijos numeris iki kito reikšminio ne nulinio bito. Jei daugiau nėra reikšminių bitų, tuomet imame esamą bito poziciją.
   b. Naudojant šią vertę, vykdyti postūmio į dešinę operaciją, pagal tą bitų skaičių, kuris yra saugomas kaip tarpinė reikšmė.
3. Jei visi ne nuliniai bitai yra suskaičiuoti, tuomet tarpinis rezultatas dabar turėtų būti galutinis rezultatas. Kitu atveju sudėtis (d) yra tarpinis rezultatas ir 2 žingsnį reikėtų vykdyti iki pačio svarbiausio bito.

Išvedimas

Apskritai, dvejetainių skaičių bitų reikšmių ($d_3d_2d_1d_0$) sandauga yra:

$$(d_3 2^3 + d_2 2^2 + d_1 2^1 + d_0 2^0)m = d_3 2^3 m + d_2 2^2 m + d_1 2^1 m + d_0 2^0 m$$

Šiame algoritmo etape numatyti kaip bus susidorojama su nulinėmis vertėmis (jų galimai sukeltų amžinų ciklų sustabdymas), taip, kad vien dvejetainiai koeficientai, lygūs

vienam, būtų skaičiuojami. Todėl nekyla daugybos ir dalybos iš nulio problemos nepaisant šio poveikio daugybos lygtyje.

$$= d_0\left(m + 2\frac{d_1}{d_0}\left(m + 2\frac{d_2}{d_1}\left(m + 2\frac{d_3}{d_2}\right)\right)\right)$$

Vardikliai visi vienodi (arba jų iš viso nėra), todėl juos galima užrašyti tokia forma:

$$= d_0\left(m + 2d_1\left(m + 2d_2\left(m + 2d_3\left(m\right)\right)\right)\right)$$

minėtam rezultatui analogiškas užrašymas būtų:

$$= d_3\left(m + 2^{-1}d_2\left(m + 2^{-1}d_1\left(m + d_0\left(m\right)\right)\right)\right)$$

Dvejetainėje matematikoje (kurios bazė yra 2), daugyba laipsniu 2 yra paprasčiausia postūmio operacija. Taigi, daugyba iš dviejų dvejetainėje sistemoje yra skaičiuojama kaip aritmetinė postūmio operacija. Forma 2^{-1} yra aritmetinis postūmis į dešinę, nulinis rezultatas yra jokios operacijos nevykdymas, o2^{1} yra aritmetinis postūmis į kairę. Daugybos rezultatas gali būti greitai apskaičiuojamas vien tiktai naudojantis aritmetinėmis postūmio operacijomis, sudėtimi ir atimtimi.

Metodas yra ypač greitas procesoriuose palaikančiuose vienos instrukcijos postūmį ir sudėties kaupimą. Palyginus su C slankaus kablelio biblioteka, Hornerio metodas praranda šiek tiek tikslumo, tačiau jis formaliai yra 13 kartų greitesnis ir naudoja tik 20% programinio kodo vietos.

Dadde algoritmas

Apdorojant dalinių sandaugų matricą Dadde pasiūlė stadijų nuoseklumą sumažinant matricos aukštį. Realizuojant si algoritmą, ankstesnės matricos aukštis turi būti ne daugiau kaip 1,5 aukščio sekančios stadijos matricos aukščio. Penktoje lentelėje pateikti standartiniai dalinių sandaugų apdirbimo stadijų kiekiai esant skirtingam daugiklio skilčių skaičiui.

5. lentelė: Dalinių sandaugų apdorojimo stadijos Dadde struktūroje

Daugintuvo skilčių kiekis (N)	Rezultato gavimo stadijų kiekis (S)
2	0
3	1
4	2
5-6	3
7-9	4
10-13	5
14-19	6
20-28	7
29-42	8

43-63	9
64-94	10

Bendras Dadde struktūros skaičiavimo algoritmas yra sekantis:

1. Pradinė sąlyga $d1 = 2$ ir $dj+1 = [1,5 \cdot dj]$, kur dj – is galo j stadijos matricos aukštis. Is sąlygos $dj+1 = [1,5 \cdot dj]$ randame j stadijų skaičių, kuris reikalingas dalinių sandaugų matricos apdorojimui.
2. Is galo j stadijoje panaudojami (3,2) ir (2,2) sumatoriai tam, kad gauti supaprastintą matricą, kurios stulpelių aukštis neviršija dj.
3. Kartojame antrą punktą tol, kol gauname matricą, sudarytą is dviejų eilučių. Tokiu būdu apskaičiuojamos duoto algoritmo realizavimo aparatinės išlaidos.

Apdorojant N^2 dydžio matricą Dadde struktūroje galutinės matricos dydis yra $4 \cdot N–3$, kiekvienas (3,2) sumatorius turi tris įėjimus ir du išėjimus. Is to seka, kad kiekvienas (3,2) sumatorius sutrumpina dalinių sandaugų matricą vienu bitu, o tai reiškia, kad bendras (3,2) sumatorių kiekis Dadde struktūroje yra $N2 - 4N + 3$, galutinio rezultato gavimo sumatoriaus skilčių kiekis $2N - 2$.

(2,2) sumatorių kiekis Dadde struktūroje nustatomas sekančiai:

1. Pirmoje stadijoje (2,2) sumatoriai naudojami stulpeliuose nuo d_i iki N, kur d_i – bitų kiekis, lemiamas Dadde algoritmo veikimo, esant konkrečiai skaičiavimo stadijai.
2. i-toje skaičiavimų stadijoje (2,2) sumatoriai naudojami stulpeliuose nuo d_i iki $d_i+1 – 1$.

Išdėstyti algoritmai rodo, kad (2,2) sumatoriai naudojami nuo antrojo iki N-ojo kiekvienos stadijos stulpelio, todėl bendras (2,2) sumatorių kiekis yra N-1. 1 bitų operandų skaičiavimo Dadde algoritmu iliustracija pateikta paveiksle (dauginimo pavyzdys).

Būto algoritmas

Būto daugybos algoritmas yra toks algoritmas, kuris daugina du ženklą turinčius dvejetainius skaičius dviejuose papildomuose žymėjimuose.

Vykdymas

Būto algoritmas apima pasikartojančią sudėtį vienos iš dviejų išankstinių reikšmių A ir S, kad būtų gauta reikmė P, tuomet vykdo dešinėn pusėn nukreiptą postūmį P užrašymo formoje. Tarkime m ir r yra atitinkamai dauginamasis ir daugiklis, taip pat laikykime, kad x ir y vaizduoja bitų, esančių m ir r skaičių.

