FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Realizace základních matematických funkcí s pomocí hardware
Student:	Jan Brokeš
Vedoucí:	Ing. Pavel Kubalík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1) Prozkoumejte existující řešení implementace základních matematických funkcí v jazyce VHDL.
2) Zaměřte se zejména na funkce goniometrické, logaritmické a druhou odmocninu.
3) Vytvořte model výpočtu těchto matematických funkci v programu Wolfram Mathematica.
4) Výsledné modely výpočtu matematických funkci převeďte do jazyka VHDL.
5) Pro vytvořené řešení vytvořte několik ukázek v jazyce VHDL.
6) Výsledné řešení řádně otestujte.
7) Při návrhu berte v úvahu následné nasazení materiálu ve výuce.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 7. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Realizace základních matematických funkcí s pomocí hardware

Jan Brokeš

Katedra číslicového návrhu
Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

16. května 2019

Poděkování

Děkuji především mému vedoucímu Ing. Pavlu Kubalíkovi, Ph.D. za jeho trpělivost, rady a vedení. Dále děkuji mým rodičům, Jarmile Brokešové a Milošovi Brokešovi, za podporu ve studiu i v životě. Poslední díky patří mé partnerce Kláře Opatové nejen za motivaci a kontrolu chyb.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2019 ……

Odkaz na tuto práci

Brokeš, Jan. *Realizace základních matematických funkcí s pomocí hardware.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Předmětem práce je efektivní výpočet matematických funkcí (logaritmus, druhá odmocnina, goniometrické funkce) v FPGA s následným využitím ve výuce FIT ČVUT v Praze. V práci jsou vytvořeny modely algoritmů s podrobným vysvětlením jejich principu v programu Wolfram Mathematica. Také je vytvořeno několik ukázek jejich korektní funkce, přesnosti a rychlosti. Tyto modely jsou implementovány v jazyce VHDL a simulují se příklady pro ověření správnosti. Výsledkem práce jsou přehledné a kvalitní kódy v jazyce VHDL a Mathematica vhodné pro výuku.

Klíčová slova výpočet matematických funkcí, realizace pomocí hardware, výuka FIT ČVUT, FPGA, Newtonova metoda, algoritmus CORDIC, Wolfram Mathematica, VHDL

Abstract

Goal of this thesis is an efficient calculation of mathematical functions (logarithm, square root, goniometric functions) inside an FPGA with subsequent usage in the education at the Faculty of Information technology of CTU in Prague. A part of the thesis includes programming models of the algorithms with a detailed explanation of their function in a program Wolfram Mathematica. Several samples of precision, speed, and correct behavior are created. These models are implemented in VHDL language, and examples are simulated to verify correctness. The results of the thesis are clear and quality code in VHDL and Mathematica languages suitable for use in education.

Keywords computation of mathematical functions, realization using hardware, education FIT CTU, FPGA, Newton's methond, CORDIC algorithm, Wolfram Mathematica, VHDL

Obsah

Úvod **1**

1 Cíl práce **3**

2 Rešerše **5**
2.1 Algoritmus CORDIC 5
2.2 Výpočet druhé odmocniny 5
2.3 Výpočet logaritmu 6

3 Analýza **7**
3.1 Způsoby ukládání čísel v digitálních systémech 7
3.2 Formát pohyblivé řádové čárky a sčítání/odčítání/násobení v tomto formátu 10
3.3 Wolfram Mathematica 11
3.4 Jazyk VHDL 11
3.5 Algoritmus CORDIC 12
3.6 Výpočet druhé odmocniny využívající Newtonovu metodu . . . 13
3.7 Výpočet logaritmu 15

4 Specifikace **17**

5 Návrh řešení **19**
5.1 Wolfram Mathematica 19
5.2 VHDL 20

6 Realizace **23**
6.1 Wolfram Mathematica 23
6.2 VHDL 25

7 Testování **37**

7.1 Wolfram Mathematica 37
7.2 VHDL 38

Závěr **43**

Bibliografie **45**

A Seznam použitých zkratek **47**

B Obsah přiloženého CD **49**

Seznam obrázků

3.1 Reprezentace čísla obecně 8
3.2 Číselná osa v přímém kódu 8
3.3 Číselná osa v doplňkovém kódu 9
3.4 Grafická ukázka funkce Newtonovy metody 14

5.1 Rozhraní entity pro sčítání 20
5.2 Blokové schéma implementace matematických funkcí 21

6.1 Získání nového úhlu v jedné iteraci CORDIC 30
6.2 Výpočet nové hodnoty Y v entitě CORDIC 31
6.3 Získání výstupu v entitě CORDIC v závislosti na vstupu 31
6.4 Vnitřní zapojení entit a procesů pro výpočet druhé odmocniny . . 33
6.5 Výpočet jedné iterace logaritmu 35
6.6 Výpočet výstupu logaritmu 36

Seznam tabulek

6.1 Tabulka pro určení sigma 25

7.1 Výsledky entity add 39
7.2 Výsledky entity sub 39
7.3 Výsledky entity mul 40
7.4 Výsledky entity CORDIC 41
7.5 Výsledky entity sqrt 41
7.6 Výsledky entity logarithm 42

Seznam výpisu kódu

6.1 Rozhraní entity add 26
6.2 Výpočet úhlu v entitě cordic 29

Úvod

Na magisterském studiu Fakulty informačních technologií ČVUT v Praze probíhá výuka předmětu Počítačové aritmetika pro studenty oboru Návrh a programování vestavných systémů. Někteří studenti mají problém s učením některých algoritmů z přednášek.

Z tohoto důvodu se práce zabývá přehlednou implementací některých problémových algoritmů z předmětu Počítačové aritmetika. Výsledek této práce pomůže vyučujícím i studentům s výukou a studiem předmětu.

Téma jsem si vybral z důvodů usnadnění studia a zlepšení výuky.

Práce je zaměřena na vytvoření modelů algoritmů pro rychlý výpočet goniometrických funkcí, druhé odmocniny a logaritmu v programu Wolfram Mathematica a následnou hardwarovou implementaci v jazyce VHDL.

Tato práce je rozdělena do dvou hlavních částí: analytické (jedná se o kapitoly rešerše a analýza) a praktické (specifikace, návrh řešení, realizace a testování). V rešerši jsou analyzovány existující řešení, v analýze se zjednodušuje implementace a zjišťuje její náročnost, specifikace upřesňuje nejasné detaily zadání, návrh řešení nastíní jakým způsobem bude udělána realizace, realizace popisuje praktické programování Wolfram Mathematica a implementaci do VHDL kódu, testování ověřuje správnost realizace.

Kapitola 1

Cíl práce

Hlavním cílem této práce je implementování matematických funkcí, konkrétně goniometrické funkce, výpočet druhé odmocniny a logaritmu, v jazyce VHDL za účelem využití zdrojových kódů ve výuce dalších studentů předmětu počítačová aritmetika na fakultě informačních technologií ČVUT.

Cílem analytické části (kapitoly rešerše a analýza) je zjednodušení problému, odvození teoretického řešení pro výpočet daných matematických funkcí a porovnání používaných řešení a rozhodnutí o jejich vhodnosti pro výuku ve výše uvedeném předmětu.

Cílem praktické části (kapitoly specifikace, návrh řešení, realizace, testování) je vytvoření modelů zvolených algoritmů v programu Wolfram Mathematica a zároveň podrobnější vysvětlení funkce těchto algoritmů. Na to navazuje implementace modelů v jazyce VHDL a vytvoření příkladů, které ukáží vlastnosti algoritmů a otestují správnost implementace.

Kapitola 2

Rešerše

Kapitola rešerše se zabývá hledáním stávajících řešení a jejich implementací. Dále podává důvody, proč není dané řešení vhodné pro tuto situaci.

2.1 Algoritmus CORDIC

Velmi populární algoritmus, na internetu je mnoho implementací v jazycích VHDL a C.

Existují open source implementace ve VHDL, například [1] nebo [2]. Kód zahrnuje vše potřebné pro tuto práci. Na druhou stranu je tento kód příliš rozsáhlý a obsahuje části, které nejsou vyžadované, a kvůli tomu není tak čitelný, jak by mohl být. Není vhodný pro vysvětlení základního principu algoritmu. Stejným způsobem je vytvořen CORDIC od firmy Xilinx (dokumentace [3]) integrovaný v programu Xilinx Vivado 2018.2.

Dále existují implementace v jazyce Verilog [4], který není na fakultě příliš populární, studenti častěji používají a více znají VHDL. Zmíněná implementace nevysvětluje podrobný princip funkce algoritmu, jeden z cílů této práce.

Implementace v jazyce C [5] nebo [6] pro potřeby této práce nejsou vhodné. Wolfram Mathematica lze využít ke grafickému znázornění se snadnou úpravou parametrů a větší interaktivitou.

2.2 Výpočet druhé odmocniny

Existuje několik způsobů výpočtu druhé odmocniny, předmětem této práce jsou pouze algoritmy využívající Newtonovu metodu pro výpočet. Z důvodů uvedených výše C není ideální jazyk pro tuto práci. Bohužel převážná část zdrojových kódů je napsána právě v jazyce C, například [7], [8] a [9].

2.3 Výpočet logaritmu

Pro účel této práce je nutné použít algoritmus, který je vyučován v předmětu Počítačové aritmetika. Tento algoritmus není často implementován ve VHDL ani v jiných jazycích. Pro získání hodnoty logaritmu se často používají tabulky v paměti (lookup table, například [10]). Kvalitní a přehledná implementace ve VHDL ani v Mathematica není snadno dostupná.

Kapitola 3

Analýza

Následující kapitola vysvětluje základní předpoklady a termíny, dále rozebírá problém výpočtu matematických funkcí(goniometrických funkcí, druhé odmocniny a logaritmu), hledá efektivní a rychlé řešení a dokazuje jeho správnost.

3.1 Způsoby ukládání čísel v digitálních systémech

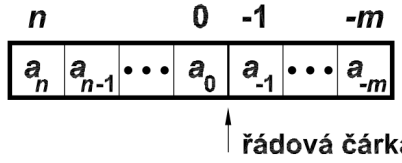
Desítková číselná soustava funguje na principu, že každá pozice v čísle odpovídá jedné mocnině 10 a číslice na dané pozici označuje počet mocnin 10. Nejlépe vysvětleno na příkladu:
Desítková soustava:
$951 = 900 + 50 + 1 = 9 * 10^2 + 5 * 10^1 + 1 * 10^0$
V desítkové soustavě je 10 základ soustavy, tj. číslo s jehož mocninami se počítá.

Nemusí se ale jednat pouze o mocniny 10. V některých situacích je vhodné jako základ použít například 2 nebo 16. Příklad ve binární soustavě (se základem 2):
$1100_2 = 1000_2 + 100_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 8 + 4 = 12$
Obecně lze vidět na obrázku 3.1, převzato z [11].

V digitálních, také nazývaných číslicových, systémech se v dnešní době používá z převážné většiny dvojková, neboli binární, soustava. Jednotlivé číslice v této soustavě se nazývají bity a „desetinná“ čárka se označuje řádová čárka. Nejnižší bit (pozice nejvíce vpravo před řádovou čárkou) v této soustavě se nazývá 0. bit, o jednu pozici vlevo následuje 1. bit, dále vlevo je 2. bit, atd. Analogicky vpravo od 0. bitu se nachází -1. bit, pokračuje -2 bit.

Pro ukládání a práci s čísly v digitálních systémech existují čtyři základní formáty reprezentace dat.

