VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY
DEPARTMENT OF MICROELECTRONICS

VÝPOČET VLASTNÍCH ČÍSEL A VLASTNÍCH VEKTORŮ HERMITOVSKÉ MATICE
COMPUTATION OF THE EIGENVALUES AND EIGENVECTORS OF HERMITIAN MATRIX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Martin Štrympl

VEDOUCÍ PRÁCE
SUPERVISOR
doc. Ing. Lukáš Fujcik, Ph.D.

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Mikroelektronika**
Ústav mikroelektroniky

Student: Bc. Martin Štrympl ***ID:*** 120617
Ročník: 2 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Výpočet vlastních čísel a vlastních vektorů hermitovské matice

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s algoritmy pro výpočet vlastních čísel a vektorů matice. Zvolte nejvhodnější algoritmus a realizujte jej nad komplexní korelační matici 4x4 v hradlovém poli. Cílovou platformou je řada Xilinx Zynq 7000. Vstupem sestaveného bloku bude korelační matice a jeho výstupem budou vlastní čísla a vlastní vektory.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 8.2.2016 ***Termín odevzdání:*** 26.5.2016

Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.
Konzultant diplomové práce: Ing. Antonín Heřmánek, Ph.D., ERA, a.s.

doc. Ing. Lukáš Fujcik, Ph.D., *předseda oborové rady*

ABSTRAKT

Tato práce se zabývá výpočtem vlastních čísel a vlastních vektorů hermitovské pozitivně–semidefinitní komplexní čtvercové matice řádu 4. Cílem je implementace výpočtu v jazyce VHDL pro hradlové pole řady Xilinx Zynq-7000.

Práce se věnuje algoritmům pro výpočet vlastních čísel a vektorů pozitivně–semidefinitních reálných symetrických čtvercových a pozitivně–semidefinitních komplexních hermitovských matic a jejich analýze s využitím programu AnalyzeAlgorithm sestaveného pro tento účel. Závěrečná část práce popisuje implementaci výpočtu do hradlového pole s využitím IP bloku Xilinx® Floating-Point Operator a programů SVAOptimalizer, SVAInterpreter a SVAToDSPCompiler.

KLÍČOVÁ SLOVA

matice, vlastní čísla, vlastní vektory, symetrická matice, hermitovská matice, QR rozklad, třídiagonální matice, analýza algoritmů, VHDL, Xilinx Zynq-7000, Xilinx® Floating-Point Operator

ABSTRACT

This project deals with computation of eigenvalues and eigenvectors of Hermitian positive-semidefinite complex square matrix of order 4. The target is an implementation of computation in language VHDL to field-programmable gate array of type Xilinx Zynq-7000.

This master project deals with algorithms used for computation of eigenvalues and eigenvectors of positive-semidefinite symmetric real square and positive-semidefinite complex Hermitian matrix and the analysis of algorithms by AnalyzeAlgorithm program assembled for this purpose. The closing part of this project describes implementation of the computation into field-programmable gate array with use of IP core Xilinx® Floating-Point Operator and SVAOptimalizer, SVAInterpreter and SVAToDSPCompiler programs.

KEYWORDS

matrix, eigenvalues, eigenvectors, symmetrix matrix, Hermitian matrix, QR decomposition, tridiagonal matrix, algorithm analyze, VHDL, Xilinx Zynq-7000, Xilinx® Floating-Point Operator

ŠTRYMPL, Martin *Výpočet vlastních čísel a vlastních vektorů hermitovské matice*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky, 2016. 109 s. Vedoucí práce byl doc. Ing. Lukáš Fujcik, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Výpočet vlastních čísel a vlastních vektorů hermitovské matice“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

podpis autora(-ky)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

PODĚKOVÁNÍ

Rád bych poděkoval konzultantovi semestrální práce panu Ing. Antonínu Heřmánkovi, Ph.D. a zaměstnancům Vysoké učení technického v Brně doc. RNDr. Jaromíru Baštincovi, CSc. a doc. Ing. Romanovi Maršálkovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Za zapůjčení vývojové desky ZedBoard náleží poděkování Ing. Michalovi Kubíčkovi, Ph.D.

Brno ....

podpis autora(-ky)

OBSAH

Úvod **14**

1 Komplexní čísla a matice **15**
1.1 Komplexní čísla . . . 16
1.2 Matice . . . 18
1.2.1 Operace sčítání, odčítání a násobení . . . 18
1.2.2 Speciální případy matic . . . 19
1.2.3 Determinant . . . 20
1.2.4 Norma vektoru . . . 21
1.3 Vlastní čísla a vlastní vektory . . . 23
1.4 Householderova redukce na třídiagonální tvar . . . 24
1.5 Jacobiho metoda . . . 25
1.6 QR rozklad . . . 27
1.6.1 Gram-Schmidtův QR rozklad . . . 27
1.6.2 Householderův QR rozklad . . . 28
1.6.3 QR rozklad pomocí Givensovy matice . . . 29
1.6.4 Srovnání QR rozkladů . . . 30
1.7 QR algoritmus pro vlastní čísla a vlastní vektory . . . 31
1.8 Hledání vlastních čísel a vlastních vektorů hermitovské matice . . . 32

2 Analýza algoritmů **33**
2.1 Analýza redukce matice na matici třídiagonální . . . 34
2.2 Analýza QR rozkladu . . . 35
2.3 Srovnání výpočetní náročnosti algoritmů . . . 38

3 Implementace do hradlového pole **39**
3.1 IP blok pro výpočty s čísly s plovoucí desetinnou čárkou . . . 41
3.1.1 Čísla s plovoucí desetinnou čárkou . . . 41
3.1.2 Porty . . . 43
3.1.3 Neblokovací režim . . . 44
3.1.4 Blokovací režim . . . 44
3.1.5 Vygenerované bloky . . . 45
3.2 Pomocné bloky . . . 47
3.3 Reálná a komplexní aritmetická jednotka . . . 51
3.3.1 Aritmetická jednotka `APU_NB` . . . 51
3.3.2 Aritmetická jednotka `APU_NB_small` . . . 52
3.3.3 Aritmetická jednotka `APU_small` . . . 55

3.4 QR algoritmus v hradlovém poli 56
3.4.1 Blok DSP_pipelineRAM 57
3.4.2 Paměť s instrukcemi pro blok DSP_pipelineRAM 60
3.4.3 Optimalizace SVA kódu 61
3.4.4 Kompilace SVA kódu 64
3.5 Simulace 66
3.5.1 Struktura simulace 66
3.5.2 Simulace bloků APU_NB a APU_NB_small 69
3.5.3 Simulace bloku APU_small 69
3.5.4 Simulace bloku DSP_pipelineRAM 70
3.5.5 Simulace bloku QR_Algorithm 70

4 Závěr **71**

Literatura **72**

Seznam symbolů, veličin a zkratek **74**

Seznam příloh **75**

A Generování autokorelační matice zatížené šumem **76**

B Jacobiho metoda **78**

C Householderova transformace na matici třídiagonální **80**

D Gram-Schmidtův QR rozklad **81**

E Householderův QR rozklad **82**

F QR rozklad pomocí Givensovy matice **83**

G QR algoritmus pro výpočet vlastních čísel a vektorů matice **84**

H Výpis SVA kódu Householderovova QR rozkladu matice řádu 2 **85**

I Výpis SVA kódu QR rozkladu po základní optimalizaci **88**

J Výpis SVA kódu QR rozkladu po základní a redukující optimalizaci **91**

K Význam portů IP bloku **93**

L Rozhraní bloků pro výpočty s čísly s plovoucí desetinnou čárkou **95**

M Definice rozhraní bloku DSP_pipelineRAM **103**

N Definice rozhraní bloku DSP_corePipelineRAM **105**

O Simulace APU_NB a APU_NB_small **107**

P Simulace APU_small **108**

Q Obsah přiloženého CD **109**

SEZNAM OBRÁZKŮ

3.1 Číslo s plovoucí desetinnou čárkou 42
3.2 Vstupní a výstupní porty IP bloku 43
3.3 Neblokační mód 44
3.4 Blokační mód 45
3.5 Vnitřní architektura `FP_CHECK` 49
3.6 Vnitřní propojení bloků `APU_NB` během komplexního násobení 52
3.7 Vnitřní propojení bloků `APU_NB` během komplexního dělení 53
3.8 Vnitřní architektura `APU_NB_small` 54
3.9 Struktura bloku `DSP_pipelineRAM` 58
3.10 Znázornění funkce bloku `DSP_corePipelineRAM` 59
O.1 Simulace `APU_NB` a `APU_NB_small` 107
P.1 Simulace `APU_small` 108

SEZNAM TABULEK

2.1 Výsledky analýzy Householderovy redukce matice řádu 4 na matici třídiagonální 34
2.2 Výsledky analýzy Householderovy redukce matice řádu 8 na matici třídiagonální 34
2.3 Výsledky analýzy QR algoritmu pro matici řádu 2 35
2.4 Výsledky analýzy QR algoritmu pro matici řádu 4 36
2.5 Výsledky analýzy QR algoritmu pro matici řádu 8 36
2.6 Výsledky analýzy QR algoritmu pro komplexní matici řádu 2 36
2.7 Výsledky analýzy QR algoritmu pro komplexní matici řádu 4 37
2.8 Srovnání QR rozkladu pro reálné a komplexní matice 38
3.1 Speciální hodnoty čísla s plovoucí desetinnou čárkou 43
3.2 Možné hodnoty signálu `cmp_op` typu `FP_CMP_OP` 48
3.3 Funkce 50
3.4 Signál operation bloku APU_NB 52
3.5 Srovnání očekávaných a simulovaných výsledků vypočítaných aritmetickou jednotkou `APU_NB` 69
K.1 Význam portů IP bloku 93

SEZNAM VÝPISŮ

3.1 Definice typů . . . 40
3.2 Definice bloku FP_ADDSUB . . . 47
3.3 Definice bloku FP_CHECK . . . 49
3.4 Definice bloku APU_NB . . . 51
3.5 Definice bloku APU_NB_small . . . 53
3.6 Definice bloku APU_small . . . 55
3.7 Definice bloku QR_Algorithm . . . 56
3.8 Výchozí kód pro ilustraci optimalizací dle pravidel QRexpand . . . 62
3.9 Mezivýsledek optimalizace dle pravidel QRexpand . . . 62
3.10 Výsledek optimalizace dle pravidel QRexpand . . . 62
3.11 Výchozí kód pro ilustraci optimalizací dle pravidel QRreduce . . . 63
3.12 Mezivýsledek optimalizace dle pravidel QRreduce po eliminaci nadbytečných přiřazení . . . 63
3.13 Mezivýsledek optimalizace dle pravidel QRreduce po redukci přičítání nul, násobení nulou a násobení jedničkou . . . 63
3.14 Mezivýsledek optimalizace dle pravidel QRreduce po opakované redukci nadbytečných přiřazení . . . 63
3.15 Mezivýsledek optimalizace dle pravidel QRreduce po odstranění operací, jejichž výsledky nejsou použity . . . 64
3.16 Výsledek optimalizace dle pravidel QRreduce . . . 64
3.17 Simulační proces clk_process . . . 66
3.18 Signály pro proces clk_process . . . 66
3.19 Simulační proces watchdog_process . . . 67
3.20 Signály pro proces watchdog_process . . . 67
3.21 Signály pro proces sim_check_process . . . 67
3.22 Signály pro proces sim_check_process . . . 68
3.23 Obvyklé zapojení bloku FP_CHECK v simulaci . . . 68
A.1 Autokorelační matice s šumem . . . 76
B.1 Jacobiho metoda . . . 78
C.1 Householderova transformace . . . 80
D.1 QR rozklad pomocí Gram-Schmidtova algoritmu . . . 81
E.1 QR rozklad pomocí Householderových transformací . . . 82
F.1 QR rozklad pomocí Givensovy matice . . . 83
G.1 QR algoritmus pro výpočet vlastních čísel a vektorů matice . . . 84
H.1 Výpis SVA kódu QR rozkladu (přímo vygenerovaného) . . . 85
I.1 Výpis SVA kódu QR rozkladu (po základní optimalizaci) . . . 88
J.1 Výpis SVA kódu QR rozkladu (po základní a redukující optimalizaci) 91

L.1 Soubor FP.vhd 95
M.1 Definice rozhraní bloku DSP_pipelineRAM 103
N.1 Definice rozhraní bloku DSP_corePipelineRAM 105

ÚVOD

Tato práce se zabývá implementací výpočtu vlastních čísel a vlastních vektorů komplexní hermitovské pozitivně-semidefinitní čtvercové matice řádu 4 do hradlového pole řady Xilinx Zynq-7000 v jazyce VHDL v základní přesnosti čísel s plovoucí desetinnou čárkou. Pro výpočty s čísly s plovoucí desetinnou čárkou jsou využity IP bloky dodávané spolu s vývojovým prostředím Xilinx Vivado.

Potřeba výpočtu vlastních čísel a vektorů souvisí se zpracováním signálů přijímaných sekundárními radary, které pracující v oblastech s hustým leteckým provozem. Zde často dochází k časovému překryvu signálů vysílaných odpovídači jednotlivých letadel. Jeden ze způsobů řešení tohoto problému spočívá v nasazení metod pro separaci signálů. Firmou ERA a.s. vyvinutá metoda separace, pro niž je v této práci vyvinut algoritmus výpočtu vlastních čísel a vektorů, vyžaduje soustavu v prostoru vhodně umístěných antén, které jsou ideálně na sobě nezávislé. Každá z antén soustavy poté přijímá odlišný signál zatížený šumem.

Signály přijímané jednotlivými anténami jsou popsány autokorelační maticí zatíženou šumem. Tato autokorelační matice je čtvercová hermitovská a pozitivně-semidefinitní. Řád matice a počet jí příslušejících reálných kladných vlastních čísel a vektorů je dán množstvím antén.

Jestliže je přijímán jediný signál zatížený šumem, má autokorelační matice jedno nenásobné nenulové vlastní číslo a jemu odpovídající vlastní vektor, který odpovídá vektoru příjmu daného signálu. Ostatní vlastní čísla a vektory náleží šumu.

Nezávislá složka Gaussova šumu ovlivňuje velikost vlastních čísel, ale nikoli vlastní vektory příslušející vlastním číslům. Z vlastních čísel matice je možné odhadnout, kolik signálů se na přijímači právě překrývá. V případě, že je dostatečný odstup signál-šum a že jsou splněny další nutné podmínky (např. dostatečná vzájemná vzdálenost odpovídačů), lze odlišit nejen jednotlivé signály, ale i pozice jim příslušejících vysílačů v prostoru. Maximální počet separovaných signálů je dán počtem antén v soustavě. Vlastní čísla a vlastní vektory autokorelační matice určují prostor, ve kterém se nachází hledané vektory příjmu. Vektory příjmu však nelze bez další informace odhadnout (např. znalost prostorového rozložení antén).

Vektor příjmu (v případě jediného signálu vlastní vektor dominantního vlastního čísla) umožňuje, při znalosti soustavy antén, odhadnout i směr, odkud zachycený signál přišel.

V příloze A.1 lze nalézt skript pro GNU Octave generující autokorelační matici jednoho přijímaného signálu, zatíženou šumem, spolu s výpočtem vlastních čísel a vlastních vektorů s využitím prostředků GNU Octave.

1 KOMPLEXNÍ ČÍSLA A MATICE

Vstupem této práce je komplexní matice, a proto bude čtenář v této kapitole nejprve seznámen s komplexními čísly. Nikoli však vyčerpávajícím způsobem. Pro potřeby realizace výpočtů v hradlovém poli je zajímavý především výpočet součtu, rozdílu, součinu a podílu komplexních čísel.

Po popisu komplexních čísel následuje seznámení s maticemi, jejich podobami, vlastnostmi a základními výpočty (sčítání, násobení, apod.). S maticemi úzce souvisí také definice vlastních čísel a vlastních vektorů matice.

V závěru kapitoly jsou popsány různé algoritmy využitelné pro výpočet vlastních čísel a vlastních vektorů matic symetrických a hermitovských.

1.1 Komplexní čísla

Komplexní čísla jsou nástavbou reálných čísel. V oboru reálných čísel lze vypočítat většinu klasických operací jako je sčítání, odečítání, násobení a dělení (kromě dělení nulou). Nezáporná čísla lze v oboru reálných čísel také odmocňovat. Výpočet odmocniny záporných čísel lze řešit v oboru komplexních čísel.[1]

Komplexní číslo se skládá z reálné a imaginární části. Rozepsáním komplexního čísla $\boldsymbol{Z}$ na reálnou a imaginární část s použitím imaginární jednotky i dostaneme výraz

$$\boldsymbol{Z} = Z_{re} + Z_{im}\mathrm{i}\ , \tag{1.1}$$

kde je imaginární jednotka i definována vztahem

$$\mathrm{i} = \sqrt{-1}\ , \tag{1.2}$$

který je možné upravit do podoby

$$\mathrm{i}^2 = -1\ [1]. \tag{1.3}$$

Jestliže je imaginární část komplexního čísla rovna 0, lze komplexní číslo brát jako číslo reálné a tak s ním počítat. Reálné číslo lze chápat jako číslo komplexní s nulovou imaginární částí.

Dvě komplexní čísla $\boldsymbol{A}$ a $\boldsymbol{B}$ jsou si rovna, pokud jsou shodné jejich reálné (A_{re} a B_{re}) i imaginární složky (A_{im} a B_{im}). Platí-li $A_{re} = B_{re}$ a současně $A_{im} = B_{im}$, platí také $\boldsymbol{A} = \boldsymbol{B}$.

Jestliže je komplexní číslo $\boldsymbol{B}$ komplexně sdruženým číslem k číslu $\boldsymbol{A}$, pak platí

$$\begin{aligned} B_{re} &= A_{re} \\ B_{im} &= -A_{im}\ [1]. \end{aligned} \tag{1.4}$$

Jiným způsobem zapsáno $\boldsymbol{B} = \overline{\boldsymbol{A}}$.

Pro komplexní číslo $\boldsymbol{C}$ platí

$$\begin{aligned} C_{re} &= A_{re} + B_{re} \\ C_{im} &= A_{im} + B_{im}\ , \end{aligned} \tag{1.5}$$

jestliže je součtem komplexních čísel $\boldsymbol{A}$ a $\boldsymbol{B}$. Pro komplexní číslo $\boldsymbol{C}$ vzniklé rozdílem čísel $\boldsymbol{A}$ a $\boldsymbol{B}$ platí

$$\begin{aligned} C_{re} &= A_{re} - B_{re} \\ C_{im} &= A_{im} - B_{im}\ . \end{aligned} \tag{1.6}$$

Pokud je komplexní číslo $\boldsymbol{C}$ dáno součinem komplexních čísel $\boldsymbol{A}$ a $\boldsymbol{B}$, bude pro výsledek násobení platit:

$$\begin{aligned} C_{re} &= A_{re} * B_{re} - A_{im} * B_{im} \\ C_{im} &= A_{re} * B_{im} + A_{im} * B_{re} \,. \end{aligned} \tag{1.7}$$

Dělení komplexních čísel je řešitelné pomocí násobení jednotkovým zlomkem tvořeným jmenovatelem a čitatelem komplexně sdruženým k děliteli. Pakliže $\boldsymbol{C}$ značí podíl komplexních čísel $\boldsymbol{A}$ a $\boldsymbol{B}$, lze dělení komplexních čísel popsat výrazem

$$\boldsymbol{C} = \frac{\boldsymbol{A}}{\boldsymbol{B}} = \frac{A_{re} + A_{im}\mathrm{i}}{B_{re} + B_{im}\mathrm{i}} \,. \tag{1.8}$$

Roznásobením podílu komplexně sdruženou jedničkou $\frac{B_{re}-B_{im}\mathrm{i}}{B_{re}-B_{im}\mathrm{i}}$ k jmenovateli $\boldsymbol{B}$ lze vztah 1.8 upravit do tvaru

$$\boldsymbol{C} = \frac{A_{re} + A_{im}\mathrm{i}}{B_{re} + B_{im}\mathrm{i}} = \frac{A_{re} + A_{im}\mathrm{i}}{B_{re} + B_{im}\mathrm{i}} \frac{B_{re} - B_{im}\mathrm{i}}{B_{re} - B_{im}\mathrm{i}} = \frac{(A_{re} + A_{im}\mathrm{i})\,(B_{re} - B_{im}\mathrm{i})}{B_{re}^2 + B_{im}^2} \,, \tag{1.9}$$

který je možné rozepsat do podoby

$$\begin{aligned} C_{re} &= \frac{A_{re} * B_{re} + A_{im} * B_{im}}{B_{re}^2 + B_{im}^2} \\ C_{im} &= \frac{A_{im} * B_{re} - A_{re} * B_{im}}{B_{re}^2 + B_{im}^2} \,. \end{aligned} \tag{1.10}$$

1.2 Matice

Matice $\mathbf{A}$ typu (m, n) je obecně definována

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix} = (a_{ij})_m^n \text{ [2].} \tag{1.11}$$

Pokud je matice reálná, lze psát $\mathbf{A} \in \mathbb{R}^{n\times m}$, případně $\mathbf{A} \in \mathbb{C}^{n\times m}$ pro komplexní matici.

Čísla a_{ij} se nazývají prvky matice. Vektor tvořený prvky $(a_{i1}, a_{i2}, \ldots, a_{in})$ je i-tým řádkem matice, zatímco vektor $(a_{1j}, a_{2j}, \ldots, a_{nj})$ je j-tým sloupcem matice.[2]

1.2.1 Operace sčítání, odčítání a násobení

Dále jsou pro potřeby této práce definovány základní aritmetické operace nad maticemi.

Matice $\mathbf{C} = (c_{ij})_m^n$ řádu (m, n) je součtem matic $\mathbf{A} = (a_{ij})_m^n$ řádu (m, n) a $\mathbf{B} = (b_{ij})_m^n$ řádu (m, n), jestliže pro prvky matice $\mathbf{C}$ platí

$$c_{ij} = a_{ij} + b_{ij} \text{ [2].} \tag{1.12}$$

V maticovém zápisu psáno $\mathbf{C} = \mathbf{A} + \mathbf{B}$.

Obdobně je matice $\mathbf{C}$ rozdílem matic $\mathbf{A}$ a $\mathbf{B}$, psáno $\mathbf{C} = \mathbf{A} - \mathbf{B}$, pokud pro prvky matice $\mathbf{C}$ platí

$$c_{ij} = a_{ij} - b_{ij}. \tag{1.13}$$

Platí-li pro prvky matice $\mathbf{C}$

$$c_{ij} = -a_{ij}, \tag{1.14}$$

lze tuto skutečnost zapsat výrazem $\mathbf{C} = -\mathbf{A}$.

Je-li matice $\mathbf{C}$ získána vynásobením matice $\mathbf{A}$ číslem α, zapsáno $\mathbf{C} = \alpha\mathbf{A}$, platí pro prvky matice $\mathbf{C}$ vztah

$$c_{ij} = \alpha a_{ij} \text{ [2].} \tag{1.15}$$

Součin matice $\mathbf{A} = (a_{ij})_m^p$ typu (m, p) s maticí $\mathbf{B} = (b_{ij})_p^n$ typu (p, n) tvoří matici $\mathbf{C} = (c_{ij})_m^n$ typu (m, n), pro jejíž prvky platí

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ip}b_{pj} = \sum_{r=1}^{p} a_{ir}b_{rj} \text{ .[2]} \tag{1.16}$$

Bude-li součinem matic matice typu $(1,1)$, s výsledkem je možné pracovat jako s reálným nebo komplexním číslem podle typu násobených matic.

1.2.2 Speciální případy matic

Speciálními případy matic jsou rozuměny matice mající určité specifické vlastnosti, které je odlišují od matic ostatních.