1. Apibrėžkime reikšmes A ir S ir pradinę P reikšmę. Visi šie skaičiai turi turėti ilgį, lygų (x+y+1).
    a. A: užpildykime svarbiausius (kairiausią) bitus m reikšme. Užpildykime kintančias reikšmes (y+1) nuliais.

b. S: užpildykime svarbiausius bitus –m reikšme dviem papildomais žymėjimais. Užpildykime kintančias reikšmes (y+1) nuliais
c. P: Užpildykime svarbiausius x bitus nuliais. Dešinėje šio užrašo pridėkime r vertę. Užpildykime mažiausiai svarbius bitus nuliais.
2. Nustatykite du mažiausia svarbius (dešiniausius) P bitus.
   a. Jei jie yra 01, raskite P+A reikšmę. Ignoruokite bet kokį perpildymą.
   b. Jei jie yra 10, raskite P+S reikšmę. Ignoruokite bet kokį perpildymą.
   c. Jei jie yra 00, nedarykite nieko. Tiesiog naudokite P kitame žingsnyje.
   d. Jei jie yra 11, nedarykite nieko. Tiesiog naudokite P kitame žingsnyje.
3. Aritmetiškai poslinkio reikšmė gaunama antrame žingsnyje, vienu postūmiu į dešinę.Tegul P dabar būna lygi šiai naujai reikšmei.
4. Kartokite 2 ir 3 žingsnį tol, kol jie bus įvykdyti y kartų.
5. Atmeskite mažiausiai reikšmingą (dešiniausią) bitą nuo P. Tai m ir r daugybos rezultatas

Pavyzdys

Raskime 3 × (-4), su m=3 ir r=-4, bei x=4 ir y=4

A = 0011 0000 0

S = 1101 0000 0

P = 0000 1100 0

Vykdome šį ciklą keturis kartus:

1. P = 0000 110**0 0**. Galiniai du bitai yra 00.
   P = 0000 011**0 0**. Aritmetiškai pastumiama į dešinę.
2. P = 0000 011**0 0**. Galiniai du bitai yra 00.
   P = 0000 001**1 0**. Aritmetiškai pastumiama į dešinę.
3. P = 0000 001**1 0**. Galiniai du bitai yra 10.
   P = 1101 001**1 0**. P = P + S
   P = 0110 100**1 1**. Aritmetiškai pastumiama į dešinę.
4. P = 1110 100**1 1**. Galiniai du bitai yra 11.
   P = 1111 0100 **1**. Aritmetiškai pastumiama į dešinę.

Gauname 1111 0100, kuris yra -12.

Šis būdas yra nepakankamas, kai daugiklis yra didesnis neigiamas skaičius, nei kuris galėtų būti atvaizduojamas (pavyzdžiui daugiklis turi 4 bitus ir jo reikšmė yra -8). Vienas galimas problemos sprendimo būdas yra pridėti daugiau bitų kairėje A, S ir P. Apačioje rodomas patobulintas daugybos būdas, kai -8 dauginami iš 2 naudojant 4 bitus dauginamojo ir daugiklio:

A = 1 1000 0000 0

S = 0 1000 0000 0

P = 0 0000 0010 0

Vykdom ciklą keturis kartus

1. P = 0 0000 001**0 0**. Paskutiniai du bitai yra 00.

   P = 0 0000 0001 0. Postūmis į dešinę.

2. P = 0 0000 000**1 0**. Paskutiniai du bitai yra 10.

   P = 0 1000 0001 0. P = P + S.

   P = 0 0100 0000 1. Postūmis į dešinę.

3. P = 0 0100 000**0 1**. Paskutiniai du bitai yra 01.

   P = 1 1100 0000 1. P = P + A.

   P = 1 1110 0000 0. Postūmis į dešinę.

4. P = 1 1110 000**0 0**. Paskutiniai du bitai yra 00.

   P = 1 1111 0000 0. Postūmis į dešinę.

Gautas rezultatas yra 11110000 (po pirmo ir paskutinio bito atmetimo) kuris dešimtainėje sistemoje yra −16.

Kaip jis veikia

Nagrinėjamas teigiamas daugiklis, susidedantis iš vienetų blokų, apsuptų 0. Pavyzdžiui, 00111110. Daugyba atrodys taip:

M×“00111110“=M×(2^5+2^4+2^3+2^2+2^1)=M×62

kur M yra daugiklis. Operacijų skaičius gali būti sumažintas iki 2, perrašant taip:

M×“010000-10“=M×(2^6-2^1)=M×62

Tiesą sakant tai gali būti parodyta taip, kad bet kokia eilė vienetų galima skelti į dviejų dvejetainių skaičių skirtumą:

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2.$$

Todėl mes galime pakeisti daugybą vienetų eilute pradiniuose skaičiuose paprastesnėmis operacijomis, pridedant daugiklį, perkeliant dalinį rezultatą į atitinkamas vietas ir tada galutinai atimant daugiklį. Pasinaudojama tuo faktu, kad nedaroma nieko, išskyrus postūmį, kai susiduriama su nuliu dvejetainiame daugiklyje ir yra panašus į matematinius apskaičiavimus, kai 99=100-1, kai daliname iš 99.

Ši schema gali būti išplėsta iki bet kokio vienetų blokų skaičiaus daugiklyje (įskaitant vienetų atvejį bloke). Taigi:

M×“00111010“=M×(2^5+2^4+2^3+2^1)=M×58

M×“0100-1010“=M×(2^6-2^3+2^1)=M×58

Būto algoritmas pagal šią schemą vyksta taip: sudėtis vykdoma, kai susiduriama su vienetų bloko pirmu skaitmeniu (0 1) ir atimtis vykdoma, kai susiduriama su bloki galu (1 0). Tai taip pat tinkama ir neigiamam daugikliui. Kai vienetai daugikliuose sugrupuojami į ilgus blokus, būto algoritmas vykdo mažiau sudėties ir atimties veiksmų nei įprastas daugybos algoritmas.

2.4 Techniniai algoritmų apskaičiavimo aspektai

Procesorių atliekamų operacijų įrenginiai gali būti daugiau arba mažiau universalūs: gali būti paprastesni, universalesni, tačiau norint realizuoti visus būtinus operacijų algoritmus reikia daug mikrokomandų, arba labiau sudėtingi ir specializuoti, kuriuose operacijos atliekamos žymiai greičiau ir nereikia daug valdančių mikrokodų. Universalesnius įrenginius galima vadinti procedūrinio tipo įrenginiais. Kadangi norint realizuoti kokios nors aritmetinės operacijos algoritmą, reikia nuosekliai atlikti keletą veiksmų. Specializuotuose renginiuose skaičiavimo algoritmas realizuojamas aparatiniu būdu ir jie dar vadinami stabilios struktūros įrenginiais.