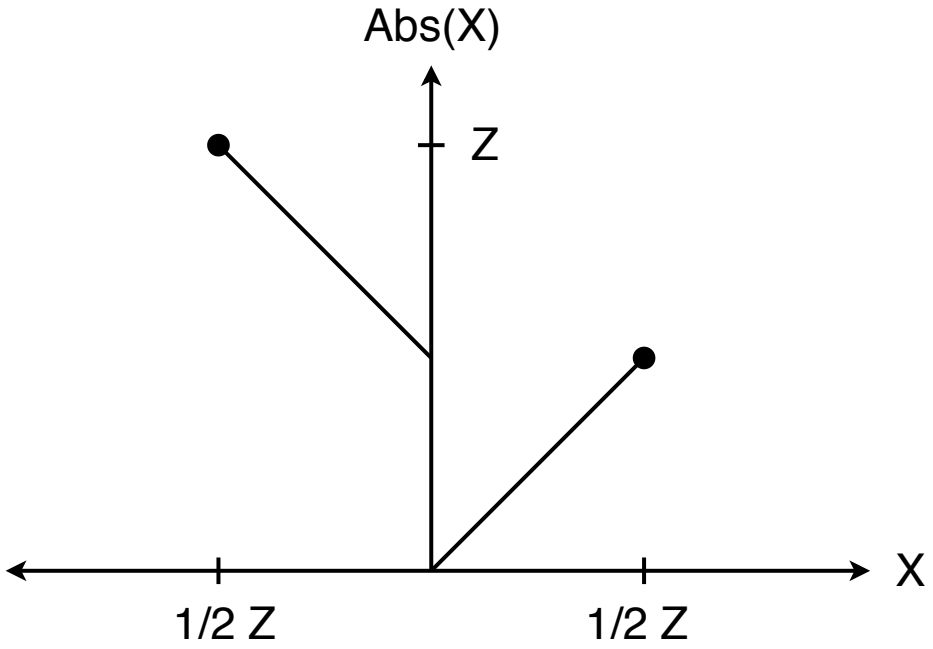
Následující dva jsou pro ukládání celých čísel.



n ... nejvyšší řád
$-m$ ... nejnižší řád

$$A \sim a_n a_{n-1} \dots a_0, a_{-1} \dots a_{-m}$$
$$A = a_n z^n + a_{n-1} z^{n-1} + \dots + a_0 + a_{-1} z^{-1} \dots a_{-m} z^{-m}$$
z ... základ číselné soustavy

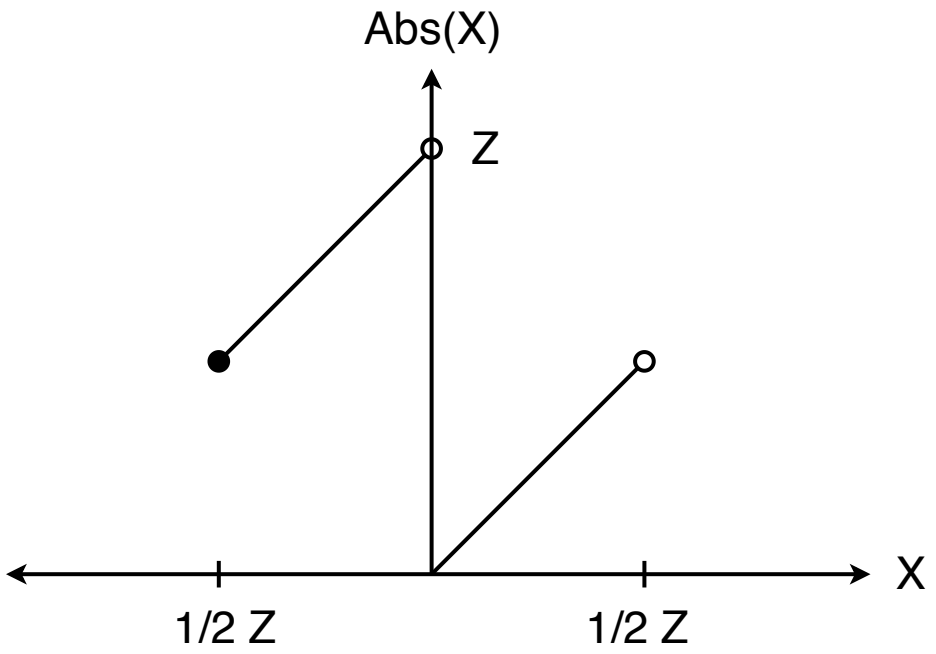
Obrázek 3.1: Reprezentace čísla obecně



Obrázek 3.2: Číselná osa v přímém kódu

Přímý kód je velmi podobný zapisování čísel v desítkové soustavě v běžném životě. Každé číslo je zaznamenáno ve dvojkové soustavě s jednou pozicí navíc pro jeho znaménko. Použití tohoto formátu má za následky dvojí zápis nuly (kladná a záporná), náročné sčítání a odčítání – odčítání vyžaduje zcela jiný postup a hardware než sčítání, porovnávání se realizuje jinak u kladných a záporných čísel. Zvláštní vlastnosti lze vidět na obrázku 3.1, inspirováno prezentací [12].

Doplňkový kód pro kladná čísla vypadá stejně jako přímý. Liší se v zápisu záporných. Nejvyšší záporné číslo (-1) se reprezentuje samými



Obrázek 3.3: Číselná osa v doplňkovém kódu

1 (11...11). Naopak nejnižší číslo začíná 1 a následují pouze 0 (100...00). Díky vlastnosti přetečení (overflow) se sčítání realizuje stejným obvodem jako odčítání. Například přičtení samých 1 (11...11) je ekvivalentní odečtení čísla 1. Získání absolutní hodnoty záporného čísla se provede bitovou negací a přičtením 1. Tato operace je náročnější než v přímém kódu. Pro porovnání stačí pouze negovat první bit. Lepší vlastnosti jsou naznačeny na obrázku 3.2, vytvořeno podle přednášky [12].

Další dva formáty jsou používány pro racionální čísla.

Pevná řádová čárka funguje stejně jako doplňkový kód (může také vycházet z přímého kódu), s tím rozdílem, že se jednotlivé bity interpretují s jinou hodnotou. Nejnižší bit již není 2^0, ale reprezentuje 2^{-n}, kde n je počet bitů vpravo od řádové čárky, n se liší v závislosti na implementaci a potřebné přesnosti.

Pohyblivá řádová čárka je popsána v následující kapitole.

3.2 Formát pohyblivé řádové čárky a sčítání/odčítání/násobení v tomto formátu

3.2.1 Popis formátu

Čísla ve formátu pohyblivé řádové čárky se skládají ze dvou složek:

- mantisa
- exponent

Mantisa je základ čísla, ze standardního formátu se získá posunutím první jedničky na 0. bit.

Exponent označuje mocninu, na kterou je nutné umocnit základ soustavy aby se vynásobením mantisy získalo původní číslo v přímém kódu.
Zapsáno vzorcem: $X_{float} = mantisa_X * 2^{exponent_X}$.

Příklad zápisu čísla 24:
$24_{10} = 11000_2 = 1.1_2 * 2^4 = 1.5_{10} * 2^4 = 1.5_{10} * 16_{10}$

3.2.2 Sčítání

K provedení sčítání je nutné, aby bity na stejných pozicích reprezentovaly stejnou hodnotu. Toho lze dosáhnout posunutím čísla s nižším exponentem doprava (respektive posunout operand s vyšším exponentem doleva) o tolik pozic, o kolik je exponent menší (resp. větší). Potom lze toto číslo sečíst se zbylým operandem použitím stejné operace jako pro doplňkový kód.

Při sčítání může nastat přenos, potom se výsledek sčítání musí posunout o jednu pozici doprava a nový exponent se rovná většímu z exponentů +1. Pokud přenos nenastane, výsledek sčítání je výsledek celé operace a novým exponentem se stane větší z exponentů operandů.

3.2.3 Odčítání

Odčítání začíná stejně jako sčítání, hodnoty bitů se pomocí binárních posunů vyrovnají. Poté se provede odčítání v doplňkovém kódu.

Pokud je druhý operand větší, výsledek bude záporný v doplňkovém kódu. Z toho se musí získat absolutní hodnota a znaménko zapsat zvlášť. V tuto chvíli může mít výsledek první 1 na jakékoliv pozici, proto se musí ověřit všechny pozice, následně se číslo posune o odpovídající počet a tím vznikne mantisa výsledku. Exponent se získá z většího exponentu upraveného podle posunu mantisy.

3.2.4 Násobení

Násobení čísel ve formátu pohyblivé řádové čárky probíhá jednodušeji než sčítání a odčítání, pro výpočet není potřeba binárních posunů.

$$A_{float} * B_{float} = mantisa_A * 2^{exponentA} * mantisa_B * 2^{exponentB}$$
$$A_{float} * B_{float} = mantisa_A * mantisa_B * 2^{exponentA} * 2^{exponentB}$$
$$A_{float} * B_{float} = mantisa_A * mantisa_B * 2^{exponentA+exponentB}$$

Komutativnost násobení a jeho další vlastnosti umožňují nezávisle na exponentech vynásobit mantisy. Pro získání výsledného exponentu se sečtou exponenty operandů. Opět může nastat přenos do vyššího řádu, proto je k získání platného výsledku nutné ho zkontrolovat a případně ošetřit stejným způsobem jako v případě sčítání.

3.3 Wolfram Mathematica

Program Wolfram Mathematica byl zvolen z několika důvodů. Studenti ho většinou znají, pracuje se s ním v předmětu Číslicové a analogové obvody (BI-CAO) v prvním ročníku bakalářského studia. Obsahuje grafické nástroje vhodné pro tvorbu grafů, které pomáhají s vizualizací řešení. Kód a komentáře jsou mezi sebou, to je vhodné pro matematické zápisy a rovnice, ale také může zhoršit čitelnost, tento problém lze odstranit funkcemi pro výpis.

3.4 Jazyk VHDL

Jazyk VHDL byl vybrán, protože se vyučuje celý semestr v předmětu Praktika v návrhu číslicových obvodů (BI-PNO). Alternativou by byl jazyk Verilog, který se učil pouze část semestru v předmětu Architektury počítačových systémů (BI-APS). Předpokládá se že studenti pravděpodobně absolvovali oba předměty. VHDL se věnovalo více času a mezi studenty je podle zkušeností oblíbenější.

3.4.1 VHDL knihovny

V práci jsem použil 2 knihovny:

std logic 1164 definuje několik symbolů popisující stavy, které mohou nastat na vodiči uvnitř digitálních obvodů. Tyto hodnoty pomáhají se simulací situací v digitálních systémech. Obsahuje například U pro neznámou hodnotu, 1 a 0 pro logické hodnoty 1 a 0 a X pro označení výsledku počítání s neznámou hodnotou.

numeric std definuje nové datové typy a jednoduché aritmetické operace využívající nové typy. V práci jsou využity operace sčítání a odčítání, spolu s operacemi definovanými v základním VHDL, tj. zřetězení, negace (NOT), logické násobení (AND) a logické sčítání (OR).

3.5 Algoritmus CORDIC

Algoritmus CORDIC (coordinate rotation digital computer, česky digitální počítač pro rotaci souřadnic) lze použít k vypočítání sinu a cosinu z polárních souřadnic.

Následující postup vytvořen podle [5] a [13].

Polární souřadnice počátečního bodu A se jmenují X_A a Y_A. Souřadnice cílového bodu B se označí X_B a Y_B. Cílový úhel lze zapsat jako součet počátečního úhlu (ϕ) a rozdílu úhlů (α).
$X_A = r * \cos(\phi)$ $\quad$ $X_B = r * \cos(\phi + \alpha)$
$Y_A = r * \sin(\phi)$ $\quad$ $Y_B = r * \sin(\phi + \alpha)$
Pro sinus a cosinus platí následující vztahy:
$\sin(\alpha + \beta) = \sin(\alpha) * \cos(\beta) + \cos(\alpha) * \sin(\beta)$
$\cos(\alpha + \beta) = \cos(\alpha) * \cos(\beta) - \sin(\alpha) * \sin(\beta)$
Po dosazení těchto vztahů vypadají definice polárních souřadnic bodu B následovně:
$X_B = r * (\cos\phi * \cos(\alpha) - \sin\phi * \sin(\alpha))$
$Y_B = r * (\sin\phi * \cos(\alpha) + \cos\phi * \sin(\alpha))$
Roznásobením závorky vznikne:
$X_B = (r * \cos\phi * \cos(\alpha) - r * \sin\phi * \sin(\alpha))$
$Y_B = (r * \sin\phi * \cos(\alpha) + r * \cos\phi * \sin(\alpha))$
V závorce se objeví polární souřadnice bodu A.
$X_B = X_A \cos(\alpha) - Y_A * \sin(\alpha)$
$Y_B = Y_A \cos(\alpha) + X_A * \sin(\alpha)$
Pro získání rovnice pro implementaci je nutné využít vztahu
$\sin(\alpha)/\cos(\alpha) = \mathrm{tg}(\alpha)$, tento zlomek lze získat vydělením $\cos(\alpha)$.
$X_B/\cos(\alpha) = X_A - Y_A \sin(\alpha)/\cos(\alpha)$
$Y_B/\cos(\alpha) = Y_A + X_A \sin(\alpha)/\cos(\alpha)$
Po dosazení $tg(\alpha)$:
$X_B/\cos(\alpha) = X_A - Y_A * \mathrm{tg}(\alpha)$
$Y_B/\cos(\alpha) = Y_A + X_A * \mathrm{tg}(\alpha)$
Další úprava je vynásobení $\cos(\alpha)$ pro získání osamostatněných X_B a Y_B na levé straně.
$X_B = \cos(\alpha) * (X_A - Y_A * \mathrm{tg}(\alpha))$
$Y_B = \cos(\alpha) * (Y_A + X_A * \mathrm{tg}(\alpha))$

V této fázi rovnice obsahují pouze dvě goniometrické funkce, navíc se stejnými úhly. Pro výpočet $\mathrm{tg}(\alpha)$ lze dosadit vhodné úhly α_i pro dosáhnutí $\mathrm{tg}(\alpha) = 2^{-i}$, což je vhodné pro výpočet v hardware. α lze rozložit na součty a rozdíly několika úhlů, které splňují $\mathrm{tg}(\alpha_i) = 2^{-i}$. K tomu stačí vypočítat $\mathrm{arctg}(2^{-n})$. Výsledky funkce se nemění a uloží do paměti ROM (read-only memory, lze měnit pouze v kódu před nahráním do FPGA). Hodnoty n začínají na 0. Maximální efektivní hodnota n nastane, když lze další hodnoty získat pouhým dělením dvěma. Za jak dlouho tato situace nastane závisí na přesnosti formátu ukládání čísel.