Matice $\mathbf{A}$ typu (m,n) je nazývána čtvercovou maticí $\mathbf{A}$ řádu n v případě, kdy je m rovno n. Pro čtvercovou matici $\mathbf{A}$ je možné zápis 1.11 zjednodušit do tvaru

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nj} & \cdots & a_{nn} \end{bmatrix} = (a_{ij})_n^n \ . \tag{1.17}$$

Prvky $a_{ii}, i = 1, \ldots, n$ tvoří hlavní diagonálu čtvercové matice $\mathbf{A}$ řádu n. Jsou-li všechny prvky matice neležící na hlavní diagonále rovny nule, nazýváme matici diagonální.

Jednotkovou maticí je myšlena diagonální matice, která má všechny prvky na hlavní diagonále rovné jedné.

Horní trojúhelníková matice má všechny prvky pod hlavní diagonálou rovné nule. Dolní trojúhelníková matice má rovné nule naopak všechny prvky nad hlavní diagonálou.

Čtvercová matice $\mathbf{A} = (a_{ij})_n^n$, pro jejíž prvky platí $a_{ij} = a_{ji}$, se nazývá symetrická[2].

Třídiagonální čtvercová matice řádu n má podobu

$$\begin{bmatrix} b_1 & c_1 & & \\ d_1 & \ddots & \ddots & \\ & \ddots & \ddots & c_n - 1 \\ & & d_{n-1} & b_n \end{bmatrix} \tag{1.18}$$

a symetrická třídiagonální čtvercová matice řádu n

$$\begin{bmatrix} b_1 & c_1 & & \\ c_1 & \ddots & \ddots & \\ & \ddots & \ddots & c_n - 1 \\ & & c_{n-1} & b_n \end{bmatrix} \text{ [3].} \tag{1.19}$$

Čtvercová matice $\mathbf{B}$ řádu n je transponovaná k čtvercové matici $\mathbf{A}$ řádu n, jestliže platí $a_{ij} = b_{ji}, i = 1, \ldots, n, j = 1, \ldots, n$. Matici $\mathbf{B}$ lze zapsat jako $\mathbf{A}^T$. Je-li $\mathbf{A}$ symetrická matice, platí $\mathbf{A}^T = \mathbf{A}$[2].

Čtvercová matice $\mathbf{B}$ řádu n je inverzní maticí čtvercové matice $\mathbf{A}$ řádu n, jestliže platí vztah

$$\mathbf{BA} = \mathbf{AB} = \mathbf{I}\ [2], \tag{1.20}$$

kde $\mathbf{I}$ je jednotková matice řádu n. Inverzní matice $\mathbf{B}$ k matici $\mathbf{A}$ je značena symbolem $\mathbf{A}^{-1}$.

Pro komplexně sdruženou matici $\mathbf{B}$ řádu n k matici $\mathbf{A}$ řádu n platí $b_{ij} = \overline{a_{ij}}, i = 1, \ldots, n, j = 1, \ldots, n$ za předpokladu, že $\overline{a_{ij}}$ značí komplexně sdružené číslo ke komplexnímu číslu a_{ij}. Komplexně sdružená matice k matici $\mathbf{A}$ je značena zápisem $\overline{\mathbf{A}}$.

Hermitovskou maticí je matice, pro niž platí $\mathbf{A} = \overline{\mathbf{A}^T}$[4]. Což lze značit jednodušeji jako $\mathbf{A} = \mathbf{A}^H$.

Ortogonální maticí je rozuměna matice, pro kterou platí $\mathbf{A}^{-1} = \mathbf{A}^T$[4].

Pokud platí $\mathbf{A}^{-1} = \overline{\mathbf{A}^T}$, je matice nazývána unitární.[4]

Aritmetický vektor dimense n je v této práci chápán jako sloupcová matice typu $(n, 1)$, tedy

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}\ [2]. \tag{1.21}$$

Reálná symetrická či hermitovská matice $\mathbf{A}$ řádu n je považována za matici pozitivně definitní, jestliže pro každý nenulový reálný či komplexní vektor $\mathbf{x}$ dimense n platí

$$\mathbf{x}^H\mathbf{Ax} > 0\ [5]. \tag{1.22}$$

Platí-li pro každý nenulový reálný či komplexní vektor $\mathbf{x}$ dimense n

$$\mathbf{x}^H\mathbf{Ax} \geq 0\ [5], \tag{1.23}$$

je matice pozitivně semidefinitní.

1.2.3 Determinant

Pro definici determinantu je nutné nejprve definovat permutaci.

Pro množinu M o velikosti n je permutací libovolná n-tice složená z prvků množiny M, přičemž se žádný prvek neopakuje. Příkladem permutací pro množinu $M = (a, b, c, d)$ jsou n-tice prvků (a, b, d, c) nebo (d, b, a, c). Počet všech různých permutací je roven

$$P(n) = n!\ [6], \tag{1.24}$$

kde $n!$ značí faktoriál n.[6]

Inverzí v permutaci $p = (p(1)p(2)\dots p(n))$ je zvána dvojice (i, j) taková, že $i < j, p(i) > p(j)$. Permutace p je sudá (resp. lichá), má-li sudý (resp. lichý) počet inverzí. Číslo $(-1)^k$, kde k je počet inverzí v permutaci p, se nazývá znaménko permutace p a značí se $sgn(p)$.[2]

Nechť je dána čtvercová matice $\mathbf{A} = (a_{ij})_n^n$ řádu n. Nechť $(p(1)p(2)\dots p(n))$ je libovolná permutace čísel $1, 2, \dots, n$ (permutací je $n!$). Součet součinů $a_{1p(1)} \cdot a_{2p(2)} \cdot a_{3p(3)} \cdot \dots \cdot a_{np(n)}$ provedených pro všechny permutace a vynásobených číslem (-1) v případě, že je permutace lichá a číslem 1 v případě sudé permutace, se nazývá determinant n-tého řádu matice $\mathbf{A}$ a označuje se $|\mathbf{A}|$. Takových součinů je $n!$.[2]

Platí tedy

$$|\mathbf{A}| = \sum_p sgn(p) a_{1p(1)} \cdot a_{2p(2)} \cdot \dots \cdot a_{np(n)} \text{ [2]}, \tag{1.25}$$

kde se sčítá přes všechny permutace p množiny $1, 2, \dots, n$.[2]

Pro čtvercovou matici $\mathbf{A}$ platí $|\mathbf{A}^T| = |\mathbf{A}|$[2].

1.2.4 Norma vektoru

Normou vektoru $\mathbf{x}$ dimense n

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{C}^{n \times 1} \tag{1.26}$$

rozumíme reálnou funkci $\|\mathbf{x}\|$ na vektorovém prostoru V, platí-li pro libovolné dva vektory $\mathbf{x}$, $\mathbf{y}$ z V a libovolné $k \in \mathbb{C}$[7]:

1. $\|\mathbf{x}\| \geq 0$ (pozitivnost)[7],
2. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ (trojúhelníková nerovnost)[7],
3. $\|k\mathbf{x}\| = |k| \cdot \|\mathbf{x}\|$ (homogenita) [7] a
4. $\|\mathbf{x}\| = 0 \Leftrightarrow \mathbf{x} = \mathbf{0}$ (pozitivní definitnost)[7].

P-norma vektoru $\mathbf{x} \in \mathbb{C}^{n \times 1}$ dimense n je pro reálné číslo $p \geq 1$ definována vztahem

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}} \text{ [8]}, \tag{1.27}$$

kde $|x_i|$ značí absolutní hodnotu.

Speciálním a často používaným případem je norma Euklidovská. Jedná se o p-normu, kde $p = 2$[8]. Euklidovskou normu lze zapsat ve tvaru

$$\|\mathbf{x}\|_2 = \sqrt{\left(\sum_{i=1}^{n} x_i^2 \right)} = \sqrt{x_1^2 + \dots + x_n^2}\,. \tag{1.28}$$

Nebo jednodušeji pomocí násobení vektorů pro $\mathbf{x} \in \mathbb{R}^{n\times 1}$

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T\mathbf{x}}\ [8], \tag{1.29}$$

respektive $\mathbf{x} \in \mathbb{C}^{n\times 1}$

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^H\mathbf{x}}\ [8]. \tag{1.30}$$

1.3 Vlastní čísla a vlastní vektory

Pro čtvercovou matici $\mathbf{A}$ řádu n jsou vlastní čísla $\lambda_1, ..., \lambda_n$ definována jako kořeny charakteristické rovnice matice

$$det(\mathbf{A} - \lambda I) = 0 \text{ [9]}, \tag{1.31}$$

kde matice $\mathbf{I}$ představuje jednotkovou matici. Roznásobením lze dostat charakteristickou rovnici ve tvaru

$$f(\lambda) = \lambda^n + a_{n-1}\lambda^{n-1} + ... + a_1\lambda^1 + a_0 = 0 \text{ [9]}. \tag{1.32}$$

Ke každému vlastnímu číslu λ_i existuje alespoň jedno nenulové řešení soustavy rovnic

$$\mathbf{Ax} = \lambda_i\mathbf{x} \text{ [9]}. \tag{1.33}$$

Toto řešení $\mathbf{x_i}$, kde $\mathbf{x_i}^T = (x_i^{(1)}, ..., x_i^{(n)})$, je pravým vlastním vektorem matice $\mathbf{A}$[9]. V dalším textu je pod pojmem vlastní vektor chápán pravý vlastní vektor matice.

Vlastní čísla matice lze hledat přímo jako kořeny charakteristické rovnice matice. Tento postup je však u velkých matic značně nepraktický. V případě počítačového zpracování je problematická i nevylučitelná nejednoznačnost řešení soustavy rovnic [9].

V případě čtvercové komplexní hermitovské pozitivně-semidefinitní matice jsou vlastní čísla kladná a reálná[5].

V rámci této práce je popsána Jacobiho metoda a QR algoritmus. V obou případech jde o numerické metody pro hledání vlastních čísel a vlastních vektorů matice. Jedná se o iterační metody aplikovatelné na matici reálnou i komplexní. Aby QR algoritmus rychleji konvergoval, lze matici, pokud je reálná, nejdříve upravit pomocí Householderovi redukce na třídiagonální tvar. (dle [9] a [10])

1.4 Householderova redukce na třídiagonální tvar

Householderovi transformace je možné použít pro převedení symetrické reálné matice na matici třídiagonální. Výsledná třídiagonální matice má stejná vlastní čísla jako matice původní. Avšak hledání vlastních čísel třídiagonální matice je snazší než hledání vlastních čísel matice původní. (dle [9] a [10])

Je-li $\mathbf{A}$ čtvercovou symetrickou reálnou maticí řádu n, je postupně určeno $n-2$ ortogonálních matic $\mathbf{H_1}, \ldots, \mathbf{H_{n-2}}$. Matice $\mathbf{A_0}$ je rovna matici $\mathbf{A}$, jež je transformována do třídiagonální podoby. Následně je možno psát

$$\begin{aligned} \mathbf{A_k} &= \mathbf{H_{k-1}}^T \cdot \mathbf{A_{k-1}} \cdot \mathbf{H_{k-1}} = \\ &= \left(\mathbf{H_1} \ldots \mathbf{H_{k-1}}\right)^T \cdot \mathbf{A} \cdot \left(\mathbf{H_1} \ldots \mathbf{H_{k-1}}\right), \; k = 1, \ldots, n-2 \; [9]. \end{aligned} \tag{1.34}$$

Platnosti 1.34 lze dosáhnout volbou vektoru $\mathbf{v_k}$ v podobě

$$\mathbf{v_k} = \left[0, \ldots, 0, a_{k+1,k}^{k-1} \pm \left(\sum_{i=k+1}^{n} |a_{i,k}^{k-1}|^2\right), a_{k+1,k}^{k-1}, \ldots, a_{n,k}^{k-1}\right]^T .[9] \tag{1.35}$$

Vektor $\mathbf{w_k}$ je pak dán vztahem

$$\mathbf{w_k} = \left(\mathbf{v_k}^T \mathbf{v_k}\right)^{-0.5} \mathbf{v_k} \; [9] \tag{1.36}$$

a vektor $\mathbf{q_k}$ výrazem

$$\mathbf{q_k} = 2(I - \mathbf{w_k}\mathbf{w_k}^T)\mathbf{A_{k-1}}\mathbf{w_k} \; [9], \tag{1.37}$$

kde $\mathbf{I}$ představuje jednotkovou matici řádu n.

To umožňuje vyjádřit matici $\mathbf{A_k}$ pomocí matice $\mathbf{A_{k-1}}$ a vektorů $\mathbf{w_k}$ a $\mathbf{q_k}$:

$$\mathbf{A_k} = \mathbf{A_{k-1}} - \mathbf{w_k}\mathbf{q_k}^T - \mathbf{q_k}\mathbf{w_k}^T \; [9]. \tag{1.38}$$

Algoritmus je zpracován v GNU Octave viz. příloha C.1.

1.5 Jacobiho metoda

Jacobiho iterační metodou lze nalézt vlastní čísla a vlastní vektory symetrické reálné matice i hermitovské komplexní matice[11]. Vhodná je hlavně pro plné matice[9].

K symetrické, popřípadě hermitovské, matici existuje ortogonální matice $\mathbf{T}$ složená z vlastních vektorů[9], pro niž platí vztah

$$\mathbf{A} = \mathbf{T^T} diag(\lambda_1, \ldots, \lambda_n)\mathbf{T} \text{ [9]}, \tag{1.39}$$

kde $diag(\lambda_1, \ldots, \lambda_n)$ představuje diagonální matici s prvky $\lambda_1, \ldots, \lambda_n$ na diagonále.

Tato metoda využívá Givensovy matice rotace ve tvaru

$$\mathbf{G}(p,q) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\phi & \cdots & \sin\phi & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin\phi & \cdots & \cos\phi & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \text{ [11]}. \tag{1.40}$$

Goniometrické funkce se nacházejí na řádcích a sloupcích s indexy p a q, kde $p < q$. Givensovu matici použijeme k vynulování prvku v p-tém řádku a q-tém sloupci matice $\mathbf{S_k}$ (kde k značí iteraci a $k \geq 1$).

Během výpočtu je sestavena posloupnost elementárních ortogonálních matic $\mathbf{S_k}$ taková, aby platil vztah

$$\mathbf{A_{k+1}} = \mathbf{S_k}^T \mathbf{A_k} \mathbf{S_k} = (\mathbf{S_1} \ldots \mathbf{S_k})^T \mathbf{A_1} (\mathbf{S_1} \ldots \mathbf{S_k}) \text{ [9]}, \tag{1.41}$$

kde $\mathbf{A_1} = \mathbf{A}$.

Protože matice $\mathbf{A_k}$ jsou podobné matici $\mathbf{A}$, mají stejná vlastní čísla[9].

Při transformaci $\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}$ se mění pouze p-té řádky a q-té sloupce matice $\mathbf{A}$, přesněji pro zvolené ϕ:

- $b_{ij} = a_{ij}$ pro $i \neq p, q$ a $j \neq p, q$ [9]
- $b_{pj} = b_{ip} = a_{pi}\cos\phi - a_{qi}\sin\phi$ pro $i \neq p, q$ [9]
- $b_{qi} = b_{iq} = a_{pi}\sin\phi - a_{qi}\cos\phi$ pro $i \neq p, q$ [9]
- $b_{pp} = a_{pp}\cos^2\phi a_{qq}\sin^2\phi - a_{pq}\sin 2\phi$ [9]
- $b_{qq} = a_{pp}\sin^2\phi a_{qq}\cos^2\phi - a_{pq}\sin 2\phi$ [9]
- $b_{pq} = b_{qp} = a_{pq}\cos 2\phi \frac{1}{2}(a_{pp} - a_{qq})\sin 2\phi$ [9]

V každém kroku Jacobiho metody je sestrojena z matice $\mathbf{A_k} = \left[a_{ij}^k\right]$ matice $\mathbf{A_{k+1}}$. Čísla p a q jsou volena tak, aby platilo

$$a_{pq}^k \neq 0 \text{ [9]}. \tag{1.42}$$

Matice $\mathbf{S_k}$ je sestrojena jako Givensova matice pro $\phi \in (-\frac{\pi}{4}, 0) \cup (0, \frac{\pi}{4})$ dle vztahu

$$\cot 2\phi_k = \frac{a_{qq}^k - a_{pp}^k}{2a_{pq}^k} \text{ [9]}. \tag{1.43}$$

Čísla (p, q) lze volit dle různých strategií. Při klasické metodě jsou hodnoty (p, q) určeny tak, aby prvek a_{pq}^k byl největším nediagonálním prvkem matice $\mathbf{A_k}$. Další možností výběru je cyklická metoda volby (p, q). Při této metodě jsou postupně ve smyčce nulovány prvky matice. Nulové prvky a_{pq}^k se přeskakují. Příkladem volby (p, q) je $(1,2)(1,3)\ldots(1,n);(2,3)\ldots(2,n);\ldots;(n-1,n)$. Prahová metoda volby (p, q) se liší od cyklické metody nulováním prvků a_{pq}^k, jejichž hodnota je větší než určitá prahová hodnota, která se zmenšuje s každou smyčkou.[9]

Zapsáno v GNU Octave viz příloha B.1 (dle Martina Mrovce[11]).

1.6 QR rozklad

Jednou z metod výpočtu vlastních čísel a vlastních vektorů matice je QR algoritmus, jehož podstatou je opakovaný výpočet QR rozkladu matice. Z hlediska hledání vlastních čísel a vektorů se jedná o iterační algoritmus. Před nasazením QR rozkladu matice je možné matici upravit kupříkladu na třídiagonální a tím urychlit konvergenci. QR rozklad lze aplikovat na reálné i komplexní matice.

Princip QR rozkladu matice $\mathbf{A}$ spočívá v nalezení matic $\mathbf{Q}$ a $\mathbf{R}$ tak, aby platilo

$$\mathbf{A} = \mathbf{QR} \text{ [9]}. \tag{1.44}$$

Přičemž $\mathbf{Q}$ je ortogonální matice a $\mathbf{R}$ je horní trojúhelníková matice.

Existují různé postupy výpočtu QR rozkladu. Zde jsou blíže popsány realizace QR rozkladu pomocí Gram-Schmidtova algoritmu, Givensova algoritmu a Householderových matic.

1.6.1 Gram-Schmidtův QR rozklad

Pro potřeby Gram-Schmidtova QR rozkladu zapíšeme čtvercovou matici $\mathbf{A}$ řádu n pomocí sloupcových vektorů:

$$\mathbf{A} = (\mathbf{a_1}|\dots|\mathbf{a_n}) \text{ [9]}. \tag{1.45}$$

Vektory $\mathbf{u_k}$ a $\mathbf{e_k}$ lze pak zapsat

$$\begin{aligned} \mathbf{u_k} &= \mathbf{a_k} - \sum_{j=1}^{k-1} (\mathbf{a_k}^T \mathbf{e_j}) \mathbf{e_j} \\ \mathbf{e_k} &= \frac{\mathbf{u_k}}{\|\mathbf{u_k}\|_2} \text{ [12]}, \end{aligned} \tag{1.46}$$

přičemž $k \in 1, \dots, n$.

Matice $\mathbf{Q}$ je nyní tvořena sloupcovými vektory $\mathbf{e_1}, \dots, \mathbf{e_k}$ a lze ji zapsat:

$$\mathbf{Q} = (\mathbf{e_1}|\dots|\mathbf{e_n}) \text{ [9]}. \tag{1.47}$$

Matici $\mathbf{R}$ je možné dopočítat dle vztahu

$$\mathbf{R} = \mathbf{Q}^T \mathbf{A} \text{ [9]}. \tag{1.48}$$

Přepsáno do GNU Octave viz. příloha D.1.

1.6.2 Householderův QR rozklad

Každou čtvercovou matici $\mathbf{A}$ řádu n lze pomocí $n-1$ Householderových matic rozložit na součin $\mathbf{QR}$ tak, že platí

$$\mathbf{H_{n-1}H_{n-2}}\cdots\mathbf{H_2H_1A} = \mathbf{Q}^T\mathbf{A} = \mathbf{R}\ [9]. \tag{1.49}$$

Principem výpočtu QR rozkladu pomocí Householderových matic je postupné nulování všech prvků pod hlavní diagonálou matice. Postupuje se od prvního sloupce až po $n-1$ sloupec.

Matice $\mathbf{A_0}$ se rovná matici $\mathbf{A}$, jejíž QR rozklad je hledán. Matice $\mathbf{H_k}$ pro k-tý sloupec je pak určena tak, aby matice $\mathbf{H_kA_{k-1}}$ měla v k-tém sloupci pod diagonálou samé nuly. Toho lze dosáhnout pomocí vektoru $\mathbf{u_k}$, voleného tak, že pro

$$\mathbf{H_k} = \mathbf{E} - 2\frac{\mathbf{u_k}\mathbf{u_k}^T}{\mathbf{u_k}^T\mathbf{u_k}}\ [9] \tag{1.50}$$

jsou prvky matice $\mathbf{H_k}$ pod hlavní diagonálou v prvním až k-tém sloupci nulové. To vše za předpokladu, že $\mathbf{E}$ je jednotková matice řádu n a vektor $\mathbf{u_k}$ je sloupcový. Pro matici $\mathbf{A_k}$ pak platí

$$\mathbf{A_k} = \mathbf{H_kA_{k-1}}\ [9], \tag{1.51}$$

což odpovídá volbě

$$\mathbf{u_k} = \mathbf{x_k} - \|\mathbf{x_k}\|_2\mathbf{e_k}\ [9], \tag{1.52}$$

kde vektor $\mathbf{x_k}$ je k-tým sloupcem matice $\mathbf{A_{k-1}}$ a $\mathbf{e_k}$ je sloupcový vektor jehož prvky vyjma k-tého prvku jsou rovny nule a k-tý prvek je roven jedné.

Pro $k = 1, \ldots, n-1$ platí

$$\mathbf{R} = \mathbf{A_{n-1}} = \mathbf{H_{n-1}} * \mathbf{A_{n-2}} = \ldots = \mathbf{H_{n-1}H_{n-2}}\cdots\mathbf{H_2H_1A}\ [9] \tag{1.53}$$

a

$$\mathbf{Q} = \mathbf{H_1}^T\mathbf{H_2}^T\cdots\mathbf{H_{n-1}}^T = \ldots = \mathbf{H_1H_2}\cdots\mathbf{H_{n-1}}\ [9]. \tag{1.54}$$

Výpočetní náročnost rozkladu čtvercové matice řádu n odpovídá přibližně

$$n^2\left(3 - \frac{n}{3}\right)\ [9] \tag{1.55}$$

operacím. Pokud je vyjádřena i matice $\mathbf{Q}$, celkový počet operací je dán vztahem

$$2m^2n - mn^2 + \frac{1}{3}n^3\ [9]. \tag{1.56}$$

Jelikož je vstupní matice čtvercová, platí $m = n$, a proto lze rovnici 1.56 zjednodušit do tvaru

$$2n^3 - n^3 + \frac{1}{3}n^3 = \frac{4}{3}n^3\ . \tag{1.57}$$

Přepsáno do GNU Octave viz. příloha E.1.

1.6.3 QR rozklad pomocí Givensovy matice

Givensovou maticí nazýváme matice ve tvaru

$$\mathbf{G}(i,j,c,s) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \text{ [9]}, \tag{1.58}$$

kde $c^2 + s^2 = 1$. Indexy i a j označují řádky a sloupce s prvky c a s[13].

