VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

C++ KNIHOVNA PRO PRÁCI S ČÍSLY V POHYBLIVÉ ŘÁDOVÉ ČÁRCE S LIBOVOLNOU PŘESNOSTÍ

C++ ARBITRARY PRECISION FLOATING POINT LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. VLADISLAV ZÁVADA**
AUTHOR

VEDOUCÍ PRÁCE — **prof. Ing. TOMÁŠ HRUŠKA, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Závada Vladislav, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **C++ knihovna pro práci s čísly v pohyblivé řádové čárce s libovolnou přesností**
C++ Arbitrary Precision Floating Point Library
Kategorie: Překladače

Zadání:

1. Nastudujte formát reprezentace čísel v pohyblivé řádové čárce dané standardem IEEE 754-2008. Nastudujte algoritmy provádějící standardní operace s těmito čísly. Proveďte zhodnocení.
2. Navrhněte a vytvořte generickou knihovnu pro práci s čísly v pohyblivé řádové čárce o libovolné, konfigurovatelné přesnosti v jazyce C++14.
3. Implementujte kompletní sadu operací imitující standardní typy jazyka C++ (např. float, double).
4. Vytvořte dostatečnou sadu jednotkových testů.
5. Optimalizujte knihovnu na výkon a dobu kompilace.
6. Porovnejte výkon a dobu kompilace s konkurenčními, open-source knihovnami. Navrhněte a proveďte možné optimalizace.

Literatura:

- Donald Ervin Knuth: Umění programování. 2. díl, Seminumerické algoritmy, Computer Press 2010, ISBN 978-80-251-2898-5
- 754-2008 - IEEE Standard for Floating-Point Arithmetic, Wikipedia https://en.wikipedia.org/wiki/IEEE_754-1985

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Přikryl Zdeněk, Ing., Ph.D., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Tato práce se zabývá návrhem floating point modulu, který umožní provádět operace s operandy v plovoucí řadové čárce, které mají libovolnou bitovou šířku. K tomuto účelu je modul implementován jako šablonová třída v jazyce C++. Modul je navržen tak, aby umožňoval jeho použití při návrhu aplikačně specifického procesoru. Nejprve je popsán standard floating point čísel a šablonové funkce v jazyce C++. V praktické části jsou poté popsány algoritmy jednotlivých operací a návrh samotného modulu jako šablonové knihovny. V práci je také popsáno testování knihovny pomocí jednotkových testů a srovnání konkurenčních řešení.

Abstract

This thesis deals with the design of a floating point module, which allows to perform operations with floating point operands that have any bit width. For this purpose, the module is implemented as a template class in C ++. The module is designed to allow it to be used when designing an application-specific processor. First, the floating point number and template functions in C++ are described. In the practical part the algorithms of the individual operations and the design of the module itself are described as template libraries. There is description of unitesting and other solutions comparison.

Klíčová slova

C++, šablony, IEEE754, pohyblivá řadová čárka, FPU, procesor, knihovna, jednotkové testy

Keywords

C++, templates, IEEE754, floating point, FPU, processor, library, unit tests

Citace

ZÁVADA, Vladislav. *C++ knihovna pro práci s čísly v pohyblivé řádové čárce s libovolnou přesností.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, Ph.D.

C++ knihovna pro práci s čísly v pohyblivé řádové čárce s libovolnou přesností

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Hrušky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vladislav Závada
20. května 2019

Poděkování

Tímto bych chtěl poděkovat svému školiteli prof. Ing. Tomáši Hruškovi Ph.D. za odborné vedení a cenné připomínky. Taky bych chtěl poděkovat pánům Ing. Zdeňku Přikrylovi Ph.D. a Ing. Josefu Potěšilovi z firmy Codasip za přínosné konzultace a skvělou spolupráci. Mé poděkování patří také rodině a přátelům, kteří mě podporovali.

Obsah

Kapitola 1

Úvod

Tato práce řeší návrh a implementaci modulu pro počítání s čísly v pohyblivé řádové čárce, která mohou mít libovolnou datovou velikost. Tedy bude možné zvlášť definovat velikost mantisy a exponentu. Motivací pro tuto práci je nedostatečnost rozsahu datových typů aritmetiky v pohyblivé řádové čárce které jsou definované standardem IEE 754 [4]. Procesory s klasickou architekturou jako je například ARM nebo Intel x86 a jejich FPU jednotky podporují pouze standardní datové typy pro floating point operace. Tyto datové typy nemusí být, ale pro některé operace efektivní. V moderních oborech, jako je strojové učení a umělá inteligence, není například vyžadována vysoká přesnost, ale primárně je kladen důraz na efektivitu a rychlost výpočtu velmi velkého množství operací násobení. V těchto případech by například dostačoval pro výpočty osmibitový datový typ s nízkou přesností, ale vysokou rychlostí výpočtu. Tento typ však klasické architektury nepodporují a je potřeba jej neefektivně emulovat klasickými datovými typy definovanými ve standardu.

Pro tento účel však existují aplikačně specifické procesory, které jsou navrhovány přímo k účelu k němuž mají sloužit. Mají upravenou instrukční sadu tak, aby výpočty které provádějí byly efektivní. Tyto procesory mají často upravený nebo přímo dedikovaný floating point modul.

Cílem této práce je tedy proto vytvořit návrh floating point modulu pro aplikačně specifický procesor. Tento modul bude umožňovat provádět všechny operace s datovými typy definovanými ve standardu IEE 754 a dále tyto datové typy rozšíří o uživatelem definované datové typy s libovolnou bitovou šířkou. Modul bude implementován jako knihovna v jazyce C++. Pro dosažení generické funkčnosti tak, aby bylo možné definovat velikosti datových typů, bude v knihovně použito šablonových tříd.

Vzhledem k tomu, že se modul bude využívat při návrhu hardwaru, klade se velký důraz na správnost daného modulu. Proto jsou už při implementaci knihovny tvořeny jednotkové testy, které verifikují správnost jednotlivých operací. V těchto testech se generují náhodné hodnoty operandů a výsledek se porovnává s referenčním řešením.

V tomto dokumentu je nejdříve popsán standard pro čísla v pohyblivé řádové čárce IEEE 754, jsou zde uvedeny věci na, které je potřeba při návrhu knihovny dávat pozor, jako například zaokrouhlovávání a výjimky. A jsou zde popsány všechny definované datové typy které budeme rozšiřovat.

V další kapitole jsou popsány šablonové třídy. Je to nástroj jazyka C++ pro vytváření kódu, které pracuje s různými datovými typy a proto se hodí pro implementaci modulu který pracuje s libovolnou přesností.

Dále tato práce popisuje návrh samotného modulu a jeho rozhraní a také algoritmy, které jsou použity pro jednotlivé aritmetické operace s datovými typy v plovoucí řádové čárce s libovolnou přesností.

Samostatná kapitola je věnována také jednotkovým testům, které slouží k verifikaci výsledků jednotlivých operací. K tvorbě těchto testů byl použit testovací framework GTest od společnosti Google.

V poslední kapitole jsou popsány konkurenční řešení dané problematiky a jejich výkonové srovnání s mnou navrženým modulem.

Kapitola 2

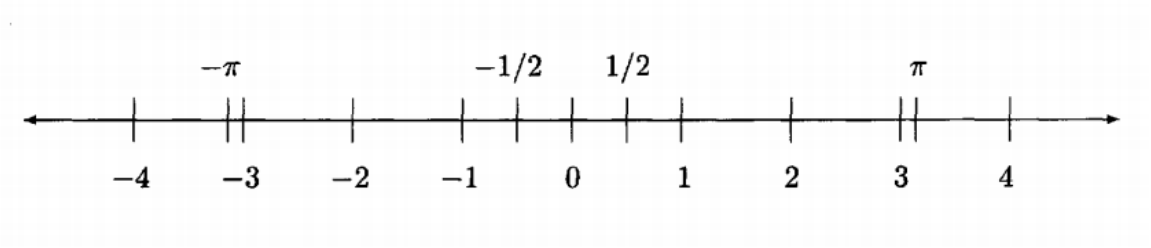
Čísla v plovoucí řádové čárce

Informace v téhle části jsou čerpány především ze standardu IEEE-754 [4].

2.1 Reprezentace čísel

Text a obrázky v této sekci byly převzaty z knihy [14].

Reálná čísla můžou být krásně reprezentována na přímce. Každý bod na téhle přímce, jež můžeme vidět na obrázku 2.1, odpovídá jednomu reálnému číslu, ale jen některé z nich jsou vyznačeny. Tato přímka se roztahuje do nekonečna na obou stranách, k nekonečnu a mínus nekonečnu, které nejsou samy tak úplně reálnými čísly ale v tomto případě je budeme zahrnovat mezi rozšířená reálná čísla. Celá čísla jsou čísla jako 0, 1 , -1, 2, -2. Říkáme, že je zde nekonečný počet spočitatelných čísel tedy celých čísel. Tímto myslíme, že když budeme počítat dostatečně dlouho a zapisovat všechny hodnoty. Každé celé číslo se jednou na tomto listu objeví, ale nikdy nemůžeme spočítat úplně všechny. Racionální čísla, jsou taková která jsou tvořena zlomkem dvou celých čísel jako například 1/2, 2/3, 4/17 atd. Některé z nich, jako například 8/4 jsou celými čísly. Pro představu toho, že jsou všechna reálná čísla spočitatelná je možné si je představit jako dvou dimenzionální pole tak, jak je naznačeno na obrázku 2.2. Sepsat řádek po řádku by nefungovalo, protože řádky jsou nekonečné. Na místo toho vygenerujeme všechna racionální čísla na diagonále: první 0, poté +-1/1; poté +-2/1, +-1/2; poté +-3/1, +-2/2, +-1/3 a tak dále. Tímto způsobem můžeme vygenerovat všechna racionální čísla (včetně všech celých čísel). Nicméně každé racionální číslo má unikání reprezentaci dosaženou pokrácení takovou, aby čitatel a jmenovatel byl co nejmenší (tedy 2/4 se pokrátí na 1/2). Iracionální čísla jsou čísla, která jsou reálná, ale nejsou racionální. Příkladem může být odmocnina ze dvou nebo číslo pí. Všechna iracionální čísla není možné sepsat a tudíž říkáme, že jsou nepočitatelná.



Obrázek 2.1: Přímka reálných čísel

	1	2	3	4	...
1	±1/1	±1/2	±1/3	±1/4	...
2	±2/1	±2/2	±2/3	±2/4	...
3	±3/1	±3/2	±3/3	±3/4	...
4	±4/1	±4/2	±4/3	±4/4	...
...	...	...	...	...	...

Obrázek 2.2: Tabulka racionálních čísel

2.2 Poziční číselné systémy

Text v této sekci byl převzat z knihy [14].

Myšlenka reprezentovat čísla za pomocí mocnin 10 byla použita mnoha starověkými národy. Hebrejci, Řeky i Číňany, ale poziční systémy které používáme dnes ne. Římané používali systém, kde každá mocnina 10 vyžadovala jiný symbol. X pro 10, C pro 100, M pro 1000 atd., a opakování těchto symbolů znamenalo, kolik z těchto mocnin 10 je přítomno. Čínský systém, který se stále používá, je podobný, ale na místo opakování, jsou symboly 1 až 9 používány k modifikování každé mocniny 10. Tyto systémy byly vhodné k počítání s počítadlem, ale již nejsou vhodné pro počítání s tužkou a papírem.

Tyto systémy neumožňovali dobře reprezentovat velmi velká čísla. Poziční notace používaná dnes celosvětově vyžaduje klíčovou myšlenku: Reprezentaci nuly jako symbolu. Podle toho co víme byla nula poprvé použita babyloňany asi 300 let před naším letopočtem. Decimální systém byl vynalezen v Indii a po století byl používán Araby a mezi lety 1200 - 1600 přešel do Evropy. Tento systém nazvaný decimální, tedy se základem 10, vyžaduje 10 symbolů. Reprezentovaných od 0 do 9. Systém je zvaný poziční, protože význam číslice je přisuzován jeho pozici v čísle. Nula je vyžadována například pro odlišení čísla 601 od čísla 61. Důvod základu 10 je pravděpodobně dán biologickým faktem, že člověk má 10 prstů na kterých tyto čísla počítal.

Existují i jiné systémy s různými základy. Například základ 60, používaný Babyloňany, který se dodnes používá k určovaní času, nebo systém se základem 20 který používaly Mayové k astronomickým kalkulacím. Mayové jsou jediný národ který, nezávisle na Babyloňanech

vynalezl nulu.

Decimální systém byl původně užíván jen pro celá čísla a nikoliv pro zlomky, až do 17. století.

Navzdory tomu, že je decimální systém vhodný pro lidi, není úplně nejvhodnější k použití v digitálních počítačích.

Binární soustava, neboli soustava, se základem 2 je mnohem vhodnější. V této soustavě je každé číslo reprezentováno jako řetězec bitů, které mohou mít hodnotu 0 nebo 1. Každý bit koresponduje s různou mocninou 2, tak jako každá číslice decimálního čísla koresponduje s různou mocninou čísla 10. Všechna záznamové média v digitálních strojích používají binární reprezentaci, základní jednotka je nazvána bit a je chápána jako entita která má dva stavy: nastaveno a nenastaveno neboli zapnuto a vypnuto. Bity jsou seskupovány do osmic zvaných Byte, které mohou mít hodnotu 0 až 255.

Zajímavostí je, že ačkoliv nebyl binární systém příliš používán před nástupem počítačů. Myšlenka používat čísla reprezentována jako součet mocnin se základem 2 je velmi stará. Byla objevena v algoritmu popisujícím násobení na Rhindském papyrusu, napsaném téměř před 4000 lety.

2.3 Reprezentace čísel v počítači

Pro práci s daty používají moderní digitální přístroje různé formáty. Těchto formátů byla v historii celá řada. Nejpoužívanější formáty zápisu čísel byly časem standardizovány tak, aby byly systémy kompatibilní a práce s nimi jednodušší.

Pro tuto práci je nejdůležitější práce s reálnými čísly a proto se zde budeme věnovat hlavně této problematice. Dnes rozlišujeme dva hlavní binární formáty zápisu reálných čísel v digitální podobě a to čísla v pevné řádové čárce a čísla v pohyblivé řádové čárce.

Pevná řadová čárka

V této reprezentaci může být reálná hodnota zapsána v binárním tvaru významově rozdělena na tři části. První částí je jednobitová reprezentace znaménka. Další částí jsou bity, které reprezentují hodnotu před desetinnou čárkou a poslední částí jsou bity, které reprezentují hodnotu za desetinnou čárkou.

Systém s pevnou řadovou čárkou je velmi limitován velikostí čísel, která je schopen na daném bitovém rozsahu uložit. Na 32-bitech lze například uložit číslo od 2^16 do 2^15. Kvůli tomu se pevná řadová čárka pro reálné výpočty příliš nepoužívá. A v této práci se věnujeme číslům v plovoucí řádové čárce.

2.4 Čísla v plovoucí řádové čárce

2.4.1 Normalizovaný tvar

Čísla v pohyblivé řádové čárce jsou reprezentována v exponenciálním tvaru. Tento tvar má ovšem více možností zápisu stejného čísla jak lze vidět například v zápise: 2.1.

$$56.123 = 56123 * 10^{-3} = 561230 * 10^{-4} = 5612300 * 10^{-5} \tag{2.1}$$

Čísla která mají stejnou hodnotu ale různou reprezentaci, se nazývají kohorty. Různé reprezentace stejných čísel by způsobovaly zbytečné problémy a tudíž je standardem IEE 754 určeno, že se vždy použije ten zástupce kohorty, který má nejmenší možný exponent, který dovoluje, aby hodnota byla zapsána přesně. Tento zástupce kohorty je nazván normalizovaný tvar čísla.

2.4.2 Standart IEEE 754-1985

Standart IEEE 754-1985 z roku 1985 byl vydán v Institute of Electrical and Electronics Engineers. Je to technický standard pro dvojkovou aritmetiku v pohyblivé řádové čárce a je to nejpoužívanější standart, který so používá například v mikroprocesorech nebo ve FPU (floating point unit - koprocesor k výpočtům v pohyblivé řádové čárce). Tento standard vyřešil problémy s rozdílnou implementací výpočtů v pohyblivé řádové čárce, pokud je tedy implementace provedena podle standardu, je zaručena jeho kompatibilita na ostatních systémech, které tento standard používají.

V této verzi byly definovány čtyři základní formáty s různou přesností. Těmito formáty jsou, single s velikostí 32 bitů, double s velikostí 64bitů a dva rozšířené typy s velikostí větší než 43 bitů a 79 bitů. Například instrukční sada pro procesory x86 implementuje dvojitou-rozšířenou přesnost s velikostí 80bitů. Pro správnou implementaci standardu je, ale vyžadována pouze základní přesnost a ostatní jsou volitelné.

Jednotlivé bity zápisu podle IEEE 754 se dělí na 3 skupiny. Těmi jsou znaménko, exponent a mantisa.
Znaménkem je vždy první bit zápisu a platí, že pro kladné čísla je tento bit 0 a pro záporné 1.
Další částí zápisu, následovanou za znaménkem, je exponent. Jehož hodnota reprezentuje exponent se základem 2, kterým se hodnota mantisy vynásobí, abychom dostali hodnotu, kterou reprezentuje číslo v plovoucí řádové čárce. Tato hodnota je definována v soustavě s posunutou nulou do poloviny maximální hodnoty exponentu. Tudíž například u osmibitového exponentu se od hodnoty musí odečíst 127, abychom dostali reálný exponent.
Mantisa reprezentuje základ, který je exponentem posouván. Její bitová šířka má přímou souvislost s přesností, kterou číslo v plovoucí řadové čárce reprezentuje. Pro ušetření datového prostoru je první bit mantisy implicitní a je nastaven na hodnotu jedna.