Procedūriniu įrenginiu gali būti netiesioginio dauginimo įrenginys. Atlikus nedidelius pakeitimus šio tipo įrenginius galima naudoti ir kitų algoritmų realizacijai – paprasto sudėjimo su ženklu, dalybos atlikimui, operacijoms su slankiuoju kableliu.

Specializuotas aparatinis daugintuvas (matricinis) yra stabilios struktūros įrenginys, skirtas konkrečios operacijos atlikimui, esant konkrečiam skilčių skaičiui ir tam tikram kodavimui.

Norint padidinti procesoriaus darbo našumą atliekant operacijas, jis gali būti padarytas blokiniu principu. Juose yra keletas funkcionaliai nepriklausomų vykdančiųjų įrenginių, vykdančių atskiras operacijas arba operacijų grupes, pavyzdžiui, trys sveikų skaičių sudėjimo blokai, du sveikų skaičių dauginimo blokai, po vieną dalybos, sudėties ir daugybos su slankiuoju kableliu bloką it kt. Šie įrenginiai dirba lygiagrečiai. Šių įrenginių valdymas atliekamas taip vadinamų ilgų komandinių žodžių pagalba (Very Long Instruction Word - VLIW). Komandiniame žodyje yra instrukcijos kiekvienam vykdančiam renginiui, o taip pat operandai arba nurodymai juos. Blokinių įrenginių trūkumas yra tai, kad šių įrenginių panaudojimas ne visada yra efektyvus, kadangi ne visada yra galimybė kiekvieno takto metu pilnai apkrauti visus vykdančiuosius įrenginius.

Dažnai daug efektyvesni būna nuoseklūs (dar kitaip vadinami konvejeriniai) operaciniai įrenginiai, kadangi nuosekliai atlikti skaičiavimus daugeliu atvejų yra paprasčiau, nei juos paskirstyti. Tipišku konvejerinio operacinio renginio pavyzdžiu gali būti matricinis

daugintuvas. Savo pavadinimą jie gavo pirmiausia dėl to, kad turi matricą operacinių elementų (sumatorių), o taip pat, kad jie dažniausiai naudojami matricų sudauginimui.

Apžvelgsime dviejų keturių skilčių dvejetainių skaičių sandaugos procesą:

$$
\begin{array}{ccccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & + & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & + & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & + & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 &
\end{array}
$$

Naudojant netiesioginio dauginimo įrenginį, turint vieną sumatorių ir šiam dauginimui atlikti reikalingų registrų rinkinį, bendru atveju tai atliekama per 4 žingsnius. Kiekviename žingsnyje yra atliekamas a reikšmės dauginimas iš atitinkamos b reikšmes, gautos ab reikšmės sudėjimas su dalinių rezultatų suma ir naujos gautos sumos perstūmimas viena skiltimi į kairę. Tokiu būdu, šiam sandaugos veiksmui atlikti reikalingą laiką galima įvertinti taip:

$T_{\mathrm{sand}} = 4(t_{\&} + 4{*}t_{\mathrm{sm}} + t_{sh})$,

Kur: $t_{\&}$ - vėlinimas viename loginiame ventilyje atliekant a ir b dauginimą,

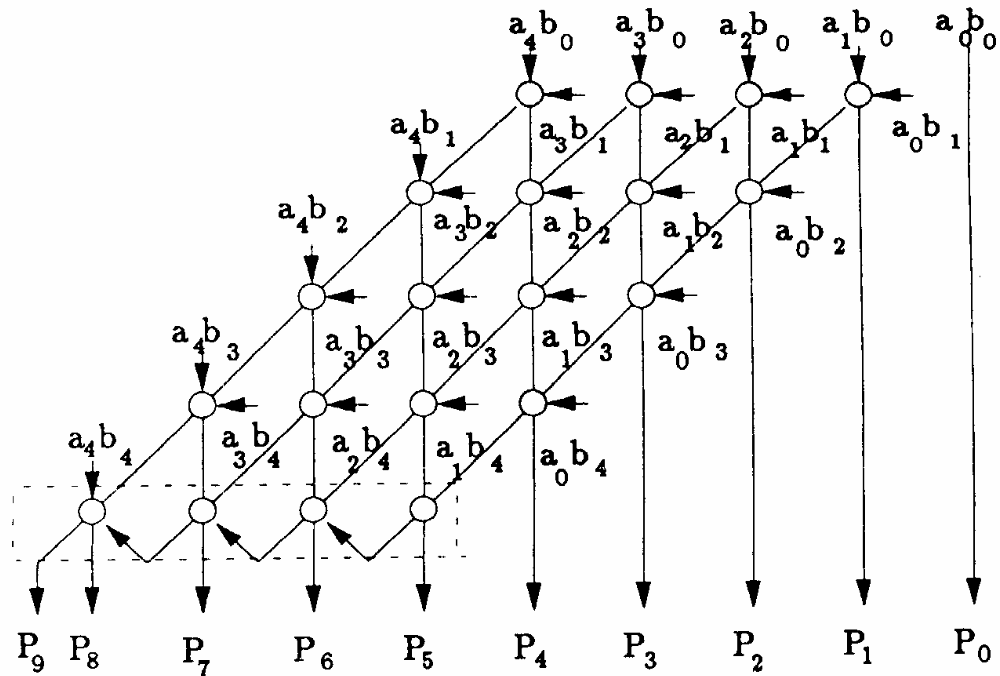
$4t_{\mathrm{sm}}$ – vėlinimas atliekant sudėti naudojant nuoseklaus pernešimo sumatorių,

t_{sm} – pilno vienos skilties dvejetainio sumatoriaus vėlinimas, kuri galima priimti lygų $2t_{\&}$,

t_{sh} – vieno poslinkio vėlinimas, kuri taip pat galima prilyginti $2t_{\&}$.

Sandaugos laikas bus lygus 44 $t_{\&}$.

Kita vertus, kadangi visas sandaugas iš principo galima atlikti lygiagrečiai, poslinkius taip pat galima apibrėžti, o pernešimus atliekant sudėti galima įvertinti tiktai atliekant baigiamąją visų keturių dedamųjų (dalinių rezultatų) sudėtį. Tokiu būdu sandaugos procesą galima gerokai pagreitinti, jeigu realizuoti matricinio dauginimo schemą, kuri parodyta žemiau esančiame paveikslėlyje.



4 pav.: Brauno daugintuvo schema. (paveikslėlis paimtas iš 13 šaltinio).