Nyní známé úhly α_i se využijí pro zjištění znaménka úhlu. $\sum_{\forall i} \alpha_i$ se musí rovnat celkovému úhlu α. Zadaný úhel bude z prvního nebo čtvrtého kvadrantu kartézské soustavy souřadnic (od $-90°$ do $90°$) . Součet α_i je větší než $90°$ a tedy pokryje celý první, případně čtvrtý kvadrant. Z toho vyplývá, že úhel α se dá složit sčítáním či odčítáním jednotlivých α_i. Začne se sčítáním α_i, a pokud hodnota součtu přesáhne α, další α_i se odečte. Podle operace se zjistí znaménko současného úhlu S_i. Pro každou iteraci bude platit tento vztah:
$X_{i+1} = K_i * (X_i - Y_i * S_i * 2^{-i})$
$Y_{i+1} = K_i * (Y_i + X_i * S_i * 2^{-i})$
Kde K_i označuje $\cos(\alpha_i)$

Násobení K_i v každé iteraci lze nahradit jedním násobením číslem K. Ukázka pro X_{i+2} s nahrazením $X_i + 1$ a Y_{i+1} z předchozí iterace jejich pravou stranou rovnice výše.
$X_{i+2} = K_{i+1} * \Big(K_i * (X_i - Y_i * S_i * 2^{-i}) - K_i * (Y_i + X_i * S_i * 2^{-i}) * S_{i+1} * 2^{-i+1}\Big)$
Ze závorky jde vytknout K_i a vznikne:
$X_{i+2} = K_{i+1} * K_i * \Big((X_i - Y_i * S_i * 2^{-i}) - (Y_i + X_i * S_i * 2^{-i}) * S_{i+1} * 2^{-i+1}\Big)$
Z toho je vidět, že K_i se násobí nezávisle na hodnotách X_i, Y_i a S_i. Navíc K_i je konstantní, protože hodnoty jednotlivých α_i se nemění, pouze jejich znaménka. Díky tomu lze pro zjednodušení výpočtu všechna vynásobit K_i mezi sebou.
$K = \prod_i K_i = \cos(\alpha_1) * \cos(\alpha_2) * \cos(\alpha_3) \ldots$
Tím vznikne K závislé pouze na počtu iterací, který je znám při implementaci a násobení konstantním K je možné integrovat přímo do návrhu a jedna iterace se zjednoduší na:
$X_{i+1} = X_i - Y_i * S_i * 2^{-i}$
$Y_{i+1} = Y_i + X_i * S_i * 2^{-i}$

Tyto rovnice se implementují a vypočítají v jedné iteraci.

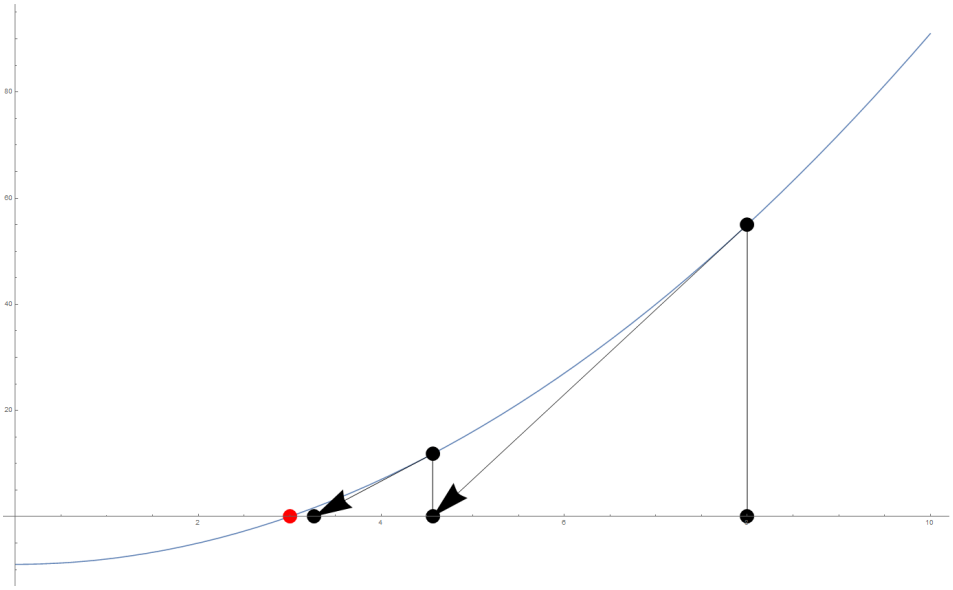
3.6 Výpočet druhé odmocniny využívající Newtonovu metodu

3.6.1 Newtonova metoda

Sekce se zakládá na prezentaci [13].

Newtonova metoda dokáže velmi rychle vyhledat kořen funkce, pokud je splněno několik nutných předpokladů:

- existence a znalost derivace
- vhodný odhad počáteční hodnoty
- funkce je spojitá, hladká a monotónní



Obrázek 3.4: Grafická ukázka funkce Newtonovy metody

Derivaci je nutné znát, protože se využívá ve vzorci pro výpočet. Některé funkce se za použití této metody oddalují od jejich kořene, pokud je počáteční hodnota příliš vzdálená. Když by funkce nebyla spojitá a hladká (= její derivace by nebyla spojitá), ve výpočtu derivace by se mohlo objevit nekonečno. Monotónnost zaručuje neexistenci lokálního minima, kde by se hodnoty mohly začít opakovat přiblížení ke kořenu.

Za daných předpokladů lze využít vztah $x_{i+1} = x_i - f(x_i)/f'(x_i)$, který přiblíží hodnotu x ke kořenu funkce. K tomu se využije tečna ke grafu funkce v bodě x_i. Bod, kde tečna protne osu X, se označí x_{i+1}. Díky požadovaným vlastnostem se x_{i+1} vždy přiblíží kořenu, při omezené přesnosti se mu po několika iteracích začne rovnat [13]. Tento proces je znázorněn na obrázku 3.4 – červený bod označuje kořen, šipky naznačují tečny, body na ose X reprezentují odhady.

3.6.2 Druhá odmocnina

Pro výpočet druhé odmocniny stačí funkce $f(x) = x^2 - A$. Derivaci lze snadno spočítat, přijatelný odhad se získá jednoduše díky formátu pohyblivé řádové čárky a funkce má všechny vlastnosti pro třetí nutnou podmínku [13].

Derivace funkce vypadá následovně: $f'(x) = 2x$
Funkce i její derivace se dosadí do vzorce Newtonovy metody:
$x_{i+1} = x_i - \frac{x_i^2 - A}{2x_i}$
Jedna polovina se vytkne:
$\frac{1}{2}\left(2x_i - \frac{x_i^2 - A}{x_i}\right)$

$2x_i$ se přenese do čitatele zlomku:
$\frac{1}{2}\Big(-\frac{-2x_i^2+x_i^2-A}{x_i}\Big) = \frac{1}{2}\Big(\frac{2x_i^2-x_i^2+A}{x_i}\Big)$
Odečte se x_i^2:
$\frac{1}{2}\Big(\frac{x_i^2+A}{x_i}\Big) = \frac{1}{2}\Big(\frac{x_i^2}{x_i}+\frac{A}{x_i}\Big) = \frac{1}{2}\Big(x_i+\frac{A}{x_i}\Big)$

Použití funkce $f(x) = x^2 - A$ by tedy vyžadovalo dělení při výpočtu v hardware. To je velmi náročné jak na využité prostředky, tak především na hodinové cykly. Více optimální řešení se zakládá na multiplikativní inverzi odmocniny.

3.6.3 Inverze druhé odmocniny

$f(x) = \frac{1}{x^2} - A$ splňuje všechny předpoklady pro použití Newtonovy metody [13]. Kořen funkce leží v bodě $1/\sqrt{A}$. Pro získání $\sqrt{A}$ stačí výsledek funkce vynásobit A ($1/\sqrt{A} * A = \sqrt{A}$).

Derivace se vypočítá snadno:
$f'(x) = -\frac{2}{x^3}$

Vzorec Newtonovy metody pro tuto funkci vypadá následovně:
$x_{i+1} = x_i - \frac{1/x^2-A}{-2/x^3} = x_i + \frac{1/x^2-A}{2/x^3}$
V dalším kroku se čitatel vynásobí jmenovatelem a poté se zkrátí zlomek.
$x_i + \frac{x_i^3}{2x_i^2} - A * \frac{x_i^3}{2} = x + \frac{x_i}{2} - A * \frac{x_i^3}{2}$
Na konec se vytkne $\frac{x_i}{2}$ před závorku.
$\frac{x_i}{2} * (2 + 1 - A * x_i^2) = \frac{x_i}{2} * (3 - A * x_i^2)$

Jak je možné vidět z výsledné rovnice, tato možnost využívá pouze násobení ($x_i^2 = x_i * x_i$), odčítání a dělení 2, realizovatelné bitovým posunem nebo snížením exponentu. Každou z těchto operací lze realizovat efektivně vzhledem k fyzickým prostředkům i hodinovým cyklům.

Jako odhad první hodnoty lze použít $\frac{1}{1^{A_{exp}/2}}$. Odvození:
$x = \frac{1}{\sqrt{A}} = \frac{1}{\sqrt{man_A * 2^{exp_A}}} = \frac{1}{\sqrt{man_A}} * \frac{1}{\sqrt{2^{exp_A}}}$
$x = \frac{1}{\sqrt{man_A}} * \frac{1}{2^{exp_A/2}} \approx \frac{1}{2^{exp_A/2}} = 2^{-exp_A/2}$

3.7 Výpočet logaritmu

Následující sekce rozvíjí přednášku [13].

Pokud se na vzorec formátu aplikuje logaritmus, jeho vlastnosti značně zjednoduší výpočet.
$A_{float} = mantisa_A * 2^{exponent_A}$
$\log_b A_{float} = \log_b mantisa_A * \log_b 2^{exponent_A}$
$\log_b A_{float} = \log_b mantisa_A + exponent_A * \log_b 2$

Za b se dosadí 2, v této práci se počítá logaritmus o základu 2.
$\log_2 A_{float} = \log_2 mantisa_A + exponent_A * \log_2 2$
Poslední člen $\log_2 2 = L$ se rovná 1 a tento člen je možné z rovnice vynechat.
$\log_2 A_{float} = \log_2 mantisa_A + exponent_A$
Exponent se přičte až na konci výpočtu a zbývající problém je vypočítat logaritmus mantisy. Dále bude $mantisa_A$ uváděno jako A.

Existuje nějaké číslo X, pro které platí $A * X = 1$. Toto číslo X se rozloží na několik čísel x_i.

$A * \prod_i x_i = 1$

Na tuto rovnici se použije přirozený logaritmus a tím se získá způsob jak vypočítat logaritmus A, pokud je známé x_i.
$\ln A + \sum_i \ln x_i = 0$
$\ln A = -\sum_i \ln x_i$
Dále stačí zajistit, aby hodnoty x_i splňovaly první z rovnic výše a hodnoty jejich logaritmů byly známé.

Hodnota A na počátku výpočtu je mezi 1 a 2, protože je to mantisa čísla. Pokud by za řádovou čárkou následovaly pouze nulové bity, číslo by se rovnalo 1, analogicky pokud by číslo začínalo 0 a pokračovalo jedničkovými bity za řádovou čárkou, bylo by rovno 1 (při neomezené přesnosti).