Počet bitů, které mantisu a exponent reprezentují je u každého formátu dán standardem. Jejich výpis shrnuje tabulka 2.2.

Obecný vztah pro výpočet hodnoty čísla je následující:

$$hodnota = (-1)^{Sign} * 2^{Exponent-emax} * (1 + Mantisa) \tag{2.2}$$

Pro základní přesnost s velikostí 32 bitů (single precision) tedy platí tento vztah:

$$hodnota = (-1)^{Sign} * 2^{Exponent-127} * (1 + Mantisa) \tag{2.3}$$

Ze vzorce je jasně patrné, že pokud je bit znaménka nulový, výsledné číslo je kladné a pokud je roven jedné číslo je záporné. Při výpočtu hodnoty se používá pro exponent kódování s posunutou nulou(aditivní kód). Od hodnoty se tedy vždy odečte konstanta rovná polovině rozsahu exponentu, nula je tedy reprezentována exponentem, který začíná nulovým bitem a pokračuje jedničkami, nižší exponent než tato konstanta je záporný.

Jak již bylo řečeno výše, standardem je určeno použití člena kohorty s nejnižším možným exponentem. Tím pádem musí být číslo zapsáno tak, aby byly první kladný bit posunut na nejlevější pozici. Standard tedy považuje tento bit za implicitní a ze zápisu ho vynechává, tím je možné zvýšit přesnost se zachováním bitové šířky. Mimo tento popsaný výpočet hodnoty, existují speciální případy hodnot, které se interpretují jiným způsobem. Tyto budou popsány v následující podkapitole

2.4.3 Speciální případy

Existují hodnoty, které se způsobem který je popsán v předchozí podkapitole nedají interpretovat. Těmito hodnotami jsou například nekonečno, nebo různé chybové hodnoty, které jsou reprezentovány jako NaN (not a number). Tyto speciální případy se označují jako výjimky.

V tabulce 2.1 jsou shrnuty některé operace způsobující výjimky a standardní výstup těchto operací tak jak je definován ve standardu IEEE 754.

Nula se znaménkem

Nula je podle IEEE 754 interpretována jako číslo které má nulový exponent a nulovou mantisu(v případě nulového exponentu se implicitní první bit mantisy nepřidává). Podle standardu existují ale dvě nuly lišící se znaménkem. Existuje tedy kladná a záporná nula. Ve většině případů se tyto hodnoty chovají stejně. Pouze některé operace jako například dělení nulou vrací s různými znaménky nuly výsledek buď plus nebo mínus nekonečno. Další funkce, které se chovají rozdílně je například signum() nebo log().

NaN

IEEE 754 specifikuje speciální hodnoty, které se nazývají NaN (not a number). Tato hodnota nejčastěji vznikne jako výsledek nevalidní operace. Těmito operacemi jsou například jakákoliv aritmetická operace, která má jako operand NaN, nebo operace které nemají matematicky definovaný výsledek ve formátu, který by se dal zapsat číslem v pohyblivé řádové čárce. Těmito čísly jsou například výsledky operací:

$$0/0 = NaN, \inf * 0 = NaN; sqrt(-1) = NaN \tag{2.4}$$

V binární podobě je NaN reprezentován hodnotou s libovolným znaménkem, kde všechny bity exponentu jsou rovny jedné a mantisa je libovolná nenulová hodnota. Podle IEEE 754 existují 2 typy hodnoty NaN. Jedním je tzv. signal NaN, který ve chvíli kdy je předán jako operand do operací způsobuje vyvolání výjimky nevalidní operace. Druhým typem je tzv. quiet (tichý) Nan, který se propaguje přes většinu operací a nezpůsobuje žádné další výjimky. V binární podobě rozlišuje quiet a signaled NaN první bit mantisy. Nulový bit značí signaling NaN, jedničkový bit značí quiet NaN. Ostatní bity mantisy nejsou standardem definovány, ale různé implementace umožňují s jejich pomocí kódovat chybu která nastala.

Nekonečno

V IEEE 754 standardu jsou definovány dvě speciální hodnoty pro kladné a záporné nekonečno. Binárně jsou reprezentovány jako exponent jenž má všechny bity rovny jedné a mantisa, jenž má všechny bity rovny nule. Bit znaménka poté určuje kladné a záporné nekonečno.

Nekonečno bývá používáno jako hodnota při přetečení nebo podtečení hodnoty přes mez, která je dána rozsahem platných hodnot pro číslo dané bitové šířky. Také je definováno jako výsledek dělení nulou.

Subnormální čísla

Subnormální čísla jsou definovaná v IEEE 754 jako velmi malá čísla blízko nule, která nedokážeme běžným způsobem na dané bitové šířce reprezentovat. Tyto čísla jsou v binární podobě reprezentována jako exponent, který má všechny bity rovny 0 a mantisa je nenulová. Znaménko funguje stejně jako při klasické reprezentaci jako nula pro kladné čísla a jedna pro záporná.

Rozdíl oproti klasické reprezentaci je takový, že implicitní bit před mantisou je roven nule a né jedné. To umožňuje, aby před prvním jedničkovým bitem mantisy předcházely nuly. Tento zápis umožní reprezentovat menší čísla než umožňuje záporný rozsah exponentu.

Nejednoznačná hodnota

Nejednoznačná hodnota není úplně výjimka, protože nastává vždy, když je výsledkem aritmetické operace hodnota, která není číslo v plovoucí řadové čárce a tudíž se musí zaokrouhlit.

Tabulka 2.1: Standardní výstup výjimek

Nevalidní operace	Nastav výstup na NaN
Dělení nulou	Nastav výstup na +-nekonečno
Přetečení	Nastav výstup na +- nekonečno nebo maximální hodnotu
Podtečení	Nastav výstup na +-0, minimální hodnotu nebo subnormální číslo
Nejednoznačná hodnota	Nastav výstup na zaokrouhlenou hodnotu

2.4.4 IEEE 754-2008

Nová verze standartu IEE 754-2008, která vznikla v roce 2008 rozšiřuje standardní typy sigle double a extended, které byly standardizovány v předchozí verzi na binary16, binary32, binary64 a binary128 jenž mají šířku bitů podle čísla uvedeného v názvu a používají binární reprezentaci stejnou jako formáty z IEEE 754-1985.

A dále definují další 3 decimální formáty decimal32, decimal64 a decimal128. Jejich bitová šířka je dána stejně jako v případě binárních formátů číslem v jejich názvu. Rozdíl v těchto formátech je, že exponent a mantisa jsou reprezentovány jako decimální hodnoty zakódované do binární podoby buď jako klasické integery nebo pomocí DPD formátu kdy jsou tři decimální číslice zakódovány do deseti bitů.

Výhodou tohoto kódování je vyhnutí se zaokrouhlovacím chybám při konverzi mezi decimálními zlomky běžnými, v lidmi zadávaných datech, a zlomky binárními.

2.4.5 Souhrn používaných formátů

Tabulka 2.2: Souhrn parametrů binárních formátů podle standartu IEEE 754-2008

	binary16	binary32	binary64	binary128
Šířka v bitech	16b	32b	64b	128b
znaménko	1b	1b	1b	1b
Šířka exponentu	5b	8b	11b	15b
Šířka mantissy	10+1b	23+1b	52+1b	112+1b
emax - maximum exponentu	15	127	1023	16383

Dále podle IEE 754-2008 existují formáty, které mají radix neboli základ deset. Desítkové formáty se příliš nepoužívají a standard připouští dvě různé implementace, které mohou být i funkčně odlišné. Liší se v kódování mantisy. Jedna implementace používá binární kódování a druhá, využívá kódování tří desítkových číslic do deseti bitů. Pro výše zmíněné důvody se v této práci se desítkové kódování dále neřeší a je zde zmíněno pouze pro úplnost formátů definovaných standardem IEE 754-2008.

Tabulka 2.3: Souhrn parametrů decimálních formátů podle standartu IEEE 754-2008

	decimal32	decimal64	decimal128
Šířka v bitech	32b	64b	128b
znaménko	1b	1b	1b
Počet číslic mantisy	7	16	34
Šířka mantissy	20	50	110
emax - maximum exponentu	96	384	6144

2.4.6 Zaokrouhlování

Zatímco zaokrouhlování decimálních čísel je pro člověka přirozené, zaokrouhlování v binárním systému může způsobovat jisté problémy. Existuje více možností jak toto zaokrouhlování provést. V této sekci bude popsáno zaokrouhlování binárních čísel, tak jak je definováno ve standardu IEEE 754-2008.

Je zde pět zaokrouhlovacích metod, které jsou popsány ve standardu IEEE 754. Většinu z nich zcela vystihuje jejich název. Dvě metody zaokrouhlují směrem k nejbližšímu. Těmito metodami jsou zaokrouhlování k sudému číslu a zaokrouhlování směrem od nuly. Ostatní metody se nazývají zaokrouhlení v určitém směru. Těmito metodami jsou zaokrouhlování směrem k nule, zaokrouhlování směrem k plus nekonečnu a zaokrouhlování k mínus nekonečnu.

Tato pravidla lze jasněji zobrazit a jednodušeji pochopit na příkladu v decimální soustavě. Tento příklad je zobrazen v následující tabulce 2.4 V tomto příkladu se zaokrouhluje na celá čísla.

Tabulka 2.4: Příklad zaokrouhlování podle jednotlivých metod IEEE 754-2008

	+42.5	+41.5	-42.5	-41.5
k nejbližšímu směrem k sudému číslu	+42.0	+42.0	-42.0	-42.0
k nejbližšímu, směrem od nuly	+43.0	+42.0	-43.0	-42.0
směrem k nule	+42.0	+41.0	-42.0	-41.0
směrem k plus nekonečnu	+43.0	+42.0	-42.0	-41.0
směrem k mínus nekonečnu	+42.0	+41.0	-43.0	-42.0

Jednotlivé algoritmy zaokrouhlování jsou popsány v kapitole: 5.11.

Zde je jen stručný popis základních zaokrouhlovacích módů. Jejich zevrubný popis je možné najít v knize [12].

Zaokrouhlování k nejbližšímu směrem k sudému číslu

V této sekci je popsána pravděpodobně nejsložitější a také nejvíce používaná metoda, která je nastavená ve většině překladačů jazyka C++ jako výchozí.
V binární podobě pracuje tak, je číslo zaokrouhleno vždy k nejbližší možné hodnotě. V případě kdy leží zaokrouhlovaná hodnota přesně mezi hodnotami na, které by se mohlo zaokrouhlit, použije se druhá část pravidla a zaokrouhlí se na to sudé. To je číslo, které v binární podobě končí nulou.

Zaokrouhlování k nejbližšímu, směrem od nuly

Stejně jako v předchozím pravidle se zaokrouhluje k nejbližšímu možnému číslu. V případě, že zaokrouhlovaná hodnota leží přesně mezi čísly, na které se dá zaokrouhlit zaokrouhlí se na to jehož absolutní hodnota je větší než zaokrouhlované číslo. Tedy pro kladné hodnoty na vyšší hodnotu a pro záporné na nižší hodnotu.

Zaokrouhlování k nule

V tomto módu na rozdíl od předchozích nezáleží, které možné výsledné hodnotě je zaokrouhlované číslo nejblíže. Vždy se zaokrouhluje směrem k nule. Tedy na hodnotu jejíž absolutní hodnota je menší než zaokrouhlované číslo.

Zaokrouhlování k plus nekonečnu

Stejně jako v předchozím módu nezáleží na blízkosti hodnoty zaokrouhlovaného čísla k možným výsledkům. Vždy se zaokrouhluje na hodnotu, která je větší než zaokrouhlované číslo.

Zaokrouhlování k mínus nekonečnu

Tento mód je stejný jako předchozí. Jediným rozdílem je směr zaokrouhlování směrem dolů. Tedy vždy na hodnotu menší než je hodnota zaokrouhlovaného čísla.

2.4.7 Zaokrouhlovací chyby

Vzhledem k formátu jaký je definován pro čísla v plovoucí řádové čárce, kdy hodnota čísla není reprezentována přesně, ale pouze v rozsahu určité přesnosti, která je dána bitovou šířkou datového typu, ve kterém je hodnota uložena, dochází při aritmetických a jiných datových operacích s těmito čísly k chybám.
Tyto chyby jsou způsobeny převážně zaokrouhlováním. Toto zaokrouhlení je nutné pro to, že pro reprezentaci nekonečného množství reálných čísel na omezený počet bitů je potřeba tyto hodnoty aproximovat. Většina operací s reálnými čísly totiž produkuje hodnoty, které nelze reprezentovat na omezeném množství bitů a proto musí být zaokrouhlen tak, aby přesně pasoval do konečné reprezentace. Tato zaokrouhlovací chyba je tedy u čísel v plovoucí řádové čárce charakteristická.

Když tedy jakákoliv operace vytváří vždy nějakou zaokrouhlovací chybu, nastává otázka, zda je nutné vůbec řešit další zaokrouhlování, které tuhle chybu pouze nepatrně zvýší? Odpověď je ano, protože kdyby tomu tak nebylo, vznikla by spousta implementací algoritmů, kde by každý produkoval jiné výsledky. Tuhle otázku řeší IEEE 754 standard definováním algoritmů zaokrouhlování tak, že výsledek operace na jakémkoliv stroji, který splňuje tento standard, je na bit stejný.

Kapitola 3

Knihovny a šablonové třídy

Součástí této práce je vytvoření knihovny pro práci s čísly v pohyblivé řádové čárce v jazyce C++ tak, aby možnost používat tuto knihovnu s datovými typy o libovolné bitové šířce. Tohoto bude docíleno nástrojem, jenž je standardní součástí jazyka C++ a tím jsou šablony. Vytvoření a použití šablon je popsáno v této kapitole.

3.1 Knihovna

Knihovny jsou důležitou součástí jazyka C++. Obsahují struktury a sady funkcí, které spolu úzce souvisí. Tyto knihovny jsou implementovány ve vlastních zdrojových souborech a je úlohou linkeru tyto knihovny připojit k programu. Narazí-li linker ve zdrojovém kódu na použití nějaké knihovny, může provést linkování dvěma různými způsoby. Může zkopírovat kód knihovny do linkované aplikace, čemuž se říká statické linkování a nebo zařídit, aby byly potřebné funkce dostupné při běhu aplikace.

Příkladem knihovny je standardní knihovna jazyka C++

3.2 Šablony - Templates

Teoretické informace obsažené v této sekci byly čerpány převážně z knihy "C++ Templates - The complete guide"[15] a praktické ukázky a tipy pro používání šablonových funkcí je také možné najít v knize "Modern C++ design: generic programming and design patterns"[5], která již je poněkud staršího data, ale praktické rady a tipy jsou stále relevantní.

Jazyk C++ vyžaduje, abychom deklarovali proměnné, funkce a většinu jiných druhů entit použitím specifického datového typu. Na druhou stranu, velké množství kódu je stejné pro různé datové typy. Zvláště když se implementují algoritmy jako například quicksort nebo různé operace jako přiřazení nebo kopírování.
Pokud by programovací jazyk nepodporoval speciální prostředky jak tohle vyřešit, zbývaly by nám pouze špatné alternativy. Na příklad implementovat to samé chování vždy znovu a tím by vzniklo spoustu duplikátního kódu. Nebo bychom například mohli použít preprocesor a makra.

Templates, neboli šablony, jsou nástroj jazyka C++, který umožňuje psát generické programy a jsou přesně tím řešením, které řeší problém typů bez zpátečnických postupů.

Jsou to funkce nebo třídy, které jsou napsány pro více datových typů, které nejsou specifikovány. Ve chvíli kdy je šablona použita, je předán datový typ jako argument, buď implicitně nebo explicitně. A tím že jsou šablonové třídy součástí jazyka C++, máme přímou podporu ověřování datových typů a datového rozsahu.

V dnešních programech, jsou šablony velmi často používanou funkcí. Například ve standardní knihovně jazyka C++ je téměř veškerý kód implementován za pomocí šablon. Knihovna tak může poskytovat sadu algoritmů a objektů pro různé datové typy.

3.2.1 Šablony funkcí

Šablona funkce funguje podobně jako klasická funkce, má také vstupní parametry a vrací výstup. Jediným rozdílem je, že šablonová funkce dokáže pracovat s různými datovými typy.

Pokud tedy potřebujeme stejnou funkci provádět nad několika různými datovými typy použijeme šablonu. Překladač pak ve chvíli, kdy je argument daného datového typu předán funkci, vygeneruje novou verzi funkce s odpovídajícími datovými typy.

Ukázka definice šablonové funkce: 3.1. V ukázce je uveden argument šablony T. Tento argument akceptuje různé datové typy (např. int, double, float). Argumenty se zapisují ihned za klíčové slovo **template** do lomených závorek. Jednotlivé argumenty jsou odděleny čárkou. Klíčové slovo **typename** představuje typ argumentu. Je to nejpoužívanější datový typ pro šablony. Parametr typu je tedy v této ukázce T a reprezentuje libovolný typ, který je specifikován při volání funkce. Může být použit jakýkoliv typ, který podporuje operace, které jsou uvnitř šablonové třídy a které pracují s tímto typem.
Z historických důvodů lze kvůli zpětné kompatibilitě místo klíčového slova typename použít klíčové slovo class. Tato záměna nemá žádný vliv na funkci.

```
template <typename T>
void function(T parameter)
{
T value = parameter;
}
```

Výpis 3.1: Šablona funkce

3.2.2 Šablony tříd

Podobně jako šablony funkcí i šablony tříd slouží k vytvoření generických tříd, které mohou mít operace a proměnné s různými datovými typy. Pokud je potřeba mít třídy, které mají stejnou funkčnost a pouze pracují s různými datovými typy. Bez použití šablon by bylo potřeba vytvořit pro každý datový typ samostatnou třídu. Šablona třídy ale umožní třídu implementovat pouze jednou a překladač podobně jako u šablony třídy vygeneruje při předání daných datových typů do šablony kopii třídy s danými datovými typy.

```
template <class T>
class className
{
        public:
        T variable;
        T operation(T parameter)
        {
                ...
        }
}
```

Výpis 3.2: Šablona třídy

Zde 3.2 je uveden příklad deklarace šablony třídy. V této deklaraci je uveden argument šablony T, který zastupuje datový typ, který bude použit. Uvnitř těla třídy se nachází třídní proměnná variable s tímto datovým typem a také metoda operation() také s typem T, která přijmá jeden parametr také s datovým typem T. Místo tohoto zástupného typu může být použit libovolný datový typ (např. int, float) .