4 pav. pateiktoje schemoje neparodyti ventiliai, reikalingi dalinių rezultatų gavimui, o taip pat fiksatoriai, reikalingi kiekvienos sumatorių eilutės įėjime. Kiekviena sumatorių eilutė yra taip vadinamas sumatorius su pernešimo išsaugojimu, kuris plačiai naudojamas įvairiuose aritmetiniuose įrenginiuose. Piešinyje punktyru apvestas lygiagretus sumatorius, kuris gali būti realizuotas kaip sumatorius su nuosekliu pernešimu (parodytas piešinyje), arba pagal pagreitinto pernešimo schemą. Šio daugintuvo dauginimo laikas yra:

$$T_{\text{sand}} = t_{\&} + (\text{n} + \text{m} - 2)*t_{\text{sm}} ,$$

Kur: n – dauginamojo skilčių skaičius

m – daugiklio skilčių skaičius.

Formulėje neįvertintas pernešimų užlaikymas, kadangi jie apibrėžiami sujungiant sumatorių linijas. Mūsų atveju dauginimo laikas bus lygus 13 $t_{\&}$. Tokiu būdu šio daugintuvo sparta, lyginant su paprasta schema, daugiau kaip 3 kartus didesnė. Be to daugintuvas gali dirbti konvejerio režimu. Pikinis konvejerio našumas esant pilnai apkrovai: 1 rezultatas gaunamas per $2t_{\&}$ laiką, t.y. 20 kartų greičiau, nei paprastoje schemoje. Toks rezultatas pasiekiamas 4-5 kartus padidėjus, lyginant su paprasta schema, aparatinėms sąnaudoms.

Brauno daugintuvuose naudojami keletas pagrindinių našumo padidinimo metodų:

- skaičiavimų atlikimas lygiagrečiai (tuo pačiu metu skaičiuojami visi ab_l);
- skaičiavimų konvejerizavimas (dauginimo ciklas atliekamas paeiliui laipsniuose, tarpskiltiniai pernešimai išsaugomi ir perduodami i sekant laipsni);

- aparatinė realizacija ir skaičiavimų specializacija leidžia išvengti poslinkio, kuris apibrėžimas, sąnaudų, pernešimo išsaugojimu ir taip pat lemiamas pasirinktu tai aparatinei realizacijai algoritmu.

Kaip jau buvo minėta, pagrindiniu matricinio daugintuvo elementu yra sumatorius su pernešimo išsaugojimu (Carry Save Adder - CSA). Jis naudojamas ne tik daugintuvuose, bet visur, kur reikia pagreitinti N skaičių sudėti. Žemiau pateiktame paveikslėlyje parodytas trijų skaičių sudėjimo sumatorius, padarytas CSA pagrindu.

2.5 Rekomendacijos naujam algoritmui

Išnagrinėjus įvairią medžiagą apie slankiojo kablelio daugybos problemas, galima sukurti bendras gaires naujai kuriamam algoritmui. Dėl aparatūrinėje įrangoje ir programinėje įrangoje realizuotoje slankiojo kablelio spartos skirtumų ieškoma tokio slankiojo kablelio daugybos apskaičiavimo algoritmo varianto, kuris būtų tinkamas ne tik vykdyti tam pritaikytoje aparatūrinėje įrangoje, bet ir tam nepritaikytoje aparatūrinėje įrangoje, taip pat neprarasti spartos ir algoritmo apskaičiavimo kokybės. Greičiausiai apskaičiuojami rezultatai pagal paprasčiausius algoritmus. Tam kad būtų lengva gauti tikslų rezultatą, algoritmo realizacija turėtų vykti dalimis. Tai yra algoritmas turėtų būti vykdomas nepriklausomai realizuotais žingsniais, kurie atskirai paimti galėtų būti taikomi kitose sistemose.

Nepriklausomai nuo aparatūrinės įrangos savybių, slankiojo kablelio algoritmas turi atitikti šiuos keliamus tam tikrus reikalavimus, kurie reikalingi užtikrinti apskaičiavimų tikslumą, kokybę ir spartą:

- Patikimumas
- Sparta
- Tikslumas
- Universalumas
- Resursų poreikis

Visos kitos galimos aritmetinių veiksmų apskaičiavimo savybės nėra tokios pamatinės, kaip mano minėtosios. Kitos galimos algoritmų savybės gali būti tinkamos kaip tokių algoritmų patobulinimai, tačiau jos vis tiek neužgožtų esminių reikalavimų kodui. Toliau, kiekvieną iš jų panagrinėsiu šiek tiek plačiau.

Patikimumas. Algoritmas turėtų būti kiek įmanoma stabilesnis, tai yra „neužstrigti“ susidūręs su teigiama ar neigiama begalybe, neapibrėžtumais vienodumais ir kitais. Kaip vykdomas algoritmas su jais susidūręs turėtų elgtis yra aprašyta 2.1 skyrelyje

esančioje 4 lentelėje. Visi lentelėje minėti atvejai nenumatyti ir neapbrėžti apskaičiavimo sistemoje gali turėti negrįžtamų ir labai brangių pasekmių. Kuo daugiau ir tiksliau numatoma galimų neapibrėžtumų, klaidų ir nesklandumų, tuo patikimesnis yra algoritmas. Algoritmo patikimumas taip pat išaugai ir numatant galimų klaidų patikimumo ištaisymus.

Sparta. Esant aukštam technologiniam išsivystymui ir smarkiai išaugus tikslumo poreikiui būtina atsižvelgti į atliekamų veiksmų spartą, tam kad skaičiavimų rezultatą gautume pakankamai greitai ir jį galėtume panaudoti. Tokiais atvejais skaičiavimų metu atliekamas greitas apskaičiavimas praranda dalį tikslumo. Sparčiausi apskaičiavimai vykdomi konkrečiai tam veiksmui atlikti skirtoje aparatūroje, tačiau siaurai sričiai skirtos aparatūros paklausa pakankamai maža, o dažnai pasirenkama tokia aparatūrai, kurios pritaikymo galimybės yra pakankamai plačios.

Tikslumas. Tam tikrais atvejais spartos poreikis tampa mažiau svarbus nei tikslumo. Tokiais atvejais aukojama sparta (skaičiavimai užtrunka nuo kelių iki keliolikos kartų ilgiau), tačiau gautas rezultatas būna tikslesnis. Rezultato tikslinimas atliekamas papildomais veiksmais, tikslesniais, bet sudėtingesniais algoritmais. Tikslumas yra ypač svarbus mokslininkams, darantiems tyrimus ieškant panašumų ir dėsnių, galinčių patobulinti esamus išradimus, bandant numatyti gamtos stichijų šėlsmą ir kitais atvejais.

Universalumas. Juk norisi, kad kiekvienoje sistemoje apskaičiavimai būtų ir tikslūs ir spartūs, tad ieškomas geriausias variantas turi būti pakankamai universalus, kad susidorotų su užduotimis įvairiose sistemose. Kuriant universalią sistemą stengiamasi sukurti kiek įmanoma tikslesnes funkcijų realizacijas, prarandant kiek įmanoma mažiau savybių. Dažniausiai realizacija skaidoma į tam tikras dalis ir tos dalys realizuojamos tam tikrais etapais, o po to sujungiamos.