Ve výpočtu se budou čísla rovnat 1 do nějakého $-n$. bitu (n je přirozené číslo). Pokud jsou menší než 1 a vynásobí se číslem větším než 1, které se také rovná 1 do $-n$. bitu, výsledek bude roven 1 alespoň do $-(n+1)$. bitu [13].

Nejméně náročný způsob jak získat vhodné hodnoty pro násobení je přičtení $(+2^{-n+1})$ či odečtení (-2^{-n}) určitého bitu, tato hodnota se nazve sigma. Jaká operace se použije záleží na 0. bitu a i. bitu (i je rovno počtu iterací).

Výsledek součtu sigma s 1 se nazve x_i. Tím se vynásobí A, aby se A přiblížilo 1 a od Y se odečte $\ln x_i$, aby se zvýšila přesnost vypočítaného logaritmu.
$A_{i+1} = A_i * x_i$
$Y_{i+1} = Y_i - \ln x_i$
Tyto dvě rovnice jsou základem VHDL implementace.

Pro získání logaritmu o základu 2 ze známého přirozeného logaritmu se využije vzorec pro převádění logaritmů: [13]
$\log_b A = \log_a A * (\log_a b)^{-1}$
$\log_2 A = \ln A * (\ln 2)^{-1}$
Po vypočítání $\ln A$ se provede násobení konstantou $(\ln 2)^{-1}$ a tím se získá konečný výsledek $\log_2 A$

Kapitola 4

Specifikace

Dále jsou popsány přesné detaily mé práce.

V programu Wolfram Mathematica vytvořím algoritmus, který využívá pouze jednoduché operace, přímo implementovatelné ve VHDL.

VHDL kód bude zaměřený pouze na simulaci, ne na implementaci do FPGA. Kód budu simulovat v programu Vidado 2018.2 pomocí simulace chování (behavioral simulation).

Součástí práce v hardwarové realizaci nebude operace sčítání a odčítání (pro doplňkový kód), využiji vestavěná implementace ve VHDL knihovnách. Naopak součástí práce budou všechny složitější operace, například násobení.

Čísla v jazyce VHDL budu ukládat ve formátu pohyblivé řádové čárky s délkou 16 bitů, z toho prvních 10 bitů vyhradím pro mantisu a zbylých 6 bitů pro exponent. Mantisa uložím v přímém kódu bez znaménka, exponent v doplňkovém. Pokud bude potřeba znaménko, přidám jeden bit jako další signál.

Kapitola 5

Návrh řešení

V této kapitole uvádím obecný pohled na řešení výpočtu goniometrických funkcí, druhé odmocniny a logaritmu v programu Wolfram Mathematica a v jazyce VHDL.

5.1 Wolfram Mathematica

Řešení budu mít ve formě tří prezentací s několika stránkami. Pro každou funkci budu mít vlastní soubor (notebook).

Na prvních stránkách uvedu vzorce či rovnice, které jsou nutné k pochopení algoritmů a postupně vysvětlím princip jejich funkce. K tomu budu používat funkce v Mathematica pro potvrzení některých definic a výroků.

Na základě rovnic vytvořím první fungující algoritmus, který názorně ukáže funkci algoritmu a jeho přesnost v každé iteraci, za cenu přehlednosti samotného kódu nebo omezeného počtu iterací. Zde budu využívat vestavěné funkce pro zobrazení vizuálních pomůcek.

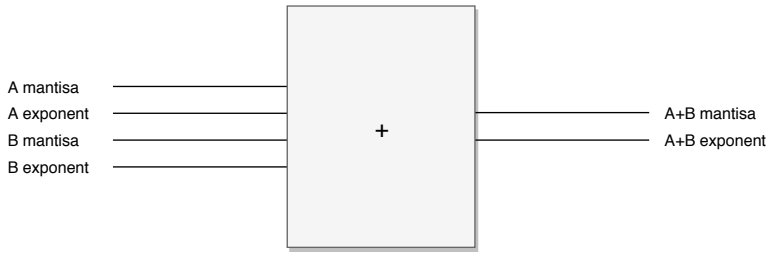
Na následující stránce vytvořím efektivní algoritmus, vhodný pro převedení do jazyka VHDL. Bude využívat pouze základní operace a funkce. Případné složitější operace či funkce budou napsané pomocí jednodušších v Mathematica tak, aby na konci bylo možné celý kód jednoduše převést do jazyka VHDL.

Funkce simulující implementaci potom porovnám s vestavěnými funkcemi z hlediska správnosti, přesnosti a rychlosti.

Dále popisuji jaké funkce budu v každém souboru vytvářet.

5.1.1 CORDIC

Goniometrické funkce jsou uvnitř programu, dalších funkcí budu potřebovat minimum. Vytvořím jednu funkci pro výpočet konstanty KN, kterou se vynásobí výsledek z algoritmu a druhou funkci pro určení znaménka čísla, aby byl zápis v Mathematica co nejvíce podobný vzorci $X_{i+1} = X_i - Y_i * S_i * 2^{-i}$.



Obrázek 5.1: Rozhraní entity pro sčítání

Pro samotný výpočet vytvořím jednu funkci, která bude průběžné výsledky ukládat do seznamu. Funkce vrací tento seznam spolu s posledními vypočtenými hodnotami.

5.1.2 Druhá odmocnina

Pro ilustraci ani pro výpočet iterace nebudu potřebovat žádné pomocné funkce. Postačí mi čtyři funkce, dvě pro každou z rovnic ($x^2 - A = 0$ a $\frac{1}{x^2} - A = 0$). Jednu z nich využiji pro ilustraci a druhou pro vytvoření kódu k implementaci.

Ve všech funkcích budu potřebovat číslo, pro které mám odmocninu vypočítat. Ve funkcích pro ilustraci vytvořím graf a na konci ho vrátím. Některé možnosti přizpůsobení grafu mohu předat uživateli pomocí parametrů. Efektivní funkce budu počítat na omezený počet iterací, ten bude nutné funkci předat.

5.1.3 Logaritmus

Budu si potřebovat vytvořit alespoň dvě pomocné funkce; první vrátí bit na určené pozici ze zadaného čísla a druhá určí sigma podle současné hodnoty a pořadí iterace.

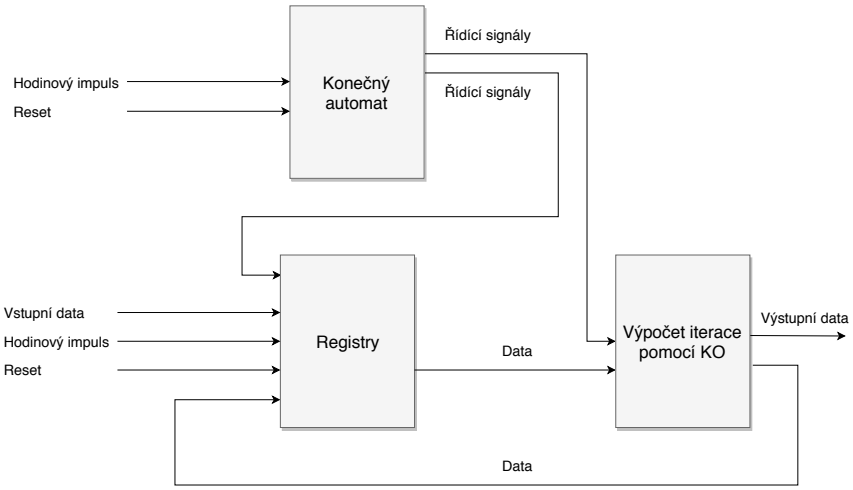
Dále vytvořím funkci, která vypočítá ze zadané hodnoty logaritmus. Stejně jako předchozí bude mít omezený počtu iterací.

5.2 VHDL

Pro každou hlavní funkci vytvořím vlastní entitu. Uvnitř těchto entit použiji několik společných menších entit pro výpočet základních operací.

5.2.1 Společné entity

Ve výpočtu matematickým funkcí budu potřebovat tři operace (sčítání, odčítání a násobení) v mé verzi formátu pohyblivé řádové čárky. Tyto operace budou využité ve více funkcích, proto je uvádím ve vlastní sekci.



Obrázek 5.2: Blokové schéma implementace matematických funkcí

Vytvořím tři entity, pro každou operaci jednu, se shodným rozhraním. Ve všech operacích ze dvou vstupních operandů vypočítám jednu výstupní hodnotu a každé z těchto tří čísel rozdělím na mantisu a exponent. Toto rozhraní je vidět na obrázku 5.1. Operace budu implementovat pouze s pomocí kombinačních obvodů bez registrů, takže by signály pro resetování nebo hodinový impuls byly zbytečné.

5.2.2 Hlavní entity

Implementace všech tří hlavních funkcí má společný základ znázorněný na obrázku 5.2. Pro řídící signály použiji konečný automat. V hlavní části budu mít čistě kombinační obvody pro výpočet nové iterace a registry k jejímu uložení.

5.2.2.1 CORDIC

Vstupní hodnota bude úhel, pro který mám vypočítat sinus a cosinus, spolu se signály pro reset a hodinový signál.

Uvnitř entity budu postupovat podle vzorce $X_{i+1} = X_i - Y_i * S_i * 2^{-i}$, podobně pro Y_i. Nejdříve zjistím číslo současné iterace i, vytvořím opačné číslo a to přičtu k exponentům Y_i a X_i. Podle znaménka úhlu potom tyto čísla přičtu nebo odečtu od původních X_i a Y_i. Tím získám výsledek iterace, který uložím v registru. Paralelně snižuji současný úhel o hodnotu, kterou vyhledám v entitě `arctan_lookup`, kam pošlu číslo iterace.

Iterace algoritmu budou probíhat, dokud od řadiče nepřijde signál, že je výpočet dokončen. Na výstupu z entity budu vystavovat za běhu algoritmu nejlepší odhad sinu a cosinu.

5.2.2.2 Druhá odmocnina

Na vstupu do entity budu mít hodiny, reset a číslo, pro které chci vypočítat druhou odmocninu.

V entitě vypočítám $\frac{x_i}{2} * (3 - A * x_i^2)$ pomocí několika entit `mul` pro násobení a entity `three_sub` pro odečtení od 3. Nové číslo uchovám v registru a použiji jako x_i v další iteraci.

Zároveň výsledek z této iterace ještě vynásobím původním číslem na vstupu, abych získal současný odhad druhé odmocniny, který pošlu na výstup.

5.2.2.3 Logaritmus

Vstupní signály, které budu potřebovat, jsou reset, signál hodin a číslo, pro které mám zjistit logaritmus o základu 2.

Hlavní entitě `logarithm` bude obsahovat entitu `get_xi`, která provede velkou část výpočtu. Podle současného čísla A, které se blíží 1, a pořadí iterace zjistí správnou dočasnou proměnnou x. x se přičte k odhadu přirozeného logaritmu Y a vynásobí se s A pro přiblížení k 1.

Y vynásobím konstantou, abych ho převedl do logaritmu o základu 2. Výsledek tohoto násobení pošlu na výstup entity `logarithm`.

Kapitola 6

Realizace

V této kapitole popisuji, jak jsem vytvořil fungující modely algoritmů v programu Wolfram Mathematica a moji implementaci v jazyce VHDL na základě návrhu z předchozí kapitoly. První sekce se týká mého řešení v Mathematica. Ve druhé sekci popisuji VHDL realizaci společných operací pro více funkcí. V dalších sekcích vysvětluji VHDL řešení matematických funkcí.

6.1 Wolfram Mathematica

6.1.1 CORDIC

Na prvních stránkách odvozuji jádro algoritmu CORDIC – rovnici pro získání lepšího odhadu.

Uvnitř algoritmu využívám dvě pomocné funkce: `Kn` a `sign`. `Kn` má jeden vstupní parametr, počet iterací, a podle toho vrací hodnotu konstanty pro správný výpočet Sinu a Cosinu. Funkce `sign` vrací znaménko čísla, které předám jako parametr, pro lepší ilustraci v rovnici. Také využívám `ArcTan` pro 2^{-i}, ve VHDL implementaci je nahrazen tabulkou v paměti (vytvořené podle [14]).

Ve funkci `cordic` nastavím počáteční hodnoty, počítám nové iterace a vrátím výsledky.