Pro vytvoření šablony třídy je nutné definovat datový typ uvnitř lomených závorek tak jak je uvedeno v příkladu 3.3. Datový typ určuje překladači s jakými datovými typy má být třída vytvořena. V tomto případě je použit typ float. Překladač pak při překladu šablony 3.2 vytvoří kopii, která funkčností odpovídá kódu uvedenému zde 3.4. Jak lze vidět všechny výskyty zástupného datového typu T byly nahrazeny typem float.

```
className<float> classObject;
```

Výpis 3.3: Instanciace objektu šablony

```
class className
{
public:
float variable;
float operation(float parameter)
{
...
}
}
```

Výpis 3.4: Třída vygenerovaná z šablony

Kapitola 4

Návrh knihovny pro práci s čísly v pohyblivé řádové čárce

Tato kapitola je věnována popisu návrhu a implementace vytvořeného modulu a je zde uveden popis algoritmů, které tento modul využívá. Základ algoritmů byl čerpán z knihy Umění programování 2.díl Seminumerické algoritmy [9]. Tyto algoritmy však byly pro možnost počítat s různými bitovými šířkami upraveny.

4.1 Šablonová třída

Aby mohlo být dosaženo možnosti pracovat s datovými typy s různou bitovou šířkou, bylo nutné navrhnout způsob, jakým je možné vytvořit generický kód, který by s těmito datovými typy pracoval. Po úvaze bylo zvoleno použití nástroje v jazyce C++ jímž jsou šablonové třídy, jež byly popsány v předchozí kapitole.

První fází návrhu třídy bylo zvolit parametry šablonové funkce. V ukázce kódu 4.1 je patrné, že byly zvoleny dvě hodnoty typu unsigned int, které udávají šířku exponentu v bitech a šířku mantisy v bitech, tyto hodnoty mají názvy EXPONENT_WIDTH a MANTISSA_WIDTH. Bitová šířka výsledného datového typu je pak součet těchto parametrů plus jeden bit pro hodnotu znaménka, které je ve všech datových typech.

```
template <unsigned EXPONENT_WIDTH, unsigned MANTISSA_WIDTH>
class FloatNumber
{
        ...
}
```

Výpis 4.1: Hlavička definice šablonové třídy pro floating point číslo libovolné velikosti

Součástí šablonové třídy jsou data, která jsou uložena ve třech částech s různými datovými typy. Jak je vidět v ukázce 4.2 První částí je znaménko, které je uloženo jako hodnota Uint s parametrem SIGN_WIDTH, který má hodnotu jedna. Toto řešení je zatím dočasné, protože v tomto případě zabírá znaménko 8 bitů a je omezeno použitím knihovny pro výpočty v pevné řadové čárce s libovolnou přesností, kterou modul využívá. Další částí je exponent. Ten je uložen v proměnné m_Exponent, jako bez-znaménkové

číslo v pevné řádové čárce s datovým typem Uint a velikostí danou parametrem šablonové funkce EXPONENT_WIDTH. Obdobně je uložena část, která představuje mantisu v proměnné m_Mantissa pouze s tím rozdílem, že velikost této proměnné určuje parametr MANTISSA_WIDTH.
Uint je typ v pevné řádové čárce bez znaménka, který je implementován podobně jako tato knihovna pomocí šablonové třídy. Umožňuje vytvořit datový typ o libovolné šířce definované parametrem této šablonové třídy. Tímto parametrem je unsigned int, který udává datovou šířku v bitech. Datový typ pro pevnou řadovou čárku je implementován ve vlastním modulu jako privátní řešení společnosti, pro kterou vyvíjím svůj modul pro práci s čísly v pohyblivé řádové čárce a bude součástí vyvíjeného systému. Tento modul je popsán v následující sekci.

```
class FloatNumber
{
        ...

        private:
                Uint<SIGN_WIDTH> m_Sign;
                Uint<EXPONENT_WIDTH> m_Exponent;
                Uint<MANTISSA_WIDTH> m_Mantissa;

        ...
}
```

Výpis 4.2: Data uložená ve třídě FloatNumber

4.2 Knihovna pro práci s čísly v pevné řádové čárce s libovolnou přesností

K tomu, aby navržená knihovna mohla správně pracovat s čísly v plovoucí řadové čárce s libovolnou přesností, je potřeba ukládat také mantisu a exponent uvnitř této knihovny s libovolnou přesností.

Vyhledem k tomu, že byl tento nástroj vyvíjen ve spolupráci se společností Codasip, byl k tomuto účelu použit jejich modul pro práci s integery v libovolné bitové šířce. V tomto modulu je implementována většina potřebných operací, operujících s těmito čísly.

Tento modul je implementován stejně jako floating point modul pomocí šablonových tříd. Název hlavní třídy je IntegerNumber, která má dva parametry. Prvním je bitová šířka čísla nazvaná _CODASIP_BITS, která je definována jeko unsigned typ. Druhým parametrem je boolovská hodnota _CODASIP_SIGN, která určuje zda první bit reprezentuje znaménko či nikoliv a číslo je tedy definováno jako unsigned (beznaménkové). Definici šablonové třídy CodasipInt lze vidět v ukázce 4.3.

Velkou výhodou této třídy je, že používá různé implementace pro různé bitové šířky a že například pro 32,64,80 bitové čísla jsou použity standardní typy jazyka C++, které jsou optimalizované.

```
/**
* \addtogroup CODASIP_INT
* \{
* \class codasip::integer::IntegerNumber
* \brief Simple and lightweight wrapper around integer backend. Implements
* full range of unary and binary operators as well as cooperation with
* standard integer types.
* \}
*/
template<unsigned _CODASIP_BITS, bool _CODASIP_SIGN>
class IntegerNumber
{
...
}
```

Výpis 4.3: Hlavička definice šablonové třídy pro floating point číslo libovolné velikosti

Druhou variantou, se kterou byla knihovna testována je za použití multiprecision integer knihovny, která je součástí systému boost [11].

4.3 Použití knihovny v systému firmy Codasip

Jak již bylo řečeno v úvodu, má práce se zabývá tvorbou floating pooint modulu pro systém firmy Codasip, která vyvíjí aplikačně specifické procesory s architekturou RISC-V. Tato firma má řešení, které umožňuje převést zdrojový kód implementovaný v jazyce C++ ve formě šablonových tříd na speciální jazyk Codal, což je jazyk určený k modelování procesorů.

Detaily transformace zdrojových kódů na návrh aplikačně specifických procesorů přesahuje rozsah této práce a jsou know-how a firemním tajemstvím firmy Codasip. Z důvodů zachování tohoto tajemství, zde tedy nemůže být uveden.

4.3.1 Aplikačně specifické procesory - ASIP

Informace v této sekci byly čerpány z knihy [10].

Aplikačně specifické procesory jsou procesory s instrukční sadou, která je vytvořena takříkajíc na míru účelu, pro který je daný procesor vyroben. Tyto procesory jsou vyrobeny tak, aby sloužily pro malou množinu specifických úloh a pro tuto činnost jsou specializovány a optimalizovány. Instrukční sada je navržena tak, aby akcelerovala nejnáročnější a nejvíce používané funkce.

Při návrhu aplikačně specifického procesoru je velmi dbáno na cenu použitého hardwaru a velmi důležitá je také výsledná spotřeba elektrické energie při jejímž růstu se zároveň zvyšuje odpadní teplo a vznikají vyšší požadavky na chlazení. Požadavek na nízkou spotřebu je zvláště důležitý, pokud je zařízení napájeno akumulátorem neboť je tím značně ovlivněna celková výdrž zařízení.

Kapitola 5

Návrh a implementace algoritmů aritmetických operací

V této kapitole je popsána implementace operací v plovoucí řádové čárce s libovolnou přesností. Také jsou zde popsány algoritmy řešící tyto operace a jsou zde vyzdviženy zajímavé části těchto algoritmů, které bylo potřeba vyřešit.

Inspirace algoritmů byla čerpána z knih [12] a [7].

5.1 Sčítání

Nyní následuje zevrubný postup navrženého algoritmu pro sčítání dvou čísel s libovolnou bitovou šířkou exponentů a mantis.

5.1.1 Speciání hodnoty operandů

Nejprve je nutné vyřešit možné speciální hodnoty operandů, které vstupují do operace a které se vymykají navrženému algoritmu sčítání v plovoucí řádové čárce. Těmito hodnotami jsou například nula, nekonečno a NaN. Tyto speciální hodnoty jsou definované ve standardu IEEE 754 a jejich stručný popis je zde: 2.4.3. Následující pravidla se vyhodnocují v pořadí tak jak jsou zde sepsána od prvního k poslednímu. Tedy pokud je například hodnota jednoho operandu NaN, toto pravidlo má větší prioritu, než ta co za ním nasledují. Algoritmus sčítání se provede až poté co je ověřeno, že žádná z hodnot operandů není speciální hodnota.

NaN

Pokud je jedním z operandů NaN, výsledkem je hodnota NaN. Pokud mají oba operandy hodnotu NaN, výsledkem je také hodnota NaN.

Nekonečno

Pokud má jeden z operandů hodnotu nekonečno, výsledkem je tento operand. Pokud mají oba operandy hodnotu nekonečno, výsledkem je hodnota nekonečno se znaménkem prvního operandu.

Nula

Pokud má jeden z operandů hodnotu nula, výsledkem sčítání je vždy hodnota druhého operandu. Pokud mají oba operandy hodnotu nula. Výsledkem sčítání je hodnota nula.

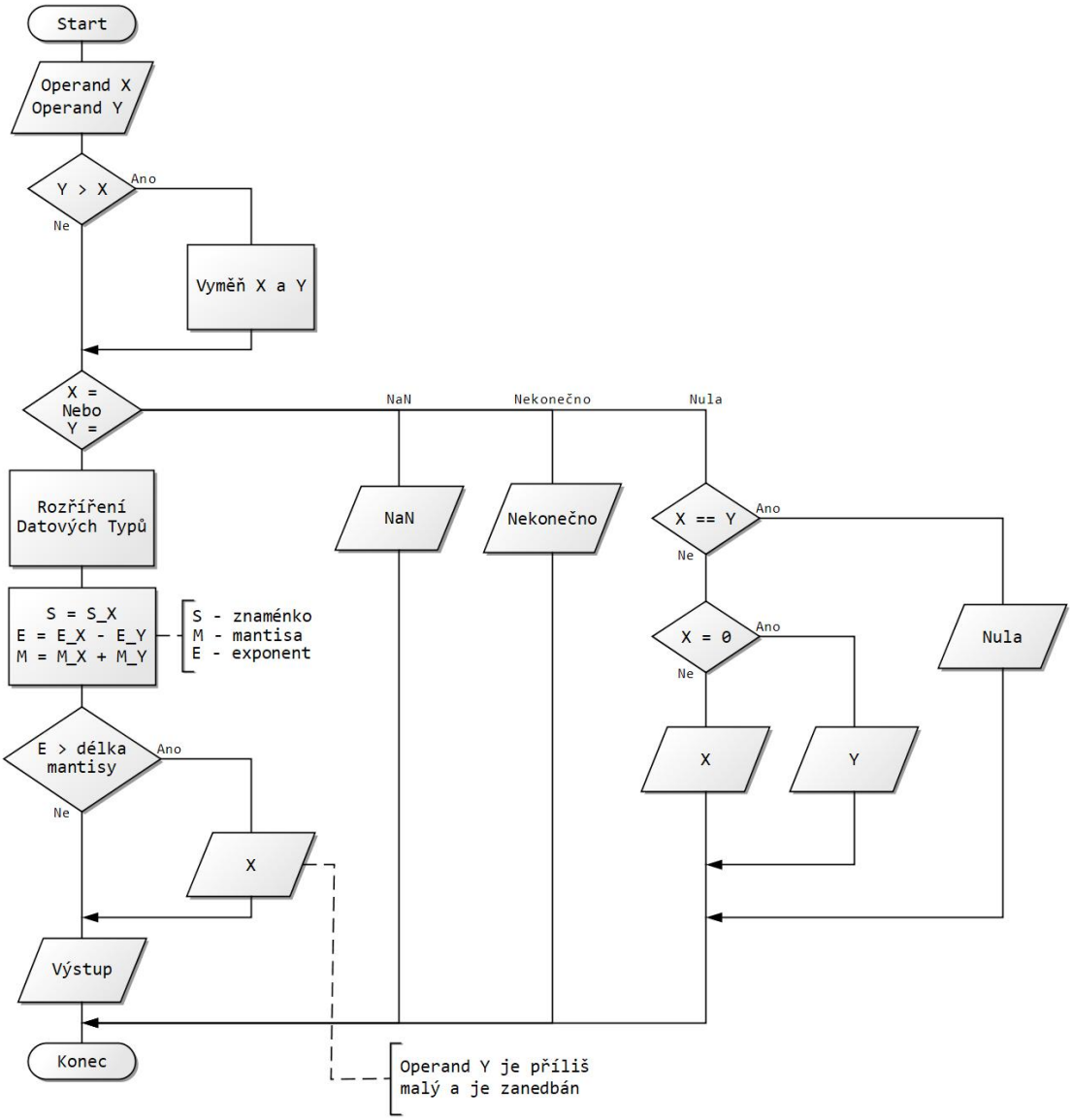
5.1.2 Algoritmus sčítání

Před samotným algoritmem sčítání se provede porovnání znamének operandů. Pokud se rovnají, provede se sčítání, pokud se nerovnají, provede se algoritmus odčítání.

Algoritmus sčítání je následující: Nejdříve pojmenujeme operandy, první bude x a druhý y a k nim příslušné exponenty, mantisy a znaménka. Poté se porovnají exponenty obou operandů, pokud je exponent x menší než exponent y, obrátí se pořadí operandů tak, aby operand x měl vždy větší nebo rovný exponent než y. Provede se rozdíl exponentů a jeho hodnota se uloží. Nastávají dvě možné situace.

a.) Pokud je rozdíl exponentů menší, obě mantisy se rozšíří na dvojnásobnou velikost větší mantisy a nad mantisou x se provede bitový posun doprava o tolik bitů kolik je rozdíl mezi exponenty. Poté se provede bitový součet obou rozšířených mantis. Výsledek tohoto součtu se zaokrouhlí na velikost mantisy a zbytek bitů se ořízne. Hodnota výsledného exponentu je rovna exponentu y.Znaménko je rovno znaménku operandu x, protože je vždy větší a znaménko není ovlivněno. Výsledek tohoto algoritmu je potřeba ještě tzv. normalizovat. To znamená posunout mantisu doleva tak, aby první jedničkový bit byl na implicitní pozici před mantisou a o velikost tohoto posunu je potřeba inkrementovat výsledný exponent. Tento výsledek už je konečným výsledkem operace sčítání.

b.) Pokud je tento rozdíl větší než velikost té mantisy, která je větší z obou operandů, výsledkem je hodnota operandu x, protože dojde k podtečení rozsahu hodnot y a tento operand se tedy ve výsledku vlivem zaokrouhlení a ořezání neprojeví.



Obrázek 5.1: Vývojový diagram operace sčítání

5.2 Odečítání

Algoritmus odečítání je implementován ve funkci s následující deklarací. 5.1 Tato funkce pracuje se dvěma hodnotami. První je uložena v objektu a druhou hodnotu přijímá funkce jako parametr. Při popisu algoritmu jsou označovány jako x a y. Jako výstup vrací rozdíl hodnot x - y.

```
/**
* \brief substract this with param value
* \param y - second operand of substraction
* \return result of substraction this - y param value
*/
template <unsigned _CODASIP_EXPONENT, unsigned _CODASIP_MANTISSA>
reference_type Substraction(value_type y)
```

Výpis 5.1: Deklarace funkce ověřující rovnost

Nejdříve je nutné ověřit speciální hodnoty.

NaN

Pokud je jedním z operandů NaN, výsledkem je hodnota NaN. Pokud mají oba operandy hodnotu NaN, výsledkem je také hodnota NaN.

Nekonečno

Pokud má jeden z operandů hodnotu nekonečno, výsledkem je tento operand. Pokud mají oba operandy hodnotu nekonečno, výsledkem je hodnota nekonečno se znaménkem prvního operandu.