Je možné volit $c = \cos\alpha$ a $s = \sin\alpha$ pro určité α. Odpovídající Givensova matice je značena $\mathbf{G}(i,j,\alpha)$.[9]

Je-li vektor $\mathbf{x} \in R^n$ a $\mathbf{y} = \mathbf{G}(i,j)^T\mathbf{x}$[13], potom

$$y_k = \begin{cases} cx_i - sx_l & prok = i \\ sx_i + cx_k & prok = j \\ x_j & prok \neq i,j \end{cases} \quad [13]. \tag{1.59}$$

Je potřebné, aby k-tá složka vektoru y byla nulová. Musí tedy platit

$$c = \frac{x_i}{\sqrt{x_i^2+x_j^2}}, \quad s = \frac{-x_j}{\sqrt{x_i^2+x_j^2}} \quad [13]. \tag{1.60}$$

Středem zájmu jsou pouze řádky a sloupce i a j, proto je uvažován prostor R^2. Cílem je vynulovat druhou složku vektoru $y \in R^2$. To je potřebné pro transformaci matice $\mathbf{A}$ na matici $\mathbf{R}$, která bude mít pod diagonálou nuly. Rozepsáním uvažovaného vztahu $\mathbf{y} = \mathbf{G}(i,j)^T\mathbf{x}$ dostaneme vztah[13]

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \lambda \begin{bmatrix} r \\ 0 \end{bmatrix} \text{ [13]}. \tag{1.61}$$

K poddiagonálním prvkům zadané čtvercové matice $\mathbf{A}$ řádu n jsou počítána čísla c, s určující Givensovu rotaci. Příslušný blok matice $\mathbf{A}$ je poté zleva přenásoben maticí rotace, a tím jsou v matici $\mathbf{A}$ získávány nuly pod diagonálou. Platí $\mathbf{Q}^T\mathbf{A} = \mathbf{R}$. Ortogonální matici $\mathbf{Q}$ dopočítáme z matice Givensovy rotace $\mathbf{G_g}$ dle vztahu $\mathbf{Q} = \mathbf{G_1} \ldots \mathbf{G_t}$, kde t značí počet rotací.[13]

Výpočet rozkladu čtvercové matice řádu n vyžaduje

$$2n^2\left(3 - \frac{n}{3}\right) \text{ [9]} \tag{1.62}$$

operací pro vyjádření matice $\mathbf{R}$.

Přepsáno do GNU Octave viz. příloha F.1.

1.6.4 Srovnání QR rozkladů

Gram-Schmidtův algoritmus QR rozkladu vykazuje velice špatnou ortogonalitu matice $\mathbf{Q}$[13]. Podle Tomáše Oberhubera[14] je také numericky nestabilní. Gram-Schmidtův QR rozklad proto není hodnocen jako vhodný k výpočtu vlastních čísel a vlastních vektorů vstupní matice. Z hlediska výpočetní náročnosti vychází QR rozklad realizovaný pomocí Householderových matic lépe.

1.7 QR algoritmus pro vlastní čísla a vlastní vektory

Opakovaným QR rozkladem lze získat všechna vlastní čísla a všechny vlastní vektory matice. Pokud je matice $\mathbf{A}$ čtvercovou maticí řádu n, pak platí, že matice $\mathbf{T_0} = \mathbf{A}$. Následně jsou hledány QR rozklady matic $\mathbf{T_k}$ ($k \in 0, 1, 2, \ldots$). Zapsáno pro $k \geq 1$

$$\mathbf{Q_k}, \mathbf{R_k} = \mathbf{T_{k-1}} \, . \tag{1.63}$$

Matice $\mathbf{T_k}$ je dopočítána vztahem

$$\mathbf{T_k} = \mathbf{R_k}\mathbf{Q_k} \, .[10] \tag{1.64}$$

Každá z matic $\mathbf{T_k}$ je ortogonálně podobná matici $\mathbf{A}$ (dle [10]) a má tedy shodná vlastní čísla. Pokud jsou vlastní čísla matice $\mathbf{A}$ reálná a kladná (což je u čtvercové komplexní hermitovské positivně semidefinitní matice zaručeno) opakováním QR rozkladu dostáváme matice $\mathbf{R_k}$ ve tvaru horní trojúhelníkové matice. Vlastní čísla se nachází na diagonále matice $\mathbf{T_k}$. Takto nalezená vlastní čísla jsou seřazená a platí $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$. S počtem opakování iterací je dosahováno vyšší přesnosti výpočtu. (dle [9] a [10])

Vlastní sloupcové vektory x matice $\mathbf{A}$ lze získat z matic $\mathbf{Q_k}$ použitím vztahu

$$x = \mathbf{Q_0}\mathbf{Q_1} \ldots \mathbf{Q_k} \, .[9] \tag{1.65}$$

Zapsáno s pomocí GNU Octave viz. příloha G.1.

1.8 Hledání vlastních čísel a vlastních vektorů hermitovské matice

Problém určení vlastních čísel a vlastních vektorů komplexní matice je možné v případě hermitovské matice, pro kterou platí $\mathbf{A} = \bar{\mathbf{A}^T}$[4], převést na problém hledání vlastních čísel a vlastních vektorů reálné matice.

Pokud pro hermitovskou matici $\mathbf{C}$ a reálné matice $\mathbf{A}$ a $\mathbf{B}$ platí

$$\mathbf{C} = \mathbf{A} + i\mathbf{B} \text{ [15]}, \tag{1.66}$$

platí dle[15] také

$$(\mathbf{A} + \mathrm{i}\mathbf{B}) \cdot (\mathbf{u} + \mathrm{i}\mathbf{v}) = \lambda(\mathbf{u} + \mathrm{i}\mathbf{v}) \text{ [15]} \tag{1.67}$$

a

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \text{ [15]}. \tag{1.68}$$

Pro potřeby následujícího textu je dáno značení

$$\mathbf{R} = \begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \tag{1.69}$$

a

$$\mathbf{r} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}. \tag{1.70}$$

O vlastních číslech $\lambda_1, ..., \lambda_n$ matice $\mathbf{C}$ lze tvrdit

$$\lambda = (\lambda_1, \lambda_1, \lambda_2, \lambda_2, \ldots, \lambda_n, \lambda_n) \text{ [15]}. \tag{1.71}$$

Pro k-tý vlastní vektor $\mathbf{x_k}$ hermitovské matice $\mathbf{C}$ náležející vlastnímu číslu λ_k platí dle [9]

$$\mathbf{C}\mathbf{x_k} = \lambda_k \mathbf{x_k}. \tag{1.72}$$

Vlastní vektory matice $\mathbf{C}$ lze vyjádřit z vlastních vektorů $\mathbf{r}$ matice $\mathbf{R}$. Jestliže je $\mathbf{r}$ vlastním vektorem matice $\mathbf{R}$, pro první vlastní číslo z páru platí

$$\mathbf{x} = \mathbf{u} + \mathrm{i}\mathbf{v} \text{ [15]} \tag{1.73}$$

a pro druhé vlastní číslo z páru platí

$$\mathbf{x} = \mathrm{i}(\mathbf{u} + \mathrm{i}\mathbf{v}) = -\mathbf{v} + \mathrm{i}\mathbf{u} \text{ [15]}. \tag{1.74}$$

2 ANALÝZA ALGORITMŮ

Výpočet vlastních čísel a vektorů lze provést různými způsoby. Některé z nich byly podrobeny analýze s cílem výběru nejoptimálnějšího algoritmu z algoritmů posuzovaných.

Pro možnost porovnání a následné optimalizace byly algoritmus pro transformaci symetrické reálné matice pomocí Householderových transformací na matici třídiagonální a algoritmus pro výpočet QR rozkladu matice reálné i komplexní realizovány v programovacím jazyce C za použití knihovny SFENCE a potřebných rozšiřujících modulů a knihoven (zvláště modul SSEM, knihovna Calculate a modul CalculateSEM). Takto přepsané algoritmy nám umožní díky modularitě SFENCE knihovny nejen numerickou verifikaci výpočtu. Vhodnou úpravou nastavení lze získat z výpočtu další zajímavé informace při použití identického zdrojového kódu algoritmu.

K této práci přiložený program AnalyzeAlgorithm analyzuje výše zmíněné algoritmy. Výsledkem počítačového zpracování jsou informace o počtu operací násobení, dělení, sčítání, odčítání a odmocňování, které je potřeba provést během příslušného výpočtu. Pro konečnou implementaci algoritmu do hradlového pole je podstatný také programem vygenerovaný SVA kód vztažený k jednotlivým algoritmům a typům matic.

Zdrojové kódy zmíněné aplikace a knihoven jsou součástí elektronické přílohy této práce.

2.1 Analýza redukce matice na matici třídiagonální

V rámci analýzy algoritmů přiloženým programem AnalyzeAlgorithm jsou dostupné výsledky náročnosti Householderovi redukce matice na matici třídiagonální pro vstupní reálnou čtvercovou symetrickou matici řádu 4 a 8.

Výsledky pro čtvercovou matici řádu 4 jsou v tabulce 2.1. První položka (přiřazení) odpovídá jednoduchému přiřazení proměnné/čísla do proměnné bez sčítání, násobení apod.

Tab. 2.1: Výsledky analýzy Householderovy redukce matice řádu 4 na matici třídiagonální

Typ operace	**Počet provedení**
přiřazení	82×
sčítání	164×
odčítání	97×
násobení	277×
dělení	2×
odmocnění	4×

Čtvercové matici řádu 8 odpovídají data z tabulky 2.2.

Tab. 2.2: Výsledky analýzy Householderovy redukce matice řádu 8 na matici třídiagonální

Typ operace	**Počet provedení**
přiřazení	390×
sčítání	3482×
odčítání	1153×
násobení	4731×
dělení	6×
odmocnění	12×

2.2 Analýza QR rozkladu

Díky analýze QR rozkladu pomocí přiloženého programu AnalyzeAlgorithm je možné porovnat výsledky pro vstupní reálné čtvercové matice řádu 2, 4, 8 a komplexní matice řádu 2 a 4.

Zjištěná data, vztahující se k vstupní čtvercové reálné matici řádu 2, jsou uvedena v tabulce 2.3. První položka (přiřazení) odpovídá jednoduchému přiřazení proměnné/čísla do proměnné bez sčítání, násobení apod.

Tab. 2.3: Výsledky analýzy QR algoritmu pro matici řádu 2

Typ operace	**Počet provedení**
přiřazení	26×
sčítání	15×
odčítání	6×
násobení	28×
dělení	4×
odmocnění	1×

Pro čtvercovou reálnou matici řádu 2 lze také v příloze H.1 nalézt SVA neoptimalizovaný kód vygenerovaný programem AnalyzeAlgorithm. V přiloženém kódu platí pro vstupní čtvercovou reálnou matici **A** řádu 2 vztah

$$\mathbf{A} = \begin{bmatrix} \_in00 & \_in01 \\ \_in10 & \_in11 \end{bmatrix} . \tag{2.1}$$

Výstupní čtvercovou reálnou matici **Q** řádu 2 lze zapsat

$$\mathbf{Q} = \begin{bmatrix} \_Q00 & \_Q01 \\ \_Q10 & \_Q11 \end{bmatrix} . \tag{2.2}$$

Obdobně je zapsána i výstupní čtvercová reálná matice **R** řádu 2 vztahem

$$\mathbf{R} = \begin{bmatrix} \_R00 & \_R01 \\ \_R10 & \_R11 \end{bmatrix} . \tag{2.3}$$

Ve výpisu SVA kódu (příloha H.1) reprezentuje proměnná $\_ZERO$ číslo 0 a proměnná $\_ONE$ číslo 1. Proměnné ve tvaru $@N$ slouží k ukládání mezivýsledků. Proměnná N v názvu nahrazuje celé číslo, pro které platí $N \geq 0$.

V tabulkách 2.4 a 2.5 lze nalézt data odpovídající vstupním čtvercovým reálným maticím řádu 4 a řádu 8. Výsledky analýzy pro komplexní matice řádu 2 a 4 jsou v tabulkách 2.6 a 2.7.

Tab. 2.4: Výsledky analýzy QR algoritmu pro matici řádu 4

Typ operace	**Počet provedení**
přiřazení	168×
sčítání	366×
odčítání	48×
násobení	477×
dělení	48×
odmocnění	3×

Tab. 2.5: Výsledky analýzy QR algoritmu pro matici řádu 8

Typ operace	**Počet provedení**
přiřazení	1172×
sčítání	6860×
odčítání	448×
násobení	7819×
dělení	448×
odmocnění	7×

Tab. 2.6: Výsledky analýzy QR algoritmu pro komplexní matici řádu 2

Typ operace	**Počet provedení**
přiřazení	57×
sčítání	73×
odčítání	42×
násobení	142×
dělení	4×
odmocnění	1×

Tab. 2.7: Výsledky analýzy QR algoritmu pro komplexní matici řádu 4

Typ operace	**Počet provedení**
přiřazení	360×
sčítání	1327×
odčítání	633×
násobení	2220×
dělení	102×
odmocnění	15×

2.3 Srovnání výpočetní náročnosti algoritmů

Ze srovnání dat získaných analýzou QR rozkladu čtvercových matic řádu 4 a 8 (tabulky 2.1, 2.2, 2.4, 2.5 a 2.7) je patrná nižší výpočetní náročnost Householderovi redukce matice na třídiagonální matici oproti QR rozkladu. Současně je patrné, že v QR rozkladu je prováděn značně vyšší počet dělení. Počet sčítání/odčítání a násobení je však srovnatelný.

Srovnání analýzy QR rozkladu pro reálné matice řádu 4 a 8 a komplexní matici řádu 4 je uvedeno v tabulce 2.8.

Tab. 2.8: Srovnání QR rozkladu pro reálné a komplexní matice

Typ operace	**Počet provedení v QR rozkladu nad maticí** $\mathbb{R}^{4\times4}$	**Počet provedení v QR rozkladu nad maticí** $\mathbb{R}^{8\times8}$	**Počet provedení v QR rozkladu nad maticí** $\mathbb{C}^{4\times4}$
sčítání/odčítání	414×	7308×	1962×
násobení	495×	7763×	2268×
dělení	48×	448×	96×

Z dat získaných programem AnalyzeAlgorithm je patrné, že v případě hledání vlastních čísel a vlastních vektorů čtvercové komplexní hermitovské matice řádu 4 pomocí QR algoritmu, by převedení problému na hledání vlastních čísel a vlastních vektorů reálné čtvercové symetrické matice řádu 8 (využitím rovnice 1.68) neušetřilo výpočetní operace. Výpočet QR rozkladu reálné matice o řádu 8 je totiž téměř 4× náročnější než výpočet QR rozkladu komplexní matice řádu 4. Pro výpočet vlastních čísel a vektorů je proto použit komplexní QR algoritmus nad čtvercovou maticí řádu 4. Householderova redukce matice do třídiagonálního tvaru, použitelná pouze pro matice reálné, ve výpočtu vlastních čísel a vlastních vektorů není uplatněna.

3 IMPLEMENTACE DO HRADLOVÉHO POLE

Základem složitějších výpočetních operací jsou výpočetní operace v oboru reálných a komplexních čísel, přičemž komplexní výpočty lze realizovat pomocí výpočetních operací v oboru reálných čísel. Zvládnutí výpočtů v oboru reálných čísel v hradlovém poli proto tvoří základ pro implementaci obsáhlých výpočtů.

Základní bloky pro počítání s čísly s plovoucí desetinnou čárkou jsou realizovány pomocí IP bloku Xilinx® Floating-Point Operator dostupného ve vývojovém prostředí Xilinx Vivado. Digitální obvod lze popsat například v jazycích VHDL a Verilog, které jsou v Xilinx Vivado podporovány. Pro popis bloků je použit jazyk VHDL, se kterým má autor této práce bohatší zkušenosti.

Vygenerované IP bloky, provádějící výpočty s čísly s plovoucí desetinnou čárkou, jsou překryty pomocnými bloky s upraveným rozhraním. Tím je usnadněna možná budoucí změna IP bloků a implementace do jiné řady hradlových polí, kupříkladu i jiného výrobce.

Bloky `APU_NB` (viz strana 51), `APU_NB_small` (viz strana 52) a `APU_small` (viz strana 55) jsou kromě ověření funkce IP bloku Xilinx® Floating-Point Operator určeny k demonstraci různých technik použití zmíněného IP bloku.

Základem pro vnitřní architekturu bloku `DSP_corePipelineRAM` (viz strana 57), určeného pro finální implementaci a realizaci výpočtu jednotlivých iterací QR algoritmu, je blok `APU_small`. Kompletní výpočet QR algoritmu je realizován blokem `QR_Algorithm` (viz strana 56) prostřednictvím bloku `DSP_pipelineRAM` (viz strana 57).

Pokud v následujícím textu nebude uvedeno jinak, vstupní a výstupní signály bloků jsou aktivní v logické 1 a neaktivní v logické 0 a impuls na signálu je chápán jako přepnutí signálu do logické 1 na dobu jednoho platného taktu hodin (náběžná hrana hodin `clk` v době aktivního signálu `clk_en`).

V zájmu vyšší čitelnosti VHDL kódu jsou používány subtypy `FP_SINGLE_NUMBER`, `FP_NUMBER`, `COMPLEX_NUMBER` a `DSP_NUMBERS`, které jsou definovány dle výpisu 3.1.

Výpis 3.1: Definice typů reprezentujících čísla ve VHDL

```
1  subtype FP_SINGLE_NUMBER is std_logic_vector(31
     downto 0);
2  subtype FP_NUMBER is FP_SINGLE_NUMBER;
3
4  type COMPLEX_NUMBER is record
5      re : FP_NUMBER;
6      im : FP_NUMBER;
7  end record;
8
9  type DSP_NUMBERS is array(natural range <>) of
     FP_NUMBER;
```

3.1 IP blok pro výpočty s čísly s plovoucí desetinnou čárkou

IP blok představuje logický funkční a často konfigurovatelný celek plnící určitou funkci (pamět, UART komunikace, násobička, apod.). V této práci je použit IP blok Xilinx® Floating-Point Operator dodávaný spolu s vývojovým prostředím Xilinx Vivado. Jde o IP blok přizpůsobitelný z hlediska prováděné matematické operace i z hlediska velikosti/přesnosti vstupního čísla, zpoždění a rozhraní pro komunikaci s ostatními bloky[16]. Uvedené IP bloky jsou podporovány čipy z řad UltraScale™ Architecture, Zynq®-7000 a 7 Series[16].

Xilinx® Floating-Point Operator IP blok podporuje operace násobení, sčítání, odčítání, sumy ($\sum \pm A[i]$), vynásobení a sečtení ($a \times b \pm c$), dělení, výpočtu odmocniny, porovnání, výpočtu převrácené hodnoty ($\frac{1}{x}$), výpočtu převrácené hodnoty odmocniny ($\frac{1}{\sqrt{(x)}}$), určení absolutní hodnoty, výpočtu logaritmu se základem e ($\ln A$), výpočtu exponenciální funkce e^x a konverze mezi různými formáty reprezentujícími číslo s pevnou nebo plovoucí desetinnou čárkou[16].

Vzhledem k operacím využívaným v algoritmech pro výpočet vlastních čísel a vlastních vektorů jsou využity operace sčítání, odčítání, násobení, dělení a výpočtu odmocniny.

Formát čísel s plovoucí a pevnou desetinnou čárkou podporovaný IP bloky Xilinx® Floating-Point Operator je dán normou IEEE-754. S ohledem na vstupní požadavky této práce je zajímavý zvláště formát čísel s plovoucí desetinnou čárkou, který je použit. Formát čísel s pevnou desetinnou čárkou je použitým IP blokem podporován jen v omezené míře, v rámci konverzí čísel s plovoucí desetinnou čárkou na čísla s pevnou desetinnou čárkou a naopak. [16]

3.1.1 Čísla s plovoucí desetinnou čárkou

Velikost čísla v bitech je označena w a obdobně je velikost mantisy v bitech značena w_f. Pro základní přesnost je velikost čísla 32 bitů a mantisy 24 bitů[16].

Čísla s plovoucí desetinnou čárkou jsou reprezentována znaménkem, exponentem a mantisou. Pokud písmenko s označuje znaménko, znak E exponent a mantisa je zapsána pomocí bitů v podobě $b_0 b_1 b_2 \ldots b_{w_f-1}$, lze hodnotu čísla v s plovoucí desetinnou čárkou vyjádřit vztahem

$$v = (-1)^s 2^E b_0 b_1 b_2 \ldots b_{w_f-1} \text{ [16].} \tag{3.1}$$

Binární bity b_i mají váhu 2^{-i}. Mantisu lze tedy přepsat do podoby

$$b_0 b_1 b_2 \ldots b_{w_f-1} = b_0 2^0 + b_1 2^{-1} + b_2 2^{-2} + \ldots + b_{w_f-1} 2^{w_f-1} = \sum_{i=0}^{w_f-1} b_i 2^{-i} \,, \tag{3.2}$$

kde každý binární bit b_i nabývá hodnoty 0 nebo 1 a žádné jiné. Jelikož bit b_0 nabývá vždy hodnoty 1, pro hodnotu mantisy platí:

$$1 \leq b_0 b_1 b_2 \ldots b_{w_f-1} < 2 \text{ [16]}. \tag{3.3}$$

Jedná se o normalizované číslo.

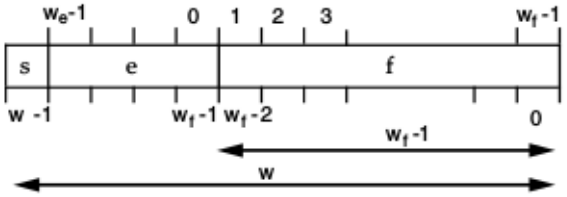
Bit b_0 se neukládá (jeho hodnota je vždy známá). Je-li označena bitová šířka pole exponentu w_e, pole exponentu e a mantisa bez bitu b_0 znakem f, pak

$$f = b_1 b_2 \ldots b_{w_f-1} \text{ [16]}. \tag{3.4}$$

Pro uložení mantisy f stačí $w_f - 1$ bitů. Pole exponentu se ukládá do $w - w_f$ bitů. Bitovou šířku pole exponentu vyjadřuje vztah

$$w_e = w - w_f \text{ [16]}. \tag{3.5}$$

Poslední bit slouží k uložení znaménka s. Vše je znázorněno na obrázku 3.1.



Obr. 3.1: Bitová pole jednotlivých částí čísla s plovoucí desetinnou čárkou [16]

Ze vztahu

$$e = \sum_{i=0}^{w_e-1} e_i 2^i \text{ [16]}, \tag{3.6}$$

je patrné, že e nabývá pouze kladných hodnot. Pro vyjádření čísel menších než 1 je nutné, aby exponent E mohl nabývat záporných hodnot. Hodnota exponentu E je určena z hodnoty e odstraněním posunutí. Postup je popsán vztahem

$$E = e - \left(2^{w_e-1} - 1\right) \text{ [16]}. \tag{3.7}$$

Kromě racionálních čísel lze do čísla s plovoucí desetinnou čárkou ukládat i speciální hodnoty dle tabulky 3.1.

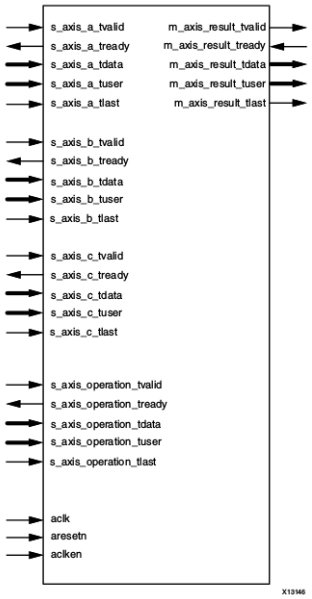
Denormalizovaná čísla jsou taková čísla, u kterých není první (explicitně nevyjádřený) bit mantisy b_0 roven 1, ale naopak 0. Výpočty s těmito velmi malými hodnotami nejsou přesné, zejména při násobení a dělení. Při ukládání denormalizovaných čísel je exponent vždy nastaven na nejnižší možnou hodnotu[17].