Resursų poreikis. Nors šiuolaikinės sistemos yra pakankamai galingos, tačiau skaičiavimų sudėtingumas yra taip pat pakankamai didelis, nes reikalavimai tikslumui yra taip pat labai aukšti. Dažniausiai kylanti problema yra ne algoritmo sukūrimas, o negalėjimas tokio algoritmo realizuoti dėl resursų, kurių nėra ar neturima poreikio.

Apibendrinant galima pasakyti, kad siekiant sukurti geriausią algoritmą ir jį realizuoti, reikia siekti, kad būtų gauti geriausi minėtų parametrų santykiai.

3 Dvigubo tikslumo slankaus kablelio daugybos realizavimas

3.1 Reikalavimai testinei programai

Kaip ir kiekviena programa taip ir testinė algoritmo realizacija turi tam tikrus reikalavimus, kuriuos turėtų atitikti. Algoritmų tyrinėjimų metu buvo sukonkretinta, kas turėtų būti testiniame programos variante:

- Skaičiaus įvedimo forma neturi būti apibrėžta. Kitaip tariant, neturėtų būti jokio skirtumo, koks yra įvestas skaičius: dvejetainis arba dešimtainis.
- Apvalinimas. Verčiant slankiojo kablelio skaičius gavus begalinį skaičių neturėtų būti problemų jo reikšmei ties tam tikru (paskutiniu) bitu nukirsti arba į jį suapvalinti po to eisiančius skaičius. Sistema (kurios pagalba atliekami skaičiavimai) visuomet skaičiuoja didesnį skaičių po kablelio kiekį nei saugo galutiniame variante, tad jei daromas apvalinimas – tai galinis skaitmuo gali skirtis naudojant apvalinimą ir jo nenaudojant.
- Jei įvestas vienas skaičius – jį kelti kvadratu.
- Išvedami (dvejetainiai skaičiai rašomi dvigubo tikslumo formatu - 64 simboliais (52 - mantisė, 11 - eksponentė, 1 - ženklas)):
    - Pradinius skaičius: dvejetainius ir dešimtainius
    - Sudaugintus skaičius (daugybos rezultatas): dvejetainis ir dešimtainis
    - Rezultato vertimas į priešingą formatą: dešimtainis ir dvejetainis
- Apsaugos:
    - Kaip eksponentė ir mantisė lygios nuliui išvesti nulį.
    - Neleisti išvesti nulio be ženklo.
    - Kai eksponentė yra nulis, o mantisė ne nulis išvesti nenormalizuotą skaičių.
    - Kai eksponentė yra tarp 1 – 2046, o mantisė – bet kas, išvedamas normalizuotas skaičius.
    - Kai eksponentė yra 2047, o mantisė – bet kas, išvedama begalybė.
    - Neleisti įvesti per didelių reikšmių.
    - Neleisti įvesti per mažų reikšmių.

3.2 Realizacijos problemos

Kuo algoritmas sudėtingesnis, tuo sudėtingesnė schema ir tuo sunkiau tokį algoritmą realizuoti. Realizacijai taip pat problemų sukelia ne tiksliai sukurtas algoritmas. Kyla ir daugiau galimai nenumatytų problemų, kurias aptariu toliau.

- Sustojimo problema – jei nenumatomas baigtinis operacijų ciklas arba situacijos, kurioms esant reikia nutraukti ciklą, kyla problemų su skaičiavimų rezultatų gavimų.
- Korektiškumas – korektiškai realizuotas programinis kodas yra aiškus kiekvienam specialistui, taip pat lengva patikrinti ar gaunami tinkami/teisingi rezultatai. Korektiškai realizuotas algoritmas taip pat bus tinkamai ištestuotas.

- Sudėtingumas – kuo sudėtingesnis algoritmas, tuo sunkiau patikrinti jo teisingumą, apskaičiuoti, kiek žingsnių turės programa įvykdyti, kad būtų gautas rezultatas. Nuo algoritmo sudėtingumo priklauso ir jo sparta, kuri mus gali tenkinti arba netenkinti (tokiu atveju reiktų peržiūrėti algoritmą).
- Resursai – galima norėti daug, galėti mažai neturint pakankamai resursų. Jei turima nepakankamai resursų, būtina algoritmą keisti taip, kad turimų resursų pakaktų. Dažnai jie įspraudžia į tam tikrus apribojimus į kuriuos turi įsitekti algoritmas.
- Efektyvumas – pagal algoritmo sudėtingumą galima spręsti apie algoritmo efektyvumą. Sudėtingų skaičiavimų labai paprastais algoritmais neįvykdysime, tačiau pati algoritmo realizacija turi būti kiek įmanoma paprastesnė.

3.3 Realizacijos rezultatai

Tinkamas ir teoriškai numatytas algoritmo vykdymas. Rezultatų lentelė pateikta kitame puslapyje.

	Testinis vektorius a	Testinis vektorius b	Laukiama reikšmė	Gauta reikšmė	Paklaida
1.	a_input=7ff0000000000000	b_input=ffffffffffffffff	expected=ffffffffffffffff	output=ffffffffffffffff	(nan)
2.	a_input=7fff000000000000	b_input=fff0000000000000	expected=7fff000000000000	output=7fff000000000000	(nan)
3.	a_input=7ff0000000000000	b_input=0000000000000000	expected=7fffffffffffffff	output=7fffffffffffffff	(nan)
4.	a_input=7ff0000000000000	b_input=3ff0000000000000	expected=7ff0000000000000	output=7ff0000000000000	(inf)
5.	a_input=3ff0000000000000	b_input=ffff000000000000	expected=ffff000000000000	output=ffff000000000000	(nan)
6.	a_input=0000000000000000	b_input=7ff0000000000000	expected=7fffffffffffffff	output=7fffffffffffffff	(nan)
7.	a_input=3ff0000000000000	b_input=7ff0000000000000	expected=7ff0000000000000	output=7ff0000000000000	(inf)
8.	a_input=0000000000000000	b_input=3ff0000000000000	expected=0000000000000000	output=0000000000000000	(0.000000)
9.	a_input=8000000000000000	b_input=3ff0000000000000	expected=8000000000000000	output=8000000000000000	(-0.000000)
10.	a_input=3ff0000000000000	b_input=0000000000000000	expected=0000000000000000	output=0000000000000000	(0.000000)
11.	a_input=3ff0000000000000	b_input=8000000000000000	expected=8000000000000000	output=8000000000000000	(-0.000000)
12.	a_input=4000000000000000	b_input=3fd0000000000000	expected=3fe0000000000000	output=3fe0000000000000	(0.500000)
13.	a_input=3fd0000000000000	b_input=4000000000000000	expected=3fe0000000000000	output=3fe0000000000000	(0.500000)
14.	a_input=c000000000000000	b_input=bfd0000000000000	expected=3fe0000000000000	output=3fe0000000000000	(0.500000)
15.	a_input=bfd0000000000000	b_input=c000000000000000	expected=3fe0000000000000	output=3fe0000000000000	(0.500000)
16.	a_input=4000000000000000	b_input=bfd0000000000000	expected=bfe0000000000000	output=bfe0000000000000	(-0.500000)
17.	a_input=bfd0000000000000	b_input=4000000000000000	expected=bfe0000000000000	output=bfe0000000000000	(-0.500000)
18.	a_input=c000000000000000	b_input=3fd0000000000000	expected=bfe0000000000000	output=bfe0000000000000	(-0.500000)
19.	a_input=3fd0000000000000	b_input=c000000000000000	expected=bfe0000000000000	output=bfe0000000000000	(-0.500000)
20.	a_input=0000000000000000	b_input=0000000000000000	expected=0000000000000000	output=0000000000000000	(0.000000)