Nula

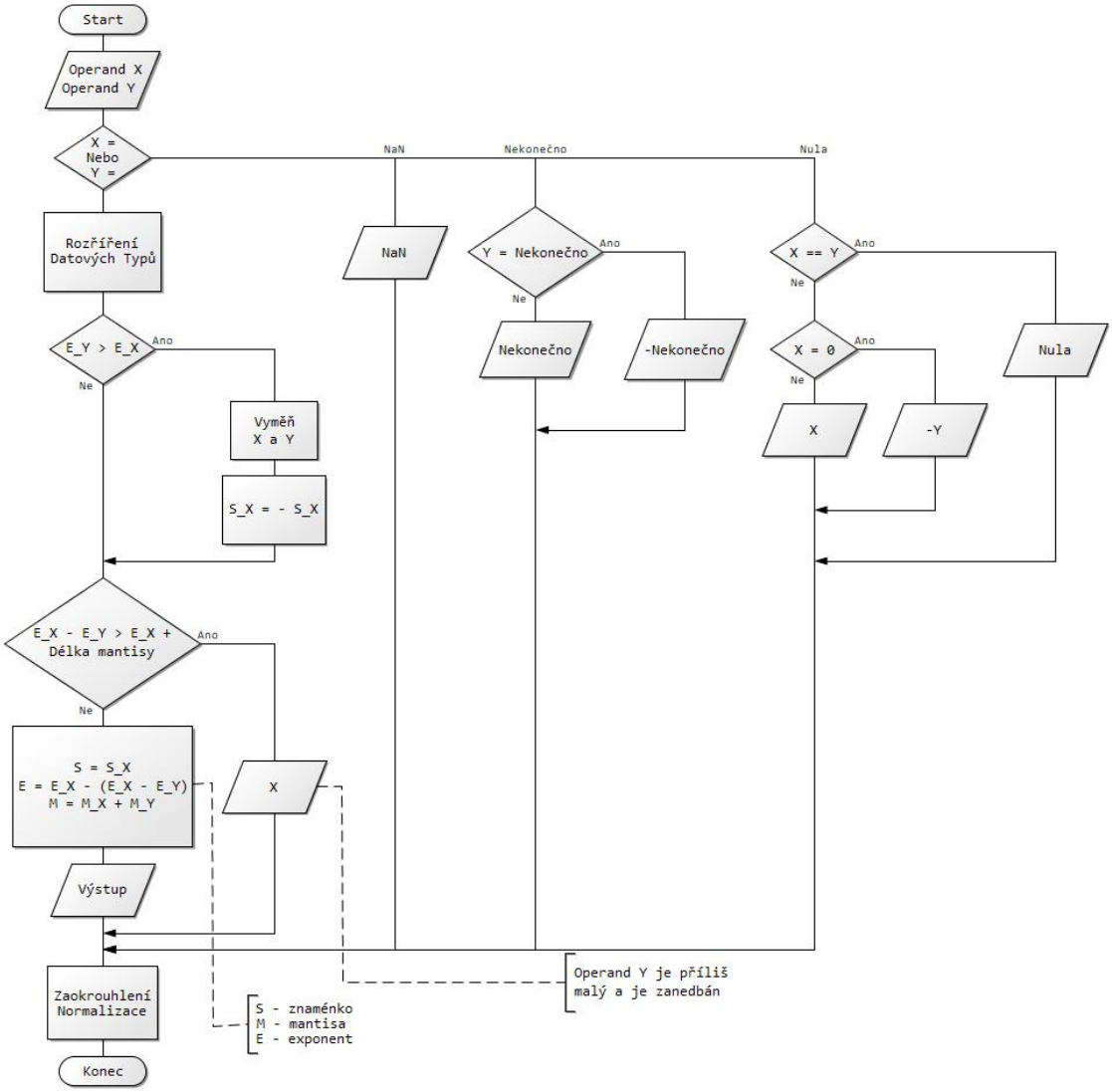
Pokud má proměnná y hodnotu nula, je návratovou hodnotou proměnná x. Pokud má x hodnotu nula je návratovou hodnotou -y.

5.2.1 Algoritmus odečítání

Nejprve je nutné oba operandy uložit do proměnné s dvojnásobnou bitovou šířkou a je k nim přidán implicitní bit, se kterým se musí počítat. Poté mohou nastat dvě situace, buď je větší operand x nebo y. Pokud je větší operand x, zjistíme jestli je jeho exponent větší než exponent y plus šířka mantisy. Pokud ne je hodnota y zanedbatelná, protože je mimo rozsah x a tudíž je vrácena hodnota x, jinak se pokračuje v algoritmu.

Zjistí se rozdíl mezi exponenty jejich odečtením. Tento rozdíl se pak odečte od exponentu x a mantisa se posune doleva o tento rozdíl bitů a následně se od této posunuté mantisy odečte hodnota mantisy y.

Pokud byla hodnota y na začátku větší provede se stejný algoritmus pouze se prohodí operandy x a y a zneguje se znaménko.



Obrázek 5.2: Vývojový diagram operace odčítání

Po těchto výpočtech je potřeba znormalizovat výslednou hodnotu posunutím mantisy a změnou hodnoty exponentu o toto posunutí. Mantisa je nutné případně zaokrouhlit pokud byli některé bity posunuty mino rozsah doleva.

5.3 Násobení

Před ověřením speciálních hodnot je možné vypočítat znaménko výsledné hodnoty. A to tak, že pokud jsou znaménka operandů stejná, bude výsledná hodnota kladná. Naopak pokud jsou znaménka operandů různá bude výsledná hodnota záporná. Následuje vyhodnocení speciálních hodnot v tomto pořadí.

NaN

Pokud je jedním z operandů NaN, výsledkem je hodnota NaN. Pokud mají oba operandy hodnotu NaN, výsledkem je také hodnota NaN.

Nula

Pokud má některý operand hodnotu nula, výsledek je hodnota nula.

Nekonečno

Pokud má jeden z operandů hodnotu nekonečno, výsledkem je tento operand. Pokud mají oba operandy hodnotu nekonečno, výsledkem je hodnota nekonečno. Znaménko je určeno výpočtem uvedeným výše.

5.3.1 Algoritmus násobení

Nyní následuje popis samotného algoritmu násobení. Nejdříve je nutné ověřit, zda při výpočtu nedojde k tzv. přetečení nebo podtečení hodnot - tedy, že výsledná hodnota exponentu nebude větší nebo menší než umožňuje bitová šířka exponentu.

K přetečení dojde pokud je součet hodnot exponentů obou operandů větší než jeden a půl násobek maximální hodnoty výsledného exponentu. Pokud k tomu dojde je hodnota výsledku nastavena na nekonečno. Tedy exponent má maximální hodnotu a mantisa je nulová.

K podtečení dojde, pokud hodnota součtu exponentů obou operandů je menší než je polovina maximální hodnoty exponentu výsledného datového typu.

Násobení je implementováno v metodě Multiplication(value_type y), která je definována s jedním parametrem, hodnotou y, dále pracuje s hodnotou uloženou v příslušném objektu ze kterého je metoda volána. Tato proměnná je označena jako x.

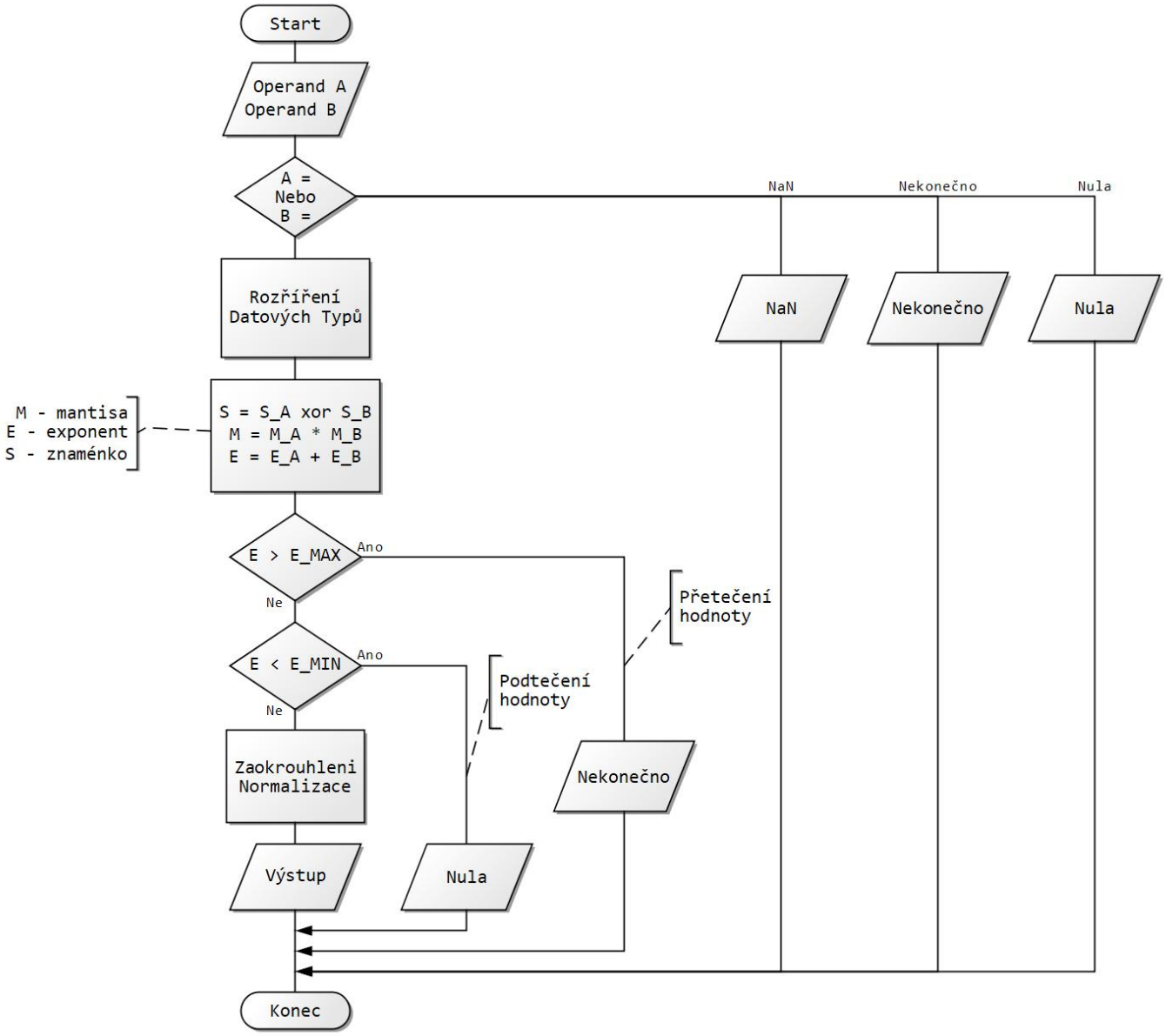
Nejprve je hodnota mantis obou operandů uložena v datovém typu s dvojnásobnou bitovou šířkou, než původní hodnota a je k ní přičtena hodnota prvního bitu, který je v reprezentaci podle standardu implicitní. Po této úpravě se mantisy mezi sebou vynásobí pomocí funkce násobení pro čísla v pevné řádové čárce s libovolnou přesností, jež je součástí knihovny Codasip_Integer již toto řešení používá. Po vynásobení je výsledná mantisa uložena opět v rozšířeném datovém typu a bude nutné ji následně oříznout a zaokrouhlit.

Po spočítání mantisy následuje výpočet exponentu. Při násobení floating point čísel se exponenty sčítají. K součtu je opět použita funkce knihovny Codasip_Integer.

Po tomto součtu exponentů je od výsledného exponentu odečtena polovina maximální hodnoty exponentu. To je dáno tím, že exponent je uložen v reprezentaci s posunutou nulou právě na polovinu maximální hodnoty. Při součtu exponentů se tedy tato hodnota přičetla dvakrát a je potřeba ji odečíst.

Dále je od výsledné hodnoty odečtena hodnota bitové šířky mantisy. To je dáno tím, že při násobení mantis se hodnota rozšířila na dvojnásobnou délku a při zaokrouhlování, které bude následovat bude tato mantisa posunuta doleva a tím oříznuta. O toto oříznutí se tedy musí snížit exponent.

Následuje už zmíněné posunutí a zaokrouhlení. Tím dojde k normalizování výsledku tak, aby byl reprezentován s implicitním bitem před mantisou a správnou bitovou šířkou. Velikost tohoto posunu se přičte k exponentu.



Obrázek 5.3: Vývojový diagram operace násobení

5.4 Dělení

Dělení je implementováno v metodě Division(value_type y), která je definována s jedním parametrem, hodnotou y což je v této operaci dělitel, dále pracuje s hodnotou uloženou v příslušném objektu, ze kterého je metoda volána. Tato proměnná je označena jako x tedy dělenec.

Následuje vyhodnocení speciálních hodnot v tomto pořadí.

NaN

Pokud je jedním z operandů NaN, výsledkem je hodnota NaN. Pokud mají oba operandy hodnotu NaN, výsledkem je také hodnota NaN.

Nula

Pokud má operand y hodnotu nula, výsledek je NaN. Pokud má operand x hodnotu nula, výsledek je hodnota nula.

Nekonečno

Pokud má jeden operand x hodnotu nekonečno, výsledkem je tento operand. Pokud má operand y hodnotu nekonečno, výsledkem je nula. Pokud mají oba operandy hodnotu nekonečno, výsledkem je hodnota nula. Znaménko je určeno výpočtem uvedeným výše.

5.4.1 Algoritmus dělení

Nyní následuje popis samotného algoritmu dělení.

Nejdříve je nutné ověřit, zda při výpočtu nedojde k tzv. přetečení nebo podtečení hodnot - tedy, že výsledná hodnota exponentu nebude větší nebo menší než umožňuje bitová šířka exponentu.

Pokud je exponent operandu x menší než exponent operandu y mínus polovina maximální hodnoty exponentu. Dochází k podtečení hodnoty exponentu do minusových hodnot. Tudíž je výsledná hodnota dělení rovna nule.

Naopak pokud exponent x mínus exponent y je větší než polovina maximální hodnoty exponentu dochází k přetečení přes maximální hodnotu exponentu ve výsledné hodnotě, a ta je proto nastavena na hodnotu nekonečno.

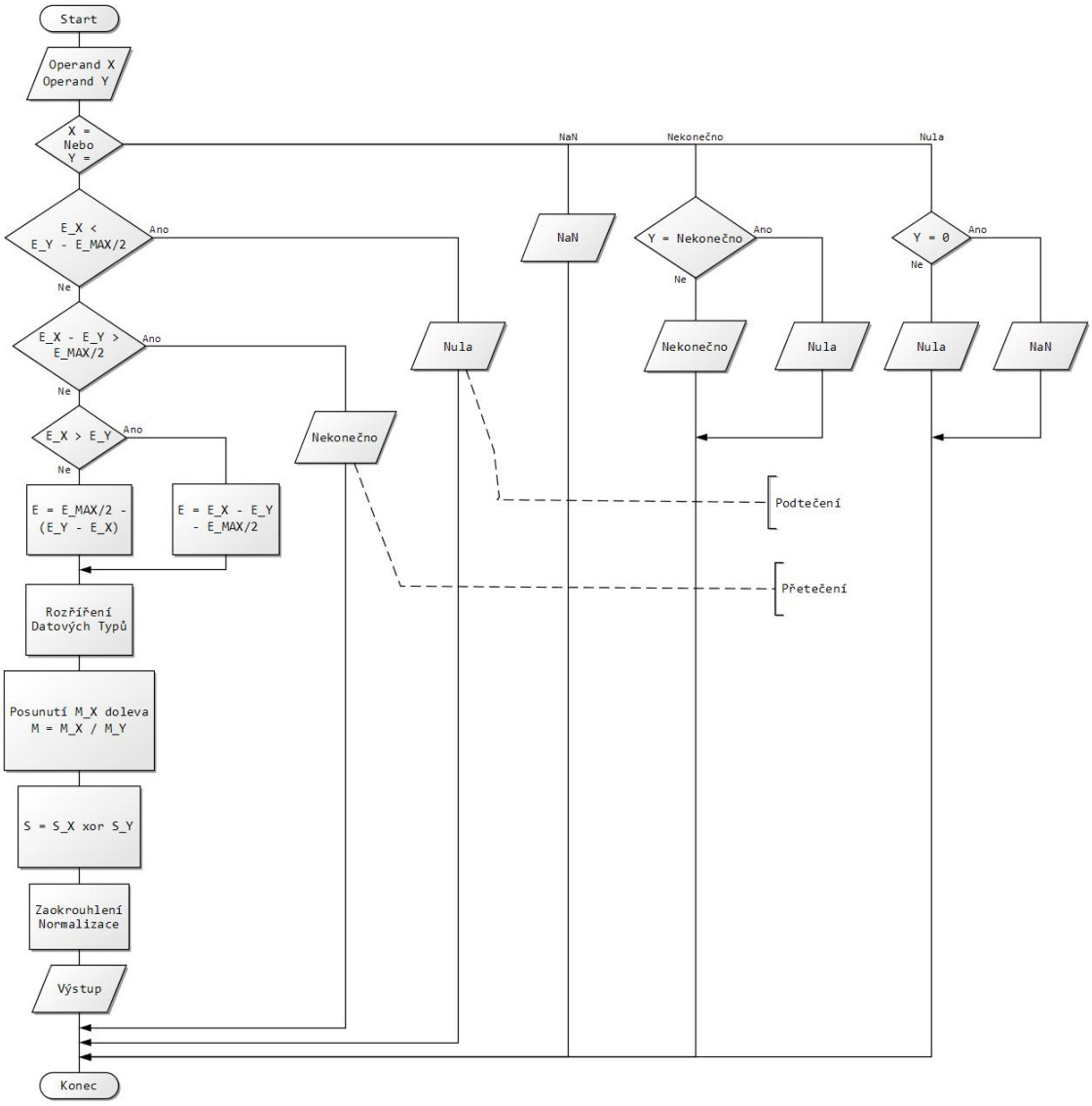
Pokud žádná z výše uvedených situací nenastane přistoupí se ke klasickému výpočtu exponentu a mantisy. Při výpočtu exponentu existují dvě možnosti.
Pokud je exponent x větší nebo roven, než exponent y. Výsledný exponent je roven rozdílu exponentů x a y mínus polovina maximální hodnoty exponentu. To je dáno tím, že je exponent reprezentován v soustavě s posunutou nulou a při odečítání je odečtena navíc.
Pokud je na druhou stranu exponent z větší něž x. Výsledný exponent je roven polovině maximální hodnoty exponentu, od které se odečte rozdíl exponentu y a x.