Vstupní parametry jsou požadovaný úhel a počet provedených iterací. Iniciální hodnoty vycházejí z [13]. β nastavím na vstupní úhel, α se nastaví na $1(2^0)$ a seznam vypočítaných hodnot začíná bodem `{0, 0}`, aby se v grafu vždy obě osy zobrazily od nuly.

V těle cyklu si nejdříve uložím hodnoty X a Y současné iterace pro zobrazení v grafu na další stránce. Následuje výpočet nových hodnot X a Y podle vzorců z analýzy ($X_{i+1} = X_i - Y_i * S_i * 2^{-i}$, $Y_{i+1} = Y_i + X_i * S_i * 2^{-i}$). Poté se zmenší současný úhel β. Načte se nová hodnota α, nová X a Y se uloží pro další iteraci.

Dále mám dvě pomocné funkce pro vykreslení grafu a porovnání výsledků. Jejich pomocí vypisuji hodnoty a komentáře k nim, aby byla poslední stránka přehledná. Na té jsou pouze parametry, které mohou uživatelé měnit, volání těchto 2 funkcí a jejich výstup.

6.1.2 Druhá odmocnina

Na první stránce popisuji Newtonovu metodu a dosazuji funkci $x^2 - A$.

Na další stranu jsem umístil funkci, která vypočítá první dvě hodnoty a graficky zobrazí, jak se vypočítá nová hodnota v závislosti na současné hodnotě tečně grafu v tom bodě. Při volání funkce dodávám tři parametry. První určuje číslo, jehož odmocninu si přeji vypočítat. Druhý parametr se použije jako první odhad hodnoty – čím lepší tento odhad je, tím lepší jsou vypočítané hodnoty. Třetí slouží pouze k zobrazení na grafu, konkrétně maximální hodnotu na ose X.

Následně nastavím parametry, zavolám funkci a tím se zobrazí graf.

4. strana obsahuje efektivní implementaci této funkce, parametry pro tuto funkci a porovnání se správnou hodnotou $\sqrt{A}$. Funkce má první dva parametry stejné jako předchozí – základ odmocniny a první odhad. Třetí udává kolik iterací se vypočítá.

Na dalších 3 stranách opakuji stejný postup (funkce s grafem, graf, algoritmus pro implementaci) s několika rozdíly:

funkce Jako základní funkci mám $\frac{1}{x^2} - A$.

odhad První odhad počítám uvnitř funkce, stejně jako ve VHDL implementaci. Odhad je $1 * 2^{exponent/2}$.

parametry Nepotřebuji první odhad jako parametr. Ve funkci s grafem je nutné pomocí parametrů specifikovat celý rozsah os X a Y, bez toho by byl graf špatně čitelný.

6.1.3 Logaritmus

Na prvních 3 stranách popisuji teorii algoritmu s využitím funkce `BaseForm` pro převod mezi desítkovou a binární soustavou.

Další strana obsahuje tabulku 6.1, na níž vysvětluji jak získat sigma, podle kterého se vypočítá x_i podle vzorce $x_i = 1 + sigma * 2^{-i}$. Hodnoty jsou podle zdroje [13].

Následuje strana s pomocnými funkcemi. První z nich (`bit`) izoluje jeden bit z čísla. Dvěma parametry se určí z jakého čísla se bit má získat a o kolikátý bit se jedná.

Další funkci `setSigma` používám pro nastavení sigma, podle tabulky 6.1 [13]. Parametry jsou současné A a pořadí iterace.

0. bit A	$-i$. **bit A**	**sigma**
0	0	2
0	1	0
1	0	0
1	1	-1

Tabulka 6.1: Tabulka pro určení sigma

Funkce `placeArrows` s parametry `pos` určující kolikátá iterace se provádí a `minus` pro určení jestli je číslo záporné a tedy posunuté o jednu pozici kvůli mínus. Touto funkcí zobrazuji šipky pod bity, které se používají k určení sigma.

Ve funkci `logExample` vycházím z algoritmu pro implementaci. Vstupní parametry jsou hodnota, kterou má vypočítat logaritmus, a počet iterací. `logExample` jsem přizpůsobil pro zobrazení hodnot ve funkci `logSlider`. Ta má stejné parametry a podle nich vytvoří okno s posuvníkem pro grafické znázornění principu algoritmu.

Efektivní algoritmus jsem vytvořil na 6. stránce. Vstupem jsou dvě čísla, hodnota A (pro výpočet $\log_A$) a počet iterací. Průběžné výsledky vypisuji do `Grid` pro zarovnání.

Poslední stránka využívá předchozí funkce. Obsahuje ukázku jak algoritmus zjišťuje hodnotu sigma i porovnání výsledků interní funkce `Log` a funkce, kterou implementuji ve VHDL.

6.2 VHDL

6.2.1 Společné entity

6.2.1.1 Entita add

Entita slouží ke sčítání v mém formátu pohyblivé řádové čárky.

Pro rozhraní mezi entity jsem se rozhodl využít typ `std_logic_vector` pro kompatibilitu mezi různými entitami. Pro přehlednější práci jsem již na úrovni rozhraní rozdělil operandy i výsledek na mantisu a exponent, lze vidět ve výpisu kódu 6.1.

Díky komutativitě sčítání nezáleží na pořadí operandů A a B, díky tomu stačí určit, který operand má větší exponent, podle toho je označit jako menší (`smaller`) a větší (`bigger`), nezávisle na jejich předchozím názvu. Tyto signály také mají o 1 větší šířku pro budoucí detekci přenosu. Do `exp_diff` si uložím rozdíl exponentů. Dále se s menším provede logický posun doprava o `exp_diff`, k tomu jsem využil knihovní funkci `shift_right`. V tu chvíli jsou čísla reprezentována v přímém kódu a lze je sečíst použitím operátoru + z knihovny `numeric_std`.

Po sečtení proběhne kontrola přenosu, jinými slovy jestli je nejvyšší bit 1, podle toho se výsledek případně upraví. V průběhu ještě proběhne kontrola

```
entity add is
    port (
        a_man       : in std_logic_vector (9 downto 0);
        a_exp       : in std_logic_vector (5 downto 0);
        b_man       : in std_logic_vector (9 downto 0);
        b_exp       : in std_logic_vector (5 downto 0);
        res_man     : out std_logic_vector (9 downto 0);
        res_exp     : out std_logic_vector (5 downto 0)
    );
end add;
```

Výpis kódu 6.1: Rozhraní entity add

jestli není jeden z operandů nulový, v takovém případě se jako výsledek označí nenulový operand.

6.2.1.2 Entita sub

Slouží k odčítání v použitém formátu. Rozhraní je stejné jako pro entitu `add`, s přidaným signálem `sign` pro označení znaménka výsledku po odčítání.

Při odčítání musí operandy zůstat ve stejném pořadí, kvůli tomu si nejdříve zjistím, který operand je větší a o kolik. Větší exponent si připravím jako základ výsledného exponentu. Potom přidám jeden vodič pro detekci přenosu, provedu bitový posun doprava pro oba operandy a podle toho, který byl větší, je správně označím pro odčítání. V této části jsou čísla reprezentována v doplňkovém kódu a lze použít operátor $-$.

Zbývá převést výsledek do formátu pohyblivé řádové čárky. K tomu určím znaménko, zaznamenáno v nejvyšším bitu výsledku odčítání. Pokud se rovná 0, odpovídá $+$, nic neměním, `sign` označím 0 a jdu na další krok. V opačném případě provedu bitovou negaci a přičtu 1, abych získal absolutní hodnotu čísla a na `sign` pošlu 1. Určím na kolikáté pozici se nachází první jednička v čísle, posunu absolutní hodnotu doleva o správný počet pozic a tento počet také odečtu od připraveného základu exponentu. Tím získám kompletní výsledek – mantisu, exponent i znaménko.

6.2.1.3 Entita multiplier

Entitu používám pro násobení v mém formátu. Rozhraní násobičky je shodné s rozhraním entity `add`.

Výsledný exponent získám prostým sečtením exponentů operandů.

K vypočítání výsledné mantisy jsem využil algoritmu add-and-shift, upravený pro práci během jednoho hodinového cyklu, založeno na principu z webové stránky [15]. Add-and-shift funguje následovně: pokud je bit B_i roven 1, operand A se přičte k mezivýsledku, pokud se bit rovná 0, s mezivýsledkem se nic

neděje, respektive přičte se 0. Operand A se binárně posune doprava o jednu pozici a i se zvětší o 1. Počet iterací je roven počtu bitů B, hodnota i začíná na 0.

Při násobení mantis může vzniknout přenos stejně jako u sčítání. V takovém případě zvětším exponent o 1 a posunu výsledek o 1 pozici doprava.

6.2.2 CORDIC

Pro všechny tři algoritmy jsem využil design s konečným automatem (KA) jako řadičem. Ve zbylé části obvodu provádím výpočet. Část kódu jsem oddělil do entity `arctan_lookup`. Diagram zapojení entit uvnitř jsem umístil na konec této sekce.

6.2.2.1 arctan lookup

Tato entita vyhledává hodnoty $\text{arctg}2^{-1}$. Vyjmul jsem ho z VHDL kódu logaritmu z důvodu zpřehlednění kódu a oddělení konstantních hodnot. Pro malou přesnost není velikost obvodu problém, proto jsem použil funkci `case`. Kód upraven podle zdroje [14].

6.2.2.2 Řadič

Konečný automat sloužící jako řadič není komplikovaný, bylo by možné stejné funkce integrovat přímo do obvodů, ale zvolil jsem rozdělení z důvodu větší přehlednosti a zjednodušení případných úprav v budoucnosti. Obsahuje pouze tři stavy a každému odpovídá jeden signál. Hlavní část jsem rozdělil klasicky do 3 procesů – registr, vstupní a výstupní kombinační obvod (KO). Dále jsem vytvořil 2 procesy pro práci s počtem iterací, jeden registr a jeden KO.

Výstupní kombinační obvod závisí pouze na současném stavu, protože jsem zvolil automat typu Moore. Výstupní KO označí 1 signál příslušící současnému stavu, na ostatní pošle 0.

Vstupní kombinační obvod je závislý na současném stavu a jestli proběhl zadaný počet iterací. Pokud proběhl, nebo pokud se KA již nachází v cílovém stavu, další stav je cílový. V opačném případě je další stav pokračování ve výpočtu.

Proces registru stavů si při resetu načte počáteční stav, v jiném případě načítá další stav.

Registr počtu iterací se nuluje při resetu, přičítá 1 ve stavu počítání a jinak udržuje stejnou hodnotu.

KO pro hlídání počtu iterací pouze porovnává, jestli se současný počet rovná cílovému (určen konstantou) a při rovnosti označí svůj signál 1.

6.2.2.3 Výpočetní část

Vycházím z těchto rovnic:
$X_{i+1} = X_i - Y_i * S_i * 2^{-i}$
$Y_{i+1} = Y_i + X_i * S_i * 2^{-i}$

Výpočet X v jedné iteraci má stejný postup jako výpočet Y, pouze se vymění hodnoty X a Y. Dále píši o Y_{i+1}, výpočet X_{i+1} probíhá analogicky.

Výpočet $X_i * 2^{-i}$ si můžu zapsat jako $X_{mantisa} * 2^{exponent} * 2^{-i}$. K výpočtu stačí, abych od exponentu odečetl i.

Zda mám sčítat nebo odčítat zjistím podle znamének Y, X a `beta` (v rovnici jako S_i). X a `beta` se mají násobit, jejich znaménka po násobení odpovídají X XOR `beta`.

Dále si vypočítám výsledky sčítání i odčítání (za použití entit `add` a `sub`) a v samostatném procesu rozhodnu, který z nich použiji podle současných znamének – znázorněno ve výpisu kódu 6.2. Tento způsob aplikuji i v dalších případech, kde se může počítat se zápornými čísly.

Pro ukládání hodnot využívám 2 registry. První z nich pro X a Y. Při resetu nastavím X na 1 a Y na 0. Při počítání si do registru uložím jejich nové hodnoty. Po dokončení počítání si pouze uchovávají poslední hodnotu. Druhý registr funguje velmi podobně. Reset způsobí načtení nové hodnoty ze vstupu, při počítání ukládá novou hodnotu úhlu, po vypočítání zůstává na stejné hodnotě.

Hodnotu úhlu α_i zjistím z entity `arctan_lookup` v prvních 7 iteracích. V dalších iteracích pouze snižuji exponent o 1, při dané přesnosti jsou následující hodnoty stejné jako dělení 2.