5 lentelė. Testiniai rezultatai.

Išvados

Kai naudojamos skirtingos skaičiavimo sistemos, verčiant ne int tipo skaičius iš vienos sistemos į kitas labai sudėtinga išsaugoti tikslumą. Vienas iš tokių pavyzdžių yra skaičiaus π vertimas į dvejetainę sistemą. Yra daug situacijų, kuriose tikslumas, apvalinimas ir slankiojo kablelio skaičiavimų tikslumas gali įtakoti rezultatus, kurie gali būti gauti kitokie nei tikėtasi. Tam, kad tokių situacijų būtų išvengta reiktų laikytis tokių taisyklių:

1. Kad būtų gautas geriausias rezultatas į skaičiavimus įtraukti reikia tiek viengubą, tiek dvigubą tikslumą. Jeigu reikia dvigubo tikslumo, turi būti visos tam tikro skaičiavimo sąlygos, įskaitant konstantas,kurios yra nurodytos dvigubam tikslumui.

2. Niekada negalvoti, kad paprasta skaitinė vertė yra tiksliai sumodeliuota kompiuteryje. Dauguma slankiojo kablelio reikšmių negali būti tiksliai pavaizduotos kaip baigtinės dvejetainės vertės.

3. Niekada nemanyti, kad rezultatas yra tikslus iki paskutinio skaičiuko dešimtainėje sistemoje. Visada yra nedideli skirtumai tarp „tikro“ atsakymo ir kas gali būti apskaičiuojamas pagal slankiojo kablelio procesoriaus galutinį tikslumą.

Rimtos ir suprantamos mokslinės medžiagos lietuvių kalba nėra, padedančios perprasti mano plėtotą temą – nėra. Su šia problema susiduria kiekvienas, kuris gilina technines žinias, tad svarbu suprasti bent vienos populiarios užsienio kalbos literatūrą.

Naudota literatūra

1. http://www.google.com/
2. http://translate.google.lt/#
3. http://en.wikipedia.org/wiki/Floating_point
4. http://en.wikipedia.org/wiki/IEEE_854-1987
5. http://www.validlab.com/goldberg/paper.pdf (pdf dokumente užrašyta: 189-190psl., atvertus – 19-20psl.)
6. http://sandbox.mc.edu/~bennet/cs110/flt/dtof.html
7. http://babbage.cs.qc.edu/IEEE-754/References.xhtml
8. http://introcs.cs.princeton.edu/91float/
9. http://www.newton.dep.anl.gov/newton/askasci/1995/math/MATH065.HTM
10. http://steve.hollasch.net/cgindex/coding/ieeefloat.html
11. http://www.tfinley.net/notes/cps104/floating.html#dec2hex
12. http://mouloudrahmani.com/Electrical/Digital/FPGAAdvanced.html
13. www.novsu.ru/file/22912
14. http://en.wikipedia.org/wiki/Horner_scheme#Floating_point_multiplication_and_division
15.

Priedai

Programos kodas

Failas dfmul.c

```
============================================================================*/
#include <stdio.h>
#include "softfloat.c"

double
ullong_to_double (unsigned long long x)
{
  union
  {
    double d;
    unsigned long long ll;
  } t;

  t.ll = x;
  return t.d;
}

/*
+--------------------------------------------------------------------------+
| * Testiniai Vektoriai                                                    |
|     a_input, b_input : įvedimo duomenys                                  |
|     z_output : laukiami rezultatų duomenys                               |
+--------------------------------------------------------------------------+
*/
#define N 20
const float64 a_input[N] = {
            /* inf, nan, inf, inf, 1.0, 0.0, 1.0 */
            0x7FF0000000000000ULL, 0x7FFF000000000000ULL, 0x7FF0000000000000ULL,
0x7FF0000000000000ULL, 0x3FF0000000000000ULL, 0x0000000000000000ULL,
0x3FF0000000000000ULL,
            /* 0.0, -0.0, 1.0, 1.0, 2.0, 0.25, -2.0 */
            0x0000000000000000ULL, 0x8000000000000000ULL, 0x3FF0000000000000ULL,
0x3FF0000000000000ULL, 0x4000000000000000ULL, 0x3FD0000000000000ULL,
0xC000000000000000ULL,
            /* -0.25, 2.0, -0.25, -2.0, 0.25, 0.0 */
            0xBFD0000000000000ULL, 0x4000000000000000ULL, 0xBFD0000000000000ULL,
0xC000000000000000ULL, 0x3FD0000000000000ULL, 0x0000000000000000ULL};

const float64 b_input[N] = {
            /* nan, -inf, 0.0, 1.0, nan, inf, inf */
            0xFFFFFFFFFFFFFFFFULL, 0xFFF0000000000000ULL, 0x0000000000000000ULL,
0x3FF0000000000000ULL, 0xFFFF000000000000ULL, 0x7FF0000000000000ULL,
0x7FF0000000000000ULL,
            /* 1.0, 1.0, 0.0, -0.0, 0.25, 2.0, -0.25 */
            0x3FF0000000000000ULL, 0x3FF0000000000000ULL, 0x0000000000000000ULL,
0x8000000000000000ULL, 0x3FD0000000000000ULL, 0x4000000000000000ULL,
0xBFD0000000000000ULL,
            /* -2.0, -0.25, -2.0, 0.25, -2.0, 0.0 */
            0xC000000000000000ULL, 0xBFD0000000000000ULL, 0x4000000000000000ULL,
0x3FD0000000000000ULL, 0xC000000000000000ULL, 0x0000000000000000ULL};

const float64 z_output[N] = {
            /* nan, nan, nan, inf, nan, nan, inf */
            0xFFFFFFFFFFFFFFFFULL, 0x7FFF000000000000ULL, 0x7FFFFFFFFFFFFFFFULL,
0x7FF0000000000000ULL, 0xFFFF000000000000ULL, 0x7FFFFFFFFFFFFFFFULL,
0x7FF0000000000000ULL,
            /* 0.0, -0.0, 0.0, -0.0, 0.5, 0.5, 0.5 */
```

```
            0x0000000000000000ULL, 0x8000000000000000ULL, 0x0000000000000000ULL,
0x8000000000000000ULL, 0x3FE0000000000000ULL, 0x3FE0000000000000ULL,
0x3FE0000000000000ULL,
            /* 0.5, -0.5, -0.5, -0.5, -0.5, 0.0 */
            0x3FE0000000000000ULL, 0xBFE0000000000000ULL, 0xBFE0000000000000ULL,
0xBFE0000000000000ULL, 0xBFE0000000000000ULL, 0x0000000000000000ULL};

int
main ()
{
  int main_result;
  int i;
  float64 x1, x2;
  main_result = 0;
  for (i = 0; i < N; i++)
    {
      float64 result;
      x1 = a_input[i];
      x2 = b_input[i];
      result = float64_mul (x1, x2);
      main_result += (result != z_output[i]);

      printf("a_input=%016llx b_input=%016llx expected=%016llx output=%016llx
(%lf)\n", a_input[i], b_input[i], z_output[i], result, ullong_to_double (result));
    }
  printf ("%d\n", main_result);
  return main_result;
}
=============================================================================*/
```