Nejprve je hodnota mantis obou operandů uložena v datovém typu s dvojnásobnou bitovou šířkou, než původní hodnota a je k ní přičtena hodnota prvního bitu, který je v reprezentaci podle standardu implicitní.

Poté se mantisa x posune o šířku mantisy doleva. Z této posunuté hodnoty se celočíselným vydělením, které je implementováno v knihovně Codasip_Integer získají hodnoty, získá hodnota kvocientu a zbytku. Pokud je zbytek větší než dvojnásobek mantisy y. Je nutné kvocient inkrementovat o jedna.
Výsledná mantisa je pak rovna hodnotě tohoto kvocientu.

Následuje posunutí a zaokrouhlení. Tím dojde k normalizování výsledku tak, aby byl reprezentován s implicitním bitem před mantisou a správnou bitovou šířkou. Velikost tohoto posunu se přičte k exponentu.

Výpočet znaménka se provádí stejně jako u operace násobení a to tak, že pokud jsou znaménka operandů stejná, bude výsledná hodnota kladná. Naopak pokud jsou znaménka operandů různá bude výsledná hodnota záporná.



Obrázek 5.4: Vývojový diagram operace dělení

5.5 Umocňování s celočíselným exponentem

V knihovně je implementována pouze varianta umocňování s celočíselným exponentem.

Operace je implementována jako metoda PowerI(const int exponent) s jedním parametrem exponent. Který definuje hodnotu exponentu na který se umocňuje.

Metoda pracuje převážně s operací násobení. Existují tři různé varianty v závislosti na hodnotě exponentu.

Za prvé - pokud je exponent nula a hodnota umocňovaného základu není NaN. Výsledná hodnota umocňování je jedna. V případě hodnoty NaN je výsledek NaN.

Za druhé, pokud je exponent větší než 0. Provede se v cyklu násobení hodnoty samy se sebou tolikrát kolikrát je hodnota exponentu mínus jedna. K násobení se používá již popsaná operace násobení 5.3, která ověřuje speciální hodnoty, tudíž se jimi nemusíme zaobírat.

Poslední třetí možností je varianta kdy je exponent menší než 0. V tomto případě probíhá výpočet stejně jako v předchozím případě kdy se v cyklu násobí hodnoty samy se sebou tolikrát kolikrát je hodnota exponentu v absolutní hodnotě mínus jedna. Speciální hodnoty jsou opět ověřeny v operaci násobení. Jediným rozdílem oproti kladnému exponentu, že výsledek je roven - jedna děleno hodnotou, která vznikla výpočtem násobení hodnot ve výše popsaném cyklu.

5.6 Odmocnina

Odmocnina je implementována v metodě SquareRoot(), která je bez parametru, pracující pouze s hodnotou uloženou v příslušném objektu, ze kterého je metoda volána. Tato proměnná bude označena jako x.

Algoritmus odmocňování je následujíc:

Nejprve jsou ověřeny speciální hodnoty.

NaN

Pokud je hodnota NaN nebo pokud je znaménko záporné je nastavena návratová hodnota na NaN.

Nekonečno

Pokud má operand hodnotu nekonečno, výsledkem je tento operand.

Nula

Pokud je hodnota x nula, zůstane nezměněna a je vrácena jako výsledek funkce.

5.6.1 Algoritmus odmocňování

Po ověření speciálních hodnot pokračuje samotný výpočet odmocniny. Nejprve je hodnota mantisy uložena v datovém typu s dvojnásobnou bitovou šířkou, než původní hodnota a poté je bitově posunuta doleva o šířku původní mantisy.

Nad touto hodnotou je pak zavolána funkce pro odmocninu čísel v pevné řádové čárce. Tato funkce je definována v knihovně pro čísla v pevné řádové čárce s libovolnou přesností, která byla vyvinuta ve společnosti Codasip. Vrací celočíselný výsledek odmocniny a také zbytek.

Výsledek odmocniny je potřeba inkrementovat v případě, že je bit na indexu šířky původní mantisy nastaven na 0 a zbytek po odmocnině je větší než výsledek odmocnění.

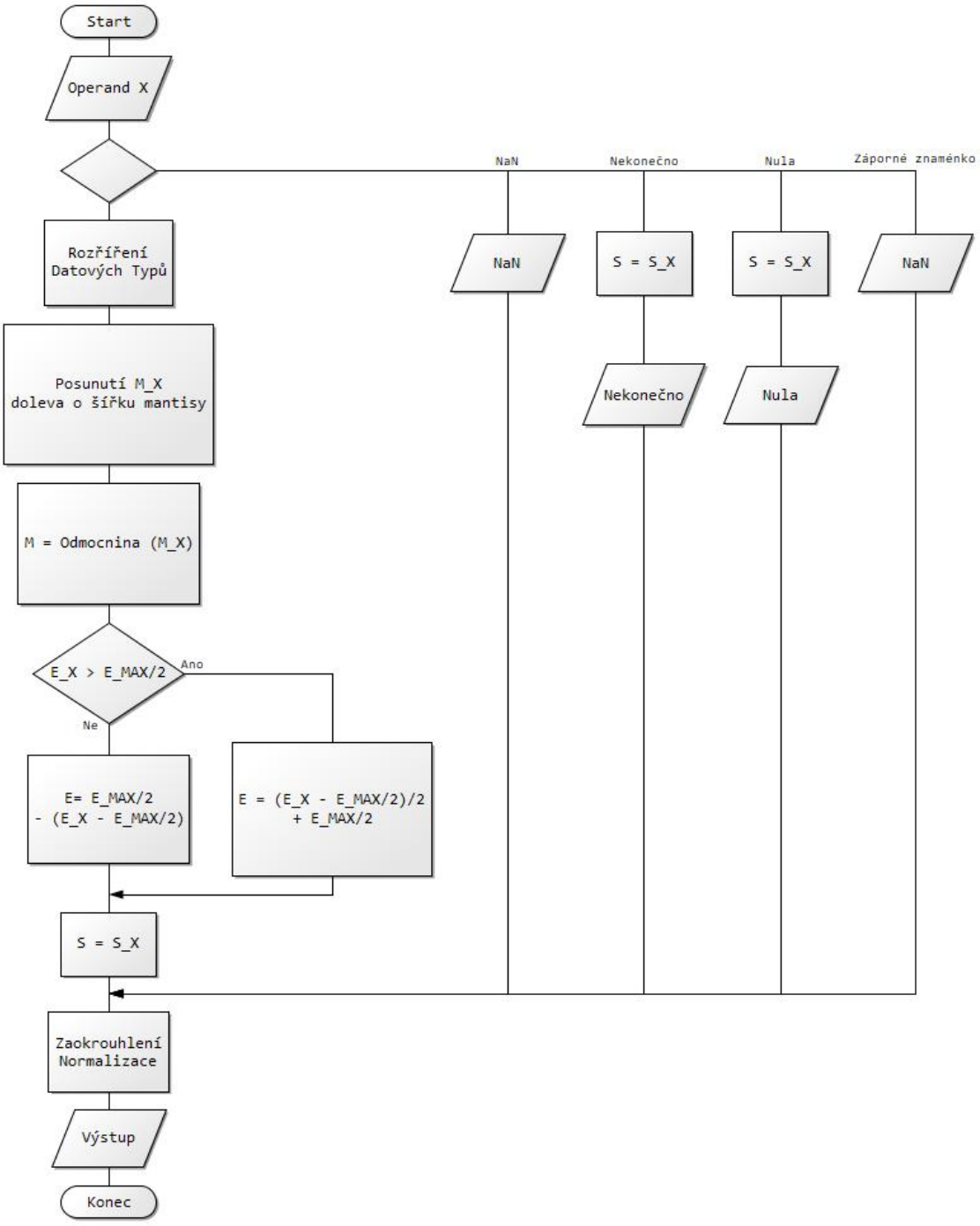
Po této operaci máme hodnotu mantisy a je potřeba spočítat hodnotu exponentu.

Exponent je potřeba upravit tak, aby jeho absolutní hodnota rozdílu od střední hodnoty byla polovina původní hodnoty. Odečteme, případně přičteme pokud je exponent menší než jeho střední hodnota, polovinu tohoto rozdílu.

Tato operace se provádí, protože exponent se při odmocňování zmenšuje na polovinu, ale v reprezentaci v plovoucí řádové čárce je exponent reprezentován v kódu s posunutou nulou právně na tuto střední hodnotu.

Pokud je tedy exponent o polovinu snížen, zbývá normalizovat výslednou hodnotu tak, aby byla mantisa posunuta do normalizovaného tvaru a velikost tohoto posunu se přičte k exponentu.

Tento algoritmus je upravenou verzí algoritmu převzatého z článku [8].



Obrázek 5.5: Vývojový diagram operace odmocňování

5.7 Porovnávání

Při porovnávání čísel v plovoucí řádové čárce může nastat několik komplikací, které při porovnávání klasických celočíselných typů nenastává. Nejprve je si třeba uvědomit, že pro jednu hodnotu čísla může existovat několik reprezentací. Tuto komplikaci knihovna řeší tak, že všechny hodnoty jsou uloženy v normalizovaném tvaru. Tento tvar reprezentuje číslo jako mantisu posunutou tak, aby nejvíce významový bit byl před mantisou uložen jako implicitní. Mantisa tedy začíná druhým významovým bitem. Po každé operaci je potřeba tedy posunou mantisu a přičíst k exponentu tento posun a tím reprezentaci normalizovat.

V tomto normalizovaném tvaru je pak možné číslo porovnávat s jiným.

Další komplikací jsou speciální hodnoty jako například Nekonečno a NaN, které se musí při porovnávání ověřovat.

Algoritmus porovnávání je implementován pomocí dvou nezávislých funkcí 5.2 ,které ověřují nerovnosti a funkce ověřující rovnost 5.3. Tyto funkce pracují se dvěma hodnotami. První je uložena v objektu a druhou hodnotu přijímá funkce jako parametr. Při popisu algoritmu jsou označovány jako x a y.

```
/***
* \brief check if this is less than y
* \return true if this is less than y
*/
bool LessThan(const value_type y) const;

/***
* \brief check if this is greater than y
* \return true if this is greater than y
*/
bool GreaterThan(const value_type y) const;
```

Výpis 5.2: Deklarace porovnávacích funkcí

```
/***
* \brief check if this is equal y
* \return true if this is equal y
*/
bool IsEqual(const value_type y) const;
```

Výpis 5.3: Deklarace funkce ověřující rovnost

Zde je popis algoritmu popisujícího operaci menší než.

Před samotným porovnáváním je potřeba ověřit speciální hodnoty.
Prvním krokem je ověření, zda je jedna z hodnot x nebo y NaN. Pokud ano je vrácena chyba. Následuje ověření, zda je právě jedna z hodnot x nebo y nekonečno nebo mínus nekonečno. Pokud je x nekonečno je vrácena hodnota true a pokud mínus nekonečno je vrácena hodnota false. Pro proměnnou y je porovnání stejné jen se vrací opačné logické hodnoty. Následuje test nulových hodnot. Pokud je jeden z operandů nula, ověřuje se jestli

je druhý operand kladný nebo záporný a podle toho je vrácena příslušná logická hodnota. Pokud jsou obě hodnoty obou operandů nulové je vrácena hodnota false.

Po ověření speciálních hodnot je možné ověřit znaménka operandů. Pokud je operand x kladný a operand y záporný je vrácena hodnota false, obdobně v opačném případě hodnota true.

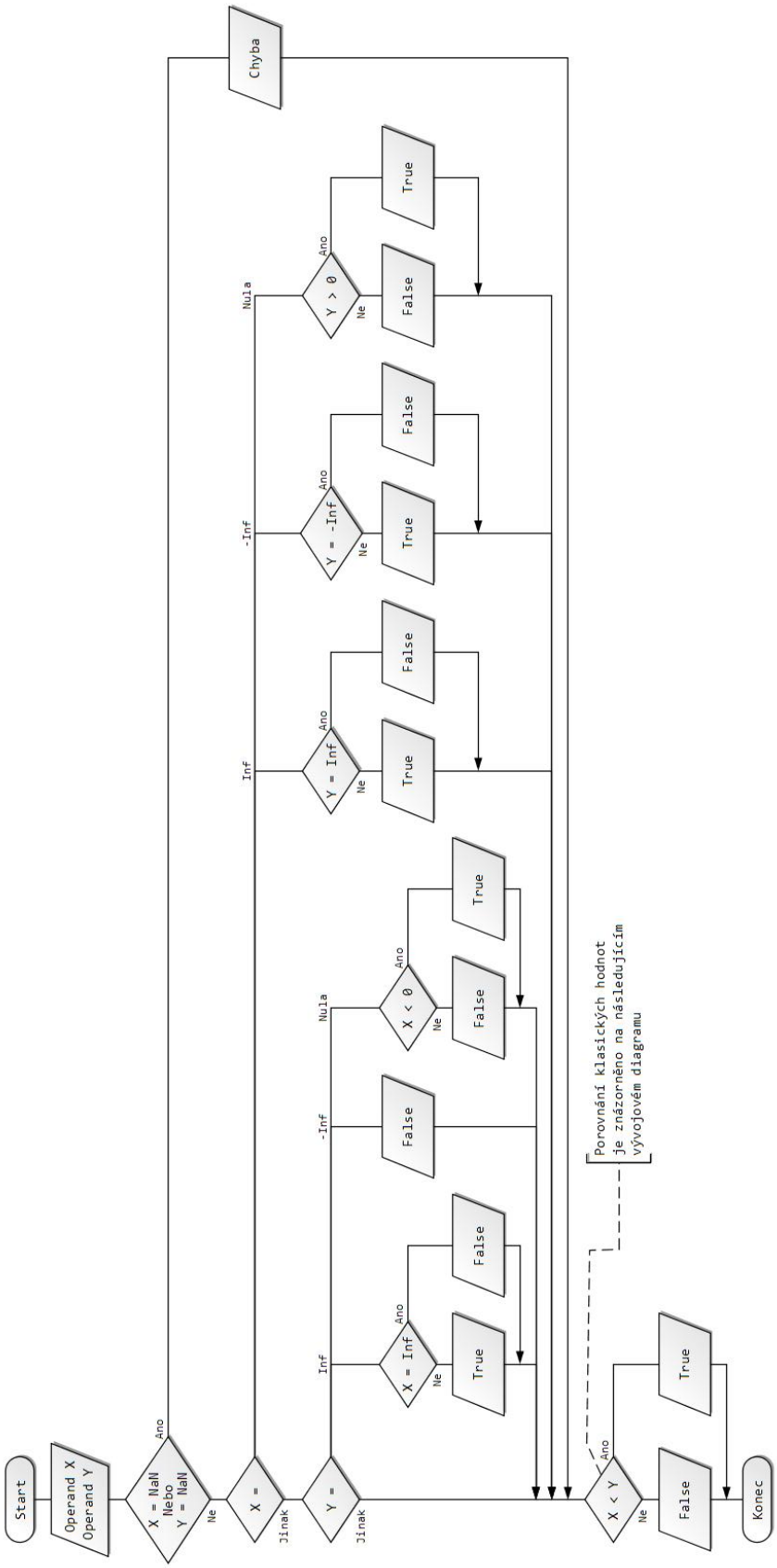
Pokud jsou znaménka operandů stejná pokračuje se k porovnávání samotných hodnot. Zde je potřeba ověřit, zda jsou obě hodnoty záporné. V tomto případě jsou všechny návratové hodnoty negovány.

V dalším kroku jsou porovnávány exponenty. Pokud je exponent proměnné x větší je návratová hodnota false, obdobně v opačném případě true. K porovnávání mantis se pokračuje pouze pokud jsou hodnoty exponentů stejné. V tomto případě se mantisy porovnávají stejně jako exponenty i se stejnými návratovými hodnotami.