Tab. 3.1: Speciální hodnoty čísla s plovoucí desetinnou čárkou[16]

Symbol speciální hodnoty	**Pole** s	**Pole** e	**Pole** f
NaN	nemá efekt	2^{w_e-1}	libovolná nenulová hodnota
$\pm\infty$	znaménko ∞	2^{w_e-1}	0
± 0	znaménko 0	0	0
denormalizované číslo	znaménko čísla	0	libovolná nenulová hodnota

3.1.2 Porty

Množství podporovaných přizpůsobení se odráží také v množství vstupně výstupních portů, které může IP blok Xilinx® Floating-Point Operator mít. Dle zvoleného nastavení jsou ve vygenerovaném bloku použity pouze potřebné porty. Všechny porty jsou vidět na obrázku 3.2.



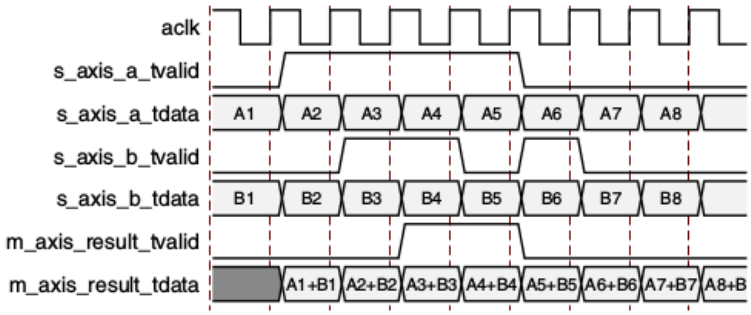
Obr. 3.2: Znázornění všech dostupných vstupních a výstupních portů IP bloku Xilinx® Floating-Point Operator [16]

Význam jednotlivých portů lze nalézt v příloze K.1.

3.1.3 Neblokovací režim

Neblokovací režim je blokem Xilinx® Floating-Point Operator podporován z důvodu kompatibility s předešlou verzí IP bloku. Lze tak s minimálními změnami syntetizovat starší projekty. Neblokační režim není doporučeno používat v nových návrzích.[16]

Princip funkce bloku v neblokačním režimu je znázorněn na obrázku 3.3.



Obr. 3.3: Neblokační mód [16]

Obrázek 3.3 znázorňuje odezvu IP bloku typu sčítačka se zpožděním jednoho cyklu. Výpočet je prováděn se vstupními daty platnými při každé náběžné hraně hodinového signálu, během které je signál aclken v logické 1. Signál aresetn je neaktivní (logická 1). Výpočet je prováděn neustále a signál m_axis_result_tvalid signalizuje, zda byl signál TVALID při zahájení výpočtu aktivní (logická 1) pro oba sčítance.[16]

Komunikační rozhraní IP bloku v neblokačním režimu je jednodušší a IP blok je proto méně náročný na využití zdrojů dostupných na cílovém hradlovém poli. [16]

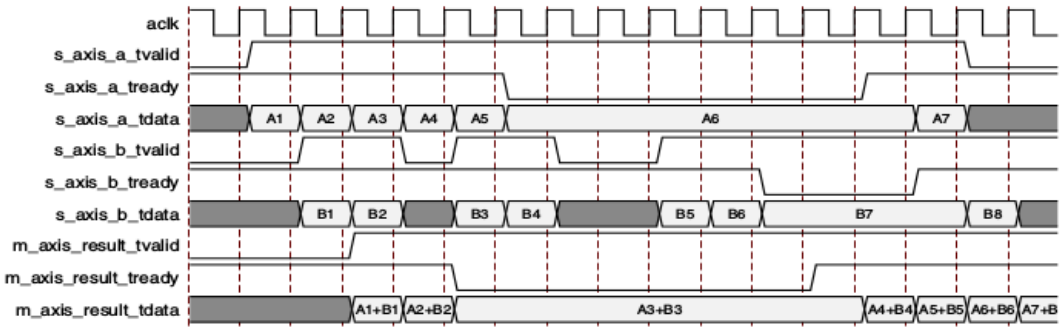
Z důvodu výkonu je signál aresetn interně ukládán do registru a efekt resetu se projevuje až se zpožděním jednoho hodinového cyklu. V případě aktivního resetu je signál m_axis_result_tvalid neaktivní (logická 0) stejně jako signál m_axis_result_tdata.[16]

3.1.4 Blokovací režim

Na každém vstupním kanálu v IP bloku Xilinx® Floating-Point Operator je umístěn buffer. Oproti neblokovacímu režimu IP blok zahajuje výpočet až ve chvíli dostupnosti validních dat ze všech vstupů. Signál TREADY na vstupu signalizuje (logická 1) schopnost vstupního bufferu pojmout další data. Naopak signál TREADY na výstupu (m_axis_result_tready) umožňuje bloku za IP blokem kontrolovat předávání výsledků z výstupu IP bloku. Pokud následující

blok není schopen přijímat další data, deaktivuje signál m_axis_result_tready (logická 0) a v okamžiku, kdy je opět schopen data přijímat a zpracovávat, signál m_axis_result_tready opět aktivuje (logická 1).

Celý proces je znázorněn pro sčítačku se zpožděním jednoho cyklu na obrázku 3.4.



Obr. 3.4: Blokační mód [16]

Významnou vlastností IP bloku v blokovacím režimu je párování validních dat. První výpočet je prováděn s prvními validními daty ze vstupů, druhý výpočet s druhými validními daty ze vstupů, třetí s třetími a tak dále.

Schopnost bufferování vstupů IP bloku umožňuje přečtení výsledků až v okamžiku, kdy je jich potřeba pro další výpočet.

3.1.5 Vygenerované bloky

Blok Xilinx® Floating-Point Operator lze nakonfigurovat pro provádění značného počtu matematických operací. V této práci jsou použity operace sčítání, odčítání, násobení, dělení a odmocňování. Pokud by později vznikla potřeba využití další matematické operace podporované IP blokem Xilinx® Floating-Point Operator, je možné dogenerovat potřebný blok.

Vygenerovanými bloky pracujícími v neblokačním režimu jsou `FP_SINGLE_NB_ADDSUB` (sčítání/odčítání), `FP_SINGLE_NB_MUL` (násobení), `FP_SINGLE_NB_DIV` (dělení) a `FP_SINGLE_NB_SQRT` (odmocňování).

V blokačním režimu pracují bloky `FP_SINGLE_ADDSUB` (sčítání a odčítání), `FP_SINGLE_ADD` (jen sčítání), `FP_SINGLE_MUL` (násobeni), `FP_SINGLE_DIV` (dělení) a `FP_SINGLE_SQRT` (odmocnění).

Pro potřeby verifikace v simulacích jsou vygenerovány i bloky `FP_SINGLE_ABS` (absolutní hodnota) a `FP_SINGLE_CMP` (porovnání), které pracují v blokačním režimu.

Rozhraní všech zmíněných bloků jsou definována v souboru FP_SINGLE.vhd (viz elektronická příloha).

3.2 Pomocné bloky

Pomocné bloky obalují základní rozhraní generovaných bloků (`FP_SINGLE_ADDSUB` apod.) více abstraktním rozhraním s kratšími jmény signálů při použití typů `FP_SINGLE_NUMBER`, `FP_NUMBER` (viz výpis 3.1). Práce s bloky je pak snazší a význam vstupů a výstupů čitelnější, čímž se zlepšuje srozumitelnost kódu.

Jedná se o pomocné bloky `FP_ADDSUB`, `FP_ADD`, `FP_MUL`, `FP_DIV`, `FP_SQRT`, `FP_CMP`, `FP_ABS`, `FP_CHECK`, `FP_ADDSUB_NB`, `FP_MUL_NB`, `FP_DIV_NB` a `FP_SQRT_NB`. Definici jejich rozhraní lze nalézt v příloze L.1.

Definice bloku `FP_ADDSUB` (výpis 3.2) obsahuje množství signálů. Velká část z nich je společná více blokům.

Výpis 3.2: Definice bloku FP_ADDSUB

```
1     component FP_ADDSUB is
2         Port ( clk : in STD_LOGIC;
3                clk_en : in STD_LOGIC;
4                reset : in STD_LOGIC;
5                sub : in STD_LOGIC;
6                A : in FP_NUMBER;
7                B : in FP_NUMBER;
8                AB_ready: out STD_LOGIC;
9                AB_ok: in STD_LOGIC;
10               Y : out FP_NUMBER;
11               Y_ready: in STD_LOGIC;
12               Y_ok : out STD_LOGIC);
13    end component;
```

Signály společné jsou zvláště `clk`, `clk_en` a `reset`. Výjimkou je blok `FP_ABS`. Následují signály `A`, `B`, `AB_ready` a `AB_ok`. Signál `B` je vynechán u bloků s jedním vstupem (`FP_ABS` a `FP_SQRT`), a proto jsou v těchto případech signály `AB_ready` a `AB_ok` přejmenovány na `A_ready` a `A_ok`. Poslední jsou signály `Y`, `Y_ready`, `Y_ok`. Signály pojmenované `AB_ok`, `Y_ok` apod. signalizují validitu vstupu/výstupu. Připravenost přijímat data je signalizována signály `AB_ready`, `Y_ready` apod., které se nevyskytují u bloků pracujících v neblokačním režimu (`FP_ADDSUB_NB` apod.). U takových bloků nenajdeme ani signál `reset`.

Hodiny jsou dány signálem `clk`, který může být povolen/zakázán signálem `clk_en`. V případě resetování je nutné podržet signál `reset` v logické 1 alespoň po dobu dvou taktů hodinového signálu. Signály `AB_ready`, `Y_ready` apod. a `AB_ok`, `Y_ok` apod.

signalizují (logická 1) připravenost k čtení či zápisu. Signály `A`, `B` a `Y` představují čísla s plovoucí desetinnou čárkou v základní přesnosti.

V případě bloku `FP_ADDSUB` je v definici uveden i specifický signál `sub`, který nabývá logické úrovně 1, pokud je žádáno odčítání. Lze také použít konstant `FP_PLUS` (pro sčítání) a `FP_MINUS` (pro odčítání) viz příloha L.1.

Blok `FP_CMP` má specifický vstup nazvaný `cmp_op` typu `FP_CMP_OP`, který určuje metodu porovnávání. Vstup může nabývat hodnot, které jsou uvedeny v tabulce 3.2. Výstup bloku se místo `Y` jmenuje `CMP` a nabývá logické 1, jestliže je zvolená podmínka pro vstupy `A` a `B` splněna.

Tab. 3.2: Možné hodnoty signálu `cmp_op` typu `FP_CMP_OP`

Konstanta	**Hodnota**	**Význam**
`FP_CMP_OP_Unordered`	`"0001"`	$(A = NaN) OR (B = NaN)$
`FP_CMP_OP_LessThan`	`"0011"`	$A < B$
`FP_CMP_OP_Equal`	`"0101"`	$A = B$
`FP_CMP_OP_LessThanEqual`	`"0111"`	$A \leq B$
`FP_CMP_OP_GreaterThan`	`"1001"`	$A > B$
`FP_CMP_OP_NotEqual`	`"1011"`	$A \neq B$
`FP_CMP_OP_GreaterThanEqual`	`"1101"`	$A \geq B$

Speciálním případem je blok `FP_CHECK` navržený pro verifikaci výsledků během simulací (viz výpis 3.3).

Význam vstupů `clk`, `clk_en` a `reset` je standardní. Následuje signál `A` (spolu s `A_ready` a `A_ok`), který udává hodnotu, jež má být zkontrolována. Hodnotu očekávanou udává signál `V`. Maximální chyba je dána vstupem `ERR`. Pro signály `V` a `ERR` jsou doplněny vstupy `VERR_ready` a `VERR_ok`. Výstupem je signál `CHECK` doplněný o `CHECK_ready` a `CHECK_ok`.

Funkci bloku `FP_CHECK` je možné přiblížit vztahem

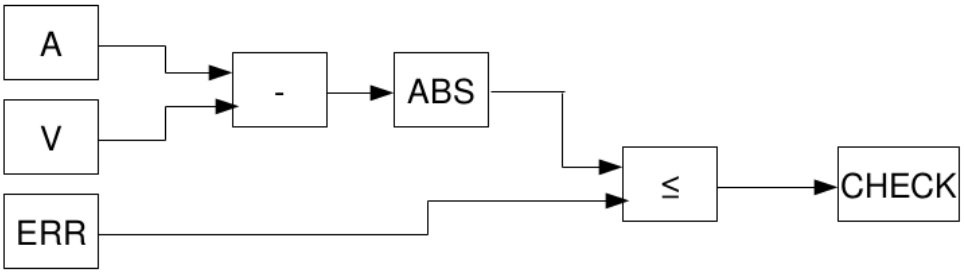
$$CHECK = |A - V| < ERR\,. \tag{3.8}$$

Výpis 3.3: Definice bloku FP_CHECK

```
1     component FP_CHECK is
2         Port ( clk : in STD_LOGIC;
3                clk_en : in STD_LOGIC;
4                reset : in STD_LOGIC;
5                A : in FP_NUMBER; -- value to be
                    checked
6                A_ready: out STD_LOGIC;
7                A_ok: in STD_LOGIC;
8                V : in FP_NUMBER; -- valid value
9                ERR : in FP_NUMBER; -- error, greater
                    than zero
10               VERR_ready: out STD_LOGIC;
11               VERR_ok: in STD_LOGIC;
12               CHECK : out STD_LOGIC;
13               CHECK_ready: in STD_LOGIC;
14               CHECK_ok : out STD_LOGIC);
15    end component;
```

Z toho je patrné, že vstupy `A` a `V` je možné zaměňovat. V simulacích jsou obvykle signály `A_ready` a `VERR_ready` nezapojeny a signál `CHECK_ready` je připojen trvale k logické 1. Za předpokladu, že výsledky k verifikaci jsou generovány pomaleji, než odpovídá rychlosti bloku `FP_CHECK`, není takové zapojení problematické.

Vnitřní struktura bloku `FP_CHECK` je znázorněna na obrázku 3.5.



Obr. 3.5: Vnitřní architektura `FP_CHECK`

Kromě již zmíněných bloků lze v souboru FP.vhd (příloha L.1) najít i definice bloků `FP_NOTZERO`, `FP_SIGNNOTZERO`, `FP_ASSIGN`, `FP_DEBABS` a `FP_BUF`. Jedná se o bloky určené pro připojení k bloku `DSP_corePipelineRAM`, které plní funkce dle tabulky 3.3.

Tab. 3.3: Funkce

Blok	**Funkce**
FP_NOTZERO	výběr nenulové hodnoty
FP_SIGNNOTZERO	funkce sign, avšak pro 0 nabývající hodnoty 1.0
FP_ASSIGN	přenesení vstupu na výstup
FP_DEBABS	výpočet absolutní hodnoty
FP_BUF	buferování výstupů jiných bloků

Blok FP_DEBABS se liší od bloku FP_ABS přítomností hodinových vstupů a resetu. Zpoždění bloku odpovídá jedné platné náběžné hraně hodin. Obdobně se i blok FP_ASSIGN liší od strohého přiřazení signálů vstupem hodin a má zpoždění stejná jako blok FP_DEVABS. Zpoždění jedné platné náběžné hrany hodin se týká i bloků FP_NOTZERO a FP_SIGNNOTZERO. U bloku FP_BUF záleží na konfiguraci a množství dat v bloku aktuálně uložených. První do bloku FP_BUF uložené číslo je z bloku vyčteno jako první.

3.3 Reálná a komplexní aritmetická jednotka

Aritmetická jednotka je blokem, který je schopen provést operace sčítání, odčítání, násobení a dělení nad dvěma reálnými, popřípadě komplexními čísly. Aritmetickou jednotku lze obecně řešit jako logický blok synchronní či asynchronní. Vzhledem k využití synchronních IP bloků pro realizaci matematických operací s čísly s plovoucí desetinnou čárkou, je architektura všech aritmetických jednotek synchronní.

3.3.1 Aritmetická jednotka `APU_NB`

Blok `APU_NB` je základní verzí aritmetické jednotky navržené s ohledem na jednoduchost. `APU_NB` využívá IP bloky generované pro práci v neblokačním režimu. Definice bloku `APU_NB` viz výpis 3.4.

Výpis 3.4: Definice bloku APU_NB

```
1  entity APU_NB is
2      Port ( clk : in STD_LOGIC;
3             clk_en : in STD_LOGIC;
4             operation : in STD_LOGIC_VECTOR (1 downto
                0);
5             complex : in STD_LOGIC;
6             A : in COMPLEX_NUMBER;
7             B : in COMPLEX_NUMBER;
8             AB_ok: in STD_LOGIC;
9             Y : out COMPLEX_NUMBER;
10            Y_ok : out STD_LOGIC);
11 end APU_NB;
```

Signály `clk` a `clk_en` udávají hodiny. `APU_NB` reaguje na náběžnou hranu hodinového signálu `clk` v případě, že je signál `clk_en` aktivní (logická 1). Signál `operation` vybírá matematickou operaci (sčítání, odčítání, násobení, dělení). Přehled viz tabulka 3.4.

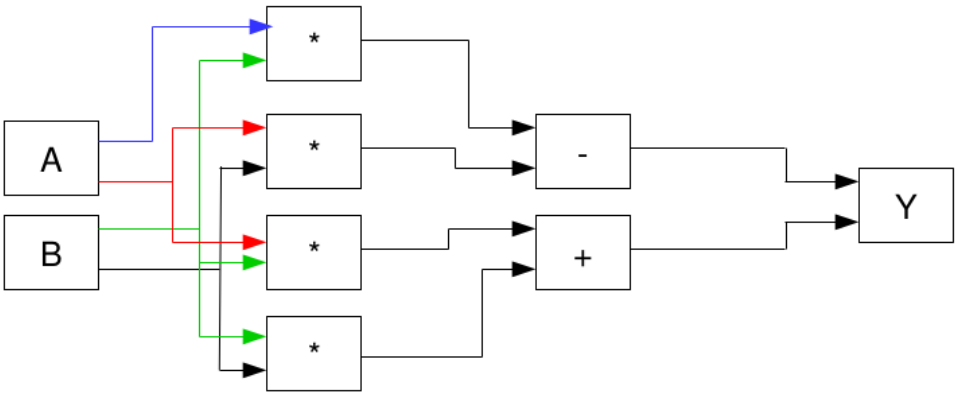
Signál `complex` je aktivní (logická 1), pokud je žádáno provedení odpovídající matematické operace v oboru komplexních čísel. Signály `A` a `B` reprezentují vstupní komplexní čísla (viz výpis 3.1). V případě, že je signál `complex` neaktivní (logická 0), se pro výpočet použijí pouze reálné části komplexních čísel (signály `A.re` a `B.re`). Signál `AB_ok`, aktivní v logické 1, signalizuje validitu vstupních dat. Výsledek matematické operace lze vyčíst ze signálu `Y`. Signál `Y_ok` signalizuje validitu výsledku logickou 1.

Tab. 3.4: Signál `operation` bloku `APU_NB`

Signál `operation`	**Význam**
`"00"`	sčítání $A + B$
`"01"`	odčítání $A - B$
`"10"`	násobení $A \times B$
`"11"`	dělení A/B

Vnitřní architektura bloku `APU_NB` je zaměřená na jednoduchost. Z hlediska využití zdrojů je značně neefektivní bez potenciálu pro zlepšení. `APU_NB` využívá tři sčítačky, šest násobiček a dvě děličky.

Vnitřní propojení sčítaček, násobiček a děliček bloku `APU_NB` se mění v závislosti na prováděné matematické operaci. Na obrázku 3.6 je znázorněno vnitřní propojení bloků v případě komplexního násobení.

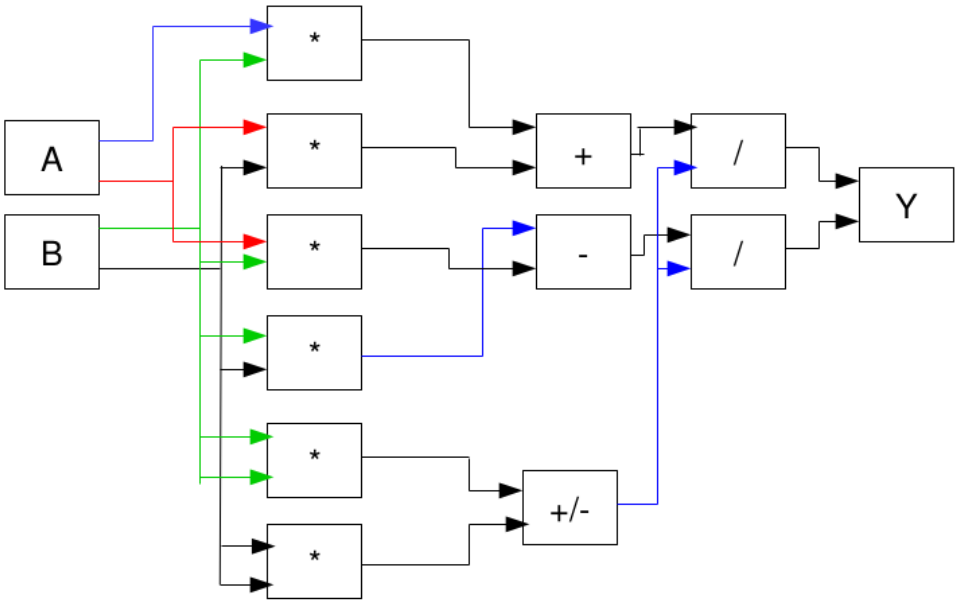


Obr. 3.6: Vnitřní propojení bloků `APU_NB` během komplexního násobení

Obrázek 3.7 (viz strana 53) naproti tomu znázorňuje vnitřní propojení bloků při výpočtu podílu dvou komplexních čísel.

3.3.2 Aritmetická jednotka `APU_NB_small`

Oproti bloku `APU_NB` blok `APU_NB_small` využívá pouze tři IP bloky, ale opakovaně. `APU_NB_small` pracuje v neblokačním režimu, stejně jako `APU_NB`. Definici rozhraní bloku `APU_NB_small` nalezneme ve výpisu 3.5.