Failas softfloat.c

```
=============================================================================*/
#include "milieu.h"
#include "softfloat.h"

/*----------------------------------------------------------------------------
| Floating-point rounding mode, extended double-precision rounding precision,
| and exception flags.
*----------------------------------------------------------------------------*/
int8 float_rounding_mode = float_round_nearest_even;
int8 float_exception_flags = 0;

/*----------------------------------------------------------------------------
| Primitive arithmetic functions, including multi-word arithmetic, and
| division and square root approximations.  (Can be specialized to target if
| desired.)
*----------------------------------------------------------------------------*/
#include "softfloat-macros"

/*----------------------------------------------------------------------------
| Functions and definitions to determine:  (1) whether tininess for underflow
| is detected before or after rounding by default, (2) what (if anything)
| happens when exceptions are raised, (3) how signaling NaNs are distinguished
| from quiet NaNs, (4) the default generated quiet NaNs, and (5) how NaNs
| are propagated from function inputs to output.  These details are target-
| specific.
*----------------------------------------------------------------------------*/
#include "softfloat-specialize"

/*----------------------------------------------------------------------------
| Returns the fraction bits of the double-precision floating-point value `a'.
*----------------------------------------------------------------------------*/

INLINE bits64
extractFloat64Frac (float64 a)
{

  return a & LIT64 (0x000FFFFFFFFFFFFF);
```

```
}

/*----------------------------------------------------------------------------
| Returns the exponent bits of the double-precision floating-point value `a'.
*----------------------------------------------------------------------------*/

INLINE int16
extractFloat64Exp (float64 a)
{

  return (a >> 52) & 0x7FF;

}

/*----------------------------------------------------------------------------
| Returns the sign bit of the double-precision floating-point value `a'.
*----------------------------------------------------------------------------*/

INLINE flag
extractFloat64Sign (float64 a)
{

  return a >> 63;

}

/*----------------------------------------------------------------------------
| Normalizes the subnormal double-precision floating-point value represented
| by the denormalized significand `aSig'.  The normalized exponent and
| significand are stored at the locations pointed to by `zExpPtr' and
| `zSigPtr', respectively.
*----------------------------------------------------------------------------*/

static void
normalizeFloat64Subnormal (bits64 aSig, int16 * zExpPtr, bits64 * zSigPtr)
{
  int8 shiftCount;

  shiftCount = countLeadingZeros64 (aSig) - 11;
  *zSigPtr = aSig << shiftCount;
  *zExpPtr = 1 - shiftCount;

}

/*----------------------------------------------------------------------------
| Packs the sign `zSign', exponent `zExp', and significand `zSig' into a
| double-precision floating-point value, returning the result.  After being
| shifted into the proper positions, the three fields are simply added
| together to form the result.  This means that any integer portion of `zSig'
| will be added into the exponent.  Since a properly normalized significand
| will have an integer portion equal to 1, the `zExp' input should be 1 less
| than the desired result exponent whenever `zSig' is a complete, normalized
| significand.
*----------------------------------------------------------------------------*/

INLINE float64
packFloat64 (flag zSign, int16 zExp, bits64 zSig)
{

  return (((bits64) zSign) << 63) + (((bits64) zExp) << 52) + zSig;

}

/*----------------------------------------------------------------------------
| Takes an abstract floating-point value having sign `zSign', exponent `zExp',
| and significand `zSig', and returns the proper double-precision floating-
| point value corresponding to the abstract input.  Ordinarily, the abstract
| value is simply rounded and packed into the double-precision format, with
| the inexact exception raised if the abstract input cannot be represented
| exactly.  However, if the abstract value is too large, the overflow and
| inexact exceptions are raised and an infinity or maximal finite value is
```

```
| returned.  If the abstract value is too small, the input value is rounded
| to a subnormal number, and the underflow and inexact exceptions are raised
| if the abstract input cannot be represented exactly as a subnormal double-
| precision floating-point number.
|     The input significand `zSig' has its binary point between bits 62
| and 61, which is 10 bits to the left of the usual location.  This shifted
| significand must be normalized or smaller.  If `zSig' is not normalized,
| `zExp' must be 0; in that case, the result returned is a subnormal number,
| and it must not require rounding.  In the usual case that `zSig' is
| normalized, `zExp' must be 1 less than the ``true'' floating-point exponent.
| The handling of underflow and overflow follows the IEC/IEEE Standard for
| Binary Floating-Point Arithmetic.
*----------------------------------------------------------------------------*/

static float64
roundAndPackFloat64 (flag zSign, int16 zExp, bits64 zSig)
{
  int8 roundingMode;
  flag roundNearestEven, isTiny;
  int16 roundIncrement, roundBits;

  roundingMode = float_rounding_mode;
  roundNearestEven = (roundingMode == float_round_nearest_even);
  roundIncrement = 0x200;
  if (!roundNearestEven)
    {
      if (roundingMode == float_round_to_zero)
            {
              roundIncrement = 0;
            }
      else
            {
              roundIncrement = 0x3FF;
              if (zSign)
                {
                  if (roundingMode == float_round_up)
                        roundIncrement = 0;
                }
              else
                {
                  if (roundingMode == float_round_down)
                        roundIncrement = 0;
                }
            }
    }
  roundBits = zSig & 0x3FF;
  if (0x7FD <= (bits16) zExp)
    {
      if ((0x7FD < zExp) || ((zExp == 0x7FD) && ((sbits64) (zSig + roundIncrement)
< 0)))
            {
              float_raise (float_flag_overflow | float_flag_inexact);
              return packFloat64 (zSign, 0x7FF, 0) - (roundIncrement == 0);
            }
      if (zExp < 0)
            {
              isTiny = (float_detect_tininess == float_tininess_before_rounding)
                || (zExp < -1) || (zSig + roundIncrement < LIT64
(0x8000000000000000));
              shift64RightJamming (zSig, -zExp, &zSig);
              zExp = 0;
              roundBits = zSig & 0x3FF;
              if (isTiny && roundBits)
                float_raise (float_flag_underflow);
            }
    }
  if (roundBits)
    float_exception_flags |= float_flag_inexact;
  zSig = (zSig + roundIncrement) >> 10;
  zSig &= ~(((roundBits ^ 0x200) == 0) & roundNearestEven);
  if (zSig == 0)
```

```
    zExp = 0;
  return packFloat64 (zSign, zExp, zSig);

}

/*----------------------------------------------------------------------------
| Returns the result of multiplying the double-precision floating-point values
| `a' and `b'.  The operation is performed according to the IEC/IEEE Standard
| for Binary Floating-Point Arithmetic.
*----------------------------------------------------------------------------*/

float64
float64_mul (float64 a, float64 b)
{
  flag aSign, bSign, zSign;
  int16 aExp, bExp, zExp;
  bits64 aSig, bSig, zSig0, zSig1;

  aSig = extractFloat64Frac (a);
  aExp = extractFloat64Exp (a);
  aSign = extractFloat64Sign (a);
  bSig = extractFloat64Frac (b);
  bExp = extractFloat64Exp (b);
  bSign = extractFloat64Sign (b);
  zSign = aSign ^ bSign;
  if (aExp == 0x7FF)
    {
      if (aSig || ((bExp == 0x7FF) && bSig))
	        return propagateFloat64NaN (a, b);
      if ((bExp | bSig) == 0)
	      {
	        float_raise (float_flag_invalid);
	        return float64_default_nan;
	      }
      return packFloat64 (zSign, 0x7FF, 0);
    }
  if (bExp == 0x7FF)
    {
      if (bSig)
	      return propagateFloat64NaN (a, b);
      if ((aExp | aSig) == 0)
	      {
	        float_raise (float_flag_invalid);
	        return float64_default_nan;
	      }
      return packFloat64 (zSign, 0x7FF, 0);
    }
  if (aExp == 0)
    {
      if (aSig == 0)
	      return packFloat64 (zSign, 0, 0);
      normalizeFloat64Subnormal (aSig, &aExp, &aSig);
    }
  if (bExp == 0)
    {
      if (bSig == 0)
	      return packFloat64 (zSign, 0, 0);
      normalizeFloat64Subnormal (bSig, &bExp, &bSig);
    }
  zExp = aExp + bExp - 0x3FF;
  aSig = (aSig | LIT64 (0x0010000000000000)) << 10;
  bSig = (bSig | LIT64 (0x0010000000000000)) << 11;
  mul64To128 (aSig, bSig, &zSig0, &zSig1);
  zSig0 |= (zSig1 != 0);
  if (0 <= (sbits64) (zSig0 << 1))
    {
      zSig0 <<= 1;
      --zExp;
    }
  return roundAndPackFloat64 (zSign, zExp, zSig0);
```

```
}
===============================================================================*/
```

Failas softfloat.h

```
===============================================================================*/

/*----------------------------------------------------------------------------
| Software IEC/IEEE floating-point types.
*----------------------------------------------------------------------------*/
typedef unsigned int float32;
typedef unsigned long long float64;

/*----------------------------------------------------------------------------
| Software IEC/IEEE floating-point underflow tininess-detection mode.
*----------------------------------------------------------------------------*/
#define float_tininess_after_rounding   0
#define float_tininess_before_rounding  1

/*----------------------------------------------------------------------------
| Software IEC/IEEE floating-point rounding mode.
*----------------------------------------------------------------------------*/
#define float_round_nearest_even  0
#define float_round_to_zero       1
#define float_round_up            2
#define float_round_down          3

/*----------------------------------------------------------------------------
| Software IEC/IEEE floating-point exception flags.
*----------------------------------------------------------------------------*/
#define float_flag_inexact     1
#define float_flag_divbyzero   2
#define float_flag_underflow   4
#define float_flag_overflow    8
#define float_flag_invalid    16
===============================================================================*/
```

Failas milieu.h

```
===============================================================================*/

/*----------------------------------------------------------------------------
| Include common integer types and flags.
*----------------------------------------------------------------------------*/
#include "SPARC-GCC.h"
===============================================================================*/
```

Failas SPARC-GCC.h

```
===============================================================================*/

/*----------------------------------------------------------------------------
| Each of the following `typedef's defines the most convenient type that holds
| integers of at least as many bits as specified.  For example, `uint8' should
| be the most convenient type that can hold unsigned integers of as many as
| 8 bits.  The `flag' type must be able to hold either a 0 or 1.  For most
| implementations of C, `flag', `uint8', and `int8' should all be `typedef'ed
| to the same as `int'.
*----------------------------------------------------------------------------*/
typedef int flag;
typedef int int8;
typedef int int16;

/*----------------------------------------------------------------------------
| Each of the following `typedef's defines a type that holds integers
| of _exactly_ the number of bits specified.  For instance, for most
| implementation of C, `bits16' and `sbits16' should be `typedef'ed to
| `unsigned short int' and `signed short int' (or `short int'), respectively.
```

```
*----------------------------------------------------------------------------*/
typedef unsigned short int bits16;
typedef unsigned int bits32;
typedef unsigned long long int bits64;
typedef signed long long int sbits64;

/*----------------------------------------------------------------------------
| The `LIT64' macro takes as its argument a textual integer literal and
| if necessary ``marks'' the literal as having a 64-bit integer type.
| For example, the GNU C Compiler (`gcc') requires that 64-bit literals be
| appended with the letters `LL' standing for `long long', which is `gcc's
| name for the 64-bit integer type.  Some compilers may allow `LIT64' to be
| defined as the identity macro:  `#define LIT64( a ) a'.
*----------------------------------------------------------------------------*/
#define LIT64( a ) a##LL

/*----------------------------------------------------------------------------
| The macro `INLINE' can be used before functions that should be inlined.  If
| a compiler does not support explicit inlining, this macro should be defined
| to be `static'.
*----------------------------------------------------------------------------*/
#define INLINE
=============================================================================*/
```