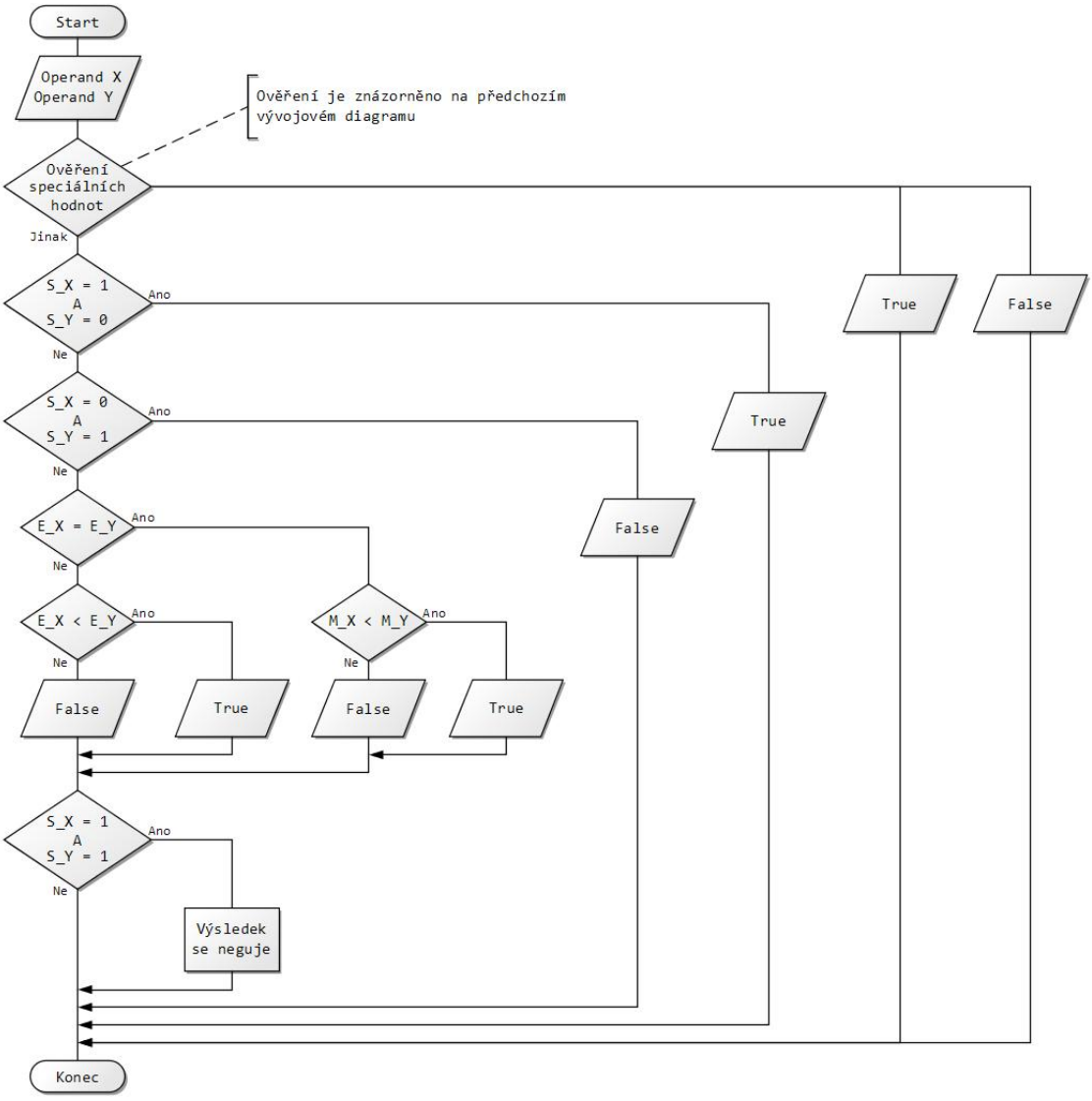
Algoritmus operace větší než je pak obdobný, pouze vrací opačné logické hodnoty.

Pro operace menší-rovno a větší rovno se ještě ověřuje pomocí funkce IsEqual(). Zde jsou stejným způsobem ověřovány speciální hodnoty a znaménka a poté exponenty, případně mantisy. Pouze se zde v případě, že jsou odpovídající hodnoty stejné pokračuje k ověřování další položky, popřípadě se vrací false, pokud jsou hodnoty různé.

Postup je názorně ukázán na vývojových diagramech **??** a **??**



Obrázek 5.6: Vývojový diagram ověřování speciálních hodnot operace menší než



Obrázek 5.7: Vývojový diagram operace menší než

5.8 Konverze na standardní datové typy

Součástí navržené knihovny je i konverze na základní floating point datové typy definované ve standardu IEEE 754. Při konverzi na standardní datové typy se rozlišují tři možné případy: Konverze na typ, který ma větší bitovou šířku a je tedy potřeba datový typ rozšířit a naopak typ který má menší bitovou šířku a je potřeba ořezat. Poslední možností je datový typ který má stejnou datovou šířku.

5.8.1 Konverze na datový typ stejné velikosti

V tomto případě je postup nejjednodušší. Díky tomu, že je navržená knihovna implementována podle standardu IEEE 754 a datové typy jsou navzájem kompatibilní se prostě jednotlivé části čísla jako je znaménko, exponent a mantisa jednoduše zkopírují na příslušné pozice ve standardních datových typech. Tedy jednobitové znaménko následované exponentem v příslušné bitové šířce následované mantisou.

5.8.2 Konverze na větší datový typ

Při převodu na větší datový typ je proces velmi jednoduchý. Mantisa se pouze zezadu rozšíří o daný počet bitů, které se nastaví na nulu. Pokud mají být dva exponenty různých datových šířek rovny, musí být roven rozdíl jejich hodnoty a jejich střední hodnoty, kterou může exponent nabývat. Při výpočtu exponentu o větší šířce tedy vypočítáme hodnotu rozdílu jeho střední hodnoty a aktuální hodnoty a k této hodnotě přičteme střední hodnotu rozsahu nového exponentu a tím získáme jeho novou hodnotu. Znaménko zůstává zachováno.

5.8.3 Konverze na menší datový typ

Konverze na datové typy s menší bitovou šířkou je nejprve nutné mantisu zaokrouhlit na požadovanou velikost a poté přebytečné bity oříznout. Zaokrouhlení se řídí definovaným zaokrouhlovacím módem, které byli definovány výše 5.11. Při konverzi exponentu se používá stejný postup jako při konverzi na větší datový typ popsaný v sekci výše. Při operaci konverze na menší datový typ je nutné brát na vědomí, že pokud je hodnota proměnné, která je konvertována mimo rozsah hodnot menšího datového typu, může dojít ke snížení přesnosti dané hodnoty popřípadě k její úplné směně pokud je ořezána hodnota exponentu.

5.9 Fma

Je funkce definovaná tak, že přijímá tři parametry x,y a z. A vrací výslednou hodnotu spočítanou jako x * y + z. Tato funkce je implementována pomocí již popsaných algoritmů násobení a sčítání.

5.10 Absolutní hodnota

Výpočet absolutní hodnoty je implementován v metodě void fabs(),která je bez parametru a pracuje pouze s hodnotou uloženou v objektu, ze kterého je volána.

Jedinou věc, kterou tahle metoda provádí je uložení hodnoty nula do bitu reprezentujícího znaménko. Tedy změní jakoukoliv hodnotu na kladnou.

5.11 Zaokrouhlování

Po každé operaci je potřeba zjistit, zda je výsledek v normalizovaném stavu a má správnou bitovou šířku. O tuto operaci se stará funkce CopyAndRound().

Tato funkce ověří nejdříve šířku mantisy. To se děje proto, že navržené algoritmy pracují s rozšířenou mantisou, ke které je navíc přičtena hodnota implicitního bitu před mantisou.

Nejdříve se tedy ověří, zda má rozšířená mantisa délku finální mantisy + 1. Právě kvůli implicitnímu bitu.

a.)
Pokud ano, tento bit se ořízne a oříznutá mantisa se uloží do výsledné proměnné.

b.)
Další možností je, že nejvýše významový bit mantisy leží v rozsahu finální mantisy, na kterou zaokrouhlujeme. V tom případě se bity posunou doleva o rozdíl mezi indexem nejvýznamnějšího bitu a šířky mantisy tak, aby byl tento bit posunut mimo rozsah a tím se oříznul implicitní bit. Pravá strana mantisy se pak doplní nulami. Po této operaci je nutné odečíst od exponentu hodnotu velikosti posunu mantisy. Aby tato hodnota normalizovaného čísla odpovídala hodnotě před normalizací.

c.)
V obou předchozích možnostech nebylo potřeba zaokrouhlovat, protože se mantisa posouvala doleva a ořezával se pouze implicitní bit.

Poslední variantou je stav kdy je pozice nejvíce významového bitu vyšší než je šířka mantisy, na kterou normalizujeme. V tomto případě je potřeba posunout bity doprava tak aby nejvíce významový bit byl oříznut přesně mimo rozsah mantisy. Opět se z něj stane implicitní bit. A určitý počet nejméně významových bitů je tímto posunem oříznut z mantisy vpravo. Kvůli tomuto oříznutí je potřeba mantisu zaokrouhlovat. Podle standardu IEEE 754 existuje pět zaokrouhlovacích módů. Algoritmy jimiž se toto zaokrouhlení realizuje jsou popsány v následujících sekcích. Po posunu mantisy doprava je potřeba zvýšit hodnotu exponentu o tento posun, aby tato hodnota normalizovaného čísla odpovídala hodnotě před normalizací.

5.11.1 Zaokrouhlovací módy podle IEEE 754

Zaokrouhlování k nejbližšímu, směrem k sudému číslu

V této sekci je popsána pravděpodobně nejsložitější a také nejvíce používaná metoda, která je nastavená ve většině překladačů jazyka C++ jako výchozí.

Základním pravidlem při zaokrouhlování k nejbližšímu u binárních čísel na n-tou pozici je zjistit hodnotu bitu na pozici n+1. Pokud je tato hodnota nulová, tak by se mělo číslo vždy zaokrouhlit dolů. Pokud je hodnota bitu jedna a jakýkoliv následující bit v daném čísle má hodnotu jedna, tak by se číslo mělo zaokrouhlit nahoru. Na druhou stranu pokud žádný následující bit za pozicí n+1 není nastaven na jedna poté se platí druhá část tohoto zaokrouhlovacího pravidla a to zaokrouhlení směrem k sudému číslu. Tohoto můžeme dosáhnout tak, že vždy zaokrouhlujeme na číslo končící nulou a je tudíž sudé.

Zaokrouhlování k nejbližšímu, směrem od nuly

Stejně jako u předchozího pravidla se zde zaokrouhluje k nejbližšímu. Zjistíme hodnotu bitu na pozici n+1. (Pozice n je ta, na kterou zaokrouhlujeme.) Pokud je tato hodnota nulová, zbytek bitů se ořízne a tím se zaokrouhlí dolů. Pokud je hodnota bitu jedna a jakýkoliv

následující bit v daném čísle má hodnotu jedna, zaokrouhluje se nahoru, ostatní bity za pozicí n se ořízno u a výsledek se inkrementuje.

Zaokrouhlování k nule

Na rozdíl od předchozích zaokrouhlovacích módů se při použití zaokrouhlování k nule nehledí na nejbližší možné číslo, na které je možné při definované bitové šířce zaokrouhlit. Výsledná hodnota zaokrouhlení při použití tohoto módu je definovaná pro kladné čísla jako maximální hodnota, která je menší než zaokrouhlované číslo. Pro záporné hodnoty je definována jako nejmenší možná hodnota, která je větší než zaokrouhlované číslo. Při použití těchto pravidel se tedy hodnota zaokrouhluje vždy k nule.

Zaokrouhlování k plus nekonečnu

Při použití tohoto módu se pět nehledá nejbližší možné číslo, ale zaokrouhluje se na nejmenší možnou hodnotu, která splňuje bitovou šířku danou zaokrouhlováním a která je větší než zaokrouhlovaná hodnota. Hodnota se tedy vždy zaokrouhluje směrem k plus nekonečnu.

Zaokrouhlování k mínus nekonečnu

Zaokrouhlování směrem k mínus nekonečnu je opak zaokrouhlování k plus nekonečnu. Hledá se hodnota, která má bitovou šířku danou zaokrouhlováním a je největší hodnotou, která je menší než zaokrouhlované číslo. Tedy se zaokrouhlí vždy k směrem mínus nekonečnu.

5.12 Ceil

Je implementována v metodě value_type Ceil() bez parametru. Jedinou vstupní hodnotou je hodnota uložená v objektu ze kterého je metoda volána.

Funkce ceil je definována tak, že jejím výsledkem je hodnota v plovoucí řádové čárce, která reprezentuje integrální hodnotu, tedy pouze celá čísla bez desetinných míst. Tato hodnota je nejmenší integrální hodnotou, která není menší než vstupní hodnota.

Před samotným výpočtem je nutné ověřit speciální hodnoty v tomto pořadí.

NaN

Pokud je vstupní hodnota NaN je nastavena návratová hodnota na NaN.

Nula

Pokud je vstupní hodnota nula, je tato hodnota beze změny vrácena.

Nekonečno

Pokud je vstupní hodnota nekonečno, je tato hodnota beze změny vrácena.

Po ověření speciálních hodnot následuje výpočet integrální části hodnoty. Vstupní hodnotu v něm označujeme jako x.

Pokud je exponent x menší než polovina maximální hodnoty exponentu mínus jedna, je výsledek pro kladné čísla jedna, protože hodnota reprezentuje pouze desetinnou část a ta je zaokrouhlena nahoru na hodnotu jedna. Pro záporné čísla znamená zaokrouhlení nahoru výslednou hodnotu nula.

Pokud je exponent větší. Mantisa je uložena do datového typu o dvojnásobné datové šířce původní mantisy a je do ní uložena hodnota mantisy. Poté se tato hodnota posune doprava o rozdíl šířky mantisy a kladné hodnoty exponentu, která je rovna velikosti exponentu, mínus poloviny maximální hodnoty exponentu. Pokud byli tímto posunem ořezán alespoň jeden bit s hodnotou jedna je posunutá hodnota inkrementována, kvůli zaokrouhlení nahoru, které je nutné provést a které vyplývá z definice operace Ceil. Po tom je mantisa posunuta o stejnou velikost zpátky a zleva jsou nasunuty nuly.
Touto operací byla oříznuta desetinná část a zbývá provést normalizaci hodnoty jejím posunutím na odpovídající šířku, která počítá s prvním implicitním bitem před mantisou.

5.13 Trunc

Je implementována v metodě value_type Trunc() bez parametru. Jedinou vstupní hodnotou je hodnota uložená v objektu, ze kterého je metoda volána.

Funkce Trunc je definována tak, že jejím výsledkem je hodnota v plovoucí řádové čárce, která reprezentuje integrální hodnotu, tedy pouze celá čísla bez desetinných míst. Tato hodnota je nejbližší integrální hodnotou, která není v amplitudě větší než vstupní hodnota.

Před samotným výpočtem je nutné ověřit speciální hodnoty v tomto pořadí.

NaN

Pokud je vstupní hodnota NaN je nastavena návratová hodnota na NaN.

Nula

Pokud je vstupní hodnota nula, je tato hodnota beze změny vrácena.

Nekonečno

Pokud je vstupní hodnota nekonečno, je tato hodnota beze změny vrácena.

Po ověření speciálních hodnot následuje výpočet integrální části hodnoty. Vstupní hodnotu v něm označujeme jako x.

Algoritmus zaokrouhlování Trunc

Postup výpočtu integrální hodnoty je velmi podobný jako u funkce ceil. Pokud je exponent menší než polovina maximální hodnoty exponentu. Je výsledná integrální hodnota nula, protože se zaokrouhluje vždy dolů. Pokud je větší než nula. Je nutné oříznout bity mantisy, které reprezentují desetinné hodnoty. To se provede posunutím vpravo a zpětným nasunutím nul. Počet bitů o které se posunuje je vypočítán jako šířka mantisy mínus rozdíl hodnoty exponentu a poloviny maximální hodnoty exponentu. Polovina maximální hodnoty exponentu se odečítá kvůli reprezentaci exponentu v soustavě s posunutou nulou.

5.14 Round

Operace Round je definována tak, že zaokrouhluje reálná čísla na nejbližší celé číslo. V případě, že je zaokrouhlovaná hodnota přesně uprostřed mezi dvěma celými čísly, vždy se zaokrouhluje směrem od nuly, to znamená pro kladné čísla na číslo, které je nejmenším možným větším číslem než zaokrouhlovaná hodnota. Pro záporné čísla je to největší možná hodnota, která je menší než zaokrouhlované číslo.

Před samotným výpočtem je nutné ověřit speciální hodnoty v tomto pořadí.

NaN

Pokud je vstupní hodnota NaN je nastavena návratová hodnota na NaN.

Nula

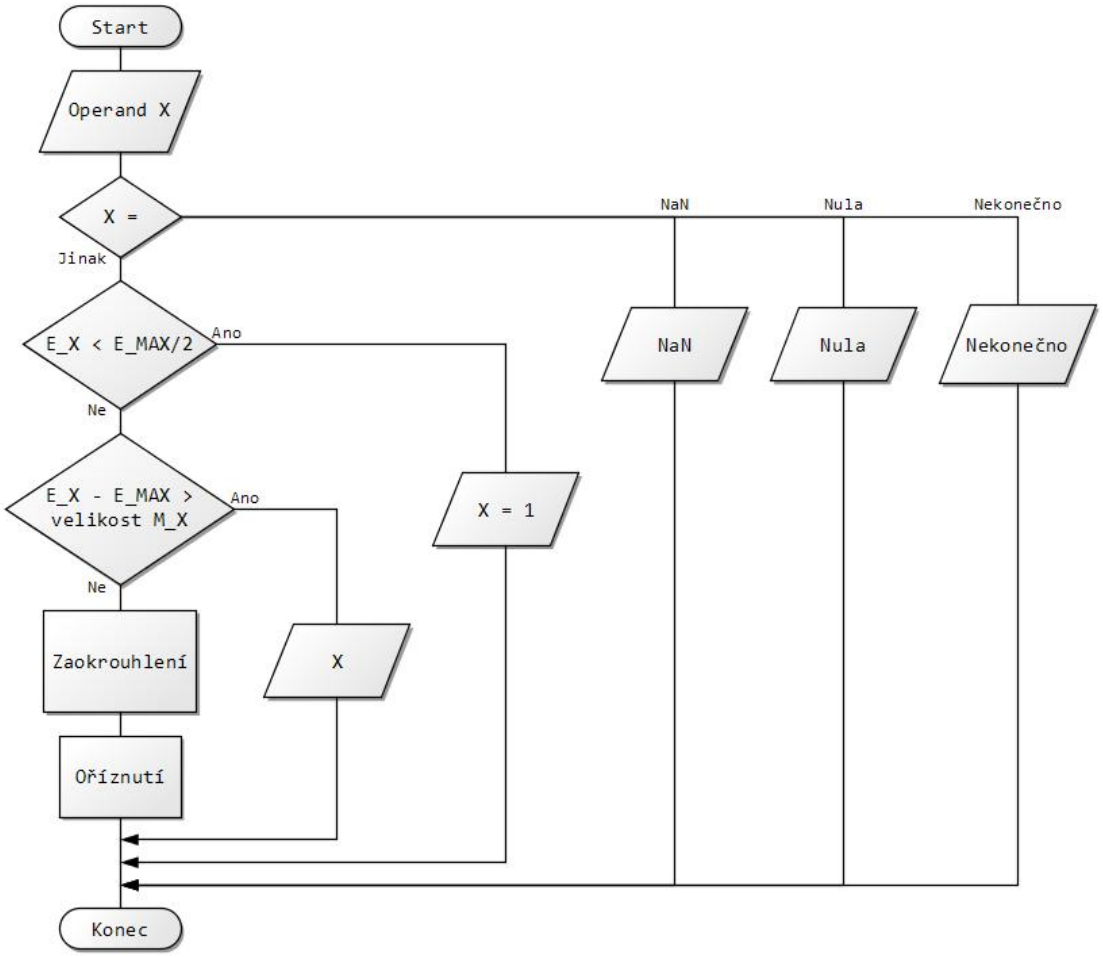
Pokud je vstupní hodnota nula, je tato hodnota beze změny vrácena.