Obr. 3.7: Vnitřní propojení bloků `APU_NB` během komplexního dělení

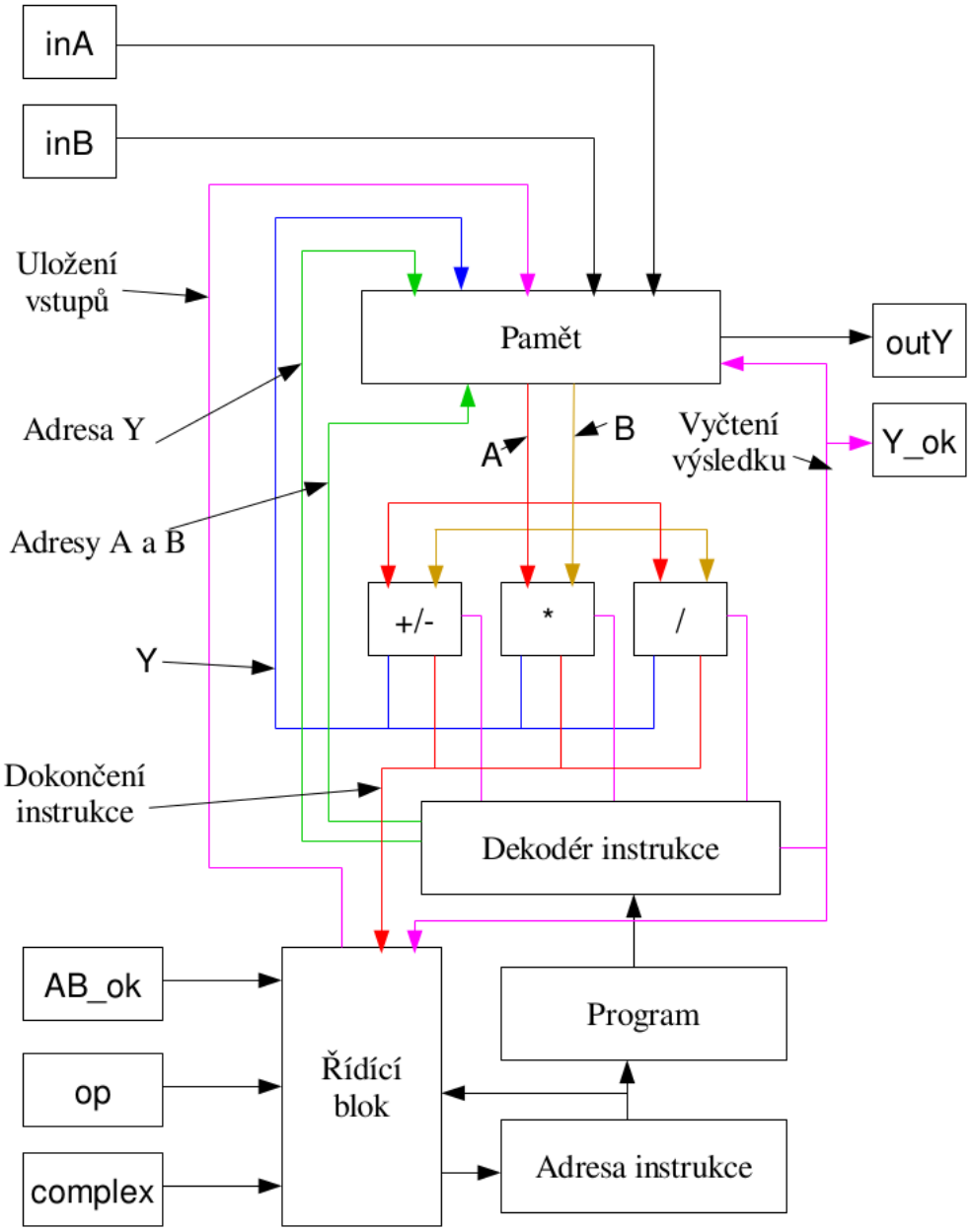
Výpis 3.5: Definice bloku APU_NB_small

```
1  entity APU_NB_small is
2      Port ( clk : in STD_LOGIC;
3             clk_en : in STD_LOGIC;
4             operation : in STD_LOGIC_VECTOR (1 downto
                 0);
5             complex : in STD_LOGIC;
6             A : in COMPLEX_NUMBER;
7             B : in COMPLEX_NUMBER;
8             AB_ok: in STD_LOGIC;
9             Y : out COMPLEX_NUMBER;
10            Y_ok : out STD_LOGIC);
11 end APU_NB_small;
```

Význam signálů je identický se stejnými signály bloku `APU_NB` (viz strana 51). Rozdílná je vnitřní architektura. Blok `APU_NB_small` pracuje jako jednoduchý programovatelný stavový automat ovládaný kódem, který řídí použití sčítačky, násobičky a děličky. Použitím jen jedné sčítačky, jedné násobičky a jedné děličky se blok `APU_NB_small` stává šetrnějším vůči dostupným prostředkům hradlového pole.

Vnitřní architektura bloku je pro lepší představu znázorněna na obrázku 3.8.

Vstupy inA a inB v obrázku 3.8 reprezentují signály `A` a `B`. Výstupní signál `Y` je



Obr. 3.8: Vnitřní architektura `APU_NB_small`

zakreslen pod jménem outY. Signál `operation` je nazván op. Signály `clk` a `clk_en` nejsou znázorněny.

Řídící blok `APU_NB_small` dává povel k nahrání vstupů do paměti a spouští program uložený v paměti, který odpovídá matematické operaci, jejíž provedení je žádáno. Jednotlivé instrukce řídí načítání signálů `A` a `B` z paměti a aktivují odpovídající IP blok (sčítačku/odčítačku, násobičku nebo děličku). Program je ukončen vyčtením výsledků na výstup. Během běhu programu jsou vstupy ignorovány. Další výpočet

lze zahájit až po dokončení probíhajícího výpočtu.

3.3.3 Aritmetická jednotka `APU_small`

Aritmetická jednotka `APU_small` využívá IP bloků pracujících v blokačním režimu. Vnitřní architektura je identická s architekturou bloku `APU_NB_small`. `APU_small` je též programovatelným stavovým automatem. Definice vstupních a výstupních signálů `APU_small` se nalézá ve výpisu 3.6.

Výpis 3.6: Definice bloku APU_small

```
1  entity APU_small is
2      Port ( clk : in STD_LOGIC;
3             clk_en : in STD_LOGIC;
4             operation : in STD_LOGIC_VECTOR (1 downto
                0);
5             complex : in STD_LOGIC;
6             A : in COMPLEX_NUMBER;
7             B : in COMPLEX_NUMBER;
8             AB_ok: in STD_LOGIC;
9             Y : out COMPLEX_NUMBER;
10            Y_ok : out STD_LOGIC);
11 end APU_small;
```

Jak je patrné, vnější rozhraní je identické s bloky `APU_NB` a `APU_NB_small`. Význam signálů viz blok `APU_NB` strana 51.

Vnitřní architektura bloku `APU_small` odpovídá architektuře bloku `APU_NB_small` (viz obrázek 3.8).

3.4 QR algoritmus v hradlovém poli

Výpočet vlastních čísel a vlastních vektorů zajišťuje blok `QR_Algorithm`, jehož definici lze nalézt ve výpisu 3.7.

Výpis 3.7: Definice bloku QR_Algorithm

```
1  entity QR_Algorithm is
2      Port ( clk : in STD_LOGIC;
3             clk_en : in STD_LOGIC;
4             reset : in STD_LOGIC;
5             matrix_real : in DSP_NUMBERS (0 to 15);
6             matrix_imag : in DSP_NUMBERS (0 to 15);
7             error : in FP_NUMBER;
8             matrix_valid: in STD_LOGIC;
9             matrix_saved: out STD_LOGIC;
10            eigValues : out DSP_NUMBERS (0 to 3);
11            eigVectors_real : out DSP_NUMBERS (0 to
                15);
12            eigVectors_imag : out DSP_NUMBERS (0 to
                15);
13            ok : out STD_LOGIC;
14            fail : out STD_LOGIC);
15 end QR_Algorithm;
```

Vstupem bloku `QR_Algorithm` jsou kromě signálů hodin (`clk` a `clk_en`) a resetu (`reset`) i signály určující čtvercovou komplexní hermitovskou matici řádu 4 (`matrix_real` a `matrix_imag`), cílovou chybu výpočtu (`error`) a signál informující blok o korektnosti vstupních dat (`matrix_valid`).

Výstupem je signál `matrix_saved`, na kterém se objeví impuls (log. 1) po úspěšném nahrání vstupních dat do výpočetního bloku `DSP_pipelineRAM`. Aktivace signálu (log. 1) `fail` indikuje selhání výpočtu, naopak impuls na signálu `ok` potvrzuje dokončení výpočtu. Vypočtená vlastní čísla lze nalézt v signálech `eigValues` a k nim přidružené vlastní vektory v signálech `eigVectors_real` a `eigVectors_imag`.

Jednotlivá čísla z matic jsou ukládána do polí typu `DSP_NUMBERS` po řádcích. Vlastní vektor příslušející k vlastnímu číslu λ_n ($n \in 0, 1, 2, 3$), jež je uloženo v signálu `eigValues(n)`, je možné zapsat

$$x_{\lambda_n}^T = [eigVectors\_real(n) + eigVectors\_imag(n)\text{i}, \\ eigVectors\_real(n+4) + eigVectors\_imag(n+4)\text{i}, \\ eigVectors\_real(n+8) + eigVectors\_imag(n+8)\text{i}, \\ eigVectors\_real(n+12) + eigVectors\_imag(n+12)\text{i}] \ . \quad (3.9)$$

Pro výpočet jednotlivých iterací QR algoritmu je použit blok `DSP_pipelineRAM`. Blok `QR_Algorithm` obsahuje i nezbytné podpůrné bloky (paměti RAM a ROM) pro činnost tohoto bloku. Použitý blok `DSP_pipelineRAM` pracuje s 8 Kb paměti RAM a pamětí ROM obsahující 8635 instrukcí.

3.4.1 Blok `DSP_pipelineRAM`

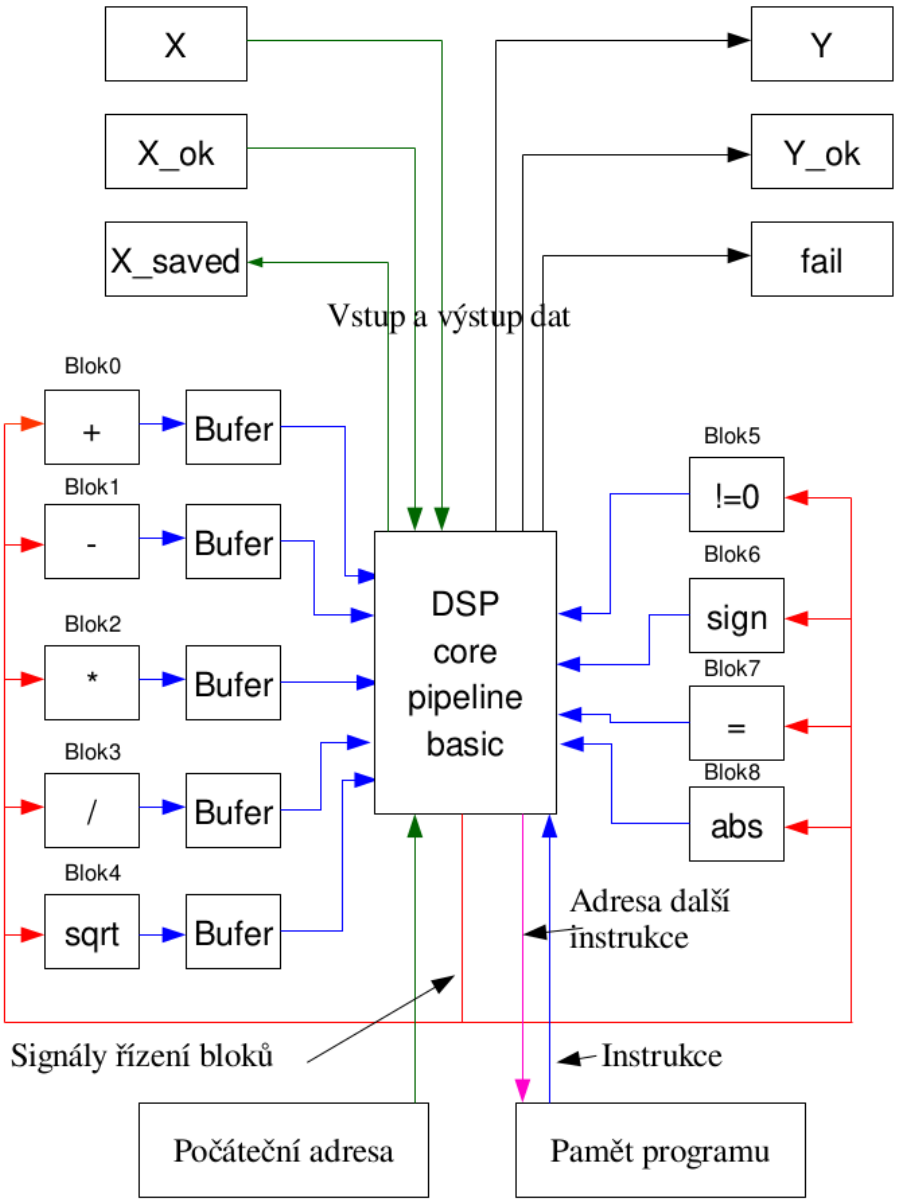
Pro realizaci výpočtu jednotlivých iterací QR algoritmu byl sestaven externí pamětí řízený blok `DSP_pipelineRAM` umožnující současné provádění více výpočetních operací a sdílení zdrojů (bloků pro výpočty s čísly s plovoucí desetinou čárkou). Základem bloku `DSP_pipelineRAM` je blok `DSP_corePipelineRAM`, jehož struktura vychází z bloku `APU_small`.

Vnitřní struktura bloků `DSP_pipelineRAM` a `DSP_corePipelineRAM` je znázorněna na obrázcích 3.9 a 3.10.

Definici rozhraní bloků `DSP_pipelineRAM` a `DSP_corePipelineRAM` lze nalézt v příloze M.1 (str. 103) a příloze N.1 (str. 59). Význam signálů `clk`, `clk_en` a `reset` je obdobný jejich významu v blocích `APU_small` apod.

Blok `DSP_corePipelineRAM` pracuje podobně jako procesor. Jeho schopnosti jsou však velmi limitovány. Každá instrukce přicházející do bloku `DSP_corePipelineRAM` obsahuje instrukční část a dvě registrové části. Instrukční část má trojí význam. Jednak může představovat číslo připojeného bloku (bloky 0 až 8 na obrázku 3.9), kterému je třeba zaslat data. V takovém případě jsou v registrových částech instrukce uloženy adresy registrů, z nichž je potřeba údaje vyčíst. Druhou možností je instrukce představující požadavek na vyčtení dat z bloku do registru. Potom je v jedné z registrových částí uloženo číslo bloku, ze kterého má být vyčítáno a v druhé registrové části adresa registru určeného pro uložení výsledku. Poslední možností je instrukce ukončující práci bloku `DSP_corePipelineRAM`, která publikuje výsledky (signál `Y`) a vygeneruje impuls (log. 1) na signálu `Y_ok`.

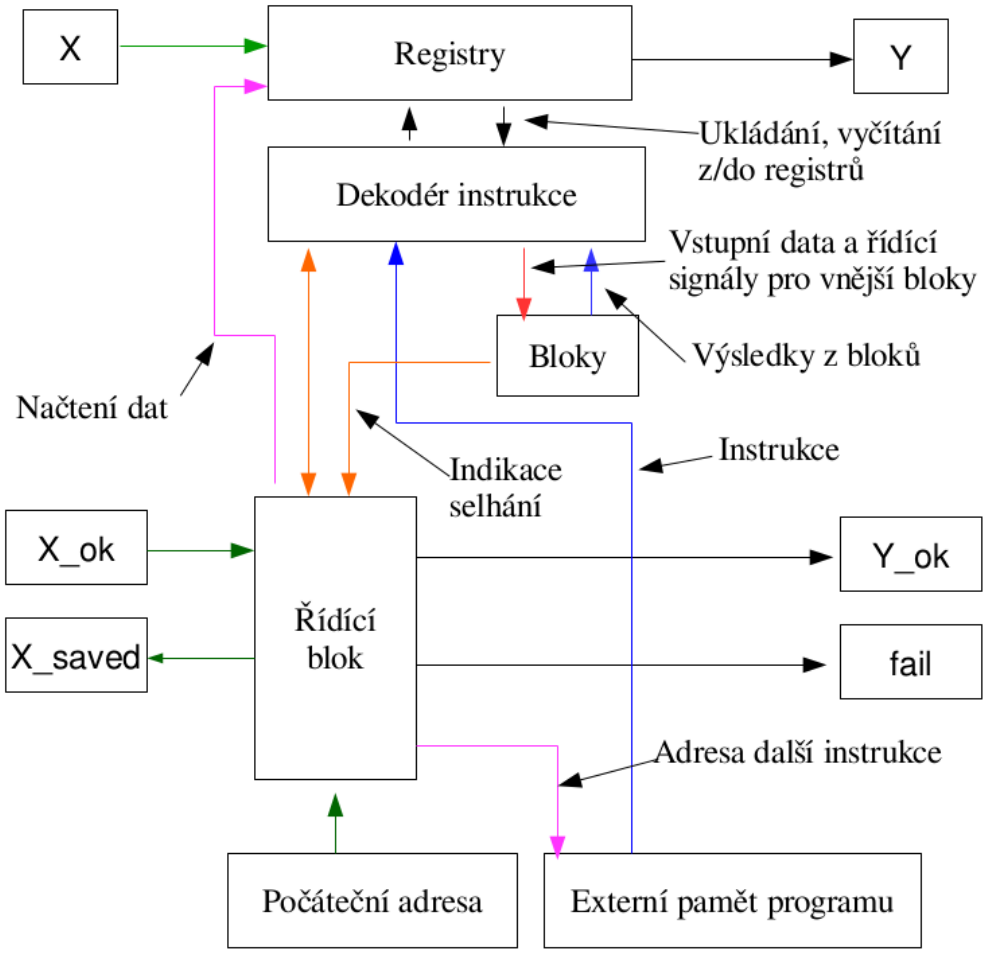
Jestliže blok `DSP_corePipelineRAM` nepracuje, po aktivaci (log. 1) signálu `X_ok` dojde ke zkopírování vstupních dat (signál `X`) do vnitřních registrů, což je potvrzeno impulsem (log. 1) na signálu `X_saved`. Následně blok `DSP_corePipelineRAM` zahajuje výpočet a další data na vstupu až do dokončení výpočtu ignoruje. Výpočet je zahájen instrukcí na adrese, jež je nastavena vnějším signálem `addressStart`, a pokračuje instrukci po instrukci až do instrukce ukončující výpočet nebo do selhání



Obr. 3.9: Struktura bloku `DSP_pipelineRAM`

některého z připojených bloků či vypršení limitu interního čítače hlídajícího zamrznutí. V případě, že výpočet končí jinak než odpovídající instrukcí, je aktivován (log. 1) signál `fail` a blok se přepíná do stavu očekávání nových vstupních dat.

Při komunikaci bloku `DSP_corePipelineRAM` s připojenými bloky jsou používány validační signály značící připravenost bloků na příjem dat (`dev_in_ready`), korektnost vstupních dat (`dev_in`), připravenost dat na vyčtení (`dev_out`), vyčtení dat blokem `DSP_corePipelineRAM` (`dev_out_ready`) a selhání činnosti (`dev_fail`). Vstupní data pro bloky jsou přenášena signály `dev_A` a `dev_B` a výsledky výpočtů signály



Obr. 3.10: Znázornění funkce bloku `DSP_corePipelineRAM`

`dev_Y`. V případě potřeby blok `DSP_corePipelineRAM` čeká, dokud připojený blok data nepřevezme nebo nevydá. Špatná sekvence instrukcí může způsobit čekání na předání výsledků z bloku, kterému nebyla předána data pro výpočet. V takovém případě po určité prodlevě zareaguje interní čítač a dojde k aktivaci signálu `fail`.

Díky výše popsané a znázorněné architektuře bloku `DSP_corePipelineRAM` není tento blok sám o sobě schopen spočítat vůbec nic. Dokonce nemusí být schopen ani přenést vstupní signály na výstup. Avšak, jak je znázorněno na obrázku 3.9, k bloku `DSP_corePipelineRAM` je možné připojit množství výpočetních bloků, které můžeme volit na míru aplikaci. Není vyloučené ani připojení více bloků stejné funkce.

Oproti blokům `DSP_corePipelineBasic` a `DSP_pipelineBasic` mají bloky `DSP_pipelineRAM` a `DSP_corePipelineRAM` vyvedené také signály `regA_en`, `regB_en`, `regA_addr`, `regB_addr`, `regA_read`, `regB_read`, `regA_we` a `regA_write` sloužící pro připojení externí paměti pro pracovní registry bloku. Paměť musí být

dvouportová, přičemž od obou portů je vyžadována podpora čtení a od jednoho portu také podpora zápisu. Důvodem pro vyvedení paměti registrů ven z bloku `DSP_corePipelineRAM` byly potíže Xilinx Vivado s efektivní syntézou a implementací bloků `DSP_corePipelineBasic` a `DSP_pipelineBasic`, které popisují interní registry bloku odlišným způsobem.

Kromě signálových vstupů a výstupů mají bloky `DSP_corePipelineRAM` a `DSP_pipelineRAM` také vstupy v podobě generických parametrů. Generické parametry bloků jsou reprezentovány celými kladnými nenulovými čísly.

Parametry `GNinputs`, `GNfirst_input`, `GNoutputs` a `GNfirst_output` určují počet vstupů `X` a výstupů `Y` procesoru a oblast registrů, do které se mají vstupy nahrát, popřípadě registrů, ze kterých se mají výstupy vyčíst. Součet `GNinputs` a `GNfirst_input` musí být nižší nebo roven počtu registrů procesoru. Totéž platí pro součet `GNoutputs` a `GNfirst_output`. Počet registrů je nastavován parametrem `GNregisters`, k němuž je přidružený parametr `GNregister_bits`, který určuje bitovou šířku signálu používaného pro adresaci registrů.

Počet připojených bloků pro výpočty s čísly s plovoucí desetinnou čárkou lze nastavit parametrem `GNdevices`, který je dostupný jen pro blok `DSP_corePipelineBasic`. S počtem připojených bloků je svázán také signál `GNoperation_bits` určující počet bitů, které určují instrukční část instrukce.

Velikost instrukce je nastavena parametrem `GNinstruction_bits`, který by měl být roven součtu `GNoperation_bits`+2∗`GNregister_bits`. Šířku signálu pro adresování instrukcí nastavuje parametr `GNaddress_bits`. Parametr `GNlatency` nastavuje zpoždění paměti instrukcí v hodinových taktech. S ním svázaný parametr `GNlatency_bits` určuje bitovou šířku čítače zpoždění.

Posledními parametry jsou `GNwatchdog_limit` a `GNwatchdog_bits`, které slouží pro nastavení limitní hodnoty interního čítače pro detekci uvíznutí a jeho bitové šířky.

Jak již bylo naznačeno, pro správnou funkci bloku `DSP_corePipelineRAM` je třeba tomuto bloku dodat korektní sekvenci instrukcí. Toho lze docílit připojením vhodně inicializované paměti na signály `address` a `instruction`. Paměť vygenerovanou pomocí IP bloku z prostředí Xilinx Vivado lze inicializovat datovým souborem typu COE.

3.4.2 Paměť s instrukcemi pro blok `DSP_pipelineRAM`

Základem k vytvoření sekvence instrukcí pro blok `DSP_pipelineRAM` je programem AnalyzeAlgorithm vygenerovaný kód SVA (soubor QR_4x4_complex_step.sva, viz elektronická příloha). Před provedením optimalizací nad tímto souborem,

ale i po jednotlivých optimalizacích, je vhodné ověřit funkčnost zdrojového SVA kódu. Toho můžeme dosáhnout použitím programu SVAInterpreter.

O vlastní optimalizace se stará program SVAOptimalizer. Nad kódem získaným z programu AnalyzeAlgorithm je vždy nutné provést optimalizaci dle sady optimalizačních pravidel QRexpand, aby byl kód SVA uvedený do tvaru kompilovatelného programem SVAToDSPCompiler.

SVA kód připravený ke kompilaci však obsahuje značný počet nadbytečných operací (zvláště operace přiřazení). Ponechání těchto operací v kódu by nenarušilo jeho funkčnost. Mělo by však negativní vliv na efektivitu výsledné sekvence instrukcí. Je proto vhodné aplikovat také sadu optimalizačních pravidel QRreduce. Výsledek je opět vhodné prověřit programem SVAInterpreter.

V přílohách H.1, I.1 a J.1 lze nalézt SVA kód pro QR rozklad reálné matice řádu 2 v základní podobě (H.1), po optimalizaci dle pravidel QRexpand (I.1) a po optimalizaci dle pravidel QRexpand a QRreduce (J.1).

Z optimalizovaného SVA kódu vygenerujeme za použití programu SVAToDSPCompiler soubor typu COE pro inicializaci paměti v prostředí Xilinx Vivado. Kompilaci provádíme pro zařízení s názvem QRstep, jehož parametry jsou v programu nakonfigurovány a odpovídají konfiguraci bloku `DSP_pipelineRAM`, který je použit v bloku `QR_Algorithm`.

Zdrojové kódy aplikací a k nim příslušejících knihoven a modulů jsou součástí elektronické přílohy této práce.

3.4.3 Optimalizace SVA kódu

Optimalizace SVA kódu spočívají v náhradě instrukcí za instrukce ekvivalentní, vyřazení instrukcí, jejichž výsledky nejsou použity a vyřazení instrukcí, jejichž provedení není nezbytně nutné pro výpočet správného výsledku. Také je optimalizováno použití proměnných tak, aby do každé proměnné byla přiřazena hodnota právě jednou. Tento krok je nezbytný pro správnou práci některých optimalizačních pravidel. Stejný krok je vyžadován i programem SVAToDSPCompiler, který jej však dokáže provést samostatně.