Celkový úhel zadaný na vstupu jsem pojmenoval beta pro odlišení od `alpha` v kódu. Již známý úhel `alpha` odečtu od `beta`, pokud je beta větší než 0, jinak `alpha` přičtu. Zapsáno rovnicí, kde `sign` je znaménko beta:
`new_beta` = `beta` − `sign` ∗ `alpha`
To si ale můžu napsat pouze jako odčítání:
`new_beta` = `sign` ∗ `|beta|` − `sign` ∗ `alpha`
`new_beta` = `sign` ∗ (`|beta|` − `alpha`)
Od beta vždy odečítám `alpha`, současné znaménko ovlivní pouze nové znaménko.

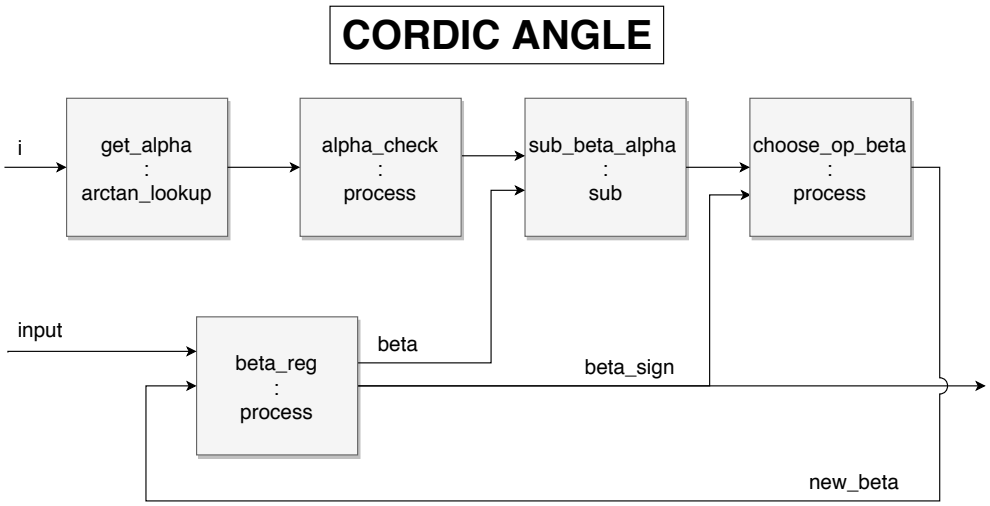
Pro dokončení výpočtu vynásobím X a Y konstantou pojmenovanou KN. Tím získám sinus a cosinus vstupního úhlu.

Zvolil jsem 10 jako počet iterací, protože v každé iteraci pracuji s hodnotami posunutými o jednu pozici doprava, proto se po 10 iteracích žádný bit mantisy nezmění.

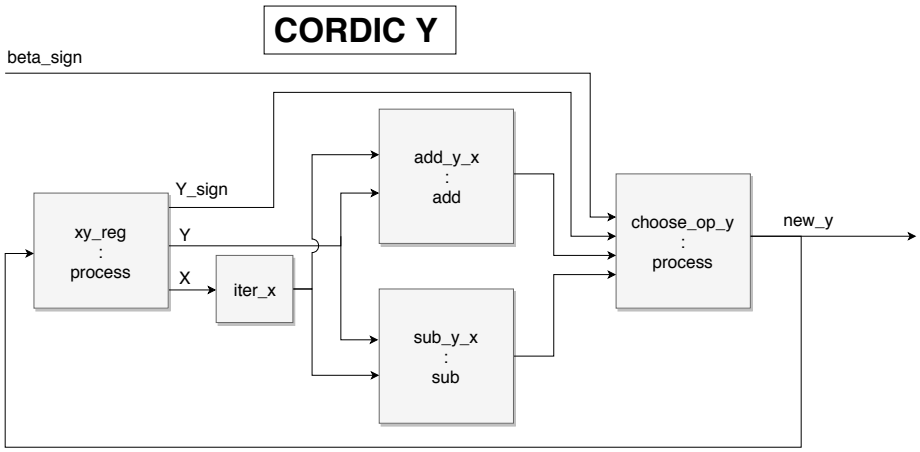
```
choose_op_y : process(y_sign, x_b_sign,
              y_sub_sign, y_sub, y_sub_exp, y_add, y_add_exp)
begin
     -- if equal signs, then add the numbers and keep the sign
     if y_sign = x_b_sign then
            new_y_sign <= y_sign;
            new_y <= y_add;
            new_y_exp <= y_add_exp;
         -- if different signs, subtract
     else
         new_y <= y_sub;
         new_y_exp <= y_sub_exp;
         -- if y is positive, sign from sub is correct
         if y_sign = '0' then
               new_y_sign <= y_sub_sign;
         -- if y is negative, sign from sub is opposite
         else
               new_y_sign <= not(y_sub_sign);
         end if;
     end if;
end process;
```

Výpis kódu 6.2: Výpočet úhlu v entitě cordic

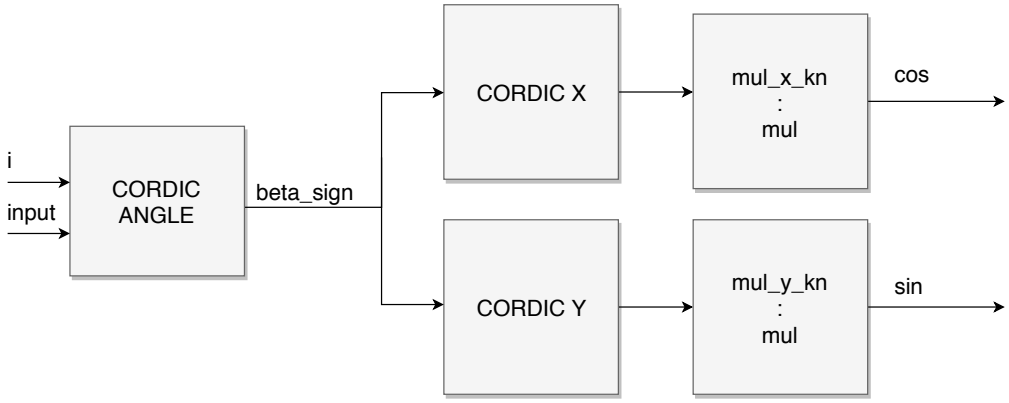
Na obrázku 6.1 popisuji, jak se počítá nový úhel beta a jeho závislost na úhlu alpha. U registru `beta_reg` jsem pro přehlednost neuvedl signály hodin, resetu a řízení. Vyobrazil jsem pouze datové signály. S výjimkou `beta_sign` a `i` mají všechny signály šířku 16 bitů.



Obrázek 6.1: Získání nového úhlu v jedné iteraci CORDIC



Obrázek 6.2: Výpočet nové hodnoty Y v entitě CORDIC



Obrázek 6.3: Získání výstupu v entitě CORDIC v závislosti na vstupu

Na obrázku 6.2 naznačuji výpočet nové hodnoty Y. Stejným způsobem se vypočítá nové X s tím rozdílem, že X se nahradí Y a Y se nahradí X. Zobrazil jsem jenom datové signály. Jediný registr `xy_reg` na obrázku ukládá hodnoty X i Y.

Na obrázku 6.3 ukazuji zapojení dříve vyobrazených částí designu. Opět jsou uvedeny pouze datové signály. Malými písmeny jsem označil instanci komponent, velkými části designu, které jsou pro přehlednost seskupeny.

6.2.3 Druhá odmocnina

Ve výpočtu používám entitu `three_sub`. Diagram zapojení se nachází na následující straně.

6.2.3.1 Entita three sub

Vrátí výsledek operace $3 - B$, kde B je vstup. Nepoužil jsem předchozí entitu `sub` z důvodu zjednodušení návrhu i případného hardware, v tomto případě zůstává jeden operand konstantní. Rozhraní má pouze jeden vstup a jeden výstup, oba s mantisou a exponentem.

Vstup nebude nikdy větší než 3, toto jsem dokázal v analýze. Díky této znalosti nemusím nikdy měnit pořadí operandů. Mantisu vstupu posunu na odpovídající místo, odečtu od konstantní 3.

Dále provádím stejné operace jako v entitě `sub` – zjistím první výskyt 1, provedu posun jedničky na nultý bit. Z důvodu vyšší přesnosti tentokrát nejvyšší bit odpovídá 2^1, pokud se 1 nachází tam, je nutné posunout doprava o jednu pozici.

Výsledný exponent se rovná 0−`lead_one`, kde `lead_one` označuje první výskyt jedničky.

6.2.3.2 Řadič

Konečný automat fungující jako řadič se skládá ze dvou registrů a tří kombinačních obvodů.

První registr používám pro zjištění počtu zbývajících iterací. Při resetu se tam nahraje požadovaný počet iterací, zadaný konstantou v kódu, na počátku hodinového cyklu se počet sníží o jedna. Za registr jsem napojil jeden z kombinačních obvodů a porovnávám jestli se snížil na nulu, podle toho posílám 1 na signál `count_is_zero`.

Druhý registr ukládá současný stav automatu, při počátku nového hodinového cyklu si uloží nový stav.

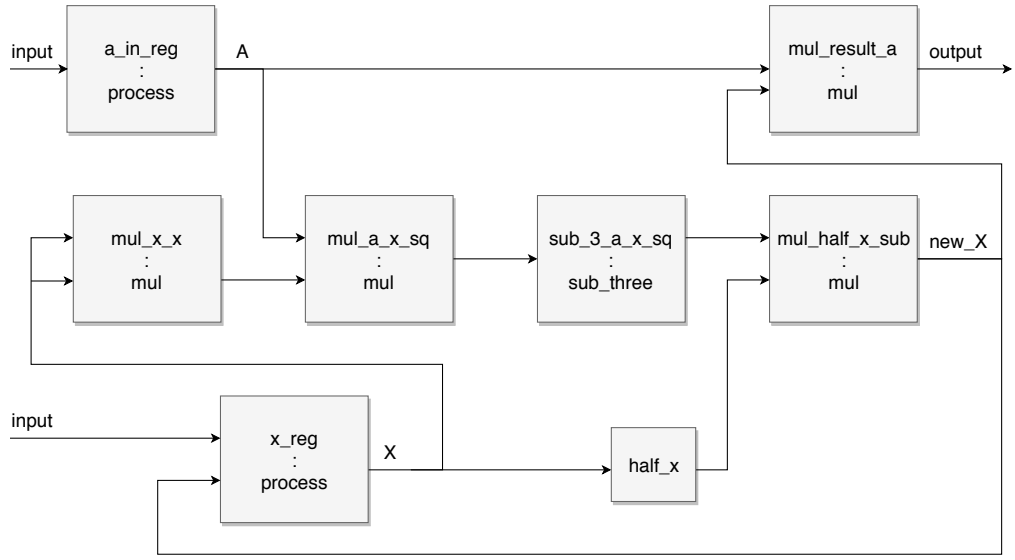
Zbývající logické obvody zajišťují získání nového stavu v závislosti na tom současném a signálu `count_is_zero`. Pokud počet klesl na nulu, nový stav je koncový, v opačném případě pokračuji další iterací.

6.2.3.3 Výpočetní část

Ve výpočetní části využívám 2 registry pro uložení hodnot a entity `multiplier` a `three_sub` pro vypočítání jedné iterace podle rovnice $(x/2)*(3-A*x^2)$ a pro vynásobení výsledku rovnice A.

Nejdříve vypočítám $x * x$, výsledek vynásobím A a pošlu jako vstup do `three_sub`. Paralelně s tím provedu $(x/2)$ pomocí snížení exponentu o jedna. Tento signál vynásobím s výstupem `three_sub` a tím získám současný odhad $1/\sqrt{A}$.

Ve stejné iteraci ihned vynásobím odhad inverze s hodnotou A, abych získal výsledek $\sqrt{A}$ při každé iteraci, nemusel vytvářet další stav a nezpomaloval celý systém o jeden hodinový cyklus.



Obrázek 6.4: Vnitřní zapojení entit a procesů pro výpočet druhé odmocniny

Jeden registr používám k uchování hodnoty vstupního operandu A. Pokud nastane reset, obsah registru se nastaví na hodnotu ze vstupu entity `sqrt`. V jiném případě se hodnota nezmění, protože A je potřeba při každé iteraci.

Ve druhém registru jsem se rozhodl mít hodnotu současného odhadu inverze x. Pokud je reset roven 1, vypočítám rychlý počáteční odhad $2^{-exponentA}$; v opačném případě se rozhodne podle signálu z řadiče. Když se ještě počítají další iterace, do registru se uloží nový odhad z poslední iterace. Pokud výpočet skončil, hodnota zůstává stejná.