Nekonečno

Pokud je vstupní hodnota nekonečno, je tato hodnota beze změny vrácena.

Algoritmus zaokrouhlování Round

Po ověření speciálních hodnot následuje samotný výpočet. Nejprve se ověří, že exponent je větší než je polovina maximální hodnoty exponentu. V opačném případě je výsledek automaticky jedna, protože hodnota je vždy menší než jedna a nulu jsme ověřili již předtím. Pokud je exponent větší pokračujeme v algoritmu: Od exponentu odečteme polovinu jeho maximální hodnoty. Pokud je tato hodnota větší než bitová šířka mantisy, je vstupní operand operace Round naším výsledkem, protože to znamená, že vstupní hodnota nemá desetinnou část. V opačném případě se od mantisy ořízne pravá část bitů. Počet těchto bitů získáme rozdílem bitové šířky mantisy a rozdílu exponentu s polovinou jeho maximální hodnoty. Touto operací jsme ořízly desetinou část hodnoty. Zbývá pouze provést zaokrouhlení hodnoty. Pokud byl první ořezaný bit nastaven na hodnotu 1, je potřeba inkrementovat mantisu. Algoritmus je ve zjednodušené podobě vyznačen na vývojovém diagramu 5.8.



Obrázek 5.8: Vývojový diagram operace Round

5.15 Floor

Je implementována v metodě value_type Floor() bez parametru. Jedinou vstupní hodnotou je hodnota uložená v objektu, ze kterého je metoda volána.

Funkce floor je definována tak, že vrací největší integrální hodnotu, která není větší než vstupní hodnota. Integrální hodnota je celočíselná hodnota reprezentována v floating point formátu.

Před samotným výpočtem je nutné ověřit speciální hodnoty v tomto pořadí.

NaN

Pokud je vstupní hodnota NaN je nastavena návratová hodnota na NaN.

Nula

Pokud je vstupní hodnota nula, je tato hodnota beze změny vrácena.

Nekonečno

Pokud je vstupní hodnota nekonečno, je tato hodnota beze změny vrácena.

Po ověření speciálních hodnot následuje výpočet integrální části hodnoty. Vstupní hodnotu v něm označujeme jako x.

Postup výpočtu integrální hodnoty je velmi podobný jako u funkce ceil nebo Trunc. Pokud je exponent menší než polovina maximální hodnoty exponentu. Je výsledná integrální hodnota nula, protože se zaokrouhluje vždy dolů. Pokud je větší než nula. Je nutné oříznout bity mantisy, které reprezentují desetinné hodnoty. To se provede posunutím vpravo a zpětným nasunutím nul. Počet bitů o které se posunuje je vypočítán jako šířka mantisy, mínus rozdíl hodnoty exponentu a poloviny maximální hodnoty exponentu. Polovina maximální hodnoty exponentu se odečítá kvůli reprezentaci exponentu v soustavě s posunutou nulou.

Rozdíl oproti funkci Trunc je v tom, že pokud je z mantisy oříznut alespoň jeden bit s hodnotou jedna a zároveň je vstupní hodnota záporná. Je potřeba před posunutím doleva ještě mantisu inkrementovat.

Poté následuje normalizace hodnoty.

5.16 Rint

Je implementována v metodě value_type Rint() bez parametru. Jedinou vstupní hodnotou je hodnota uložená v objektu, ze kterého je metoda volána.

Nejdříve je potřeba zpracovat speciální hodnoty vstupu.

NaN

Pokud je vstupní hodnota NaN je nastavena návratová hodnota na NaN.

Nula

Pokud je vstupní hodnota nula, je tato hodnota beze změny vrácena.

Nekonečno

Pokud je vstupní hodnota nekonečno, je tato hodnota beze změny vrácena.

Funkce Rint je definována tak, že vrací integrální hodnotu za použití zaokrouhlování definovaného módem podle IEEE 754 standardu viz. sekce 5.11.

Tak jako ve funkcích Floor a Rint, opět dochází pomocí posunu doprava k ořezání desetinné části mantisy. Případné zaokrouhlení pomocí inkrementování je dáno příslušným zaokrouhlovacím módem.

- zaokrouhlování směrem k pozitivnímu nekonečnu

  Při módu zaokrouhlování směrem k pozitivnímu nekonečnu, je mantisa při kladných hodnotách zaokrouhlena vždy nahoru a záporných vždy dolů.

- zaokrouhlení k negativnímu nekonečnu

  Naopak při módu zaokrouhlení k negativnímu nekonečnu, je mantisa při kladných hodnotách vždy zaokrouhlena dolů a při záporných nahoru.

- zaokrouhlování k nule V módu zaokrouhlování k nule jsou hodnoty mantisy vždy zaokrouhleny dolů.

- zaokrouhlení k sudému číslu V módu zaokrouhlení k sudému číslu se vždy kontroluje poslední bit zaokrouhlovaného čísla. Pokud je roven jedné zaokrouhluje se nahoru. Pokud je roven nule, číslo je již sudé mantisa se neinkrementuje.

5.16.1 Integrální hodnota je rovna nule

Všechny předchozí případy se provádějí pouze pokud je exponent větší než polovina jeho maximální hodnoty. Pokud je menší, je integrální část nulová. V tomto případě se výsledná hodnota opět určuje podle módu.

- zaokrouhlování směrem k pozitivnímu nekonečnu Je výsledná hodnota pro kladné čísla je vždy jedna a pro záporné nula, protože se zaokrouhluje nahoru.
- zaokrouhlení k negativnímu nekonečnu Je výsledná hodnota pro kladné čísla vždy nula, protože se zaokrouhluje dolů. Pro záporné je to mínus jedna.
- zaokrouhlování k nule Výskedná hodnota je nula, protože se vždy zaokrouhluje k nule.
- zaokrouhlení k sudému číslu Výsledná hodnota je nula.

Kapitola 6

Testování

Informace v této kapitole byli čerpány z knihy [13] a z dokumentace google test frameworku [6].

Unit testy neboli jednotkové testy slouží k validaci správnosti testovaného zdrojového kódu. Filozofií tohoto testování je rozdělit zdrojový kód na co možná nejmenší samostatně testovatelné části - jednotky a ty pak testovat nezávisle na okolním prostředí. Tím se tedy ověří, zda je daná jednotka správná. V objektově orientovaném programování může být touto jednotkou například metoda třídy nebo celá třída s jejím rozhraním.

Tyto jednotky jsou ideálně navzájem nezávislé. Pokud mezi nimi nějaká závislost existuje, je možné použít techniku zvanou mockování. Tato technika umožní izolovat jednotlivé jednotky tak, je vytvořeno falešné rozhraní mezi jednotkou, která simuluje vstupy, které jsou předávána této jednotce.

Testování jednotlivých jednotek je prováděno na principu black boxu. Do jednotky jsou tedy přes její rozhraní předány vstupní data a testuje se, zda výstupní data odpovídají předdefinovanému referenčnímu výsledku. Pokud ano test je úspěšný, v opačném případě nikoliv.

Unit testy jsou obvykle automatizovány. K tomuto účelu slouží jednotlivé unit testové frameworky.

S automatizovanými unit testy je pak možné provádět testování jednotlivých modulů při každém změně kódu a vydání nové vrze aplikace, aby se ověřilo, že změnou nebyla vytvořena jiná chyba.

6.0.1 Nevýhody jednotkového testování

Nevýhodou jednotkového testování je, že neodhalí všechny chyby, které mohou v programu nastat. To je zapříčiněno tím, že těmito testy nelze u větších programů pokrýt všechny proveditelné stavy, které mohou v programu nastat.

6.1 Google test

Google test [6] neboli zkráceně Gtest je framework určený ke zjednodušení tvorby unit testů. Tento framework byl použit kvůli kompatibilitě mého floating point modulu a existujícího řešení, do kterého se bude tento modul integrovat.

Obecně je tedy Google test framework pro tvorbu a automatické testování za pomoci unit testů. Je to multiplatformní systém pracující jak na Unixu Windows tak Linuxu a je určen pro testování programů implementovaných v jazyce C++.

Testování pomocí Gtest je založeno na jednotlivých testech jejichž součástí jsou tzv. asserty, což jsou výrazy, které vyhodnocují, zda je zadaná podmínka splněna. Výsledek assertu může být úspěch, fatální neúspěch nebo nefatální neúspěch. Rozdíl mezi fatálním a nefatálním neúspěchem je ten, že při fatálním neúspěchu je přerušen celý test jako neúspěšný. Při nefatálním neúspěchu pokračuje test dále a pouze je tento neúspěch zaznamenán.

Asserty jsou pak využity testem, aby ověřily chování testovaného zdrojového kódu testovaného programu. Pokud je některý z assertů na konci testu označen jako nespěšný - tedy false. Je jako neúspěšný označen celý test. Tento neúspěch se pak zobrazí ve výpisu testování.

Testy se v Gtest frameworku sdružují do jednotlivých skupin kvůli lepší přehlednosti.

6.1.1 Praktické použití

Asserty Gtest frameworku jsou vlastně makra, která porovnávají dvě hodnoty, jež mohou být například výsledkem nějaké metody. Asserty se dělí na fatální (začínající na ASSERT) a nefatální (Začínající na EXPECT). Rozdíl mezi nimi je, že fatální generují při neúspěchu fatální neuspěch testu a nefatální nikoliv. Příklady těchto maker ukazuje tabulka 6.1 tato tabulka byla převzata z

Tabulka 6.1: Příklady jednotlivých asertů

Fatální assert	Nefatální assert	Ověřuje
ASSERT_EQ(val1, val2);	EXPECT_EQ(val1, val2);	val1 == val2
ASSERT_NE(val1, val2);	EXPECT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPECT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPECT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPECT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

Hodnoty argumetů musí být porovnatelné pomocí odpovídajícího operátoru z této tabulky, v opačném případě nastane chyba při překladu. Asserty mohou také pracovat s nestandardními, nově definovanými datovými typy. Tyto typy pouze musí splňovat podmínku, že u nich musí být definován odpovídající porovnávací operátor. Této možnosti bylo využíváno při testování floating point modulu, kde jsou definovány datové typy s různou datovou šířkou.

Na příkladu 6.1 lze vidět jeden testovaný scénář. Lze na něm spatřit vytvoření a inicializace proměnné value, s definovaným typem ve formátu floating point a s šířkou exponentu 10 bitů a s šířkou mantisy 23 bitů. Nad touto proměnou je poté zavolána procedura msbPositionIndex, která vrací index prvního jedničkového bitu mantisy a uloží ji do proměnné result. Potom je zavolán assert EXPECT_EQ, který ověří, zda je hodnota result rovna hodnotě 6. V tomto případě bude výsledek testu úspěch.

```
TEST(CodasipFloatTest, msb_position3)
{
codasip::Uint<10> exponent(1);
codasip::Uint<23> mantissa(64);
codasip::Float<10, 23> value(exponent, mantissa, false);

int result = value.msbPositionIndex(mantissa)

EXPECT_EQ(6, result);
}
```

Výpis 6.1: Příklad testu s použitím frameworku Gtest

6.2 Návrh jednotlivých testů

6.2.1 Testování s náhodnými hodnotami

Generování náhodných hodnot

Ke generování hodnot se používá uniformní reálná distribuce hodnot, která je definována funkcí. uniform_real_distribution. Tato funkce vygeneruje distribuci z rozsahu, který je definován jejími parametry. První parametr slouží k definování minimální hodnoty a druhý parametr k definování maximální hodnoty. Ve chvíli kdy máme vytvořenou distribuci, je potřeba generátor. Je použit generátor ze standardní knihovny - default_random_generator. Tento generátor potřebuje ke správné funkci takzvaný seed, což je hodnota sloužící k inicializaci generátoru tak, aby pokaždém spuštění generoval náhodné hodnoty. Tato hodnota musí být taky náhodná a je generována pomocí třídy random device, která generuje pseudonáhodné čísla v pevné řádové čárce, jež pro účely generování náhodných čísel pro testy, při kterých není vyžadováno generování opravdu náhodných čísel, bohatě postačí.

V testech se pak tyto vygenerované hodnoty konvertují na požadovaný datový typ s daným rozsahem. Výpočet je pak spuštěn nad standardními typy jazyka C++, které slouží jako referenční řešení a typy definovanými v naší knihovně.

```
std::random_device rd;
std::default_random_engine generator(rd());
std::uniform_real_distribution<double> distribution(0, DBL_MAX);

double f1 = distribution(generator);
```

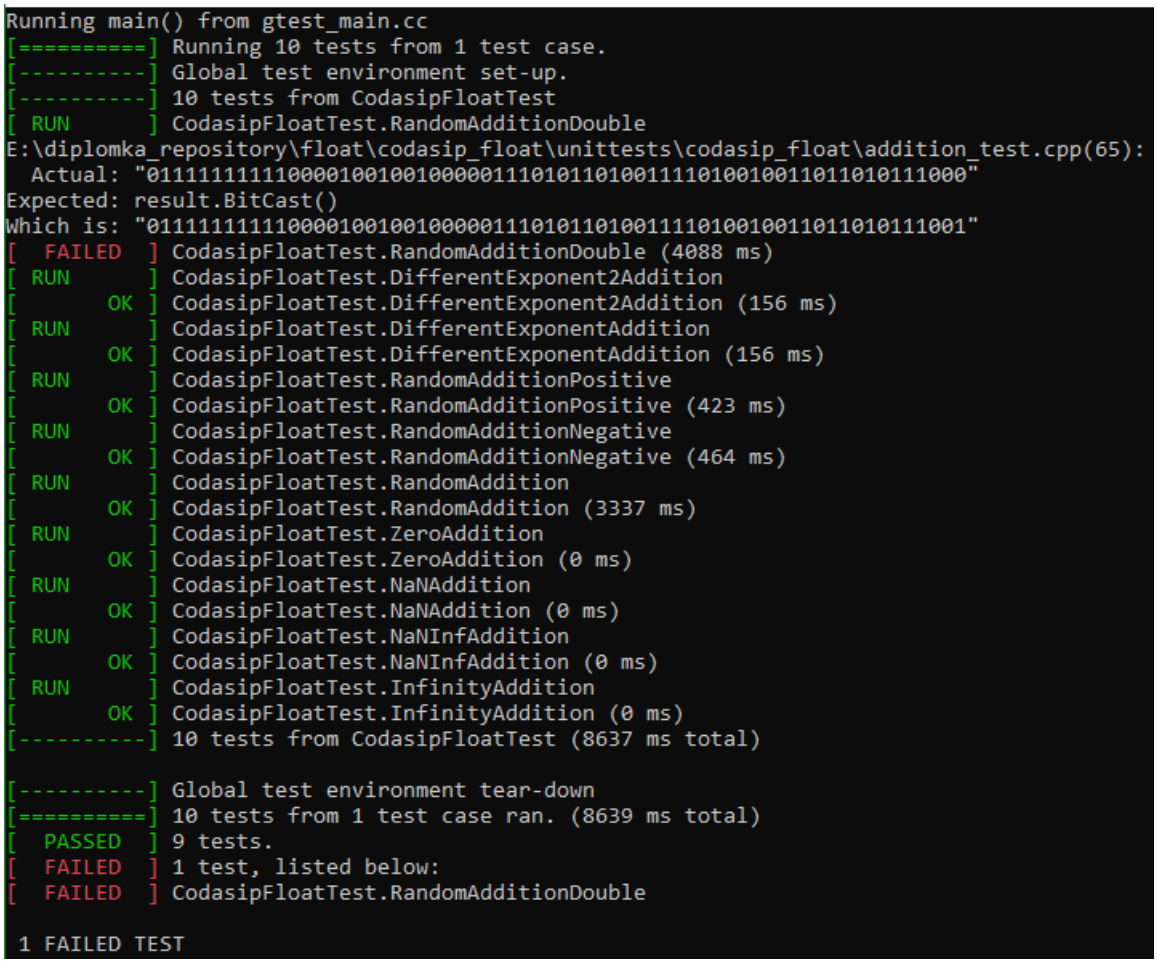
Výpis 6.2: Generování náhodných hodnot

Ověřování výsledků

Ve chvíli kdy jsou vygenerovány všechny hodnoty operandů a ty jsou konvertovány do referenčních datových typů a do typů, které testujeme je možné přistoupit k samotnému výpočtu. Ten proběhne nezávisle jak pro referenční hodnoty tak pro ty testované. A výstupy jsou uloženy v proměnných odpovídajících datových typů. Nyní je nutné výstupy ověřit. Naše výstupy ověřujeme vůči referenčním několika způsoby v závislosti na tom jakou funkcionalitu ověřujeme. Je možné ověřit bitový výpis ve formě řetězce kde jsou zaznamenány jednotlivé binární hodnoty. Toto ověřování slouží k porovnávání přesných hodnot například u operace zaokrouhlování. Dále je možné ověřit konverzi na dekadický tvar porovnáváním hodnot zaokrouhlených na určitý počet desetinných míst. Poslední možností je ověřování hodnot exponentu, mantisy a znaménka zvlášť.

K ověřování jsou použita makra, která jsou popsána v tabulce 6.1.

Výsledkem testů je výpis jednotlivých testovacích scénářů, spolu s jejich výsledným statusem, který je buď OK v případě úspěšného testu, nebo ERROR v případě kdy se referenční a ověřované hodnoty liší. Příklad takového výstupu je vidět na obrázku 6.1, kde je možné vidět jak úspěšné testy tak jeden test, který selhal.



Obrázek 6.1: Příklad výstupu jednotkových testů operace sčítání

6.2.2 Testování mezních případů

Testování s náhodnými hodnotami tak jak bylo popsáno v předchozí sekci může odhalit velkou část chyb, ale nikdy nebude stoprocentní. V této sekci se zaměříme testování speciálních případů, u kterých se předpokládá častý výskyt chyb.

Těmito příklady můžou být například speciální hodnoty operandů, při operacích floating point aritmetiky definované standardem, jako například NaN (not a number), nekonečno nebo nula. Ale také jimi může přetečení hodnoty proměnné přes její maximální mantisu nebo exponent.

Nyní budou popsány příklady těchto mezních hodnot tak, jak se testují v jednotlivých unit testech.

Sčítání

Pokud je jeden z operandů NaN výsledek bude opět NaN. Pokud je jeden právě jeden z operandů nekonečno výsledek bude nekonečno.

Test spočívá v tom, že sečteme hodnotu nekonečno s hodnotou NaN. A výsledek musí být opět NaN, protože toto pravidlo má vyšší prioritu. V testu se tedy kontroluje, zda je hodnota exponentu maximální a zda je hodnota mantisy různá od nuly. Tato konfigurace odpovídá hodnotě NaN.

Násobení

U násobení je nutné zkontrolovat hlavně jestli je korektně zpracováno přetečení hodnoty mimo platný rozsah. Pokud je jeden z operandů NaN výsledek bude opět NaN.

Dělení

U dělení je důležité otestovat dělení nulou. Tato operace by měla vyústit v chybu. Dělení nekonečnem by měla být nula. A pokud je jeden z operandů NaN výsledek bude opět NaN.

Odmocnina

Odmocnina je definována pouze pro kladné čísla. Je tedy nutné testovat, zda je vstupní hodnota větší nebo rovna nule. V opačném případě je výsledkem NaN

Kapitola 7

Existující konkurenční řešení a jejich srovnání

Součástí práce je nastudování a porovnání současných opensource řešení dané problematiky. Jedním z těchto řešení je multiprecision modul knihovny boost [11] . Dalším existujícím řešením je GNU multiple precision Arithmetic Library.

7.1 Popis konkurenčních řešení

7.1.1 Boost Multiprecision

Boost je komplementem standardní knihovny jazyka C++, obsahuje spoustu užitečných funkcí, které ve standardním jazyce chybí. Je platformě nezávislý a podporovaný v mnoha operačních systémech, včetně Windows a Linuxu. V podstatě je to soubor mnoha knihoven, založených na C++ standardu. Knihovny jsou kontinuálně vyvíjeny a aktualizovány vekou skupinou vývojářů od roku 1998 kdy byla vydána první verze. Boost má svou vlastní licenci, vyvinutou ve spolupráci s Harvardskou univerzitou, která není vázána na GPL. Její přesné znění a popis je možné nalézt zde: [1]. Tato licence umožňuje používat a modifikovat zdrojové kódy podle libosti jak ke komerčním, tak k nekomerčním použitím. Zároveň ale vyžaduje, že musí každá distribuce obsahovat tuto licenci, což může být pro komerční využití problém.

7.1.2 GNU multiple precision

Gnu multiple precision, neboli zkráceně GMP je open-source knihovna pro aritmetiku s libovolnou přesností, pracující ad integery, racionálními čísly a nad čísly v plovoucí řádové čárce. GMP má bohatou sadu funkcí a má rozhraní kompatibilní se standardními typy jazyka C++. Hlavní použití pro GMP je kryptografie a výzkumné aplikace a výpočet pro algebraické systémy. GMP bylo vyvinuto s důrazem na maximální rychlost pro malé i velké operandy. GMP bylo poprvé vydáno v roce 1991 a od té doby je soustavně vyvíjeno a pravidelnými verzemi vydávanými přibližně jednou do roka. Od verze 6 je GMP distribuováno pod dvojí licencí: GNU LGPL v3 a GNU GPL v2. Tyto licence umožňují využívat knihovnu zdarma, sdílet ji a upravovat. Bohužel mají striktní restrikce při komerčním použití. Více o těchto licencích je možné najít v oficiálních dokumentacích zde: [3], [2].

7.2 Optimalizace

Během vývoje bylo provedeno několik změn, které vedly k optimalizaci a tudíž zrychlení výpočtů. Například byl použit implicitní bit před mantisou. Tenhle bit je definován ve standardu IEEE 754 a jeho princip je takový, že mantisu považujeme vždy o jeden bit větší tak jako by před ní byl bit s hodnotou 1. Tím pádem se snižuje paměťová náročnost o jeden bit na každé floating point číslo a nepatrně se zrychlí i operace s tímto datovým typem.

Nutno také zmínit, že rychlost výpočtů v navrhovaném modulu značně závisí na implementaci knihovny v pevné řádové čárce vzhledem k tomu, že používá ve velké míře operace a datovými typy, které jsou v ní definovány. Momentálně je implementace odladěna při použití knihovny codasip_int. Pro optimální výkon bude tuto knihovnu nutno předělat nebo nahradit, protože zatím umožňuje alokovat paměť jen po celých bytech, což se neslučuje s implementací libovolné šířky pro floating point datové typy.

V tabulce 7.1 lze vidět, že knihovna je rychlejší po stránce kompilace jak ukazuje tabulka: 7.1. Toho bylo docíleno tak, že knihovna obsahuje pouze nutné operace, které byli vyžadovány a žádné redundantní prvky. Rychlost kompilace byla také optimalizována implementací knihovny pouze do jednoho zdrojového souboru a omezením počtu includovaných knihoven, což urychlí linkování během překladu.

7.3 Srovnávací test doby kompilace

Jako testovací prostředí pro měření doby kompilace byl zvolen operační systém ubuntu 14, běžící na osobním počítači. Metodika testování je následující. Pomocí programu Cmake byl vytvořen Makefile, pomocí něhož byly následně kompilovány zdrojové kódy příkazem make. Součástí kompilace byly všechny implementované unit testy, kdy byly testovány dvě varianty. Jedna varianta používá k výpočtu mnou navženou knihovnu pro floating point aritmetiku, která jako backend používá knihovnu codasip_int. Druhou variantou byla kompilace unit testů se stejnými výpočty, ale za použití knihovny Boost multiprecision [11]. Měření doby kompilace bylo provedeno programem time, který je přítomen v operačním systému ubuntu 14. Bylo provedeno deset měření jejichž výsledky jsou shrnuty v tabulce 7.1.

Tabulka 7.1: Výsledky měření doby kompilace

Knihovna	Codasip float	Boost multiprecision
	1m 5,4s	1m 11,3s
	1m 6,1s	1m 15,6s
	1m 7,6s	1m 14,3s
	1m 6,3s	1m 16,1s
	1m 6,9s	1m 12,2s
	1m 5,5s	1m 11,9s
	1m 4,8s	1m 18s
	1m 8,4s	1m 16,1s
	1m 8,7s	1m 13,2s
	1m 7,6s	1m 12,8s
Průměr	1m 6,7s s	1m 14,2s

7.4 Srovnávací test doby výpočtu

Metodika testování doby výpočtu je následující. Jako testovací stoj je použit notebook s operačním systémem Windows 10, testuje se v prostředí Visual studio 15, community edition. K měření doby výpočtu je použit testovací framework Gtest [6]. Tento framework umožňuje spuštění jednotlivých scénářů, u kterých následně ověřuje správnost výpočtu vůči referenčnímu řešení, ale také měří jaká doba byla nutná k výpočtu.

Operace sčítání a násobení mají pro nás nejvyšší prioritu, jelikož vyvíjená knihovna bude použita například při výpočtech strojového učení kde jsou tyto operace nejčastější. Tudíž jsem vybral tyto operace k testování doby výpočtu.

Test byl proveden pro 32 a 64 bitové typy. Význam sloupců v následujících tabulkách 7.2, 7.3, 7.4, 7.5 popisujících výsledky je následující: Float 32 bit a Double 64 bit reprezentují standardní typy jazyka C++. Boost 32 bit a Boost 64 bit reprezentují výpočet sčítání provedený knihovnou boost multiprecision. [11]. Datové typy Codasip 32 bit a Codasip 64 bit reprezentují mnou navrženou knihovnu.

7.4.1 Operace sčítání

Tabulka 7.2: Výsledky měření doby výpočtu operace sčítání na 32 bitech

Datový typ	Float 32bit	Boost 32 bit	Codasip 32bit
	13 ms	121 ms	389 ms
	13 ms	116 ms	743 ms
	13 ms	91 ms	476 ms
	13 ms	79 ms	431 ms
	14 ms	109 ms	475 ms
	12 ms	99 ms	394 ms
	14 ms	118 ms	418 ms
	21 ms	75 ms	417 ms
	15 ms	83 ms	399 ms
	12 ms	128 ms	748 ms
Průměr	14 ms	102 ms	489 ms

Tabulka 7.3: Výsledky měření doby výpočtu operace sčítání na 64 bitech

Datový typ	Double 64bit	Boost 64 bit	Codasip 64bit
	15 ms	74 ms	986 ms
	14 ms	72 ms	574 ms
	12 ms	82 ms	638 ms
	26 ms	71 ms	742 ms
	12 ms	75 ms	631 ms
	25 ms	84 ms	577 ms
	26 ms	78 ms	655 ms
	13 ms	77 ms	1030 ms
	21 ms	80 ms	806 ms
	12 ms	80 ms	933 ms
Průměr	18 ms	77 ms	757 ms

7.4.2 Operace násobení

Operace násobení byla testována a měřena na 100 000 iteracích. Tento test byl proveden desetkrát a byly spočítány průměrné hodnoty. Tyto hodnoty jsou vidět v tabulkách: 7.4 a 7.5.

Tabulka 7.4: Výsledky měření doby výpočtu operace násobení na 32 bitech

Datový typ	Float 32 bit	Boost 32bit	Codasip 32bit
	16 ms	11 ms	116 ms
	18 ms	11 ms	122 ms
	19 ms	11 ms	134 ms
	21 ms	11 ms	129 ms
	15 ms	10 ms	124 ms
	32 ms	11 ms	143 ms
	17 ms	11 ms	135 ms
	18 ms	13 ms	130 ms
	20 ms	14 ms	135 ms
	22 ms	12 ms	134 ms
Průměr	20 ms	12 ms	131 ms

Tabulka 7.5: Výsledky měření doby výpočtu operace násobení na 64 bitech

Datový typ	Double 64bit	Boost 64 bit	Codasip 64bit
	24 ms	14 ms	117 ms
	19 ms	12 ms	130 ms
	25 ms	16 ms	128 ms
	28 ms	18 ms	138 ms
	30 ms	18 ms	153 ms
	36 ms	11 ms	140 ms
	13 ms	11 ms	152 ms
	22 ms	16 ms	130 ms
	17 ms	11 ms	135 ms
	20 ms	17 ms	138 ms
Průměr	24 ms	15 ms	136 ms

Kapitola 8

Závěr

Cílem této práce bylo vytvořit, modul pro práci s čísly v pohyblivé řádové čárce. Modul byl vyvinut tak, aby mohl pracovat s datovými typy s libovolnou bitovou šířkou. Tato bitová šířka je definována dvěma parametry, těmi jsou bitová šířka mantisy a bitová šířka exponentu. Tyto parametry mohou mít libovolnou bez-znaménkovou hodnotou a jsou v zásadě omezeny pouze implementací modulu pro čísla v pevné řádové čárce s libovolnou přesností, kterou navrhovaný modul využívá. Tento modul byl vytvářen ve spolupráci s firmou Codasip a bude součástí jejich platformy pro návrhu procesorů. Je tedy úzce propojen s jejich implementací čísel v pevné řádové čárce, která je momentálně omezena na násobky osmi bitů. Zároveň také musí dodržovat určité parametry, pro kompatibilitu s jejich prostředím a přizpůsobit rozhraní.
Modul byl implementován v jazyce C++. Generických datových typů libovolné šířky bylo dosaženo za pomoci šablonových tříd, jež jsou součástí jazyka C++. Šablonovým třídám a jejich použití je v práci věnována celá kapitola. Jako součást práce jsem také nastudoval a zpracoval potřebné znalosti a materiály týkající se standardu pro aritmetiku v plovoucí řadové čárce, jež zahrnují například zpracování výjimek, různé zaokrouhlovací módy a v neposlední řadě algoritmy, provádějící standardní operace s těmito čísly. Mým hlavním úkolem bylo upravit tyto algoritmy tak, aby pracovali s libovolnou bitovou šířkou. Tyto algoritmy jsou popsány v kapitole: Návrh a implementace algoritmů aritmetických operací.
Dále byly implementovány algoritmy konverzí na různé datové typy, ať už standardní floating point typy jako float a double nebo mnou implementované typy o různých bitových šířkách. Mezi navrženými algoritmy je i konverze na decimální soustavu tak, aby bylo možné vypisovat data na standardní výstup, popřípadě konverze z decimálního vstupu.
Nad navrženou knihovnou byly implementovány jednotkové testy, které testují jednotlivé operace a ověřují jejich výsledky vůči referenčním hodnotám. Jsou zaměřeny převážně na testování mezních případů, jako například počítání s nekonečnem nebo hodnotou NaN. Popřípadě operací které nejsou definovány, jako například dělení nulou. Součástí testů je také vygenerování náhodných hodnot, které slouží jako operandy aritmetickým operacím. S těmito operandy se pak počítá jak v referenčních řešeních, tak v navržené knihovně a výsledky se porovnávají. Samostatnou kapitolu tvoří srovnání s konkurenčními řešeními, které řeší danou problematiku. Bylo provedeno srovnání doby výpočtu a doby překladu. Doba výpočtu bohužel dopadla hůře, než konkurenční řešení a bude potřeba provést další optimalizace. Z hlediska doby překladu dopadl mnou navržený modul lépe, než konkurenční řešení.

Všechny body zadání jsem splnil a z práce si odnesl spoustu zkušeností ohledně aritmetiky v plovoucí řádové čárce a tvorby šablonových tříd.

Jako další pokračování v této práci je možné implementovat složitější matematické operace jako například sinus a cosinus. Bude také zapotřebí hledat další optimalizace pro zrychlení operací, kde se bude nutné zaměřit také na implementaci knihovny pro pevnou řadovou čárku v libovolné bitové šířce.

Literatura

[1] BOOST Software license. https://www.boost.org/users/license.html, accessed: 2019-05-11.

[2] GNU GPL v2. https://www.gnu.org/licenses/gpl-2.0.html, accessed: 2019-05-12.

[3] GNU LGPL v3. https://www.gnu.org/licenses/lgpl.html, accessed: 2019-05-12.

[4] *IEEE standard for floating-point arithmetic*. New York, NY: Institute of Electrical and Electronics Engineers, 2008, ISBN 978-0-7381-5752-8.

[5] Alexandrescu, A.: *Modern C++ design : generic programming and design patterns applied*. Boston, MA: Addison-Wesley, 2001, ISBN 978-0201704310.

[6] Civil, G.: Google test documentation. 2019.
URL https://github.com/google/googletest

[7] Goldberg, D.: What Every Computer Scientist Should Know About Floating Point Arithmetic. *ACM Computing Surveys*, 1991.

[8] Hain, T. F.; Mercer, D. B.: Fast Floating Point Square Root. 2005.

[9] Knuth, D. E.: *Umění programování 2.díl Seminumerické algoritmy*. Computer Press, a.s., 2010, ISBN 978-80-251-2898-5.

[10] Liu, D.: *Embedded DSP processor design : application specific instruction set processors*. Amsterdam Boston: Morgan Kaufmann/Elsevier, 2008, ISBN 978-0-12-374123-3.

[11] Maddock, J.; Kormanyos, C.: Boost.Multiprecision. 2018.
URL https://www.boost.org/doc/libs/1_66_0/libs/multiprecision/doc/html/index.html

[12] Muller, J. M.: *Handbook of floating-point arithmetic*. Boston: Birkhaauser, 2010, ISBN 978-0-8176-4704-9.

[13] Osherove, R.: *The art of unit testing : with examples in C*. Shelter Island, NY: Manning Publications, 2014, ISBN 978-1-617290-89-3.

[14] Overton, M.: *Numerical computing with IEEE floating point arithmetic : including one theorem, one rule of thumb, and one hundred and one exercises*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2001, ISBN 0-89871-571-7.

[15] Vandevoorde, D.: *C++ templates : the complete guide*. Boston: Addison-Wesley, 2018, ISBN 978-0321714121.

Přílohy

Příloha A

Obsah CD

Přiložené CD obsahuje:

- Zdrojové soubory této práce ve formátu LaTeX.
- Text práce ve formátu PDF.
- Zdrojové soubory knihovny pro práci s čísly v pohyblivé řadové čárce s libovolnou přesností
- Zdrojové soubory jednotlivých jednotkových testů
- Zdrojové soubory testovacího frameworku gtest