Optimalizace dle pravidel QRexpand

V případě optimalizačních pravidel QRexpand dochází k rozpisu operátorů "+=", "-=", "*="a "/=". Pokud by byl zmíněným optimalizacím podroben zdrojový kód v jazyce C, odpovídající výpisu 3.8, optimalizovaný kód by odpovídal výpisu 3.9.

Výpis 3.8: Výchozí kód pro ilustraci optimalizací dle pravidel QRexpand

```
1    V0 = X0;
2    V0 += X0;
3    V0 -= X1;
4    V0 *= V0;
5    V1 = V0 / X3;
6    Y0 = V0 + X2;
```

Výpis 3.9: Mezivýsledek optimalizace dle pravidel QRexpand

```
1    V0 = X0;
2    V0 = V0 + X0;
3    V0 = V0 - X1;
4    V0 = V0 * V0;
5    V1 = V0 / X3;
6    Y0 = V0 + X2
```

Výsledku optimalizace pro rozšíření počtu proměnných tak, aby do každé proměnné byl výsledek přiřazen právě jednou, odpovídá kód dle výpisu 3.10.

Výpis 3.10: Výsledek optimalizace dle pravidel QRexpand

```
1    V0 = X0;
2    V2 = V0 + X0;
3    V3 = V2 - X1;
4    V4 = V3 * V3;
5    V1 = V4 / X3;
6    Y0 = V4 + X2
```

Optimalizace dle pravidel QRreduce

Dalším krokem je aplikace pravidel optimalizace QRreduce. Pro ilustraci problému je výchozím kód 3.11.

Výpis 3.11: Výchozí kód pro ilustraci optimalizací dle pravidel QRreduce

```
1    V0 = X0;
2    V2 = V0 + 0;
3    V3 = V2 - X1;
4    V4 = V2 * 0;
5    V5 = V3 * V4;
6    V6 = V3 * 1;
7    V7 = V6 + V5;
8    V1 = V7 / X3;
9    Y0 = V7 + X2
```

Z optimalizací dle pravidel QRreduce je nejprve provedena eliminace nadbytečných přiřazení. Po této optimalizaci vzniká kód dle výpisu 3.12.

Výpis 3.12: Mezivýsledek optimalizace dle pravidel QRreduce po eliminaci nadbytečných přiřazení

```
1    V2 = X0 + 0;
2    V3 = V2 - X1;
3    V4 = V2 * 0;
4    V5 = V3 * V4;
5    V6 = V3 * 1;
6    V7 = V6 + V5;
7    V1 = V7 / X3;
8    Y0 = V7 + X2
```

Následují redukce sčítání čísel s nulou, násobení čísel nulou a jedničkou. Tím je získán kód odpovídající výpisu 3.13.

Výpis 3.13: Mezivýsledek optimalizace dle pravidel QRreduce po redukci přičítání nul, násobení nulou a násobení jedničkou

```
1    V2 = X0;
2    V3 = V2 - X1;
3    V4 = 0;
4    V5 = V3 * V4;
5    V6 = V3;
6    V7 = V6 + V5;
7    V1 = V7 / X3;
8    Y0 = V7 + X2
```

Opakováním eliminace nadbytečných přiřazení je kód upraven do podoby 3.14.

Výpis 3.14: Mezivýsledek optimalizace dle pravidel QRreduce po opakované redukci nadbytečných přiřazení

```
1    V3 = X0 - X1;
2    V5 = V3 * 0;
3    V7 = V3 + V5;
4    V1 = V7 / X3;
5    Y0 = V7 + X2
```

Poslední pravidla v sadě optimalizací QRreduce zajišťují zahození výpočetních operací, jejichž výsledky nejsou použity. V ilustračním kódu jsou proměnné X považované za vstup, proměnné Y za výstup a proměnné V za dočasné. Aplikací na ilustrační kód (výpis 3.14) vznikne kód dle výpisu 3.15.

Výpis 3.15: Mezivýsledek optimalizace dle pravidel QRreduce po odstranění operací, jejichž výsledky nejsou použity

```
1    V3 = X0 - X1;
2    V5 = V3 * 0;
3    V7 = V3 + V5;
4    Y0 = V7 + X2
```

Jelikož jsou optimalizační pravidla opakována, dokud je co optimalizovat, proběhne celý cyklus optimalizací znovu a konečným výsledkem je kód odpovídající výpisu 3.16.

Výpis 3.16: Výsledek optimalizace dle pravidel QRreduce

```
1    V3 = X0 - X1;
2    Y0 = V3 + X2
```

3.4.4 Kompilace SVA kódu

Úkolem kompilace SVA kódu je převedení jednotlivých instrukcí do instrukcí bloku `DSP_pipelineRAM`. Kompilace je specifická pro příslušnou konfiguraci bloku `DSP_pipelineRAM` a pro příslušné zapojení výpočetních bloků k vnořenému bloku `DSP_corePipelineRAM`. Pokud se jedno nebo druhé změní, je nutné provést kompilaci s novým, změnám odpovídacím nastavením.

Prvním krokem kompilace je provedení pomocných optimalizací. Jedná se o již zmíněné rozšíření počtu pomocných proměnných SVA kódu tak, aby k jedné proměnné náleželo právě jedno přiřazení.

Následuje kontrola vstupního SVA kódu po aplikaci pomocných optimalizací. Kompilační program SVAToDSPCompiler v tomto kroku kontroluje, zda je do všech

použitých proměnných přiřazena právě jedna hodnota a zda proměnné nejsou použity dříve, než je jejich hodnota nastavena.

Pokud SVA kód projde kontrolou, je zahájena jeho kompilace. Každé SVA instrukci je přiřazena množina instrukcí bloku `DSP_pipelineRAM`. Instrukce bloku se dělí na vstupní a výstupní. Vstupní instrukce, které zpravidla vkládají data určená k výpočtu do příslušného výpočetního bloku, jsou provedeny okamžitě. Instrukce výstupní, které zpravidla vyčítají výsledky z příslušného výpočetního bloku, jsou uloženy do zásobníku spolu s informací, v jakém taktu bude výsledek dostupný. Do sekvence instrukcí jsou přednostně přidávány operace, jejichž vstupy jsou dostupné v registrech, popřípadě jsou již vypočítané a čekají na vyčtení. Pokud je provedení instrukce závislé na proměnné nevyčtené, je před provedení instrukce vložena odpovídající instrukce vyčtení. Po kompilaci celého SVA kódu je nakonec přidána instrukce ukončení výpočtu a prezentace výsledků.

Posledním krokem kompilace je mapování proměnných do registrů bloku `DSP_corePipelineRAM`. Přednostně jsou mapovány vstupní a výstupní proměnné. Po namapování vstupů a výstupů následuje mapování pomocných proměnných, které respektuje dobu jejich platnosti, jež je dána prvním přiřazením a posledním použitím. Posledním použitím končí i platnost vstupních proměnných.

Po namapování proměnných do registrů následuje doplnění odpovídajících adres do instrukcí a vygenerování výsledné sekvence instrukcí, která je následně uložena do souboru formátu COE s ohledem na bitovou šířku jedné instrukce. Vytvořením COE souboru je kompilace ukončena.

3.5 Simulace

V blokačním módu pracující aritmetická jednotka `APU_small` je verifikována proti výsledkům z aritmetické jednotky `APU_NB`, jejíž funkce byla ověřena autorem vůči výsledkům z programu GNU Octave. Simulace ověřující funkci bloků `QR_Algorithm` a `DSP_pipelineRAM` využívají automatické, popřípadě ruční porovnání s výsledky z GNU Octave.

3.5.1 Struktura simulace

Simulace využívají pro generování hodinového signálu procesu `clk_process` (viz. výpis 3.17).

Výpis 3.17: Simulační proces clk_process

```
1      -- Clock process definitions( clock with 50%
          duty cycle is generated here.
2      clk_process : process
3      begin
4          clk <= '0';
5          wait for clk_period/2;  --for 0.5 ns signal
              is '0'.
6          clk <= '1';
7          wait for clk_period/2;  --for next 0.5 ns
              signal is '1'.
8      end process;
```

Použití procesu `clk_process` vyžaduje definici signálu `clk` a konstanty `clk_period` (viz výpis 3.18). S hodinovým signálem souvisí také signál `clk_en` povolující hodiny, který je v simulacích trvale připojen k logické 1.

Výpis 3.18: Signály pro proces clk_process

```
1      constant clk_period : time := 10 ns;
2
3      signal clk : STD_LOGIC := '0';
4      signal clk_en: STD_LOGIC;
```

Ukončení simulace po vypršení určitého odsimulovaného časového úseku je zajištěno procesem `watchdog_process` (viz. výpis 3.19).

Výpis 3.19: Simulační proces watchdog_process

```
1       -- watchdog process
2       WATCHDOG_will <= WATCHDOG_is + 1;
3
4       watchdog_process: process (clk, clk_en,
           WATCHDOG_will)
5       begin
6          if rising_edge(clk) then
7              if (clk_en='1') then
8                  WATCHDOG_is <= WATCHDOG_will;
9
10                 assert WATCHDOG_is<WATCHDOG
11                     report "Watchdog timeout"
12                     severity FAILURE;
13
14             end if;
15         end if;
16      end process;
```

Pro použití procesu `watchdog_process` je nutné definovat signály `WATCHDOG_is` a `WATCHDOG_will` spolu s konstantou `WATCHDOG` (dle výpisu 3.20). Změnou konstanty lze ovlivnit odsimulovaný čas, který musí uplynout pro zastavení simulace.

Výpis 3.20: Signály pro proces watchdog_process

```
1       constant WATCHDOG : unsigned(7 downto 0) :=
           to_unsigned(200,8);
2
3       signal WATCHDOG_is, WATCHDOG_will : unsigned(7
           downto 0) := (others => '0');
```

Porovnání výsledků s očekávanými hodnotami je možné prostřednictvím procesu `sim_check_process` (výpis 3.22) s nímž souvisí definice signálů dle výpisu 3.21.

Výpis 3.21: Signály pro proces sim_check_process

```
1       signal CHECK_A, CHECK_V, CHECK_ERR : FP_NUMBER;
2       signal CHECK_A_ready, CHECK_A_ok : STD_LOGIC;
3       signal CHECK : STD_LOGIC;
4       signal CHECK_ok : STD_LOGIC;
```

Výpis 3.22: Signály pro proces sim_check_process

```
1   -- check real part of result process
2   sim_check_process: process (clk, clk_en, CHECK,
      CHECK_ok)
3   begin
4       if rising_edge(clk) then
5           if (clk_en='1') then
6               if (CHECK_ok='1') then
7                   assert (CHECK='1')
8                       report "Bad␣check␣of␣result"
9                       severity ERROR;
10              end if;
11          end if;
12      end if;
13  end process;
```

Čísla určená k porovnání jsou představována signály `CHECK_A` a `CHECK_V`. Velikost povolené chyby vyjadřuje signál `CHECK_ERR`. Výsledek porovnání je přenášen signálem `CHECK`, jež je validní v době aktivního signálu `CHECK_ok`. Pokud je kontrolované číslo ve shodě s číslem žádaným, je výsledek porovnání signalizován aktivitou (log. 1) signálu `CHECK`.

Použití procesu `sim_check_process` vyžaduje připojení bloku `FP_CHECK` na příslušné signály dle výpisu 3.23.

Výpis 3.23: Obvyklé zapojení bloku FP_CHECK v simulaci

```
1   -- component for check results
2   test_CHECK: FP_CHECK
3     port map (  clk => clk, clk_en => clk_en, reset
          => reset,
4                 A => CHECK_A, A_ready =>
                    CHECK_A_ready, A_ok =>
                    CHECK_A_ok,
5                 V => CHECK_V, ERR => CHECK_ERR,
                    VERR_ready => CHECK_VERR_ready,
                     VERR_ok => CHECK_VERR_ok,
6                 CHECK => CHECK, CHECK_ready =>
                    CHECK_ready, CHECK_ok =>
                    CHECK_ok);
```

Konkrétní zapojení se v různých simulacích mírně odlišuje. Signál `CHECK_ready` může být trvale aktivní, signály `CHECK_A_ready` a `CHECK_VERR_ready` mohou být nepřipojeny. Přípustné je i sjednocení signálů `CHECK_A_ok` a `CHECK_VERR_ok`.

3.5.2 Simulace bloků `APU_NB` a `APU_NB_small`

Výsledek simulace bloků `APU_NB` a `APU_NB_small` je v příloze O.1. Simulace využívá struktury popsané v kapitole 3.5.1.

Během simulace se postupně provádějí operace reálného sčítání, reálného odčítání, reálného násobení, reálného dělení, komplexního sčítání, komplexního odčítání, komplexního násobení a komplexního dělení. Hodnoty vstupních čísel jsou dané (viz tabulka 3.5). Pomocí vhodného programu (např. GNU Octave/Matlab) lze vypočítat předpokládané výsledky operací a srovnat je s výsledky zjištěnými ze simulace (tabulka 3.5). Výpočty v tabulce 3.5 jsou zaokrouhleny na tři desetinná místa.

Tab. 3.5: Srovnání očekávaných a simulovaných výsledků vypočítaných aritmetickou jednotkou `APU_NB`

`A`	`B`	**Operace**	**Výpočet GNU Octave**	**Simulovaný výsledek**
5.15	16.00589	sčítání	21.156	21.156
5.15	16.00589	odčítání	−10.856	−10.856
5.15	16.00589	násobení	82.430	82.430
5.15	16.00589	dělení	0.322	0.322
5.15 − 2.2511i	16.00589 + 3.13i	sčítání	21.156 + 0.879i	21.156 + 0.879i
5.15 − 2.2511i	16.00589 + 3.13i	odčítání	−10.856 − 5.381i	−10.856 − 5.381i
5.15 − 2.2511i	16.00589 + 3.13i	násobení	89.476 − 19.911i	89.476 − 19.911i
5.15 − 2.2511i	16.00589 + 3.13i	dělení	0.283 − 0.196i	0.283 − 0.196i

3.5.3 Simulace bloku `APU_small`

Cílem simulace bloku `APU_small` je ověření funkce aritmetické jednotky. Blok `APU_NB` je zde použit jako referenční aritmetická jednotka. Výsledek simulace bloku `APU_small` je v příloze P.1. Postup simulace je obdobný jako v případě simulace aritmetických jednotek `APU_NB` a `APU_NB_small`.

3.5.4 Simulace bloku `DSP_pipelineRAM`

Simulace bloku `DSP_pipelineRAM` je rozdělena do několika částí. Výchozí simulace (soubor "sim_DSP_NoSignFullDeviceTest.vhd", viz elektronická příloha) ověřuje základní funkce bloku. Jedná se například o schopnost bloku `DSP_corePipelineRAM` komunikovat s bloky pro výpočty nad čísly s plovoucí desetinnou čárkou, načítat vstupní data a prezentovat vypočítané výsledky. Výsledky jsou ověřeny automaticky vůči výsledkům získaným z GNU Octave.

Soubor "sim_DSP_ComplexQR_4x4.vhd" (viz elektronická příloha) ověřuje schopnost bloku `DSP_pipelineRAM` realizovat rozsáhlý výpočet (komplexní QR rozklad matice řádu 4). V tomto případě není automatická verifikace prováděna.

3.5.5 Simulace bloku `QR_Algorithm`

Výstupem této práce je blok `QR_Algorithm`. Jeho simulace (soubor "sim_QR_Algorithm.vhd", viz elektronická příloha) ověřuje schopnost bloku opakovaně provádět výpočet iterací QR algoritmu až do dosažení požadované přesnosti. Výsledky jsou kontrolovány automaticky oproti výsledkům aplikovaného algoritmu v programu GNU Octave.

Verifikace bloku `QR_Algorithm` je kromě simulace možná i skrze procesor architektury ARM integrovaný v čipu Zynq-7020, který je součástí vývojové desky ZedBoard. Takový postup ověření funkce je realizovatelný prostřednictvím syntézy a implementace bloku `topARMverification` (soubor "topARMverification.vhd", elektronická příloha) a následného nahrání výsledného bitstreamu do hradlového pole. Po spuštění programu QRverification (viz elektronická příloha) v procesoru integrovaném na čipu Zynq-7020 lze stisknutím libovolného z petice tlačítek (BTNU, BTNL, BTNC, BTNR nebo BTND) spustit vygenerování jedné náhodné hermitovské pozitivně-semidefinitní matice řádu 4. Program vygenerovaná data předá bloku `QR_Algorithm` a vyčte zpět výsledek výpočtu. Vygenerovaná matice i jí odpovídající výsledky jsou z procesoru odesílány prostřednictvím UARTu jako čistý text.

4 ZÁVĚR

V rámci této práce byly zkoumány možnosti výpočtu vlastních čísel a vektorů symetrických a hermitovských matic. Byl analyzován algoritmus Householderovy transformace reálné matice do třídiagonálního tvaru a algoritmus QR rozkladu za použití Householderových matic. Získané výsledky byly použity pro sestavení a implementaci bloku pro výpočet vlastních čísel a vlastních vektorů čtvercové komplexní hermitovské pozitivně-semidefinitní matice řádu 4 do hradlového pole řady Xilinx Zynq-7000 se zaměřením na efektivní využití zdrojů.

Za účelem analýzy algoritmů a zpracování SVA kódu byly sestaveny programy AnalyzeAlgorithm, SVAInterpreter, SVAoptimalizer a SVAToDSPCompiler. Bylo také nutné rozšířit knihovny Processingg, Calculate a Format a moduly ESEM a CalculateSEM. Pro potřeby SVAToDSPCompileru byla naprogramována knihovna HarDes a modul HarDesSEM.

Žádaný výpočet vlastních čísel a vlastních vektorů je realizován blokem `QR_Algorithm`, který realizuje výpočet využitím QR algoritmu. Implementace do hradlového pole byla verifikována pomocí simulací a procesoru ARM integrovaného na čipu Zynq 7020. Blok dokáže pracovat s frekvencí 100 MHz, přičemž jedna iterace vyžaduje 9778 taktů. Na testovací matici bylo dosaženo přesnosti v řádu 10^{-5} v pěti iteracích a čase kratším 500 μs. Během každé iterace je vykonáno 8635 instrukcí, což odpovídá 4317 výpočtům provedených nad čísly s plovoucí desetinnou čárkou. Při výpočtu je nutné celkem po dobu 1087 taktů čekat na mezivýsledky. Efektivita využití výpočetního bloku dosahuje 89 %. Bez aplikace optimalizací (program SVAoptimalizer) by bylo nutné provádět 5896 operací nad čísly s plovoucí desetinnou čárkou.

LITERATURA

[1] *Komplexní čísla* [online]. poslední aktualizace 2. 12. 2015 [cit. 2. 12. 2015]. Dostupné z URL: <http://www.matematika.cz/komplexni-cisla>.

[2] KRUPKOVÁ, Vlasta - FUCHS, Petr. *MATEMATIKA 1* 1. vyd., Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav matematiky, 2007. 288 s. ISBN 978-80-214-3438-7

[3] VICHER, Miroslav. *NUMERIKA* [online]. 26. 3. 2001,[cit. 24. 11. 2015]. Dostupné z URL: <http://physics.ujep.cz/˜jskvor/NME/Vicher/NUMER.PDF>.

[4] *Lineární algebra - Vlastní čísla a vektory matic* [online]. poslední aktualizace 30. 1. 2003 [cit. 24. 11. 2015]. Dostupné z URL: <http://artemis.osu.cz/mmmat/txt/la/mvl.htm>.

[5] *Symetrické matice* [online]. poslední aktualizace 17. 3. 2005 [cit. 26. 11. 2015]. Dostupné z URL: <https://math.feld.cvut.cz/ftp/vyuka/MA5/SymetrMatice.pdf>.

[6] *Permutace* [online]. poslední aktualizace 4. 12. 2015 [cit. 4. 12. 2015]. Dostupné z URL: <http://www.matematika.cz/permutace>.

[7] *1.2. NORMA A SKALÁRNÍ SOUČIN* [online]. poslední aktualizace 9. 9. 2002 [cit. 4. 12. 2015]. Dostupné z URL: <http://artemis.osu.cz/Matem/2/1_02.pdf>.

[8] *Sklární součin* [online]. poslední aktualizace 19. 5. 2003 [cit. 4. 12. 2015]. Dostupné z URL: <http://www.karlin.mff.cuni.cz/˜tuma/2002/NLinalg9.pdf>.

[9] BASTINEC, Jaromír - NOVÁK, Michal. *Moderní numerické metody* [online]. [cit. 19. 11. 2015]. Dostupné z URL: <http://matika.umat.feec.vutbr.cz/inovace/materialy/skripta/mmnm.pdf>.

[10] QUARTERONI, Alfio - SACCO, Riccardo - SALERI, Fausto. *Numerical Mathematics (Second Edition)* 1. vyd., Berlín: nakladatelství Springer Berlin Heidelberg, 2007. 655 s. ISBN 0939-2475

[11] MROVEC, Martin. *Řešení problémů vlastních čísel s aplikacemi v molekulové dynamice* Ostrava: Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, 2013, 51 s. Vedoucí bakalářské práce prof. RNDr. Zdeněk Dostál, DSc.

[12] *QR Decomposition with Gram-Schmidt* [online]. poslední aktualizace 24. 5. 2007 [cit. 27. 11. 2015]. Dostupné z URL: <http://www.math.ucla.edu/~yanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf>.

[13] TEJKAL, Jan. *QR-algoritmus* Praha: Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, 2010, 59 s. Vedoucí bakalářské práce Doc. RNDr. Jan Zítko, CSc.

[14] OBERHUBER, Tomáš. *Vlastní čísla matic - aplikace* [online]. poslední aktualizace 17. 1. 2015 [cit. 9. 12. 2015]. Dostupné z URL: <`http://geraldine.fjfi.cvut.cz/~oberhuber/data/vyuka/num-1/06-vlastni-cisla.pdf`>.

[15] *Numerical recipes* [online]. poslední aktualizace 17. 1. 2013 [cit. 25. 11. 2015]. Dostupné z URL: <http://www.it.uom.gr/teaching/linearalgebra/NumericalRecipiesInC/c11-4.pdf>.

[16] *LogiCORE IP Floating-Point Operator v7.0* [online]. poslední aktualizace 5. 10. 2015 [cit. 27. 11. 2015]. Dostupné z URL: <http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf>.

[17] TIŠNOVSKÝ, Petr. *Norma IEEE 754 a příbuzní: formáty plovoucí řádové tečky* [online]. poslední aktualizace 31. 5. 2006 [cit. 1. 12. 2015]. Dostupné z URL: <http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky/>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

$\mathbf{A}$	matice
$\mathbf{v}$	vektor
$\|\|\mathbf{v}\|\|_2$	Euklidovská norma vektoru $\mathbf{v}$
i	imaginární jednotka
$\boldsymbol{Z}$	komplexní číslo
$\mathbb{R}$	množina reálných čísel
$\mathbb{C}$	množina komplexních čísel
SVA	SFENCE Variable Assembler

SEZNAM PŘÍLOH

A Generování autokorelační matice zatížené šumem **76**

B Jacobiho metoda **78**

C Householderova transformace na matici třídiagonální **80**

D Gram-Schmidtův QR rozklad **81**

E Householderův QR rozklad **82**

F QR rozklad pomocí Givensovy matice **83**

G QR algoritmus pro výpočet vlastních čísel a vektorů matice **84**

H Výpis SVA kódu Householderovova QR rozkladu matice řádu 2 **85**

I Výpis SVA kódu QR rozkladu po základní optimalizaci **88**

J Výpis SVA kódu QR rozkladu po základní a redukující optimalizaci **91**

K Význam portů IP bloku **93**

L Rozhraní bloků pro výpočty s čísly s plovoucí desetinnou čárkou **95**

M Definice rozhraní bloku DSP_pipelineRAM **103**

N Definice rozhraní bloku DSP_corePipelineRAM **105**

O Simulace APU_NB a APU_NB_small **107**

P Simulace APU_small **108**

Q Obsah přiloženého CD **109**

A GENEROVÁNÍ AUTOKORELAČNÍ MATICE ZATÍŽENÉ ŠUMEM

Výpis A.1: Autokorelační matice zatížená šumem

```
%% Vypocet vlastnich cisel a vlastnich vektoru

%% Generovani hermitovske,
% pozitivne-semidefinitni komplexni ctvercove
% matice "W", jejiz vlastni vektor prislusici
% nejvetsimu vlastnimu cislu
% bude "v", ostatni vlastni cisla
% a vektory budou prisluset sumu.
% Matice "W" je velikosti "m".
m = 4;
% Vektor "v"
v = complex(randn(m,1),randn(m,1));
% Matice "W"
W = v * v';
% Pridani kladne realne konstanty k diagonale =
   pridani sumu
sigma = sqrt(norm(v))/5; % 5 x mensi
W = W + sigma^2 * eye(m,m);
% Pozn. Matice "W" odpovida autokorelacni matici
% vzorku vicekanalovych dat obsahujicich signal
% prave z jednoho zdroje a nezavisly (v case
% i mezi kanaly) gaussovsky sum.
% Tato matice bude tedy mit jedno dominantni
% vlastni cislo a zbyle na urovni sumu (sigma^2).

%% Vlastni cisla, vlastni vektory
% Vlastni vektory jsou sloupce eigenVectors, vlastni
    cisla
% jous na diagonale eigenValues
[eigenVectors, eigenValues] = eig(W);
% Vlastni cisla nejsou setrizena, provedeme
   setrizeni
[max,iSort] = sort(diag(eigenValues),'descend');
% Vlastni cislo nalezi vzdy konkretnimu vlastnimu
```

```
    vektoru,
% provedeme prislusne setrizeni vlastnich vektoru
eigenVectors = eigenVectors(:,iSort);

% Zobrazeni: Pro srovnani obou vektoru pouzijeme
    metodu
% unifikace faze na prvni hodnote =
% faze na prvim prvku vektoru bude nulova =
%prvni prvek vektoru bude realne cislo.
vUnif = v./v(1,1);
eigenVectors = eigenVectors ./ eigenVectors(1,1);
% Vektory zobrazime v komplexni rovine,
% kde osa x je realna cast prvku vektoru,
% osa y potom imaginarni cast vektoru
figure
plot(real(vUnif), imag(vUnif), 'ob');
hold on
plot(real(eigenVectors(:,1)), imag(eigenVectors(:,1))
    , 'xr');
grid on
title('Nejvetsi vlastni vektor')
legend('Skutecny','Nejvetsi vlastni vektor')
```

B JACOBIHO METODA

Výpis B.1: Jacobiho metoda[11]

```
1  % jacobi maethod
2
3  function [eigval,eigvec, iter] = JacobiDiag(A, myeps)
4    %eigval - diag. matrix of eigenvaluesm eufvec -
       matrix of eifenvectors
5    %iter - nr of iterations
6    format long;
7    iter = 0;
8    S = A;
9    V = eye(size(S,1));
10   [ i , j ] = FindPivotIndex(S);
11   while(norm(S-diag(diag(S)))>=myeps)
12     G = CreateGivensMatrix(S,i,j);
13     S = G'*S*G;
14     V = V*G;%rotate eigenvectors
15     [ i , j ] = FindPivotIndex(S);
16     iter = iter + 1;
17   end
18   eigvec = V;
19   eigval = diag(diag(S));
20 end
21
22 function [p, q] = FindPivotIndex(A)
23   format long;
24   n = size(A,1);
25   p = 1;
26   q = 2;
27   pivot = 0;
28   for i = 1:n-1
29     for j = i+1:n
30       if (abs(A(i,j )) > abs(pivot))
31         pivot = A(i, j );
32         p = i;
33         q = j;
34       end
```

```
35      end
36    end
37  end
38
39  function [G] = CreateGivensMatrix(M, i, j)
40    format long;
41    if (M(i, i ) == M(j,j ))
42      mytheta = pi/4;
43    else
44      mytheta = 1/2 * atan( 2 * M(i, j ) / (M(i, i ) -
          M(j,j)));
45    end
46    G = eye(size(M,1));
47    G(i, i ) = cos(mytheta);
48    G(j, j ) = cos(mytheta);
49    G(i, j ) = -sin(mytheta);
50    G(j, i ) = sin(mytheta);
51  end
```

C HOUSEHOLDEROVA TRANSFORMACE NA MATICI TŘÍDIAGONÁLNÍ

Výpis C.1: Householderova transformace na matici třídiagonální

```
1  % Householder transform real
2
3  function out = HouseholderTransformReal(A)
4    sizeA = size(A);
5
6    out = A;
7    ort = eye(sizeA);
8
9    for (k=2:(sizeA-1))
10     for (j=1:(k-1))
11       v(j, 1) = 0;
12     end;
13
14     hlpA = 0;
15     for (j=k:sizeA)
16       v(j, 1) = out(j,k-1);
17       hlpA += out(j,k-1)^2;
18     end;
19     v(k, 1) += sign(v(k))*sqrt(hlpA);
20
21     w = (1/(sqrt(v'*v))*v')';
22
23     q = 2*(eye(sizeA)-w*w')*out*w;
24
25     out = out - w*q' - q*w';
26   end;
27 end;
```

D GRAM-SCHMIDTŮV QR ROZKLAD

Výpis D.1: QR rozklad pomocí Gram-Schmidtova algoritmu

```
1  % GramSchmidt QR
2
3  function [Q,R] = QR_GramSchmidt(A)
4    sizeA = size(A)(1);
5
6    Q = zeros(sizeA);
7    U = Q;
8    R = Q;
9    for k = [1:sizeA]
10     U(1:sizeA,k) = A(1:sizeA,k);
11
12     for n=[1:k-1]
13       U(1:sizeA,k) = U(1:sizeA,k) - (A(1:sizeA,k)'*Q
             (1:sizeA,n))*Q(1:sizeA,n);
14     end
15     Q(1:sizeA,k) = U(1:sizeA,k)/norm(U(1:sizeA,k),2);
16   end
17
18   R = Q'*A;
19 end
```

E HOUSEHOLDERŮV QR ROZKLAD

Výpis E.1: QR rozklad pomocí Householderových transformací

```
1  % Householder QR
2
3  function [Q,R] = QR_Householder(A)
4    sizeA = size(A)(1);
5
6    R = A;
7    Q = eye(sizeA);
8
9    for (k=1:(sizeA-1))
10     z = zeros(sizeA, 1);
11     x = z;
12     z(k, 1) = 1;
13     for (j=k:sizeA)
14       x(j, 1) = R(j,k);
15     end;
16     hlpA = norm(x);
17
18     if (x(k)>=0)
19       u = x + hlpA*z;
20     else
21       u = x - hlpA*z;
22     end
23
24     H = eye(sizeA) - 2*(u*u')/(u'*u);
25
26     R = H*R;
27     Q = Q*H;
28   end;
29 end
```

F QR ROZKLAD POMOCÍ GIVENSOVY MATICE

Výpis F.1: QR rozklad pomocí Givensovy matice

```
% QR rozklad matice Givensovou metodou
function [Q, R] = QR_Givens(A)
  sizeA = size(A)(1);
  R = A;
  for ij=[1:sizeA]
    for ii=[sizeA:-1:ij+1]
      % urcime c a s
      a = R(ii-1,ij)
      b = R(ii,ij)
      if (b==0)
        c = 1;
        s = 0;
      else
       if (abs(b)>abs(a))
         tau = -a/b;
         s = 1/sqrt(1+tau*tau);
         c = s*tau;
       else
         tau = -b/a;
         c = 1/sqrt(1+tau*tau);
         s = c*tau;
       end
      end
      % vynulujeme prvek ii, ij
      Gh = [c, s;-s, c];
      R(ii-1:ii,ij:sizeA) = Gh'*R(ii-1:ii,ij:sizeA)
    end
  end
  % dopocitame matice Q
  Q = A/R;
end
```

G QR ALGORITMUS PRO VÝPOČET VLASTNÍCH ČÍSEL A VEKTORŮ MATICE

Výpis G.1: QR algoritmus pro výpočet vlastních čísel a vektorů matice

```
1  % QR Eigen problem
2
3  function [eigenVectors, eigenValues, delta, iter] =
       QR_Eigen(A, my, err)
4    eigenVectors = eye(size(A));
5    eigenValues = A;
6    Cdig = eigenVectors;
7    Coth = abs(eigenVectors - 1);
8    iter = 0;
9
10   while true
11     if (my)
12       [Q,eigenValues] = QR_Householder(eigenValues);
13     else
14       [Q,eigenValues] = qr(eigenValues);
15     end
16
17     eigenValues = eigenValues*Q
18     eigenVectors = eigenVectors*Q;
19
20    delta = (sqrt(sum(sum(abs(eigenValues.*Coth).^2)))
         /max(max(abs(eigenValues(1,1)))));
21    if (delta<err)
22      break;
23    end
24
25    iter = iter + 1;
26   end
27 end
```

H VÝPIS SVA KÓDU HOUSEHOLDEROVOVA QR ROZKLADU MATICE ŘÁDU 2

Výpis H.1: Výpis SVA kódu QR rozkladu pomocí Householderových transformací

```
1  OPERATOR "=" @0 = _ONE;
2  OPERATOR "=" @1 = _ZERO;
3  OPERATOR "=" @2 = _ZERO;
4  OPERATOR "=" @3 = _ZERO;
5  OPERATOR "=" @4 = _ZERO;
6  OPERATOR "=" @5 = _ONE;
7  OPERATOR "=" @6 = _in00;
8  OPERATOR "=" @7 = _in01;
9  OPERATOR "=" @8 = _in01;
10 OPERATOR "=" @9 = _in11;
11 OPERATOR "=" @10 = @0;
12 OPERATOR "=" @11 = @3;
13 OPERATOR "=" @12 = @2;
14 OPERATOR "=" @13 = @5;
15 OPERATOR "=" @1 = _ONE;
16 OPERATOR "=" @14 = _ZERO;
17 OPERATOR "=" @15 = @6;
18 METHOD "abs" @16 = @6;
19 OPERATOR "*=" = @16, @16;
20 OPERATOR "+=" = @14, @16;
21 OPERATOR "=" @17 = @7;
22 METHOD "abs" @16 = @7;
23 OPERATOR "*=" = @16, @16;
24 OPERATOR "+=" = @14, @16;
25 METHOD "sqrt" @16 = @14;
26 OPERATOR "*" @18 = @1, @16;
27 OPERATOR "*" @19 = @4, @16;
28 METHOD "sign_nz" @16 = @15;
29 OPERATOR "*" @20 = @18, @16;
30 OPERATOR "*" @21 = @19, @16;
31 OPERATOR "+" @22 = @15, @20;
32 OPERATOR "+" @23 = @17, @21;
33 METHOD "conj" @24 = @22;
34 METHOD "conj" @25 = @23;
```

```
35 OPERATOR "*" @18 = @22, @24;
36 OPERATOR "*" @26 = @22, @25;
37 OPERATOR "*" @19 = @23, @24;
38 OPERATOR "*" @27 = @23, @25;
39 OPERATOR "+" @20 = @18, @18;
40 OPERATOR "+" @21 = @19, @19;
41 OPERATOR "+" @28 = @26, @26;
42 OPERATOR "+" @29 = @27, @27;
43 OPERATOR "*" @30 = @24, @22;
44 OPERATOR "*" @31 = @25, @23;
45 OPERATOR "+" @18 = @31, @30;
46 OPERATOR "/" @32 = @20, @18;
47 OPERATOR "/" @33 = @21, @18;
48 OPERATOR "/" @34 = @28, @18;
49 OPERATOR "/" @35 = @29, @18;
50 OPERATOR "-" @36 = @0, @32;
51 OPERATOR "-" @37 = @3, @33;
52 OPERATOR "-" @38 = @2, @34;
53 OPERATOR "-" @39 = @5, @35;
54 OPERATOR "*" @40 = @36, @6;
55 OPERATOR "*" @41 = @38, @7;
56 OPERATOR "+" @18 = @41, @40;
57 OPERATOR "*" @40 = @36, @8;
58 OPERATOR "*" @41 = @38, @9;
59 OPERATOR "+" @42 = @41, @40;
60 OPERATOR "*" @40 = @37, @6;
61 OPERATOR "*" @41 = @39, @7;
62 OPERATOR "+" @43 = @41, @40;
63 OPERATOR "*" @40 = @37, @8;
64 OPERATOR "*" @41 = @39, @9;
65 OPERATOR "+" @44 = @41, @40;
66 OPERATOR "=" @6 = @18;
67 OPERATOR "=" @7 = @43;
68 OPERATOR "=" @8 = @42;
69 OPERATOR "=" @9 = @44;
70 OPERATOR "*" @45 = @10, @36;
71 OPERATOR "*" @46 = @12, @37;
72 OPERATOR "+" @18 = @46, @45;
73 OPERATOR "*" @45 = @10, @38;
```

```
74 OPERATOR "*" @46 = @12, @39;
75 OPERATOR "+" @42 = @46, @45;
76 OPERATOR "*" @45 = @11, @36;
77 OPERATOR "*" @46 = @13, @37;
78 OPERATOR "+" @43 = @46, @45;
79 OPERATOR "*" @45 = @11, @38;
80 OPERATOR "*" @46 = @13, @39;
81 OPERATOR "+" @44 = @46, @45;
82 OPERATOR "=" @10 = @18;
83 OPERATOR "=" @11 = @43;
84 OPERATOR "=" @12 = @42;
85 OPERATOR "=" @13 = @44;
86 OPERATOR "=" _Q00 = @10;
87 OPERATOR "=" _Q10 = @11;
88 OPERATOR "=" _Q01 = @12;
89 OPERATOR "=" _Q11 = @13;
90 OPERATOR "=" _R00 = @6;
91 OPERATOR "=" _R10 = @7;
92 OPERATOR "=" _R01 = @8;
93 OPERATOR "=" _R11 = @9;
```

I VÝPIS SVA KÓDU QR ROZKLADU PO ZÁKLADNÍ OPTIMALIZACI

Výpis I.1: Výpis SVA kódu QR rozkladu po základní optimalizaci pro kompilovatelnost programem SVAToDSPCompiler

```
1  OPERATOR "=" @0 = _ONE;
2  OPERATOR "=" @1 = _ZERO;
3  OPERATOR "=" @2 = _ZERO;
4  OPERATOR "=" @3 = _ZERO;
5  OPERATOR "=" @4 = _ZERO;
6  OPERATOR "=" @5 = _ONE;
7  OPERATOR "=" @6 = _in00;
8  OPERATOR "=" @7 = _in01;
9  OPERATOR "=" @8 = _in01;
10 OPERATOR "=" @9 = _in11;
11 OPERATOR "=" @10 = @0;
12 OPERATOR "=" @11 = @3;
13 OPERATOR "=" @12 = @2;
14 OPERATOR "=" @13 = @5;
15 OPERATOR "=" @47 = _ONE;
16 OPERATOR "=" @14 = _ZERO;
17 OPERATOR "=" @15 = @6;
18 METHOD "abs" @16 = @6;
19 OPERATOR "*" @58 = @16, @16;
20 OPERATOR "+" @56 = @14, @58;
21 OPERATOR "=" @17 = @7;
22 METHOD "abs" @59 = @7;
23 OPERATOR "*" @60 = @59, @59;
24 OPERATOR "+" @57 = @56, @60;
25 METHOD "sqrt" @61 = @57;
26 OPERATOR "*" @18 = @47, @61;
27 OPERATOR "*" @19 = @4, @61;
28 METHOD "sign_nz" @62 = @15;
29 OPERATOR "*" @20 = @18, @62;
30 OPERATOR "*" @21 = @19, @62;
31 OPERATOR "+" @22 = @15, @20;
32 OPERATOR "+" @23 = @17, @21;
33 OPERATOR "=" @24 = @22;
```

```
34 OPERATOR "=" @25 = @23;
35 OPERATOR "*" @63 = @22, @24;
36 OPERATOR "*" @26 = @22, @25;
37 OPERATOR "*" @67 = @23, @24;
38 OPERATOR "*" @27 = @23, @25;
39 OPERATOR "+" @68 = @63, @63;
40 OPERATOR "+" @69 = @67, @67;
41 OPERATOR "+" @28 = @26, @26;
42 OPERATOR "+" @29 = @27, @27;
43 OPERATOR "*" @30 = @24, @22;
44 OPERATOR "*" @31 = @25, @23;
45 OPERATOR "+" @64 = @31, @30;
46 OPERATOR "/" @32 = @68, @64;
47 OPERATOR "/" @33 = @69, @64;
48 OPERATOR "/" @34 = @28, @64;
49 OPERATOR "/" @35 = @29, @64;
50 OPERATOR "-" @36 = @0, @32;
51 OPERATOR "-" @37 = @3, @33;
52 OPERATOR "-" @38 = @2, @34;
53 OPERATOR "-" @39 = @5, @35;
54 OPERATOR "*" @40 = @36, @6;
55 OPERATOR "*" @41 = @38, @7;
56 OPERATOR "+" @65 = @41, @40;
57 OPERATOR "*" @70 = @36, @8;
58 OPERATOR "*" @73 = @38, @9;
59 OPERATOR "+" @42 = @73, @70;
60 OPERATOR "*" @71 = @37, @6;
61 OPERATOR "*" @74 = @39, @7;
62 OPERATOR "+" @43 = @74, @71;
63 OPERATOR "*" @72 = @37, @8;
64 OPERATOR "*" @75 = @39, @9;
65 OPERATOR "+" @44 = @75, @72;
66 OPERATOR "=" @48 = @65;
67 OPERATOR "=" @49 = @43;
68 OPERATOR "=" @50 = @42;
69 OPERATOR "=" @51 = @44;
70 OPERATOR "*" @45 = @10, @36;
71 OPERATOR "*" @46 = @12, @37;
72 OPERATOR "+" @66 = @46, @45;
```

```
73 OPERATOR "*" @79 = @10, @38;
74 OPERATOR "*" @82 = @12, @39;
75 OPERATOR "+" @76 = @82, @79;
76 OPERATOR "*" @80 = @11, @36;
77 OPERATOR "*" @83 = @13, @37;
78 OPERATOR "+" @77 = @83, @80;
79 OPERATOR "*" @81 = @11, @38;
80 OPERATOR "*" @84 = @13, @39;
81 OPERATOR "+" @78 = @84, @81;
82 OPERATOR "=" @52 = @66;
83 OPERATOR "=" @53 = @77;
84 OPERATOR "=" @54 = @76;
85 OPERATOR "=" @55 = @78;
86 OPERATOR "=" _Q00 = @52;
87 OPERATOR "=" _Q10 = @53;
88 OPERATOR "=" _Q01 = @54;
89 OPERATOR "=" _Q11 = @55;
90 OPERATOR "=" _R00 = @48;
91 OPERATOR "=" _R10 = @49;
92 OPERATOR "=" _R01 = @50;
93 OPERATOR "=" _R11 = @51;
```

J VÝPIS SVA KÓDU QR ROZKLADU PO ZÁKLADNÍ A REDUKUJÍCÍ OPTIMALIZACI

Výpis J.1: Výpis SVA kódu QR rozkladu po základní a redukující optimalizaci provedené programem SVAOptimalizer

```
1  METHOD "abs" @16 = _in00;
2  OPERATOR "*" @58 = @16, @16;
3  METHOD "abs" @59 = _in01;
4  OPERATOR "*" @60 = @59, @59;
5  OPERATOR "+" @57 = @58, @60;
6  METHOD "sqrt" @61 = @57;
7  METHOD "sign_nz" @62 = _in00;
8  OPERATOR "*" @20 = @61, @62;
9  OPERATOR "+" @22 = _in00, @20;
10 OPERATOR "*" @63 = @22, @22;
11 OPERATOR "*" @26 = @22, _in01;
12 OPERATOR "*" @67 = _in01, @22;
13 OPERATOR "*" @27 = _in01, _in01;
14 OPERATOR "+" @68 = @63, @63;
15 OPERATOR "+" @69 = @67, @67;
16 OPERATOR "+" @28 = @26, @26;
17 OPERATOR "+" @29 = @27, @27;
18 OPERATOR "*" @30 = @22, @22;
19 OPERATOR "*" @31 = _in01, _in01;
20 OPERATOR "+" @64 = @31, @30;
21 OPERATOR "/" @32 = @68, @64;
22 OPERATOR "/" @33 = @69, @64;
23 OPERATOR "/" @34 = @28, @64;
24 OPERATOR "/" @35 = @29, @64;
25 OPERATOR "-" _Q00 = _ONE, @32;
26 OPERATOR "-" _Q10 = _ZERO, @33;
27 OPERATOR "-" _Q01 = _ZERO, @34;
28 OPERATOR "-" _Q11 = _ONE, @35;
29 OPERATOR "*" @40 = _Q00, _in00;
30 OPERATOR "*" @41 = _Q01, _in01;
31 OPERATOR "+" _R00 = @41, @40;
```

```
32 OPERATOR "*" @70 = _Q00, _in01;
33 OPERATOR "*" @73 = _Q01, _in11;
34 OPERATOR "+" _R01 = @73, @70;
35 OPERATOR "*" @71 = _Q10, _in00;
36 OPERATOR "*" @74 = _Q11, _in01;
37 OPERATOR "+" _R10 = @74, @71;
38 OPERATOR "*" @72 = _Q10, _in01;
39 OPERATOR "*" @75 = _Q11, _in11;
40 OPERATOR "+" _R11 = @75, @72;
```

K VÝZNAM PORTŮ IP BLOKU

Tab. K.1: Význam portů IP bloku Xilinx® Floating-Point Operator

Jméno	**Port**	**Volitelný**	**Popis**
aclk	vstupní	ne	hodinový signál, obvod reaguje na náběžnou hranu
aclken	vstupní	ano	aktivní v logické jedničce, povoluje hodinový signál
aresetn	vstupní	ano	aktivní v logické nule, synchronní resetování, vždy má prioritu před signálem aclken, musí být aktivní alespoň po dobu dvou náběžných hran
s_axis_a_tvalid	vstupní	ano	TVALID pro kanál A
s_axis_a_tready	vstupní	ne	TREADY pro kanál A
s_axis_a_tdata	vstupní	ano	TDATA pro kanál A
s_axis_a_tuser	vstupní	ne	TUSER pro kanál A
s_axis_a_tlast	vstupní	ne	TLAST pro kanál A
s_axis_b_tvalid	vstupní	ne	TVALID pro kanál B
s_axis_b_tready	vstupní	ne	TREADY pro kanál B
s_axis_b_tdata	vstupní	ne	TDATA pro kanál B
s_axis_b_tuser	vstupní	ne	TUSER pro kanál B
s_axis_b_tlast	vstupní	ne	TLAST pro kanál B
s_axis_c_tvalid	vstupní	ne	TVALID pro kanál C
s_axis_c_tready	vstupní	ne	TREADY pro kanál C
s_axis_c_tdata	vstupní	ne	TDATA pro kanál C
s_axis_c_tuser	vstupní	ne	TUSER pro kanál C
s_axis_c_tlast	vstupní	ne	TLAST pro kanál C
s_axis_operation_tvalid	vstupní	ne	TVALID pro kanál OPERATION
s_axis_operation_tready	vstupní	ne	TREADY pro kanál OPERATION
s_axis_operation_tdata	vstupní	ne	TDATA pro kanál OPERATION

s_axis_operation_tuser	vstupní	ne	TUSER pro kanál OPERATION
s_axis_operation_tlast	vstupní	ne	TLAST pro kanál OPERATION
m_axis_result_tvalid	výstupní	ano	TVALID pro kanál C
m_axis_result_tready	vstupní	ne	TREADY pro kanál C
m_axis_result_tdata	výstupní	ano	TDATA pro kanál C
m_axis_result_tuser	výstupní	ne	TUSER pro kanál C
m_axis_result_tlast	výstupní	ne	TLAST pro kanál C

Vlastní překlad z dokumentace IP bloku Xilinx® Floating-Point Operator[16].

L ROZHRANÍ BLOKŮ PRO VÝPOČTY S ČÍSLY S PLOVOUCÍ DESETINNOU ČÁRKOU

Výpis L.1: Soubor FP.vhd

```
1  ----------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 10/08/2015 08:05:17 AM
6  -- Design Name:
7  -- Module Name: FP - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24 use IEEE.NUMERIC_STD.ALL;
25
26 library work;
27 use work.FP_SINGLE.ALL;
28
29 -- Uncomment the following library declaration if
      instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
```

```
32 --use UNISIM.VComponents.all;
33
34 package FP is
35
36     subtype FP_NUMBER is FP_SINGLE_NUMBER;
37
38     function negativeNumber(pos: FP_NUMBER) return
       FP_NUMBER;
39
40     constant FP_PLUS : STD_LOGIC := '0';
41     constant FP_MINUS : STD_LOGIC := '1';
42
43     subtype FP_CMP_OP is STD_LOGIC_VECTOR(3 downto 0)
       ;
44
45     constant FP_CMP_OP_Unordered : FP_CMP_OP := "0001
       ";
46     constant FP_CMP_OP_LessThan : FP_CMP_OP := "0011"
       ;
47     constant FP_CMP_OP_Equal : FP_CMP_OP := "0101";
48     constant FP_CMP_OP_LessThanEqual : FP_CMP_OP := "
       0111";
49     constant FP_CMP_OP_GreaterThan : FP_CMP_OP := "
       1001";
50     constant FP_CMP_OP_NotEqual : FP_CMP_OP := "1011"
       ;
51     constant FP_CMP_OP_GreaterThanEqual : FP_CMP_OP
       := "1101";
52
53     component FP_ADD is
54         Port ( clk : in STD_LOGIC;
55                clk_en : in STD_LOGIC;
56                reset : in STD_LOGIC;
57                A : in FP_NUMBER;
58                B : in FP_NUMBER;
59                AB_ready: out STD_LOGIC;
60                AB_ok: in STD_LOGIC;
61                Y : out FP_NUMBER;
62                Y_ready: in STD_LOGIC;
```

```
63                Y_ok : out STD_LOGIC;
64                fail : out STD_LOGIC);
65      end component;
66
67      component FP_ADDSUB is
68          Port ( clk : in STD_LOGIC;
69                 clk_en : in STD_LOGIC;
70                 reset : in STD_LOGIC;
71                 sub : in STD_LOGIC;
72                 A : in FP_NUMBER;
73                 B : in FP_NUMBER;
74                 AB_ready: out STD_LOGIC;
75                 AB_ok: in STD_LOGIC;
76                 Y : out FP_NUMBER;
77                 Y_ready: in STD_LOGIC;
78                 Y_ok : out STD_LOGIC;
79                 fail : out STD_LOGIC);
80      end component;
81
82      component FP_MUL is
83          Port ( clk : in STD_LOGIC;
84                 clk_en : in STD_LOGIC;
85                 reset : in STD_LOGIC;
86                 A : in FP_NUMBER;
87                 B : in FP_NUMBER;
88                 AB_ready: out STD_LOGIC;
89                 AB_ok: in STD_LOGIC;
90                 Y : out FP_NUMBER;
91                 Y_ready: in STD_LOGIC;
92                 Y_ok : out STD_LOGIC;
93                 fail : out STD_LOGIC);
94      end component;
95
96      component FP_DIV is
97          Port ( clk : in STD_LOGIC;
98                 clk_en : in STD_LOGIC;
99                 reset : in STD_LOGIC;
100                A : in FP_NUMBER;
101                B : in FP_NUMBER;
```

```
102                 AB_ready: out STD_LOGIC;
103                 AB_ok: in STD_LOGIC;
104                 Y : out FP_NUMBER;
105                 Y_ready: in STD_LOGIC;
106                 Y_ok : out STD_LOGIC;
107                 fail : out STD_LOGIC);
108     end component;
109 
110     component FP_SQRT is
111         Port ( clk : in STD_LOGIC;
112                 clk_en : in STD_LOGIC;
113                 reset : in STD_LOGIC;
114                 A : in FP_NUMBER;
115                 A_ready: out STD_LOGIC;
116                 A_ok: in STD_LOGIC;
117                 Y : out FP_NUMBER;
118                 Y_ready: in STD_LOGIC;
119                 Y_ok : out STD_LOGIC;
120                 fail : out STD_LOGIC);
121     end component;
122 
123     component FP_CMP is
124         Port ( clk : in STD_LOGIC;
125                 clk_en : in STD_LOGIC;
126                 reset : in STD_LOGIC;
127                 cmp_op : in FP_CMP_OP;
128                 A : in FP_NUMBER;
129                 B : in FP_NUMBER;
130                 AB_ready: out STD_LOGIC;
131                 AB_ok: in STD_LOGIC;
132                 CMP : out STD_LOGIC;
133                 CMP_ready: in STD_LOGIC;
134                 CMP_ok : out STD_LOGIC;
135                 fail : out STD_LOGIC);
136     end component;
137 
138     component FP_ABS is
139         Port ( A : in FP_NUMBER;
140                 A_ready: out STD_LOGIC;
```

```
141                 A_ok: in STD_LOGIC;
142                 Y : out FP_NUMBER;
143                 Y_ready: in STD_LOGIC;
144                 Y_ok : out STD_LOGIC);
145     end component;
146 
147     component FP_CHECK is
148         Port ( clk : in STD_LOGIC;
149                clk_en : in STD_LOGIC;
150                reset : in STD_LOGIC;
151                A : in FP_NUMBER; -- value to be
                      checked
152                A_ready: out STD_LOGIC;
153                A_ok: in STD_LOGIC;
154                V : in FP_NUMBER; -- valid value
155                ERR : in FP_NUMBER; -- error, greater
                      than zero
156                VERR_ready: out STD_LOGIC;
157                VERR_ok: in STD_LOGIC;
158                CHECK : out STD_LOGIC;
159                CHECK_ready: in STD_LOGIC;
160                CHECK_ok : out STD_LOGIC;
161                fail : out STD_LOGIC);
162     end component;
163 
164     -- no IP core
165     component FP_NOTZERO is
166         Port ( clk : in STD_LOGIC;
167                clk_en : in STD_LOGIC;
168                reset : in STD_LOGIC;
169                A : in FP_NUMBER;
170                B : in FP_NUMBER;
171                AB_ready: out STD_LOGIC;
172                AB_ok: in STD_LOGIC;
173                Y : out FP_NUMBER;
174                Y_ready: in STD_LOGIC;
175                Y_ok : out STD_LOGIC;
176                fail : out STD_LOGIC);
177     end component;
```

```
178
179     -- no IP core
180     component FP_SIGNNOTZERO is
181         Port ( clk : in STD_LOGIC;
182                clk_en : in STD_LOGIC;
183                reset : in STD_LOGIC;
184                A : in FP_NUMBER;
185                A_ready: out STD_LOGIC;
186                A_ok: in STD_LOGIC;
187                Y : out FP_NUMBER;
188                Y_ready: in STD_LOGIC;
189                Y_ok : out STD_LOGIC;
190                fail : out STD_LOGIC);
191     end component;
192
193     -- no IP core
194     component FP_ASSIGN is
195         Port ( clk : in STD_LOGIC;
196                clk_en : in STD_LOGIC;
197                reset : in STD_LOGIC;
198                A : in FP_NUMBER;
199                A_ready: out STD_LOGIC;
200                A_ok: in STD_LOGIC;
201                Y : out FP_NUMBER;
202                Y_ready: in STD_LOGIC;
203                Y_ok : out STD_LOGIC;
204                fail : out STD_LOGIC);
205     end component;
206
207     -- no IP core
208     component FP_DEVABS is
209         Port ( clk : in STD_LOGIC;
210                clk_en : in STD_LOGIC;
211                reset : in STD_LOGIC;
212                A : in FP_NUMBER;
213                A_ready: out STD_LOGIC;
214                A_ok: in STD_LOGIC;
215                Y : out FP_NUMBER;
216                Y_ready: in STD_LOGIC;
```

```
217                 Y_ok : out STD_LOGIC;
218                 fail : out STD_LOGIC);
219     end component;
220
221     -- no IP core
222     component FP_BUF is
223         Generic (   GN_bits : positive := 3;
224                     GN_numbers : positive := 8);
225         Port ( clk : in STD_LOGIC;
226                clk_en : in STD_LOGIC;
227                reset : in STD_LOGIC;
228                A : in FP_NUMBER;
229                A_ready: out STD_LOGIC;
230                A_ok: in STD_LOGIC;
231                A_fail: in STD_LOGIC;
232                Y : out FP_NUMBER;
233                Y_ready: in STD_LOGIC;
234                Y_ok : out STD_LOGIC;
235                Y_fail : out STD_LOGIC);
236     end component;
237
238     component FP_ADDSUB_NB is
239         Port ( clk : in STD_LOGIC;
240                clk_en : in STD_LOGIC;
241                sub : in STD_LOGIC;
242                A : in FP_NUMBER;
243                B : in FP_NUMBER;
244                AB_ok: in STD_LOGIC;
245                Y : out FP_NUMBER;
246                Y_ok : out STD_LOGIC);
247     end component;
248
249     component FP_MUL_NB is
250         Port ( clk : in STD_LOGIC;
251                clk_en : in STD_LOGIC;
252                A : in FP_NUMBER;
253                B : in FP_NUMBER;
254                AB_ok: in STD_LOGIC;
255                Y : out FP_NUMBER;
```

```
256                 Y_ok : out STD_LOGIC);
257     end component;
258 
259     component FP_DIV_NB is
260         Port ( clk : in STD_LOGIC;
261                clk_en : in STD_LOGIC;
262                A : in FP_NUMBER;
263                B : in FP_NUMBER;
264                AB_ok: in STD_LOGIC;
265                Y : out FP_NUMBER;
266                Y_ok : out STD_LOGIC);
267     end component;
268 
269     component FP_SQRT_NB is
270         Port ( clk : in STD_LOGIC;
271                clk_en : in STD_LOGIC;
272                A : in FP_NUMBER;
273                A_ok: in STD_LOGIC;
274                Y : out FP_NUMBER;
275                Y_ok : out STD_LOGIC);
276     end component;
277 
278 end FP;
279 
280 package body FP is
281     function negativeNumber(pos: FP_NUMBER) return
        FP_NUMBER is
282         variable neg : FP_NUMBER;
283         begin
284             neg := pos;
285             neg(pos'high) := NOT pos(pos'high);
286             return neg;
287         end negativeNumber; -- end function
288 end package body;
```

M DEFINICE ROZHRANÍ BLOKU DSP_PIPELINERAM

Výpis M.1: Definice rozhraní bloku DSP_pipelineRAM

```
1  component DSP_pipelineRAM is
2      generic (
3              GNinputs : positive := 8;
4              GNfirst_input : integer := 0;
5              GNoutputs : positive := 8;
6              GNfirst_output : integer := 8;
7              GNoperation_bits : positive := 3;
8              GNaddress_bits : positive := 8;
9              GNinstruction_bits : positive := 16;
10             GNlatency : positive := 2;
11             GNlatency_bits : positive := 2;
12             GNregisters : positive := 128;
13             GNregister_bits : positive := 7;
14             GNwatchdog_limit : positive := 768;
15             GNwatchdog_bits : positive := 10);
16     Port (  clk : in STD_LOGIC;
17             clk_en : in STD_LOGIC;
18             reset : in STD_LOGIC;
19             X : in DSP_NUMBERS (0 to GNinputs-1); --
                  input data vector
20             X_ok: in STD_LOGIC; -- valid intput data,
                   calculate can start
21             X_saved: out STD_LOGIC; -- accept intput
                  data signal
22             Y : out DSP_NUMBERS (0 to GNoutputs-1);
                  -- output data vector
23             Y_ok : out STD_LOGIC; -- output data ok
                  signal
24             fail: out STD_LOGIC; -- calculation fail
                  signal
25             -- WARNING: follows out signals are not
                  synchronized with rissing egle of
                  clock signal!
26             addressStart: in STD_LOGIC_VECTOR (
```

```
                GNaddress_bits-1 downto 0); -- addres
                of first instruction of calculation
27              address: out STD_LOGIC_VECTOR (
                GNaddress_bits-1 downto 0); -- addres
                of next instruction
28              instruction: in STD_LOGIC_VECTOR (
                GNinstruction_bits-1 downto 0); --
                instruction input (from memory)
29              -- ram/register signals
30              regA_addr: out STD_LOGIC_VECTOR (
                GNregister_bits-1 downto 0);
31              regB_addr: out STD_LOGIC_VECTOR (
                GNregister_bits-1 downto 0);
32              regA_read: in FP_NUMBER;
33              regB_read: in FP_NUMBER;
34              regA_we: out STD_LOGIC;
35              regA_write: out FP_NUMBER);
36  end component;
```

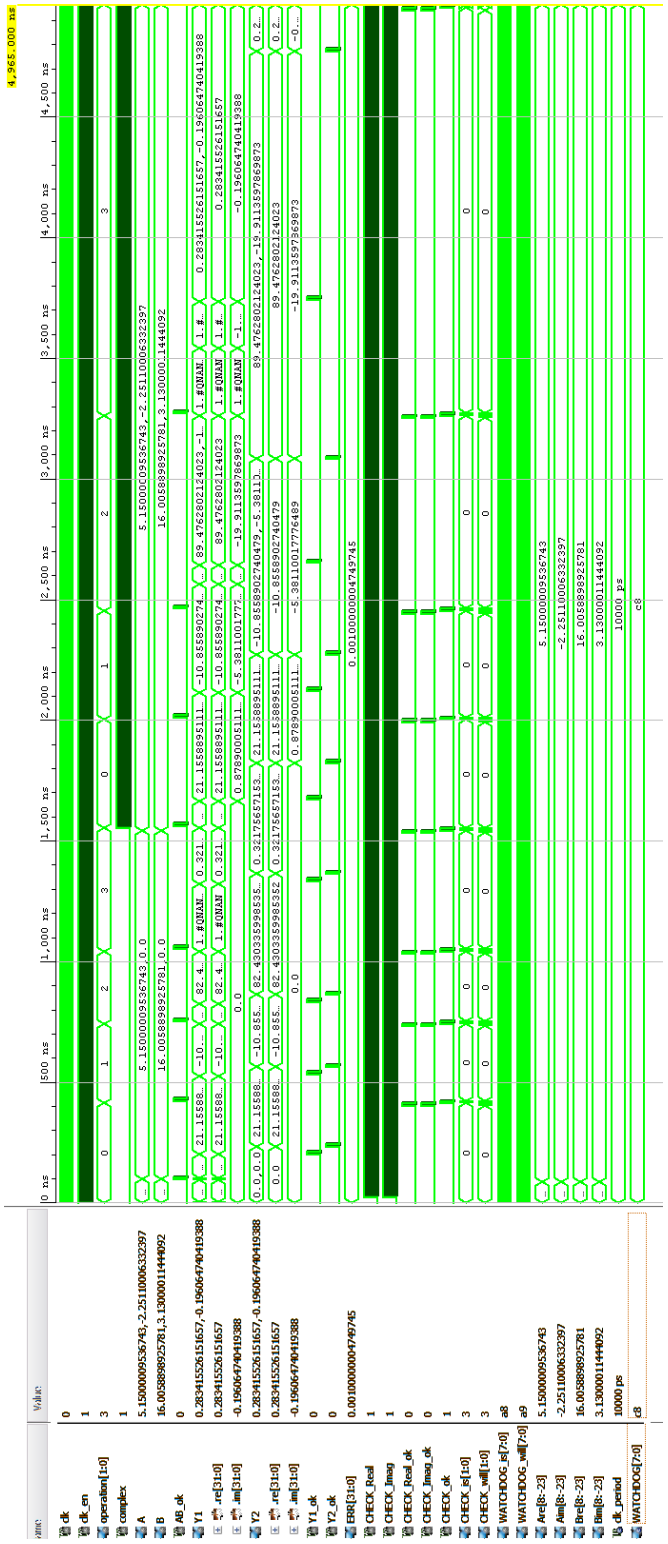
N DEFINICE ROZHRANÍ BLOKU DSP_COREPIPELINERAM

Výpis N.1: Definice rozhraní bloku DSP_corePipelineRAM

```
1  component DSP_corePipelineRAM is
2      generic (
3              GNinputs : positive := 8;
4              GNfirst_input : integer := 0;
5              GNoutputs : positive := 8;
6              GNfirst_output : integer := 8;
7              GNoperation_bits : positive := 2;
8              GNaddress_bits : positive := 5;
9              GNinstruction_bits : positive := 5;
10             GNlatency : positive := 1;
11             GNlatency_bits : positive := 1;
12             GNregisters : positive := 128;
13             GNregister_bits : positive := 7;
14             GNdevices : positive := 4;
15             GNwatchdog_limit : positive := 768;
16             GNwatchdog_bits : positive := 10);
17     Port (  clk : in STD_LOGIC;
18             clk_en : in STD_LOGIC;
19             reset : in STD_LOGIC;
20             X : in DSP_NUMBERS (0 to GNinputs-1); --
                  input data vector
21             X_ok: in STD_LOGIC; -- valid intput data,
                   calculate can start
22             X_saved: out STD_LOGIC; -- accept intput
                  data signal
23             Y : out DSP_NUMBERS (0 to GNoutputs-1);
                  -- output data vector
24             Y_ok : out STD_LOGIC; -- output data ok
                  signal
25             fail: out STD_LOGIC; -- calculation fail
                  signal
26             -- WARNING: follows out signals are not
                  synchronized with rissing egle of
                  clock signal!
```

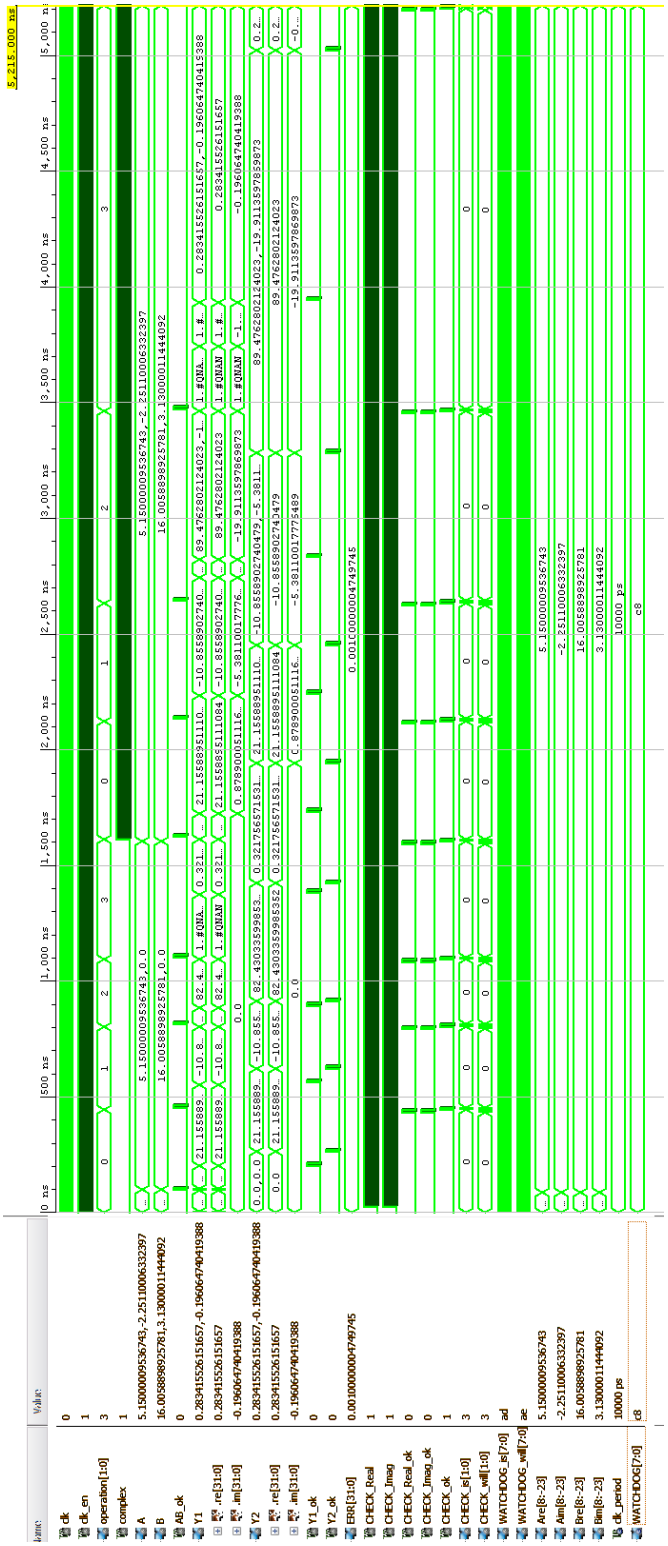
```
27              addressStart: in STD_LOGIC_VECTOR (
                   GNaddress_bits-1 downto 0); -- addres
                   of first instruction of calculation
28              address: out STD_LOGIC_VECTOR (
                   GNaddress_bits-1 downto 0); -- addres
                   of next instruction
29              instruction: in STD_LOGIC_VECTOR (
                   GNinstruction_bits-1 downto 0); --
                   instruction input (from memory)
30              -- ram/register signals
31              regA_addr: out STD_LOGIC_VECTOR (
                   GNregister_bits-1 downto 0);
32              regB_addr: out STD_LOGIC_VECTOR (
                   GNregister_bits-1 downto 0);
33              regA_read: in FP_NUMBER;
34              regB_read: in FP_NUMBER;
35              regA_we: out STD_LOGIC;
36              regA_write: out FP_NUMBER;
37              -- devices signals
38              dev_in_ready: in STD_LOGIC_VECTOR(
                   GNdevices-1 downto 0); -- signals for
                   connect devices like multipler and etc
39              dev_in: out STD_LOGIC_VECTOR(GNdevices-1
                   downto 0);
40              dev_A: out FP_NUMBER;
41              dev_B: out FP_NUMBER;
42              dev_out_ready: out STD_LOGIC_VECTOR(
                   GNdevices-1 downto 0);
43              dev_out: in STD_LOGIC_VECTOR(GNdevices-1
                   downto 0);
44              dev_Y: in DSP_NUMBERS (0 to GNdevices-1);
45              dev_fail: in STD_LOGIC_VECTOR(GNdevices-1
                    downto 0));
46  end component;
```

O SIMULACE APU_NB A APU_NB_SMALL



Obr. O.1: Simulace `APU_NB` a `APU_NB_small`

P SIMULACE APU_SMALL



Obr. P.1: Simulace `APU_small`

Q OBSAH PŘILOŽENÉHO CD

- `/` ..... kořenový adresář CD
  - `COE_SVA` . ...složka COE souborů pro inicializaci pamětí a jim odpovídajících SVA kódů
  - `COE_SVA` ... složka SVA kódů vygenerovaných programem AnalyzeAlgorithm
  - `Zynq7020`
    - `c`
      - `XilinxSDK.zip` ..... archiv projektu a dat pro Xilinx SDK
    - `vhdl`
      - `SVD.xpr.zip` ..... archiv projektu pro Xilinx Vivado
  - `dependencies` ..zdrojové kódy závislostí programů vyvinutých v rámci práce
  - `pdf`
    - `xstrym00_diplomka.pdf` ..... pdf verze práce
  - `sw` ..... zdrojové kódy programů vyvinutých v rámci práce
  - `CTIME` ..... soubor s popisem obsahu CD