V obrázku 6.4 jsem uvedl všechny vztahy mezi procesy a entitami uvnitř entity `sqrt`. Stejně jako u CORDIC jsem skryl nedatové signály. Šířka všech signálů je teoreticky 16 bitů, ale v registru `x_reg` se využívá pouze exponent, tedy 6 bitů.

6.2.4 Logaritmus

Na další stránku jsem umístil diagram zapojení.

6.2.4.1 Řadič

Opět jsem řadič rozdělil na 5 procesů: dva registry a tři kombinační obvody. V jednom z registrů uchovávám čítač iterací a ten mám napojen na kombinační obvod, který ho porovnává s požadovaným počtem iterací.

Ze zbylých procesů jsem vytvořil základ konečného automatu. Do registru načítám nový stav, pokud nenastane reset, v tom případě tam vložím počá-

teční stav. Jedním kombinačním obvodem nastavuji nový stav. Druhým KO zajišťuji korektní signály z konečného automatu.

6.2.4.2 Výpočetní část

Hodnoty Y, jeho znaménko a A ukládám v jednom registru. V případě resetu nastavím A na vstupní hodnotu a Y vynuluji. Při počítání nastavím všechny hodnoty na nově vypočítané podle . Jinak obsah registru zůstává stejný.

Nejdříve vypočítám aktuální i přičtením 1 k počtu iterací z řadiče.

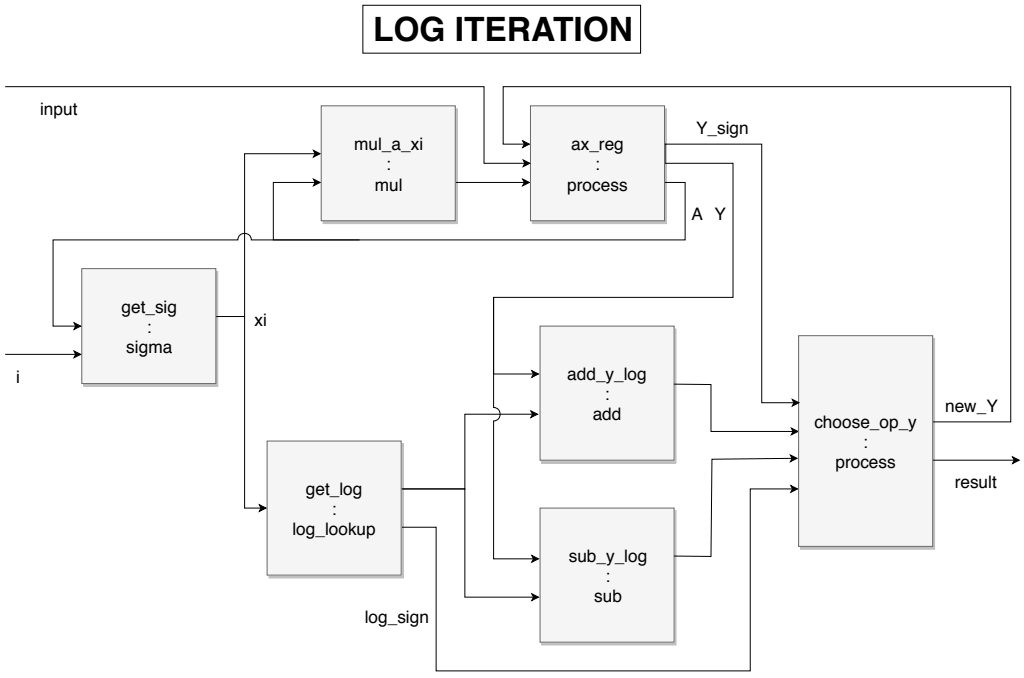
Pomocí entity `get_xi`, která na vstupu obdrží A a i, vypočítám x_i pro tuto iteraci. Uvnitř provedu několik konverzí pro jednodušší výpočet a podle tabulky 6.1 a vzorce $x_i = 1 + sigma * 2^{-i}$ vypočítám x. Před odesláním na výstup ještě číslo upravím, abych dodržel formát. Díky omezeným možnostem výsledku stačí zkontrolovat dvě podmínky a hodnotu exponentu nastavit přímo.

Nové A dostanu násobením současného A s x.

Vyhledám hodnotu logaritmu $\log_e x$ současné iterace v entitě `log_lookup`, realizované podle [14]. Ta vrací správnou hodnotu pouze podle i.

Použiji entity `add` a `sub` pro vypočítání součtu i rozdílu Y a logaritmu iterace. Podle znamének se rozhodnu, který výsledek z předchozích operací použiji. Pokud jsou znaménka stejná, využívám součet a znaménko je určeno znaménkem Y. V případě různých znamének vyberu rozdíl, když je Y kladné, výsledné znaménko je přímo závislé na `sub`, jinak ho zneguji.

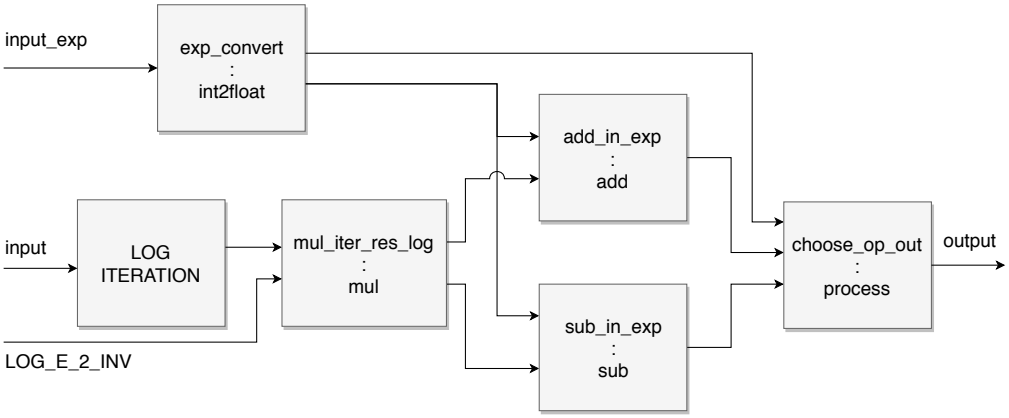
Vynásobím nové Y konstantou `LOG_E_2_INV`, abych získal hodnotu logaritmu se základem 2. Nakonec tento výsledek přičtu (případně odečtu podle znaménka) k exponentu na vstupu, abych získal binární logaritmus vstupního čísla. Exponent přichází v doplňkovém formátu, proto ho musím pomocí entity `int2float` převést do formátu pohyblivé řádové čárky. V `int2float` využívám podobný postup jako v entitě `sub`, upravený pro doplňkový kód s délkou 6 bitů.



Obrázek 6.5: Výpočet jedné iterace logaritmu

Na obr. 6.5 ukazuji způsob zapojení procesů a komponent pro počítání jedné iterace v entitě `logarithm`. U registru `ax_reg` nemám uvedeny signály hodin a resetu, ani připojení k řadiči.

Obrázek 6.6 popisuje získání celkového výsledku podle výsledku současné iterace. Opět se jedná o datové signály. LOG ITERATION reprezentuje obvod na předchozím obrázku 6.5.



Obrázek 6.6: Výpočet výstupu logaritmu

Kapitola 7

Testování

V této kapitole popisuji proces ověřování správnosti mých řešení. V programu Wolfram Mathematica jsem porovnával mé výsledky s výstupy z integrovaných funkcí programu. Implementaci v jazyce VHDL jsem simuloval v programu Vivado 2018.2 a porovnával s hodnotami z Wolfram Mathematica.

7.1 Wolfram Mathematica

Výsledky zobrazuji a porovnávám v desítkové soustavě a používám desetinná místa pro určení přesnosti.

7.1.1 CORDIC

Pro testování jsem využil vestavěné funkce `Sin` a `Cos`. Stejný úhel, který předám jako parametr mé implementaci, dám také oběma funkcím.

V závislosti na předem daném počtu iterací se výsledek mého algoritmu přibližuje správné hodnotě. Při zadání 6 iterací se správně vypočítá pouze 1. desetinné místo (navíc v některých případech by bylo nutné zaokrouhlit). Přesnost se zlepší na 3 desetinná místa ve všech testovaných případech, když zvýším počet iterací na 13. Pro většinu hodnot mi stačilo 12 iterací, ale úhly -21° a 15° se lišily na 3. místě. Správnosti na 5 desetinných místech jsem dosáhl při 20 iteracích.

Každou možnost jsem vždy ověřil alespoň s následujícími velikostmi úhlů: -43°, -21°, 0°, 15°, 30°, 37°, 67° a 90°.

7.1.2 Druhá odmocnina

Číslo, které si přeji použít jako základ odmocniny, označím A. A odmocním použitím odmocniny ($\sqrt{}$ v Mathematica) a mého algoritmu.

Při počítání $\sqrt{A}$ (pouze pro ukázku v Mathematica) nechávám uživateli možnost změnit počáteční odhad. Počet iterací nutný pro přesný odhad je

na tom závislý. Při dobrých odhadech[1] algoritmus do 7. iterace dosáhne výsledku přesného na 6 desetinných míst.

Výpočet $\frac{1}{\sqrt{A}}$, implementovaný ve VHDL, má jako parametry pouze A a počet iterací. Odhad se počítá uvnitř algoritmu, stejně jako ve VHDL. Díky tomu se odhad rovná výsledku (alespoň do 6. desetinného místa) z integrované funkce do 5. iterace, pro všechna vyzkoušená čísla, tj: 0.499999, 5.73, 31, 127, 984, 75624 a 15615915.

7.1.3 Logaritmus

Pro kontrolu správnosti jsem využil funkci logaritmu (`Log[2, A]`) v Mathematica.

Přesnost na 5 desetinných míst potřebuji okolo 20 iterací, v závislosti na konkrétním čísle, všechna testovaná dosáhla přesnosti do 25 iterací. 4 správná desetinná místa mohu získat s 15 iteracemi. Pro 2 desetinná místa mi stačí pouze 8 iterací. Testoval jsem čísla 0.15635, 0.731256, 5.73, 21.53, 341, 1365, 10922

7.2 VHDL

Pro testování používám interní simulaci v programu Vivado 2018.2. Pro každou entitu jsem vytvořil jednu testovací entitu (testbench, zkráceně TB). Korektní hodnoty získám z Wolfram Mathematica pomocí funkcí `BaseForm`, `MantissaExponent` a `N`.

7.2.1 Čistě kombinační obvody

Tyto entity nevyžadují hodinový impuls, tím se mi zjednoduší testování. Na vstupní porty testované entity nahraji operandy, počkám až se signály propagují a porovnám se správnými hodnotami.

7.2.1.1 add

V entitě `add` ověřuji s běžnými čísly a s několika neobvyklými případy, to jsou: výměna pořadí operandů, sčítání s nulou, oba operandy rovné nule a správné počítání při vzniku přenosu.

Entita fungovala mi vracela správné výsledky ve většině případů, kromě situace, kdy jeden ze vstupních operandů byl menší než 1 (exponent byl menší než 0) a druhý byl 0 (exponent 0). Protože jsem exponent a mantisu počítal naprosto nezávisle na sobě, do výsledku se dostala nenulová mantisa, posunutá o několik míst doprava, a větší z exponentů, což byla 0. V mezivýsledku se první 1 mohla objevit až na -2. nebo nižším bitu. Z toho důvodu, že kontroluji pouze 0. a -1. bity (kvůli přenosu), se mohlo stát, že výstup entity nezačínal 1.

[1] Za dobrý považuji odhad, který je od správné hodnoty vzdálený méně než o jeden řád.

Druh vstupu	vstup A	vstup B	výstup WM	výstup VHDL
Mantisa	1011010101	1010010101	1010110101	1010110101
Exponent	000001	000001	000010	000010
Mantisa	1100000000	1010000000	1100010100	1100010100
Exponent	000010	111101	000010	000010
Mantisa	1010010101	0000000000	1010010101	1010010101
Exponent	111111	000000	111111	111111

Tabulka 7.1: Výsledky entity add

Druh vstupu	vstup A	vstup B	výstup WM	výstup VHDL
Mantisa	1000000000	1000000000	1111000000	1111000000
Exponent	000101	000001	000100	000100
Mantisa	1011010101	1010010101	1000000000	1000000000
Exponent	000001	000001	111110	111110

Tabulka 7.2: Výsledky entity sub

To jsem opravil přidáním speciální podmínky pro případ nulového operandu, ostatní hodnoty začínají 1 a stejný problém tam nemůže nastat.

Po této opravě počítá entita všechna testovaná čísla korektně, jak je vidět v tabulce 7.1, kam jsem dal 3 příklady (z testovaných 8) pro ukázku. Výstup z Wofram Mathematica (označen jako výstup WM) jsem umístil do prostředního sloupce pro porovnání se správnými hodnotami.

7.2.1.2 sub

Entitu `sub` testuji čísly s podobnými exponenty, s velmi různými exponenty a s nulovou mantisou (a tedy i exponentem) na pozicích prvního i druhého operandu a obou operandů.

Po zjištění problému se sčítáním jednoho nulového čísla v entitě `add` jsem stejný problém opravil i v této entitě.

Všechny výsledky, které jsem ověřil po implementování opravy, odpovídaly výsledkům z programu Wolfram Mathematica, naznačeno v tabulce 7.2 dvěma příklady.

7.2.1.3 mul

Entita `mul` pro mě byla nejsnadnější na testování. Na velikosti exponentů téměř nezáleží, ve výpočtu se pouze sčítají. Nulová hodnota opět představuje problém, kdy entita musí vrátit nulovou mantisu i exponent, po zkušenostech s předchozími entitami jsem právě případ s nulou testoval velice pečlivě.

Druh vstupu	vstup A	vstup B	výstup WM	výstup VHDL
Mantisa	1111111111	1000000000	1111111111	1111111111
Exponent	000000	000000	000000	000000
Mantisa	1010101010	1100111001	1000100101	1000100101
Exponent	000001	111101	111111	111111
Mantisa	1010101010	1100111001	1000100101	1000100101
Exponent	111001	111001	110011	110011

Tabulka 7.3: Výsledky entity mul

Dále jsem ověřoval práci s běžnými hodnotami i v jiných krajních případech – maximální hodnoty a minimální nenulové hodnoty.

Výstup z entity se rovnal správným hodnotám z Mahtematica, jak můžete vidět v tabulce 7.3.

7.2.2 Obvody s registry

K testování jsem vytvořil proces pro generování hodinových impulsů. V tomto procesu každou polovinu periody změním hodinový signál z 0 na 1, případně z 1 na 0. K nastavení délky periody jsem na začátku těla entity přidal konstantu `PERIOD`. Tento proces jsem vložil do každé testovací entity níže.

Zbytek testování probíhá v jednom procesu pro zjednodušení synchronizace resetování s kontrolou správným hodnot. Nejdříve jsem měl reset oddělený ve vlastním procesu stejně jako hodiny, ale měl jsem problémy, kdy nastal reset dříve než se výpočet dokončil nebo naopak čekal zbytečně moc taktů po dokončení.

V testbenchi pro tři hlavní entity s několika iteracemi (`CORDIC`, `sqrt`, `logarithm`) jsem přidal konstantu `ITER_COUNT` pro určení počtu iterací, podle toho TB počká správnou dobu před kontrolou hodnot.

V entitě `CORDIC` jsem vyzkoušel pár běžným úhlů (nebyly to krajní hodnoty, byly kladné) a našel jsem problém s přesností na nejnižších bitech. Vyzkoušel jsem i ostatní hlavní entity a chování bylo vždy stejné. Počátečních 7 až 8 bitů se vypočítalo správně, v každé iteraci se rovnaly prvním bitům z Wolfram Mathematica, ale koncové bity se ustálily na špatných hodnotách.

Zdrojem problému bylo špatné zaokrouhlování. Vytvořil jsem entitu, která přičte nejvyšší zahozený bit a správně upraví zbytek čísla. Tu jsem přidal do entit pro základní operace. Po přidání měli výsledky větší přesnost.

7.2.2.1 CORDIC

Ověřoval jsem několik různých úhlů z levé poloviny kartézské soustavy souřadnic: kladné, záporné úhly i pravý úhel. Správně mi vyšly kladné i záporné úhly,

Úhel	Druh vstupu	vstupní úhel	výstup WM	výstup VHDL
30°	Mantisa	1000011000	100000000	100000000
	Exponent	111111	111111	111111
37°	Mantisa	1010010101	1001101**00**	1001101**11**
	Exponent	111111	111111	111111
67°	Mantisa	1001010110	1110101**11**	1110101**00**
	Exponent	000000	000000	000000

Tabulka 7.4: Výsledky entity CORDIC

Desítkově	Druh vstupu	vstupní úhel	výstup WM	výstup VHDL
0.75	Mantisa	1100000000	1101110111	1101110111
	Exponent	111111	111111	111111
24	Mantisa	1100000000	1001110011	1001110011
	Exponent	000100	000010	000010
846	Mantisa	1101001110	1110100011	1110100010
	Exponent	001001	000010	000100

Tabulka 7.5: Výsledky entity sqrt

které se neblížily jedné z krajních hodnot (90 nebo 0 stupňům). V takovém případě se přesnost dramaticky snížila.

Výsledky z entity `CORDIC` jsou zobrazeny v tabulce 7.4. Kvůli přehlednosti jsem uvedl pouze sinus bez záporných úhlů. Nepřesné bity jsem označil.

7.2.2.2 Druhá odmocnina

Výsledky vycházely velmi dobře, po implementaci zaokrouhlení v některých případech vyšel výsledek přesně na všech 10 bitů mantisy. Algoritmus také funguje rychleji než ostatní dva, na výpočet stačí 6 iterací.

Výsledky jsem vepsal do tabulky 7.5. Výsledky Wolfram Mathematica jsem zaokrouhlil na 10 míst v binární soustavě.

Při testování jsem narazil na chybu programu Vivado, kdy byl výstup z entity označen X, i přes to že uvnitř entity se vše vypočítalo v pořádku.

7.2.2.3 Logaritmus

Výsledky entity `logarithm` se nejvíce zlepšily s uvedením zaokrouhlení. Všechny testované hodnoty, které jsem testoval, vyšly správně, pouze s maximálně dvěma rozdílnými bity na konci. To je způsobeno omezenou přesností. Exponenty vycházejí přesně.

V tabulce 7.6 jsem uvedl 3 výpočty bez znamének, entita na první vstup správně vrací záporné znaménko.

Desítkově	**Druh vstupu**	**vstupní úhel**	**výstup WM**	**výstup VHDL**
0.6426	Mantisa	1010010010	1010001**01**	1010001**10**
	Exponent	111111	111111	111111
1.8673	Mantisa	1110111100	111001101**0**	111001101**1**
	Exponent	000000	111111	111111
7648.46	Mantisa	1110111100	1100111010	1100111010
	Exponent	001100	000011	000011

Tabulka 7.6: Výsledky entity logarithm

Závěr

Cíle mé práce byly analýza, vytvoření modelů a implementace algoritmů pro výpočet goniometrických funkcí (algoritmus CORDIC), druhé odmocniny (pomocí Newtonovy metody) a logaritmu. Modely jsem měl vytvořit v programu Wolfram Mathematica, implementovat v jazyce VHDL a ověřit pomocí simulace v programu Xilinx Vivado. Výsledky mé práce by měly doplňovat výuku předmětu Počítačové aritmetika na magisterském studiu na FIT ČVUT v Praze.

V souborech pro Mathematica jsem se ze začátku věnoval dokázání správnosti algoritmů a odvození jejich rovnic. Dále jsem ukázal vývoj v jednotlivých iteracích, k tomu jsem využil zabudované grafické funkce programu. Pokračoval jsem verzí algoritmu využívající pouze funkce a operace, které lze převést do jazyka VHDL a případně implementovat v FPGA. Na závěr porovnávám nově vytvořené funkce s těmi integrovanými v programu. Všechny mé funkce se po dostatečném počtu iterací rovnaly výsledkům vestavěných funkcí.

Ve VHDL kódu jsem vytvořil tři entity, pro každou matematickou funkci jednu. Uvnitř používám další menší entity pro základní operace (sčítání, odčítání, násobení) a pro konkrétní účely každého algoritmu. Dle mého testování jsou všechny algoritmy implementovány správně. Pro vstupy, které se příliš neblíží krajním hodnotách, se vypočítané hodnoty téměř rovnaly těm z Wolfram Mathematica. V některých případech se hodnoty lišily v posledních bitech z důvodu omezené přesnosti.

Zdrojové kódy jsou přehledně napsané a okomentované, s rozdělením do více funkcí či entit a s důrazem na budoucí využití ve výuce.

Bibliografie

1. THIBEDEAU, Kevin. *cordic.vhdl* [online]. 2014 [cit. 2019-05-10]. Dostupné z: https://github.com/kevinpt/vhdl-extras/blob/master/rtl/extras/cordic.vhdl.
2. HERVEILLE, Richard. *CORDIC core* [online]. 2001 [cit. 2019-05-13]. Dostupné z: https://opencores.org/projects/cordic.
3. XILINX. *CORDIC v6.0* [online]. 2017 [cit. 2019-05-09]. Dostupné z: https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf.
4. ZIPCPU. *Using a CORDIC to calculate sines and cosines in an FPGA* [online]. 2017 [cit. 2019-05-08]. Dostupné z: https://zipcpu.com/dsp/2017/08/30/cordic.html.
5. TIŠNOVSKÝ, Pavel. *Výpočet goniometrických funkcí algoritmem CORDIC* [online]. 2019 [cit. 2019-05-09]. Dostupné z: https://www.root.cz/clanky/vypocet-goniometrickych-funkci-algoritmem-cordic/.
6. SLINTÁK, Vlastimil. *Algoritmus CORDIC* [online]. 2012 [cit. 2019-05-08]. Dostupné z: https://uart.cz/740/algoritmus-cordic/.
7. CP-ALGORITHMS. *Newton's method for finding roots* [online]. 2019 [cit. 2019-05-09]. Dostupné z: https://cp-algorithms.com/num_methods/roots_newton.html.
8. SMIT, Abhiraj. *Program for Newton Raphson Method* [online]. 2017 [cit. 2019-05-10]. Dostupné z: https://www.geeksforgeeks.org/program-for-newton-raphson-method/.
9. HELLOACM.COM. *How to Implement Integer Square Root in C/C++?* [online]. 2013 [cit. 2019-05-10]. Dostupné z: https://helloacm.com/coding-exercise-implement-integer-square-root-c-online-judge/.

10. ALACHIOTIS, Nikolaos. *Floating-Point Logarithm Unit* [online]. 2010 [cit. 2019-05-13]. Dostupné z: `https://opencores.org/projects/fp_log`.

11. PLUHÁČEK, Alois. *Číselné soustavy, sčítání a odčítání* [online]. 2018 [cit. 2019-05-13]. Dostupné z: `https://courses.fit.cvut.cz/MI-ARI/media/lectures/ARI-1-01-A-CS.pdf`.

12. PLUHÁČEK, Alois. *Sčítání a odčítání* [online]. 2018 [cit. 2019-05-13]. Dostupné z: `https://courses.fit.cvut.cz/BI-JPO/media/lectures/JPO-1-02-B-A1.pdf`.

13. PLUHÁČEK, Alois. *Elementární funkce* [online]. 2018 [cit. 2019-05-13]. Dostupné z: `https://courses.fit.cvut.cz/MI-ARI/media/lectures/ARI-1-09-I-EF.pdf`.

14. AMANOR David N., et al. *Efficient hardware architectures for modular multiplication on FPGAs* [online]. 2005 [cit. 2019-05-13]. Dostupné z: `https://www.researchgate.net/figure/VHDL-implementation-of-lookup-table_fig5_4178804`.

15. ANDRAKA, Ray. *Multiplication* [online]. 2016 [cit. 2019-05-13]. Dostupné z: `http://www.andraka.com/multipli.php`.

Dodatek A

Seznam použitých zkratek

BI-APS Architektury počítačových systémů

BI-CAO Číslicové a analogové obvody

BI-PNO Praktika v návrhu číslicových obvodů

CORDIC Coordinate rotation digital computer

ČVUT České vysoké učení technické

FIT Fakulta informačních technologií

FPGA Field Programmable Gate Array (Programovatelná hradlová pole)

KA Konečný automat

KO Kombinační obvod

WM Wolfram Mathematica

ROM Read-only memory (česky pamět pouze pro čtení)

TB Testbench (entita pro testování)

VHDL VHSIC Hardware Description Language

Dodatek B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové soubory
  - `Mathematica` ..... notebooky pro Wolfram Mathematica
  - `VHDL` ..... zdrojové kódy ve VHDL
    - `design` ..... implementace
    - `TB` ..... testovací soubory
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF