ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

DIPLOMOVÁ PRÁCE

Implementace prediktivního řízení v obvodech programovatelné logiky

Bc. Jakub Jeníček **2018**

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jakub JENÍČEK**
Osobní číslo:	**E15N0101P**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Elektronika a aplikovaná informatika**
Název tématu:	**Implementace prediktivního řízení v obvodech programovatelné logiky**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

1. Shrňte základní metody prediktivního řízení a jejich možnosti při využití v řízení pohonu.
2. Implementujte metodu prediktivního řízení ADMM v obvodu FPGA dle doporučené literatury. Během implementace využívejte moderní trendy a nástroje pro návrh číslicových systémů.
3. Porovnejte výsledky implementace se simulací v programu Matlab.
4. Demonstrujte činnost a časové parametry výsledné implementace v reálné aplikaci řízení pohonu.
5. Celý proces návrhu detailně popište.

Rozsah grafických prací: **podle doporučení vedoucího**

Rozsah kvalifikační práce: **40 - 60 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **JEREZ, Juan Luis, et al.: Embedded online optimization for model predictive control at megahertz rates. Automatic Control, IEEE Transactions on, 2014, 59.12: 3238-3251**
2. **PINKER, Jiří; POUPA, Martin: Číslicové systémy a jazyk VHDL. BEN-technická literatura, 2006**
3. **SINGH, Desh; ENGINEER, Supervising Principal. Higher level programming abstractions for fpgas using opencl. In: Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing. 2011**
4. **GEYER, Tobias, et al.: Model predictive pulse pattern control. Industry Applications, IEEE Transactions on, 2012, 48.2: 663-676**

Vedoucí diplomové práce: **Ing. Petr Burian, Ph.D.**
Regionální inovační centrum elektrotechniky

Datum zadání diplomové práce: **10. října 2017**

Termín odevzdání diplomové práce: **24. května 2018**

Doc. Ing. Jiří Hammerbauer, Ph.D.
děkan

L.S.

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 10. října 2017

Anotace

Předkládaná diplomová práce je zaměřena na implementaci specifického příkladu metody prediktivního řízení ADMM do obvodů programovatelné logiky. Součástí práce je popis chování implementovaného systému a jeho částí. Dále jsou zde popisovány inicializační a konfigurační konstanty, které zajišťují generický návrh implementace.

Výstupem této práce je funkční návrh prototypu metody prediktivního řízení ADMM, který je určen k následným experimentům. V práci jsou dále uvedeny výsledky simulací při změnách některých parametrů a jejich dopad na syntézu.

Klíčová slova

Prediktivní řízení, ADMM, návrh implementace, paralelní zpracování, FPGA, simulace, VHDL.

Abstract

This thesis is focused on the implementation of a specific example of ADMM predictive control method in the circuits of programmable logic. Part of the thesis is a description of the behavior of the implemented system and its parts. In addition, initialization and configuration constants are provided, which provides a generic design implementation.

The output of this thesis is a functional design of the ADMM predictive method, which is intended for subsequent experiments. The thesis also presents the results of the simulations of changes in some parameters and their impact on synthesis.

Key words

Predictive control, ADMM, desing of implementation, parallel processing, FPGA, simulation, VHDL.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

Dále prohlašuji, že veškerý software, použitý při řešení této diplomové práce, je legální.

.....

podpis

V Plzni dne 24.5.2018 Bc. Jakub Jeníček

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Burianovi, Ph.D. a konzultantovi doc. Ing. Václavu Šmídlovi, Ph.D. za jejich cenné profesionální rady a za to, že jsem díky nim mohl přispět, prostřednictvím této práce, k výzkumu metod prediktivního řízení.

Obsah

OBSAH **7**

ÚVOD **8**

SEZNAM ZKRATEK **9**

1 ZÁKLADNÍ METODY PREDIKTIVNÍHO ŘÍZENÍ **10**

1.1 DĚLENÍ PREDIKTIVNÍCH ALGORITMŮ ŘÍZENÍ 10
1.2 METODA ADMM 11
1.3 POPIS SIMULACE ADMM V PROGRAMU MATLAB 12

2 NÁVRH IMPLEMENTACE ALGORITMU **14**

2.1 DEFINICE INICIALIZAČNÍCH KONSTANT A PARAMETRŮ 14
2.2 POPIS CHOVÁNÍ HLAVNÍ ENTITY 18
2.3 POPIS CHOVÁNÍ JEDNOTLIVÝCH POD-KOMPONENT 22
2.3.1 Pod-komponenta cast_Y *23*
2.3.2 Pod-komponenta cast_1 ..... *26*
2.3.3 Pod-komponenta cast_2 ..... *28*
2.3.4 Pod-komponenta cast_3 ..... *34*

3 SIMULACE A SYNTÉZA VÝSLEDNÉ IMPLEMENTACE **37**

3.1 KOMPONENTA NEW_COMPONENT 37
3.2 POROVNÁNÍ IMPLEMENTACE SE SIMULACÍ V PROGRAMU MATLAB 41
3.3 VÝSLEDKY SYNTÉZY 46

ZÁVĚR **49**

POUŽITÁ LITERATURA **50**

PŘÍLOHA I. ÚRYVEK ZE SIMULACE V PROGRAMU MATLAB (ADMM) **1**

PŘÍLOHA II. ÚRYVEK ZE SIMULACE V PROGRAMU MATLAB (HUANG) **2**

Úvod

Předkládaná práce je zaměřena na implementaci prediktivního algoritmu ADMM do obvodu programovatelné logiky při respektování moderních trendů návrhu a nástrojů pro návrh číslicových systémů. Tato implementace bude dále využívána pro výzkumné účely, a proto je práce především zaměřena na popis chování jednotlivých komponent, ze kterých se výsledná implementace skládá.

Text je rozdělen do tří částí; první část se zabývá obecným popisem prediktivních algoritmů s příkladem jejich dělení. V této části byl dále kladen větší důraz na obecný popis funkce prediktivního algoritmu ADMM a simulace v programu Matlab ve které byl tento algoritmus realizován. Tato simulace sloužila jako reference pro následný návrh v obvodech programovatelné logiky.

Druhá část této práce je zaměřena na popis návrhu implementace ADMM do FPGA. Pro návrh byl použit program Quartus Prime Lite Edition verze 17.0.2 Build 602 a pro vlastní návrh byl použit jazyk VHDL ve verzi VHDL-1993. Protože výsledná implementace má sloužit pro výzkumné účely, tak je zde kladen důraz na popis a vizualizaci chování jednotlivých komponent, vysvětlení konfiguračních konstant a popis pro vytvoření inicializačních konstant.

Poslední třetí část tohoto textu je věnována prezentaci výstupních dat simulace a jejich porovnání se simulací v programu Matlab. Pro simulaci byl použit program ModelSim – Intel FPGA Starter Edition verze 10.5b. Dále je zde popsána komponenta, která je schopna propojit navržený algoritmus v FPGA s procesorem pomocí rozhraní Avalon. V této části jsou také představeny dopady na výsledky syntézy při změnách některých parametrů. [1]

Seznam zkratek

Zkratka	Význam zkratky
ADMM	Metoda střídavého multiplikátoru (Alternating Direction Method of Multipliers)
ALM	Adaptivní logický modul (Adaptive Logic Module)
ASP	Adaptivní spínání vzorů (Adaptive Switching Pattern)
DMA	Přímý přístup do paměti (Direct Memory Access)
DMTC	Přímá regulace středního točivého momentu (Direct Mean Torque Control)
DSC	Přímé řízení (Direct Self Control)
DSPC	Přímé řízení rychlosti (Direct Speed Control)
FGM	Rychlá gradientní metoda (Fast Gradient Method)
FPGA	Programovatelné logické pole (angl. Field Programmable Gate Array)
MPC	Řízení založené na modelu (Model Predictive Control)
PROMC	Řízení pomocí paměti ROM (Programmable Read Only Memory Control)

1 Základní metody prediktivního řízení

Prediktivní řízení je možné využít v mnoha odvětví jako například v chemickém průmyslu, bankovnictví (pohyb na burze), řízení motorů a další. Metod pro realizaci prediktivního řízení je velké množství, ale některé rysy mají tyto metody společné: [2] [3]

- Obsahují matematický model pro predikci budoucího akčního zásahu
- Požadovaná hodnota řízené veličiny je předem známa
- Pro výpočet akčního zásahu je použita minimalizace účelové funkce

Na základě známého matematického modelu systému lze vypočítat posloupnost akčních zásahů na predikčním horizontu při známé požadované hodnotě řízené veličiny. Pro popis matematického modelu systému lze použít libovolný model, který dostatečně vystihne dynamické vlastnosti řízeného systému. Mezi používané popisy matematického modelu například patří impulzní funkce, přenosová funkce, stavový popis, účelová funkce a další. [3]

K výpočtu posloupnosti akčních zásahů je třeba vyřešit optimalizační problém, který se skládá z omezení vstupních a výstupních veličin a vhodné účelové funkci. Účelová funkce obsahuje budoucí hodnoty výstupu, žádané veličiny a akční zásahy. Z výsledné posloupnosti akčních zásahů je pro řízení použit jen první člen vypočítán v předchozím kroku. Tento postup se opakuje při každé periodě vzorkování což je nazýváno jako strategie pohyblivého horizontu. [3]

Řešení optimalizačního problému může být výpočetně velmi náročné, pro složité matematické modely což přináší problém při řízení systémů, které musí být řízeny v krátkém čase, jako například řízení motorů. Při použití takto výpočetně náročného algoritmu to může vést k zvětšení vzorkovací periody, aby bylo možné algoritmus vůbec použít. Pro eliminaci tohoto nedostatku je možné použít metody, které jsou navrženy tak, aby bylo možné využít paralelních výpočtů. Tyto algoritmy se hodí pro realizaci v obvodech programovatelné logiky (FPGA), díky možnosti paralelního zpracování dat. [4]

1.1 Dělení prediktivních algoritmů řízení

Pro dělení prediktivních algoritmů můžeme jednotlivé algoritmy rozdělit například z pohledu funkčního principu, nebo z délky predikčního horizontu. Při použití dělení z hlediska základního funkčního principu, můžeme prediktivní algoritmy rozdělit na tři skupiny. [2] [5]

Princip první skupiny je založen na hysterezi, kdy se algoritmus snaží udržet řízenou veličinu v určitých mezích respektive oblasti. Udržení ve vymezené oblasti je dosaženo pomocí predikce chování řízené veličiny v budoucnu a vyřešením optimalizačního problému

pro akční zásah tak, aby řízená veličiny ve vymezené oblasti zůstala co nejdéle. Zástupci této kategorie jsou ASP, PROMC pro řízení proudu a PROMC pro řízení napětí. [2] [5]

Další skupinou prediktivních algoritmů řízení je řízení založené na trajektorii. Principem tohoto typu algoritmů je přinutit řízený systém aby se držela předem spočítaných trajektorií. Jakmile je systém v jedné z těchto trajektorií, je zde držen do doby, než přijde zásah zvenčí. Mezi zástupce těchto algoritmů můžeme uvést například DSC, DSPC, DMTC atd. [2]

Třetí skupinou algoritmů z hlediska principu je založena na modelu (MPC). Jak je vidět už z názvu skupiny, je součástí těchto algoritmů podrobný matematický model systému. Díky podrobnému matematickému modelu je možné přesněji předpovídat chování systému a z tohoto důvodu i efektivněji tento systém řídit. Mezi zástupce této skupiny můžeme zařadit algoritmy FGM a ADMM. [2] [4]

Poslední dva jmenované algoritmy FGM a ADMM jsou výpočetně náročné při implementaci do běžného mikro kontroléru. Protože tyto algoritmy obsahují maticové a vektorové operace, které je nutné provádět v každé iteraci algoritmu tak, jsou tyto algoritmy vhodné pro realizaci v FPGA. [4]

1.2 Metoda ADMM

V článku „Embedded Online Optimization for Model Predictive Control at Megahertz Rates“ (viz. [4]) byly diskutovány metody prediktivního řízení pro použití v FPGA. Jednou z diskutovaných metod byla i metoda ADMM. Tato metoda zde byla použita pro řešení duálního optimalizačního problému při zavedeném vstupním a stavovém omezení systému. Pro tento účel je vytvořena kopie *y* rozhodovací proměnné *z*, která obsahuje vstupní vektor *u* a vektor stavů *x*. Pro tyto proměnné se následně řeší optimalizační problém a z výsledků těchto optimalizací je spočítán gradient. Tento postup je prováděn iteračně, jak jde vidět z převzatých rovnic (1), (2) a (3). [4]

$$y_{i+1} := arg\,min_y\, L_\rho(z_i, y, v_i) \quad (1)$$

$$z_{i+1} := arg\,min_z\, L_\rho(z, y_{i+1}, v_i) \quad (2)$$

$$v_{i+1} := v_i + \rho(y_{i+1} - z_{i+1}) \quad (3)$$

Kde gradient je aproximován pomocí $(y_{i+1} - z_{i+1})$. Parametr ρ je rozšířený Lagrangiův parametr (angl. augmented Lagrangian parameter) pro duální funkce. Tento parametr bývá při implementaci nechán jako konfigurační. [4] [6]

Výhodou této metody je možnost rozdělit optimalizační problém do dvou, případně i více, na sobě nezávislých optimalizací. Díky tomuto rozdělení se může docílit zjednodušení výpočtů těchto optimalizací, které by byly v případě společného řešení výpočetně náročné.

V některých případech měkkého omezení, kdy je velký rozdíl mezi velikostí Lagrangiova multiplikátoru v pro omezení rovnosti ($z = y$) a velikostí hlavních iterací z_i a y_i, tato metoda může velmi pomalu konvergovat. Důvod lze vidět z rovnice (3), kdy pro existenci velkého optimálního multiplikátoru v je nezbytné provést velký počet iterací, když rozdíl při aproximaci gradientu zůstává malý při každé iteraci při $\rho \approx 1$. Pro zlepšení konvergence je možné rozšířit rozhodovací proměnné o přidané proměnné obsahující podmínky pro kvadratickou regulaci, kdy se nám zvětší dimenze proměnných o dimenzi přidané proměnné, nebo můžeme lepší konvergence dosáhnout vhodnými úpravami rovnic. [4]

Samotné vytváření rovnic pro řešení optimalizačních problému pro účely implementace do vestavěných zařízení je velmi složité a v této práci tato problematika nebude popisována. V následující pod-kapitole budou stručně popisovány jednotlivé kroky výpočtu specifické aplikace této metody, která je určena pro výzkumné účely.

1.3 Popis simulace ADMM v programu Matlab

Cílem této práce byla realizace ADMM v obvodech programovatelné logiky. Úkolem tohoto algoritmu je omezení amplitudy (lineární omezení) a frekvenčního spektra (kvadratické omezení) vstupních a stavových proměnných systému. Pro tuto realizaci mi byla panem doc. Ing. Václavem Šmídlem, Ph.D. poskytnuta simulace tohoto algoritmu napsaná v programu Matlab. V této kapitole bude stručně popsána část simulace obsahující kód, který byl převáděn do FPGA v rámci této práce. Veškerý popisovaný kód je reprezentován ve stejném znění, jako je originál (viz. Příloha I. A II.).

Simulace z programu Matlab (viz. Příloha I.) je rozdělena na tři části, kde úkolem první části označené jako „u update“ je aktualizace vstupů systému *u*. Vektor *tmp* provádí sloučení výsledků optimalizace a Lagrangiova multiplikátoru (*zl* + *vl*) pro lineární omezení a (*zq* + *vq*) pro kvadratické omezení. Veličina *rho* zde reprezentuje Lagrangiův parametr a z hlediska algoritmu je konstantní. Další konstantou v této části je čtvercová matice *A_cost_inv*, která představuje parametr optimalizace. Vektor *f_cost* je z pohledu algoritmu vstupní parametr optimalizace a je počítán v nadřazeném systému (mimo smyčku „ADMM solver“) pomocí vstupní požadované hodnoty a na základě výstupního prvního prvku z vektoru *u*. [6]

Druhá část s označením „Linear constraints“ provádí výpočet optimalizace pro lineární omezení součástí tohoto výpočtu je úprava „over relaxation“ *ul* (úprava používaná pro optimalizaci), kde je konfiguračním parametrem *alp*. Výpočet optimalizace *zl* obsahuje konstanty lineárního omezení *ub* a *lb* pro limitaci amplitudy vstupního vektoru *u*. Následně je aktualizován multiplikátor pro lineární část *vl*. [6]

Poslední část s označením „Quadratic constraints“ obsahuje stejné kroky jako předchozí část. Kdy první řádek této části přestavuje obdobný výpočet jako v prvním řádku části předchozí s rozdílem, že matice *zq* obsahuje sloupcové vektory, o stejné délce jako *zl* v části dva, řazené za sebou, které korespondují s počtem podmínek pro kvadratické omezení *nq*. V posledním řádku této části si můžeme všimnout stejného výpočtu *vq* jako byl pro výpočet *vl* s tím rozdílem, že *uq* a *vq* jsou teď matice o stejných dimenzích a strukturou jako *zq*. Pro výpočet optimalizace *zq* je zde využito funkce s pojmenováním „Huang“. Z hlediska algoritmu jsou zde konstanty *Ac* (matice pro kvadratické omezení), *E_vec* (vlastní vektory v matici), *EI_vec* (inverzní vlastní vektory v matici), *ED_val* (vlastní čísla v diagonální matici), *E_val* (vlastní čísla ve vektoru). Proměnné vektory *fc* (vektor pro kvadratické omezení) a *–bc* (obsahuje podmínky pro omezení frekvenčního spektra) jsou vstupními parametry, které jsou opět aktualizovány na základě vstupního referenčního signálu a výstupu *u* (první prvek vektoru).

Ve funkci „Huang“ (viz. Příloha II.) je na základě výsledku podmínky při, které se zkoumá, zda se bod nachází ve vymezené oblasti, pokud ano tak není prováděn další zásah. Pokud podmínka neprojde, jsou hodnoty převedeny do souřadného systému, který je dán vlastními vektory. Následně se provede metoda půlení intervalu neboli bisekce na intervalu daném konstantami *mu_low* (tato proměnná z hlediska funkce Huang byla v implementaci nahrazena konstantou) a *mu_high*. Z bisekce nám vyvstává další důležitá konstanta a to je počet iterací potřebný pro provedení bisekce. Výsledek z bisekce je pak transformován zpět do původního souřadného systému.

Po ukončení předem známého počtu iterací *iter_max* (viz. Příloha I.) je první prvek z vektoru *u* použit pro aktualizaci systémových stavů a inicializačních podmínek což už nebylo součástí převodu do FPGA, proto zde už tyto výpočty nejsou uvedeny.

2 Návrh implementace algoritmu

Celý program obsahující algoritmus, který jsem měl převést, byl napsán v programu Matlab a bylo tedy nezbytné si nadefinovat kroky pro převedení algoritmu do FPGA. Pro implementaci do FPGA jsem použil jazyk VHDL ve standartu VHDL-1993. Inspiraci pro návrh jednoduchých bloků v jazyce VHDL jsem čerpal z doporučené literatury. [7]

Prvním krokem pro převod algoritmu bylo rozdělení algoritmu na menší samostatné části (pod-komponenty) ze kterých se výsledná komponenta bude skládat. Algoritmus jsem pro to rozdělil na čtyři části, funkce jednotlivých částí je popisována v pod-kapitole (viz. 2.3) níže. Sloučil jsem výpočet pro lineární a kvadratické omezení pro jednotlivé vektory, protože jejich výpočet je při implementaci do FPGA v zásadě stejný.

Dalším důležitým krokem byla definice konstant z hlediska algoritmu, který bude implementován do FPGA. Výčet těchto konstant, jejich označení ve VHDL kódu a v Matlab-u a další podrobnosti jsou uvedeny v tabulce (*Tab. 3*), popis jak vytvořit tyto konstanty je sepsán v pod-kapitole (viz. *2.1*).

Po rozdělení na jednotlivé pod-komponenty a nadefinování inicializačních konstant byla na řadě volba komunikace mezi pod-komponentami, jako je nastavování příznaků pro příjem, vysílání dat nebo samotný způsob vystavování dat (paralelně/sériově). Chování jednotlivých pod-komponent je popsáno v pod-kapitole (viz. 2.3).

Nakonec bylo potřeba vytvořit jednu komponentu, která bude schopna komunikovat s nadřazeným systémem. Tento nadřazený systém dodává data spočítané na základě výstupu z ADMM algoritmu (výsledná VHDL komponenta) a vstupu zvenčí (např. A/D převodník). Chování této komponenty a požadavky pro správnou komunikaci s ní jsou popsány v pod-kapitole (viz. 2.2).

2.1 Definice inicializačních konstant a parametrů

Pro inicializaci konstant jsem vytvořil dvě knihovny, které obsahují inicializaci konstant a nové datové typy. První knihovna „**pkg_init_positive.vhd**“ je pomocná knihovna, kterou využívá knihovna „**vektor_type.vhd**“ pro inicializaci konstant a nových datových typů, které jsou již používány pro realizaci ADMM algoritmu.

V tabulce (*Tab. 1*) jsou sepsány názvy konstant a použitý datový.

Tab. 1 *Konstanty obsažené v knihovně pkg_init_positive*

Název	Datový typ	Popis
Q_init	positive	Počet bitů za desetinou čárkou.
poc_bitu_init	positive	Počet bitů pro jeden prvek.

Název	Datový typ	Popis
poc_prvku_init	positive	Počet prvků v jednom vektoru.
poc_vektoru_init	positive	Počet vektorů pro lineární a kvadratické omezení.
poc_iter_init	positive	Počet iterací ADMM algoritmu.
poc_iter_HUANG_init	positive	Počet iterací v Bisekci.
poc_vek_mat_init	positive	Počet vektorů pro speciální matice.[1]

Knihovna „**vektor_type.vhd**“ definuje nové datové typy, které jsou závislé na konstantách z pomocné knihovny. Důvodem pro deklaraci nových datových typů, bylo zjednodušení zápisu a zlepšení orientace v kódu. Výčet nových datových typů s podrobnějším popisem je v tabulce (*Tab. 2*). Většina nových typů jsou pole obsahující různý počet prvků, jeden prvek je reprezentován bitovým polem „**std_logic_vector((poc_bitu_init - 1) downto 0)**“ (pro zkrácení zápisu se v tabulkách místo tohoto zápisu bude používat **typ_prvek**).

Tab. 2 *Výčet nových datových typů definovaných ve vektor_type*

Název	Rozsah 0 až XX	Základní datový typ	Popis
prvku_19	poc_vektoru_init - 2	typ_prvek	Tento typ je především používán pro vstupní data z nadřazeného systému (kvadratické omezení v Matlab-u označované bc).
vektor	poc_prvku_init - 1	typ_prvek	Tento typ reprezentuje jeden vektor o X prvcích.
vek_19	((poc_vektoru_init - 1) * poc_prvku_init) - 1	typ_prvek	Tento typ reprezentuje matici o X vektorech.
vek_z	poc_vek_mat_init - 1	typ_prvek	Tento typ reprezentuje pole o X vektorech, kde délka vektorů není stejná.
matice	(poc_prvku_init * poc_prvku_init) - 1	typ_prvek	Tento typ představuje čtvercovou matici korespondující s typem vektor.

[1] Toto číslo vychází ze součtu dimenzí čtvercových matic (př.: EI_vec) použitých pro výpočet zq v programu Matlab. Př.: 19 matic o DIM = 1,2,3,4,5,6,7,8,9,10,10,10,10,10,10,10,10,10,10 => $\sum DIM = 145$

Název	Rozsah 0 až XX	Základní datový typ	Popis
matice2	(poc_prvku_init * poc_vek_mat_init) - 1	typ_prvek	Tento typ reprezentuje matici obsahující jednotlivé matice o různých dimenzích se zarovnáním na délku datového typu vektor.
int_arr	poc_vektoru_init - 2	unsigned(4 downto 0)	Toto pole představuje zásobník pro porovnávání délek/dimenzí matic/vektorů/polí.

Tato knihovna dále obsahuje konstanty, které jsou používány jako parametry pro zobecnění zápisu kódu, nebo jako inicializační konstanty pro chod algoritmu. V tabulce (*Tab. 3*) jsou vypsány všechny konstanty s názvem použitým ve VHDL kódu, příslušné pojmenování z Matlab-u.

Tab. 3 *Konstanty obsažené v knihovně vektor_type*

Název (VHDL)	Název (Matlab)	Datový typ	Popis
Q		positive	Inicializováno pomocí Q_init.
poc_bitu		positive	Inicializováno pomocí poc_bitu_init.
poc_prvku		positive	Inicializováno pomocí poc_prvku_init.
poc_vektoru		positive	Inicializováno pomocí poc_vektoru_init.
poc_iter		positive	Inicializováno pomocí poc_iter_init.
poc_iter_HUANG		positive	Inicializováno pomocí poc_iter_HUANG_init.
poc_vek_mat		positive	Inicializováno pomocí poc_vek_mat_init.
lb	lb	typ_prvek	Lineární omezení spodní hranice.
ub	ub	typ_prvek	Lineární omezení horní hranice.
RO	rho	typ_prvek	Parametr ADMM.
alfa	alp	typ_prvek	Parametr ADMM.
alfa_1		typ_prvek	alfa_1 = 1 - alfa

Název (VHDL)	Název (Matlab)	Datový typ	Popis
jenda		typ_prvek	Tato konstanta představuje 1 v definovaném formátu pevné desetinné čárky.
minusJEDNA		typ_prvek	Tato konstanta představuje (-1) v definovaném formátu pevné desetinné čárky.
mu_low1	mu_low	typ_prvek	Tato konstanta představuje fixní spodní hranici při Bisekci.
mu_high	mu_high	typ_prvek	Tato konstanta představuje fixní horní hranici při Bisekci.
mu_pom2		typ_prvek	Tato hodnota představuje střed na intervalu <mu_low, mu_high>.
len		int_arr	To to pole obsahuje velikost dimenzí/délek matic/vektorů.
A_cost_inv	A_cost_inv	matice	Tato matice obsahuje prvky inverzní váhovací matice.
Ac	Ac	matice2	Matice kvadratických omezení.
EI_vec	EI_vec	matice2	Tato matice obsahuje inverzní vlastní vektory.
E_vec	E_vec	matice2	Tato matice obsahuje vlastní vektory.
E_val	E_val	vek_z	Tento vektor obsahuje vlastní čísla.

Všechny konstanty s datovým typem **typ_prvek**, **matice**, **matice2** a **vek_z** jsou zapsány ve formátu čísla s pevnou desetinnou čárkou. Tento formát musí korespondovat s konstantami **poc_bitu_init** a **Q_init**. Pro inicializaci konstanty **len** s datovým typem **int_arr** není třeba zvláštního přístupu, stačí klasická inicializace pro pole složené z unsigned.

Konstanta **A_cost_inv** obsahuje prvky transponované stejnojmenné matice v Matlab-u. Tyto prvky jsou po celých řádcích poskládány za sebou.

Pro inicializaci konstant **Ac**, **E_vec**, **EI_vec** a **E_val** bylo třeba zvolit vlastní formát seskupení prvků, protože tyto konstanty obsahují matice po případě vektory o různých dimenzích.

Konstantní pole **Ac** obsahuje transponované čtvercové matice, kde jsou jednotlivé řádky matic doplněny nulami do délky vektoru. Takto upravené matice jsou opět po celých řádcích poskládány za sebou. Stejný postup je aplikován i na konstanty **E_vec** a **EI_vec**. Konstantní pole **E_val** je složeno z vektorů o různých délkách bez doplnění nulami na stejnou délku, jako tomu bylo u předchozích konstant.

Pro generický dizajn jednotlivých čítačů v komponentách jsem použil generické parametry (*Tab. 4*), které se nastavují v hlavní komponentě. Tyto parametry určují počet bitů mínus jedna pro jednotlivé čítače.

Tab. 4 *Generické parametry pro realizaci čítačů (komponenta komplet_v1_02)*

Název	Datový typ	Maximální rozsah (0 až XX)
B_cit_200	positive	poc_prvku * poc_vektoru
B_cit_145	positive	poc_vek_mat – 1
B_cit_50	positive	poc_iter – 1
B_cit_20	positive	poc_vektoru – 1
B_cit_10	positive	poc_prvku – 1

2.2 Popis chování hlavní entity

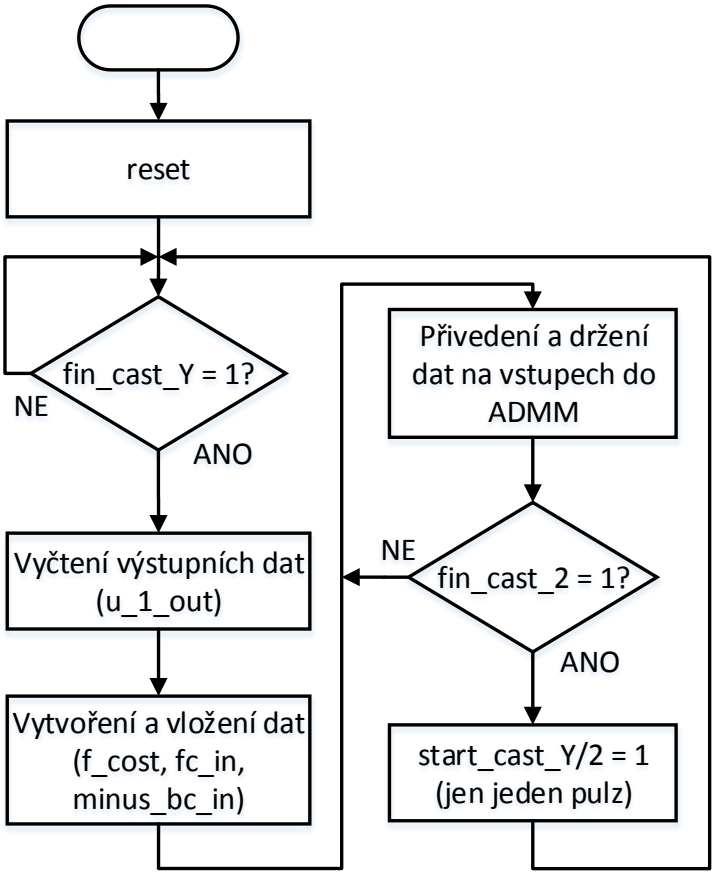
Hlavní výsledná komponenta (tzv. top level entita) **komplet_v1_02** v souboru **komplet_v1_02.vhd** propojuje jednotlivé pod-komponenty a přidává logiku pro vystavení dat, spouštění a ukončení algoritmu. Rozhraní pro vstupní a výstupní porty této komponenty je vypsáno v tabulce (*Tab. 5*). Generické parametry pro tuto komponentu jsou vypsány v tabulce (*Tab. 4*).

Tab. 5 *Rozhraní komponenty komplet_v1_02*

Název	Směr (In/Out)	Datový typ	Popis
gl_clk	In	std_logic	Společný hodinový signál. Reaguje na náběžnou hranu.
gl_reset	In	std_logic	Společný asynchronní reset. Aktivní v ‘0’.
start_cast_Y	In	std_logic	Signál pro spuštění pod-komponenty cast_Y.
start_cast_2	In	std_logic	Signál pro spuštění pod-komponenty cast_2.
fin_cast_Y	Out	std_logic	Signál indikující vystavení nových dat z algoritmu.
fin_cast_2	Out	std_logic	Signál indikující dopočítání algoritmu.

Název	Směr (In/Out)	Datový typ	Popis
busy_ADMM	Out	std_logic	Indikátor zpracovávání dat od startu algoritmu do vystavení nových dat.
f_cost	In	vektor	Vstupní data z nadřazeného systému.
fc_in	In	vek_19	Vstupní data z nadřazeného systému.
minus_bc_in	In	prvku_19	Vstupní data z nadřazeného systému.
u_1_out	Out	typ_prvek	Výstupní data z algoritmu.

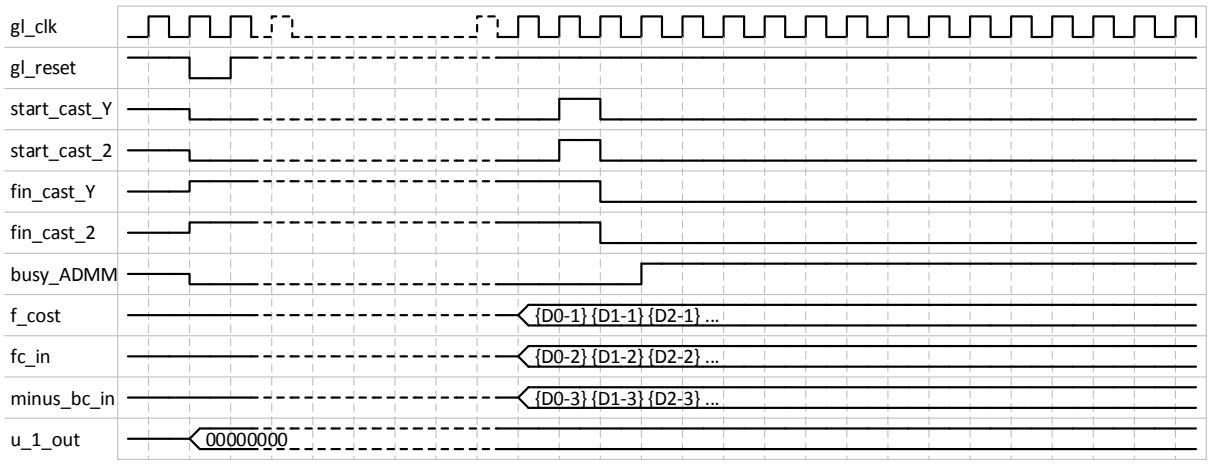
Jak už bylo řečeno, tato komponenta propojuje jednotlivé pod-komponenty a slouží jako rozhraní pro nadřazený systém. Je proto důležité dodržovat správný postup při komunikaci s ní a dále je nutné dodržovat správný souběh řídících signálů a vstupních dat. Postup při komunikaci s komponentou je znázorněn pomocí vývojového diagramu na obrázku (*Obr. 1*).



Obr. 1 *Průběh komunikace s komponentou komplet_v1_02*

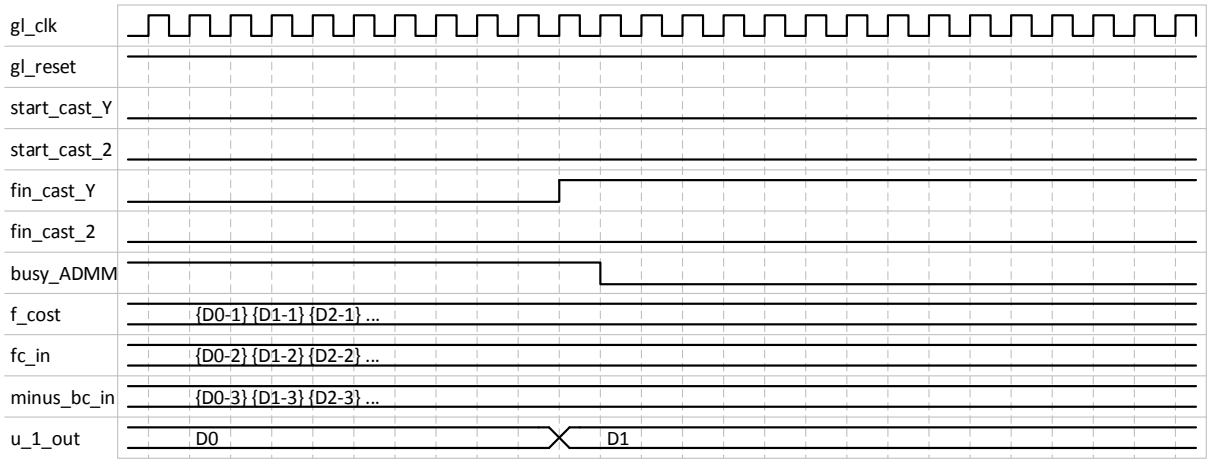
Jak jde vidět z diagramu (*Obr. 1*), tak je nutné před započetím komunikační rutiny vyvolat asynchronního reset, který je aktivní v log. 0. Provedení resetu je nutné pro uvedení algoritmu do známého výchozího stavu (viz. *Obr. 2*). Po provedeném resetu algoritmus čeká na vložení nových dat a následné spuštění pomocí uvedení signálů **start_cast_Y** a

start_cast_2 do log. 1 po dobu jednoho taktu. Při spouštění algoritmu je nezbytné, aby vstupní data zůstala konstantní.



Obr. 2 *Příklad průběhů signálů a dat od resetu ke spuštění (komplet_v1_02)*

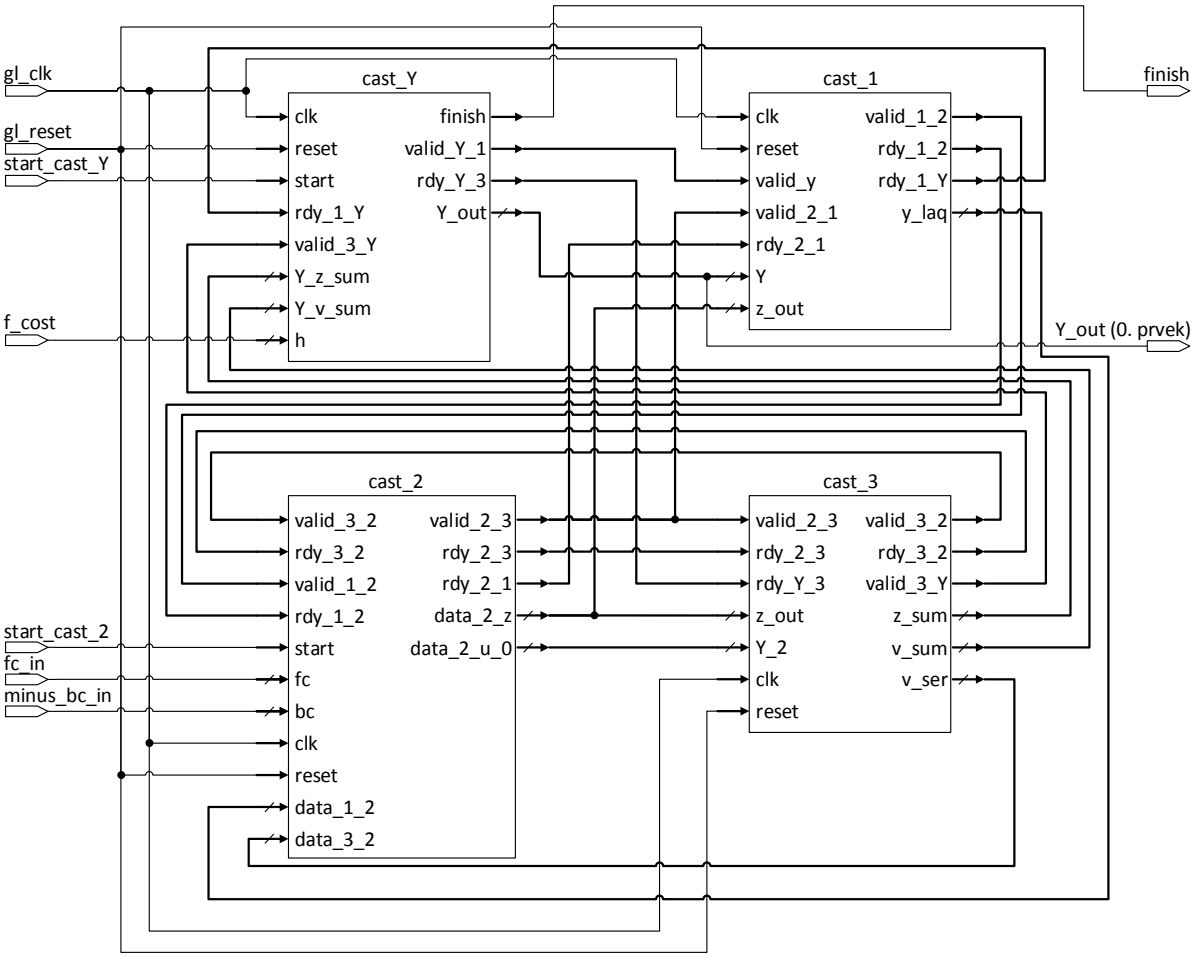
Po resetu a prvním spuštění algoritmu se čeká na výstup dat **u_1_out** (viz. *Obr. 3*), který je značen signálem **fin_cast_Y** (aktivní v log. 1). Je-li signál **fin_cast_Y** aktivní, pak se mohou data vyzvednout a v nadřazeném systému použít k výpočtu vstupních dat, ty se pak přivedou zpět na příslušné vstupy komponenty.



Obr. 3 *Průběh signálů při výstupu dat (komplet_v1_02)*

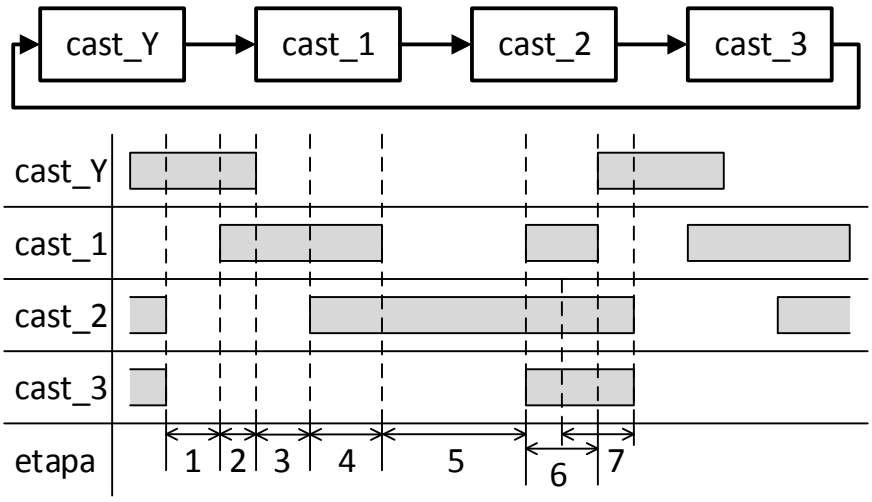
Signál **fin_cast_2** (aktivní v log. 1) značí, že pod-komponenta **cast_2** skončila (tato pod-komponenta využívá některá vstupní data, proto je nutné počkat, až také dopočítá) a je tedy možné provést inicializaci (spuštění) dalšího výpočtu algoritmu pomocí současné aktivace signálů **start_cast_Y** a **start_cast_2**.

Jak už bylo zmíněno, tak tato hlavní entita realizuje propojení jednotlivých pod-komponent. Znázornění těchto propojení je na obrázku (*Obr. 4*), pro zjednodušení zde není zakreslen proces pro zpracování výstupních signálů z hlavní entity.



Obr. 4 *Zjednodušené schéma vnitřního propojení pod-komponent*

Na obrázku (*Obr. 5*) v horní části je zobrazena sekvence zpracování dat jednotlivými pod-komponentami. Ve spodní části obrázku (*Obr. 5*) je šedými obdélníky znázorněna aktivita jednotlivých pod-komponent a je rozdělena na etapy. Podrobnější popis těchto pod-komponent je v kapitole (viz. 2.3).



Obr. 5 *Sekvence zpracování dat (nahoře); Diagram aktivity pod-komponent (dole)*

V 1. etapě probíhá zpracování v pod-komponentě **cast_Y**. Následně v 2. etapě proběhne paralelní předání dat z pod-komponenty **cast_Y** do pod-komponenty **cast_1.** V 3. a částečně i ve 4. etapě probíhá zpracování pod-komponentou **cast_1**, která ve 4. etapě sériově předá data do pod-komponenty **cast_2**, kde následně proběhne dlouhé zpracování dat (způsobeno iteračním charakterem výpočtu v této části; 5. a částečně 6. etapa). Pod-komponenta **cast_2** v 6. etapě sériově předává data pod-komponentám **cast_1** a **cast_3** v této etapě začíná zpracovávání v pod-komponentě **cast_3.** V poslední 7. etapě probíhá sériové předání dat z pod-komponenty **cast_3** do pod-komponenty **cast_2** a paralelní přenesení dat z **cast_3** zpět do pod-komponenty **cast_Y**.

2.3 Popis chování jednotlivých pod-komponent

V této pod-kapitole jsou vypsány rozhraní a generické parametry jednotlivých pod-komponent. Dále jsou zde vyobrazeny některé průběhy signálů a dat při běhu algoritmu. Veškeré průběhy signálů jsou vyobrazeny při nastavení uvedeném v tabulce (*Tab. 6*).

Tab. 6 *Hlavní nastavení algoritmu pro účely reprezentace průběhů*

Název	Hodnota	Umístění v souboru
B_cit_200	7	komplet_v1_02.vhd
B_cit_145	7	komplet_v1_02.vhd
B_cit_50	5	komplet_v1_02.vhd
B_cit_20	4	komplet_v1_02.vhd
B_cit_10	3	komplet_v1_02.vhd
Q_init	10	pkg_init_positive.vhd
poc_bitu_init	32	pkg_init_positive.vhd
poc_prvku_init	10	pkg_init_positive.vhd
poc_vektoru_init	20	pkg_init_positive.vhd
poc_iter_init	50	pkg_init_positive.vhd
poc_iter_HUANG_init	20	pkg_init_positive.vhd
poc_vek_mat_init	145	pkg_init_positive.vhd

V následující kapitole je u jednotlivých pod-komponent uveden stručný popis implementované funkce a její reprezentace v předloze v programu Matlab (viz. Příloha I. a Příloha II.).

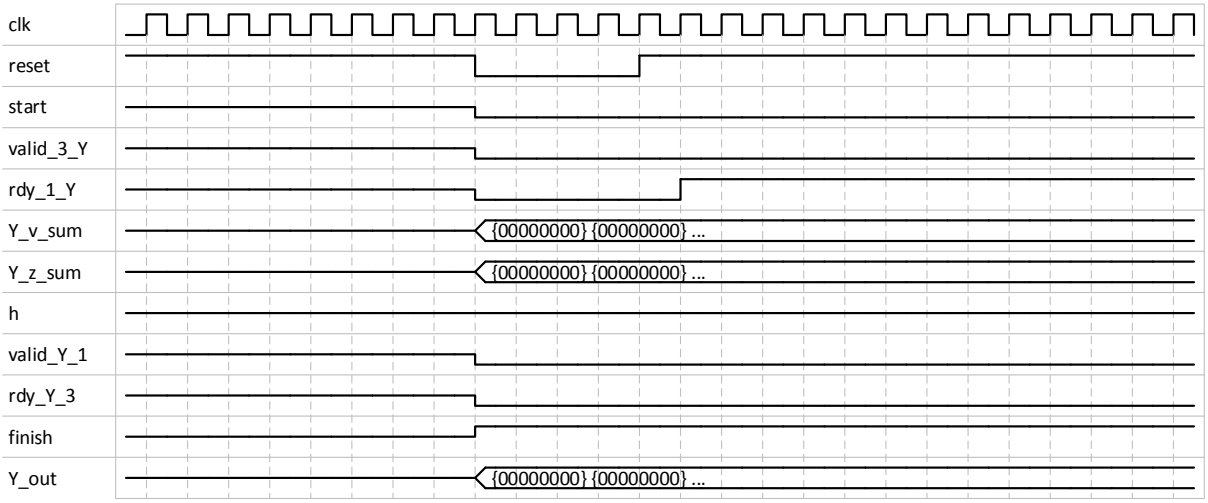
2.3.1 Pod-komponenta cast_Y

V této komponentě je implementovaná část algoritmu, který se stará o počítání provedených iterací celého algoritmu a vystavení dat z algoritmu. Hlavním úkolem této části algoritmu je aktualizace vstupních stavů. Rozhraní této komponenty je uvedeno v tabulce (Tab. 7).

Tab. 7 *Rozhraní a generické parametry komponenty cast_Y*

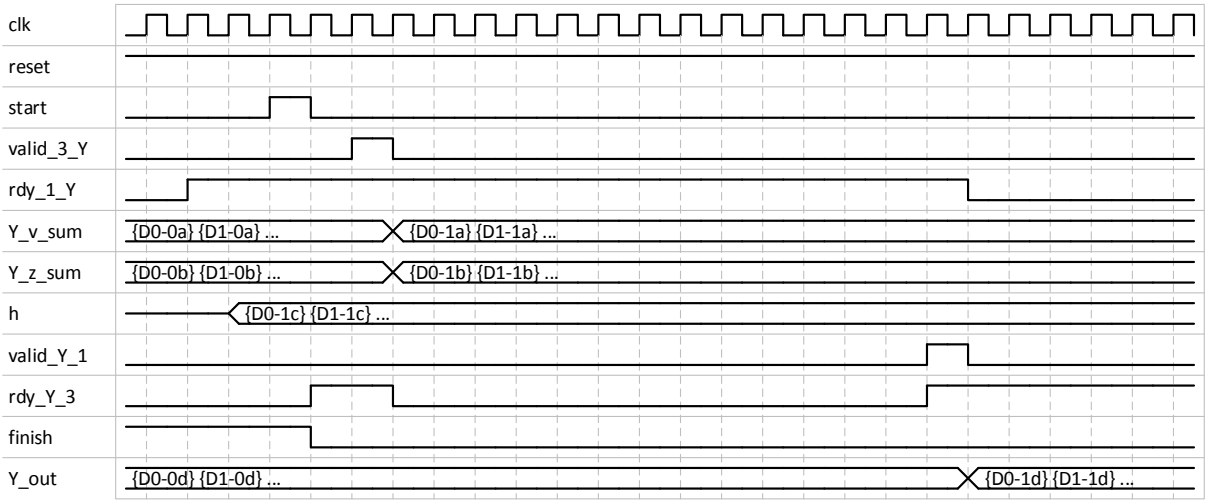
Název	Směr (In/Out)	Datový typ	Popis
w_cit_50		positive	Generický parametr inicializován pomocí **B_cit_50**.
w_cit_10		positive	Generický parametr inicializován pomocí **B_cit_10**.
start	In	std_logic	Signál pro odblokování komponenty po dopočítání algoritmu. Spouští příjem dat na portu **h** (**f_cost**).
valid_3_Y	In	std_logic	Signál z komponenty **cast_3**. Spouští příjem vystavených dat z **cast_3**.
rdy_1_Y	In	std_logic	Signál z komponenty **cast_1**. Blokuje vysílání dat do komponenty **cast_1**.
clk	In	std_logic	Hodinový signál. Reakce na náběžnou hranu.
reset	In	std_logic	Asynchronní reset. Aktivní v logické 0.
Y_v_sum	In	vektor	Paralelní vstup dat z komponenty **cast_3**.
Y_z_sum	In	vektor	Paralelní vstup dat z komponenty **cast_3**.
h	In	vektor	Paralelní vstup dat z nadřazeného systému (**f_cost**).
valid_Y_1	Out	std_logic	Signál do komponenty **cast_1**. Označuje vystavení dat z této komponenty.
rdy_Y_3	Out	std_logic	Signál do komponenty **cast_3**. Blokuje vystavení dat z komponenty **cast_3**.
finish	Out	std_logic	Příznak vystavení dat po dokončení iterací v algoritmu.
Y_out	Out	vektor	Paralelní výstup dat do **cast_1** a v případě dokončení iterací je nultý prvek použit pro výstup algoritmu.

Pro zajištění správné funkce je nutné provést reset, aby se komponenta uvedla do známého stavu (viz. *Obr. 6*).



Obr. 6 *Chování rozhraní při resetu (komponenta cast_Y)*

Po provedeném resetu komponenta drží signál **rdy_Y_3** v logické nule a tím brání v rozběhnutí algoritmu. Pro odblokování je třeba přivést na vstup **start** jeden pulz, který je zároveň použit pro příjem dat na portu **h** (viz. *Obr. 7*). Pro správný příjem dat z nadřazeného systému je nutné, aby se data na vstupním portu **h** neměnila v okolí náběžné hrany **start** signálu.



Obr. 7 *Chování rozhraní při přijmu dat z nadřazeného systému (komponenta cast_Y)*

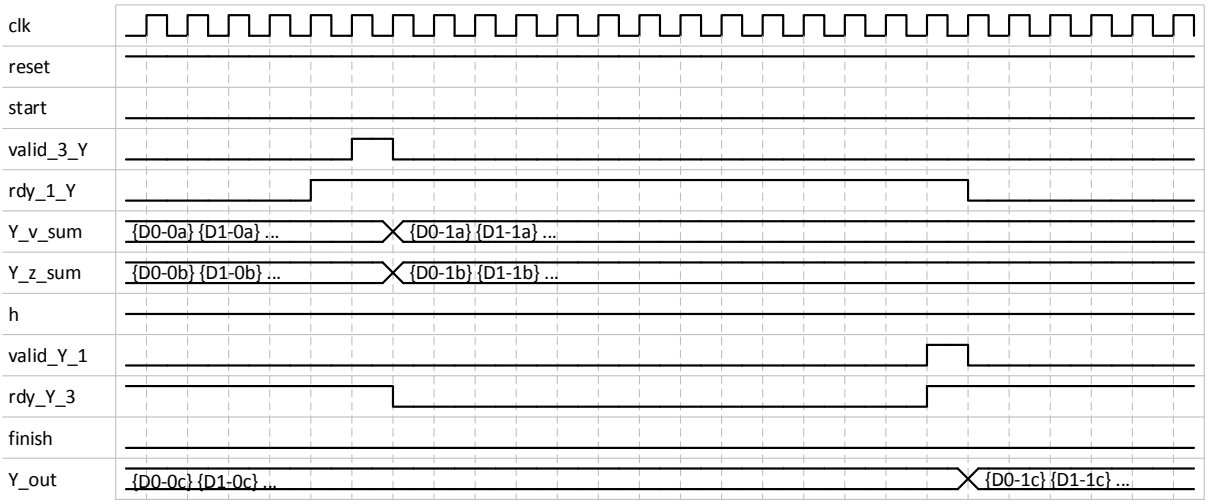
Při standartním běhu, mezi první a poslední iterací algoritmu (viz. *Obr. 8*), je po paralelním přijmu dat na portech **Y_v_sum** a **Y_z_sum** z komponenty **cast_3** (poslední komponenta z hlediska sekvence výpočtu v jedné iteraci) proveden výpočet, který lze shrnout do jedné rovnice:

$$Y\_out = A\_cost\_inv \times (h + RO \times (Y\_v\_sum + Y\_z\_sum)) \qquad (4)$$

Z výše uvedené rovnice vidíme, že zpracování dat v této komponentě odpovídá výpočtům (5) a (6) v předloze z programu Matlab (viz. Příloha I.), kde výpočet je částečně realizován v komponentě **cast_3** a částečně v této komponentě.

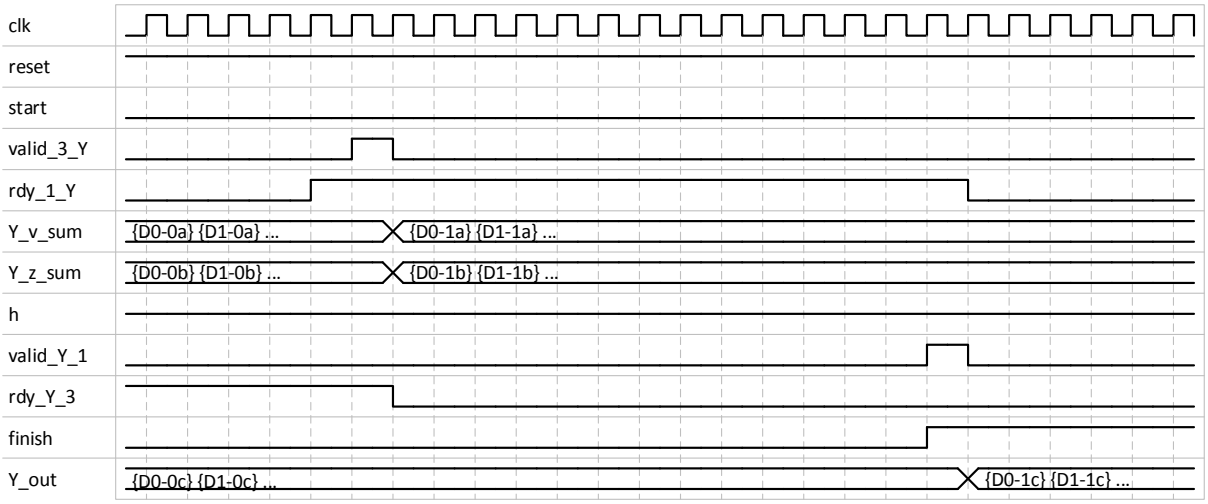
$$tmp = rho \times (zl + vl) + sum(rho \times (zq + vq), 2) \quad (5)$$

$$u = A\_cost\_inv \times (f\_cost + tmp) \quad (6)$$



Obr. 8 *Chování rozhraní při standartním běhu (komponenta cast_Y)*

Při vystavování dat v poslední iteraci výpočtu algoritmu se současně se signálem **valid_Y_1** aktivuje i signál **finish**, který značí dopočítání poslední iterace a je tedy možné vystavit data z algoritmu do nadřazeného systému (viz. *Obr. 9*). Tyto data jsou obsažena v paralelním výstupu **Y_out** z této komponenty v nultém prvku. Signál **rdy_Y_3** v tomto případě zůstává v logické 0, aby po dopočítání ostatních komponent zastavil další spuštění výpočtu v případě, že nedojde k opětovnému zahájení pomocí signálu **start.**



Obr. 9 *Chování rozhraní při vystavení dat po dokončení iterací (komponenta cast_Y)*

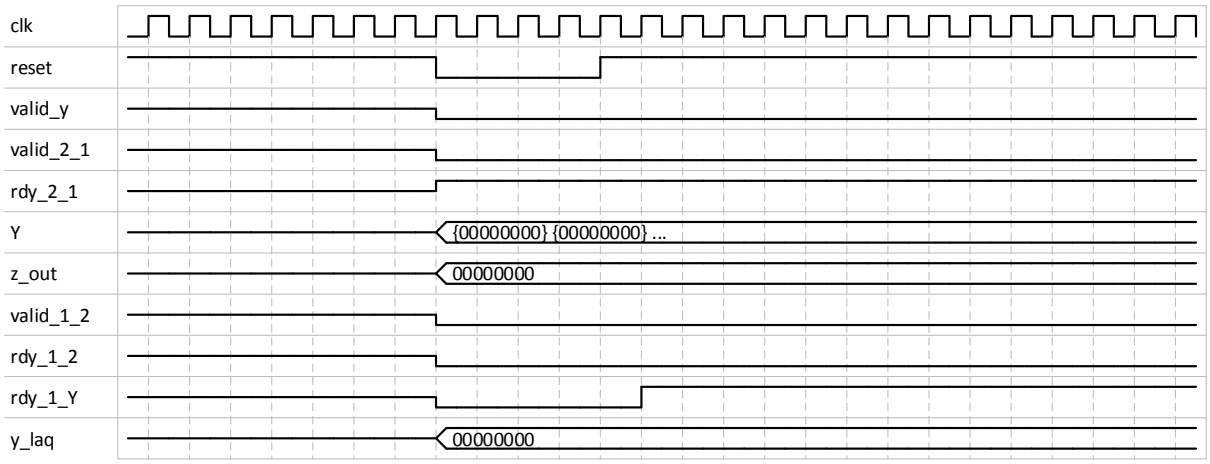
2.3.2 Pod-komponenta cast_1

Rozhraní a generické parametry této komponenty je uvedeno v tabulce (*Tab. 8*).

Tab. 8 *Rozhraní a generické parametry komponenty cast_1*

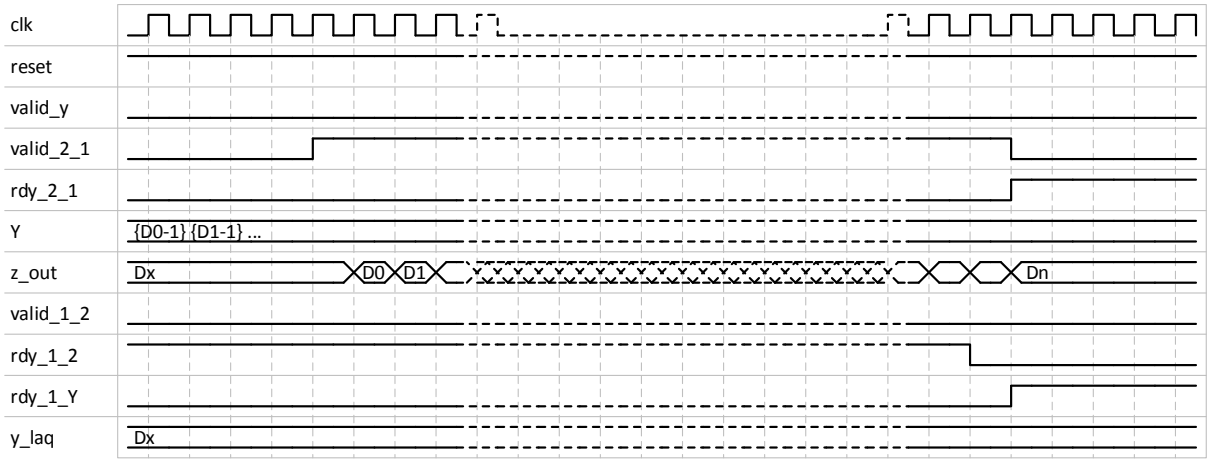
Název	Směr (In/Out)	Datový typ	Popis
w_cit_200		positive	Generický parametr inicializován pomocí **B_cit_200**.
valid_y	In	std_logic	Signál z komponenty **cast_Y**. Spouští příjem vystavených dat z **cast_Y**.
rdy_2_1	In	std_logic	Signál z komponenty **cast_2**. Zastavuje vysílání dat do komponenty **cast_2**.
valid_2_1	In	std_logic	Signál z komponenty **cast_2**. Spouští příjem vystavených dat z komponenty **cast_2**.
clk	In	std_logic	Hodinový signál. Reakce na náběžnou hranu.
reset	In	std_logic	Asynchronní reset. Aktivní v logické 0.
Y	In	vektor	Paralelní vstup dat z komponenty **cast_Y**.
z_out	In	typ_prvek	Sériový vstup dat z komponenty **cast_2**.
valid_1_2	Out	std_logic	Signál do komponenty **cast_2**. Označuje vystavení dat z této komponenty.
rdy_1_2	Out	std_logic	Signál do komponenty **cast_2**. Zastavuje přísun dat z komponenty **cast_2**.
rdy_1_Y	Out	std_logic	Signál do komponenty **cast_Y**. Blokuje vyslání dat z komponenty **cast_Y**.
y_laq	Out	typ_prvek	Sériově vysílána data do komponenty **cast_2**.

Pro nastavení komponenty do známého stavu (viz. *Obr. 10*) je potřeba provést reset komponenty. Na tomto obrázku si můžeme všimnout, že výstupní signál **rdy_1_Y** je nastaven do logické 1 jeden takt po skončení resetu. Tímto nastavení tato komponenta dává najevo, že je připravena na příjem dat z předešlé komponenty **cast_Y**.



Obr. 10 *Chování rozhraní při resetu (komponenta cast_1)*

Při standartním běhu komponenta po aktivaci signálu **valid_y** přijme data vyslané komponentou **cast_Y** skrze port **Y** (viz. *Obr. 12*). Tyto data jsou následně pře násobena konstantou **alfa** a zároveň jsou také pře násobena data přijata z komponenty **cast_2** v přechozí iteraci (viz. *Obr. 11*), konstantou **alfa_1**, vždy prvek po prvku.



Obr. 11 *Příjem dat z cast_2 (komponenta cast_1)*

Výsledky po násobení se spolu sečtou a jsou vystaveny ven skrze port **y_laq** (viz. *Obr. 12*). Tento postup se dá přiblížit zápisem:

$$y\_laq = alfa \times Y_{i \bmod (poc\_prvku\_init)} + alfa\_1 \times z\_out_i \qquad (7)$$

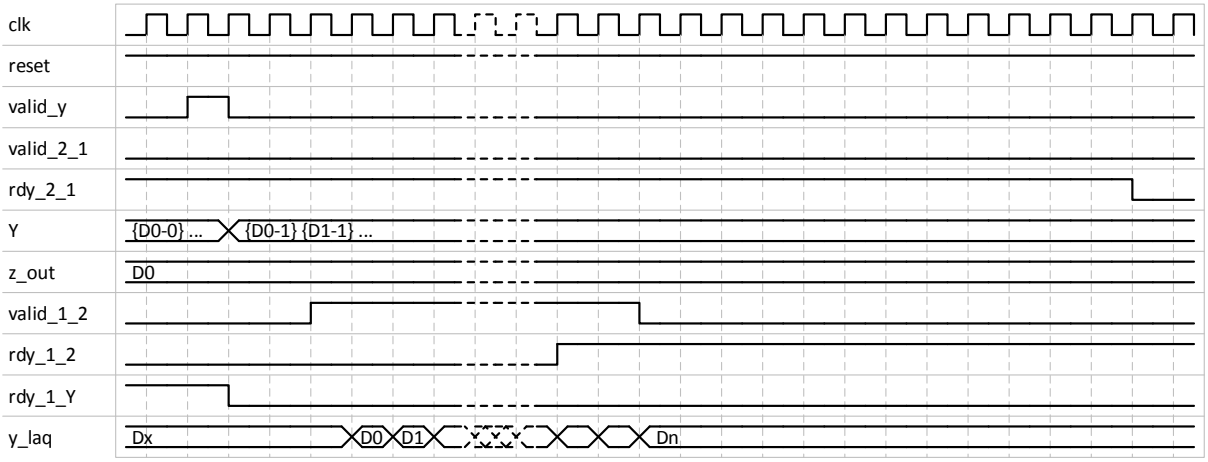
Kde index *i* označuje násobený prvek ve složeném vektoru obsahující vektor pro lineární omezení a vektory pro kvadratické omezení. Tyto data jsou sériově přijímána a následně interně uložena a proto si je v této rovnici můžeme představit jako jeden vektor (pro znázornění příslušného vstupního portu jsou zde využity názvy z rozhraní).

Výsledkem této komponenty je realizace sjednocení a realizace výpočtů z předlohy v programu Matlab (viz. Příloha I.):

$$ul = alp \times u + (1 - alp) \times zl \qquad (8)$$

$$uq = alp \times u \times ones(1, m \times nq) + (1 - alp) \times zq \qquad (9)$$

Kde výpočet *ul* představuje výpočet pro lineární omezení a výpočet *uq* naopak představuje výpočet pro kvadratické omezení.



Obr. 12 *Příjem dat z cast_Y a vysílání dat do cast_2 (komponenta cast_1)*

2.3.3 Pod-komponenta cast_2

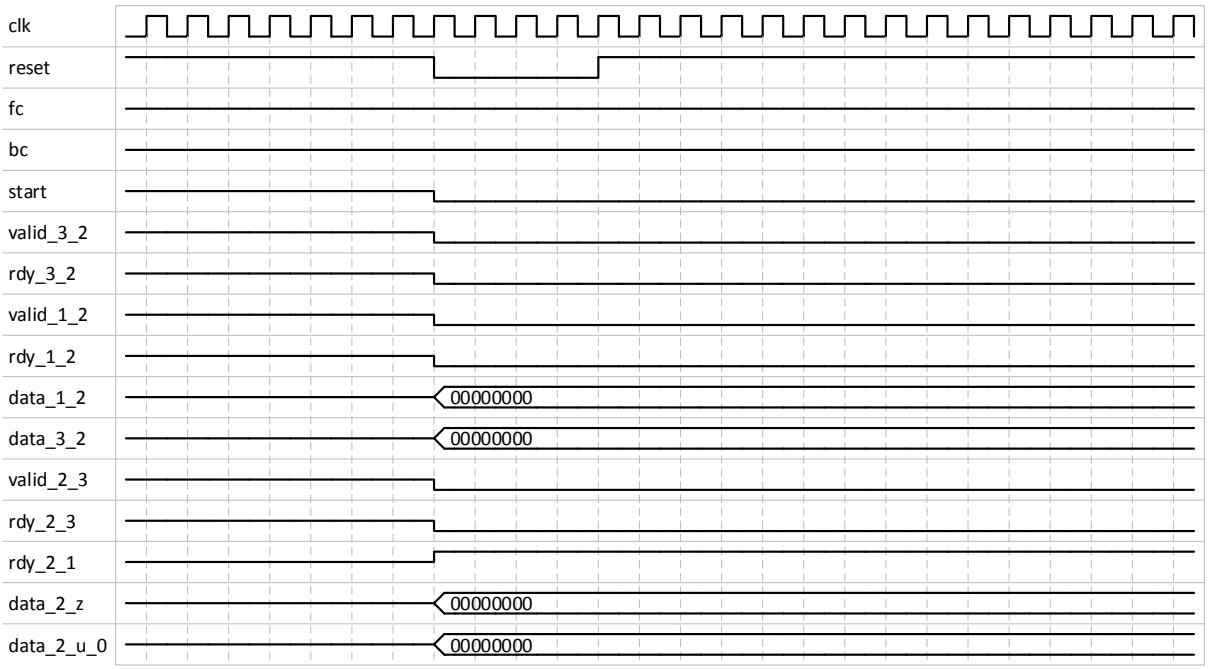
Tato komponenta pro svůj chod využívá velkou část konstant inicializovaných v souboru **vektor_type.vhd**, jako například **Ac, E_vec, EI_vec, E_val, len** a další. Dále je potřeba přivést inicializační data z nadřazeného systému na příslušné porty (viz. *Tab. 9*).

Tab. 9 *Rozhraní a generické parametry komponenty cast_2*

Název	Směr (In/Out)	Datový typ	Popis
w_cit_200		positive	Generický parametr inicializován pomocí **B_cit_200**.
w_cit_145		positive	Generický parametr inicializován pomocí **B_cit_145**.
w_cit_20		positive	Generický parametr inicializován pomocí **B_cit_20**.
w_cit_10		positive	Generický parametr inicializován pomocí **B_cit_10**.
fc	In	vek_19	Paralelní vstup dat z nadřazeného systému (**fc_in**).
bc	In	prvku_19	Paralelní vstup dat z nadřazeného systému (**minus_bc_in**).
start	In	std_logic	Signál spouští příjem dat z nadřazeného systému na portech **fc** a **bc**.
valid_3_2	In	std_logic	Signál z komponenty **cast_3**. Spouští příjem vystavených dat z **cast_3**.

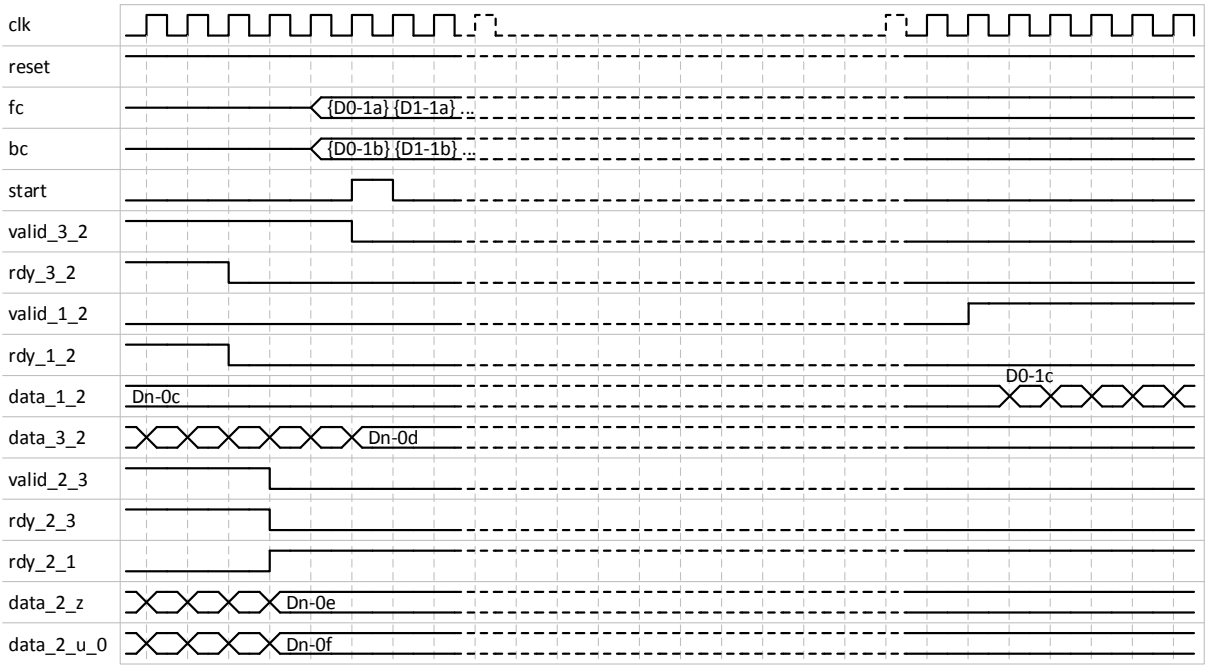
Název	Směr (In/Out)	Datový typ	Popis
rdy_3_2	In	std_logic	Signál z komponenty **cast_3**. Zastavuje vysílání dat do komponenty **cast_3**.
valid_1_2	In	std_logic	Signál z komponenty **cast_1**. Spouští příjem vystavených dat z **cast_1**.
rdy_1_2	In	std_logic	Signál z komponenty **cast_1**. Zastavuje vysílání dat do komponenty **cast_1**.
clk	In	std_logic	Hodinový signál. Reakce na náběžnou hranu.
reset	In	std_logic	Asynchronní reset. Aktivní v logické 0.
data_1_2	In	typ_prvek	Sériový vstup dat z komponenty **cast_1**.
data_3_2	In	typ_prvek	Sériový vstup dat z komponenty **cast_3**.
valid_2_3	Out	std_logic	Signál do komponenty **cast_3**. Označuje vystavení dat z této komponenty.
rdy_2_3	Out	std_logic	Signál do komponenty **cast_3**. Blokuje vysílání dat z komponenty **cast_3**.
rdy_2_1	Out	std_logic	Signál do komponenty **cast_1**. Blokuje vysílání dat z komponenty **cast_1**.
data_2_z	Out	typ_prvek	Sériově vysílaná data do komponent **cast_1** a **cast_3**.
data_2_u_0	Out	typ_prvek	Sériově vysílaná data do komponenty **cast_3**.

Pro správnou inicializaci komponenty je opět nutné provést reset (viz. *Obr. 13*). Z tohoto obrázku můžeme vidět, že tato komponenta při aktivování resetu (logická 0) okamžitě nastaví příznak **rdy_2_1** do logické 1, čímž dává najevo, že je připraven pro příjem dat z předešlé komponenty **cast_1**. Nicméně je nutné pro správnou funkci ještě provést inicializaci této komponenty vstupními daty z nadřazeného systému skrze paralelní vstupy **fc** a **bc** (viz. Obr. 14).



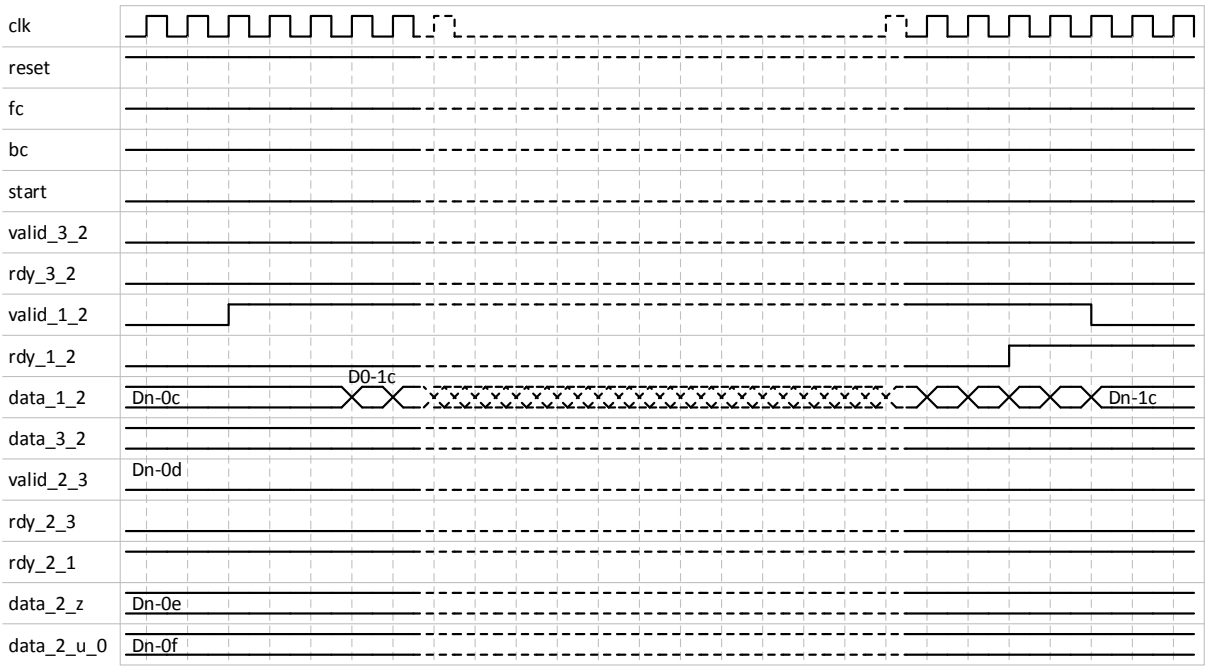
Obr. 13 *Chování rozhraní při resetu (komponenta cast_2)*

Před započetím výpočtu v první iteraci algoritmu musí být přivedeny data na porty **fc** a **bc** následované jedním pulzem signálu **start** (viz. *Obr. 14*), obdobně jako u komponenty **cast_Y**.



Obr. 14 *Chování rozhraní při přijmu dat z nadřazeného systému (komponenta cast_2)*

Po inicializaci následuje standartní chování komponenty, kdy v první fázi se čeká na příjem dat z předešlé komponenty **cast_1** (viz. *Obr. 15*).



Obr. 15 *Chování rozhraní při přijmu dat z cast_1 (komponenta cast_2)*

Po přijetí nových dat z komponenty **cast_1** se provede jejich zpracování. Toto zpracování se dá pomyslně rozdělit na pět částí:

1. Výpočet pro lineární omezení, který je v Matlab-u (viz. Příloha I.) reprezentován výpočtem:

$$zl = min(ub, max(lb, ul - vl)) \quad (10)$$

   Zde je *ul* reprezentováno prvním vektorem s počtem prvků roven **poc_prvku** vyslaný z komponenty **cast_1** skrze port **data_1_2**. Vektor *vl* je reprezentován také prvním vektorem o stejné délce jako *ul*, ale data obsahující tento vektor byla vyslána v předchozí iteraci algoritmu z komponenty **cast_3** skrze port **data_3_2**.

2. Realizace podmínky ve funkci označené v Matlab-u jako Huang (viz. Příloha II.):

$$u\_0' \times A \times u\_0 - 2 \times f' \times u\_0 < b \quad (11)$$

   Kde *u_0* je rozdíl mezi vstupními daty z komponenty **cast_1** a komponentou **cast_3** bez části pro lineární omezení. Matice *A* představuje konstantní pole **Ac**, *f* obsahuje vstupní data z nadřazeného systému **fc** a *b* také obsahuje data z nadřazeného systému **bc**.

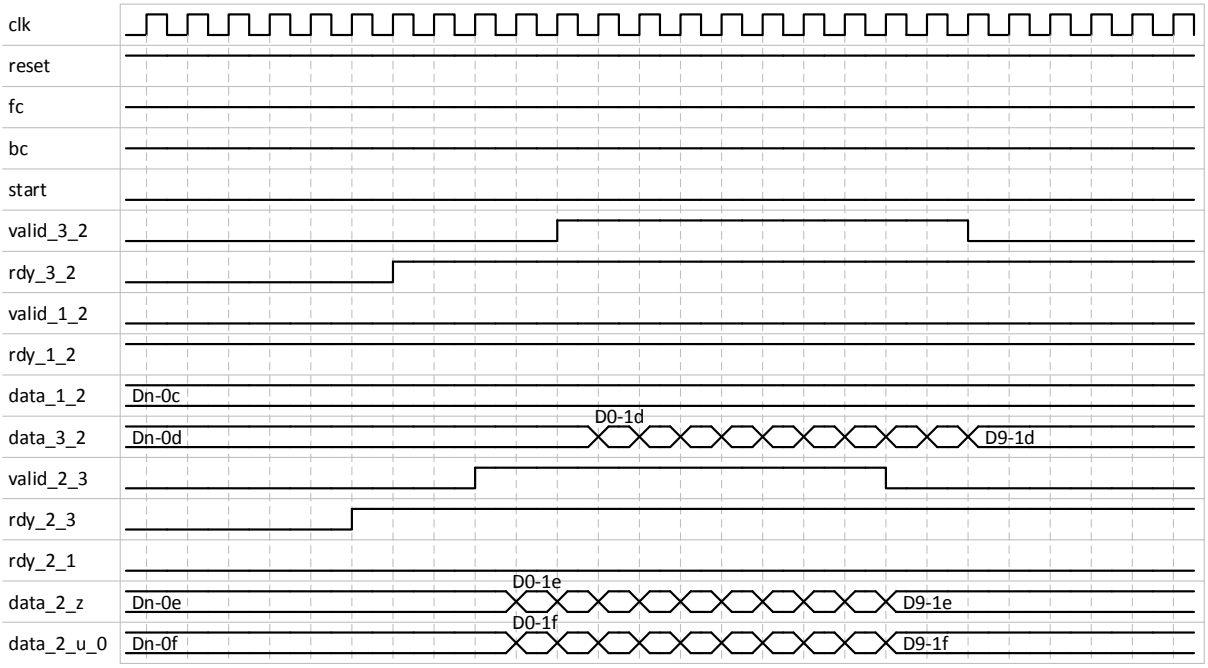
3. Realizace výpočtu pro transformaci do souřadného systému daného vlastními vektory obsažené v konstantním poli **EI_vec**. Jedná se o rovnice:

$$u\_0 = EI\_vec \times u\_0(1{:}\,len) \quad (12)$$

$$f = EI\_vec \times f(1{:}\,len) \quad (13)$$

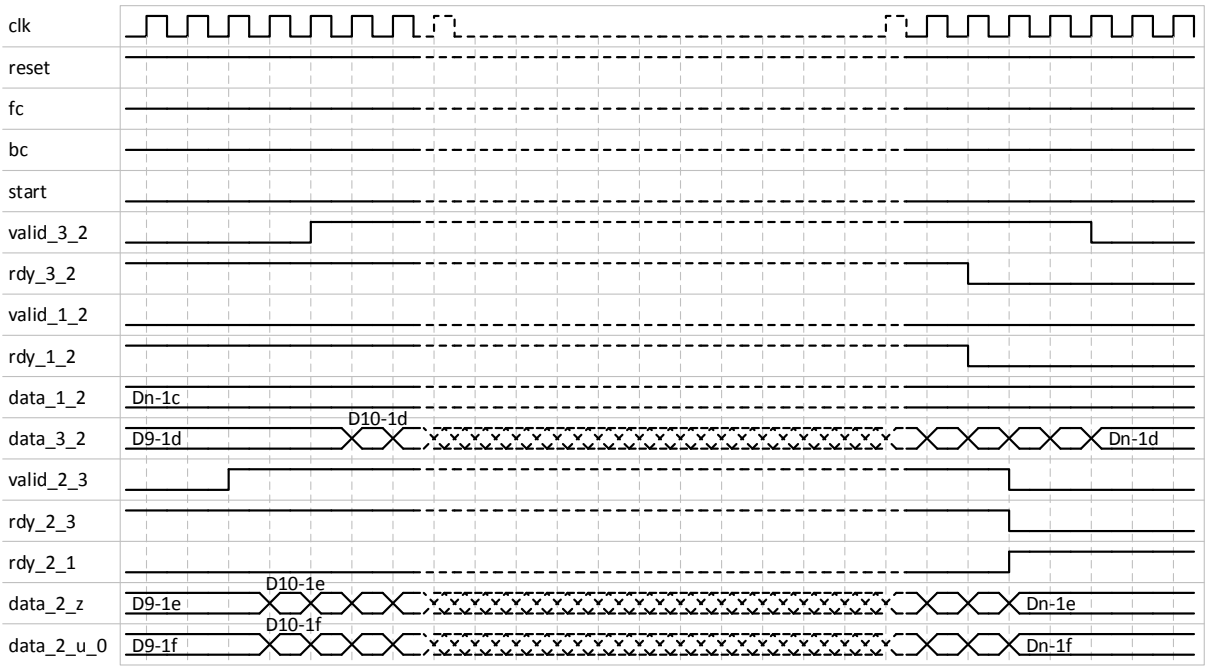
   Během této fáze jsou vyslána data pro lineární omezení (viz. *Obr. 16*). Data z portu **data_2_z** jsou vysílána do komponent **cast_1** a **cast_3**, tyto data obsahují výsledek

z výpočtu (10). Data z portu **data_2_u_0** jsou vysílaná jen do **cast_3** a obsahují rozdíl *ul – vl*.



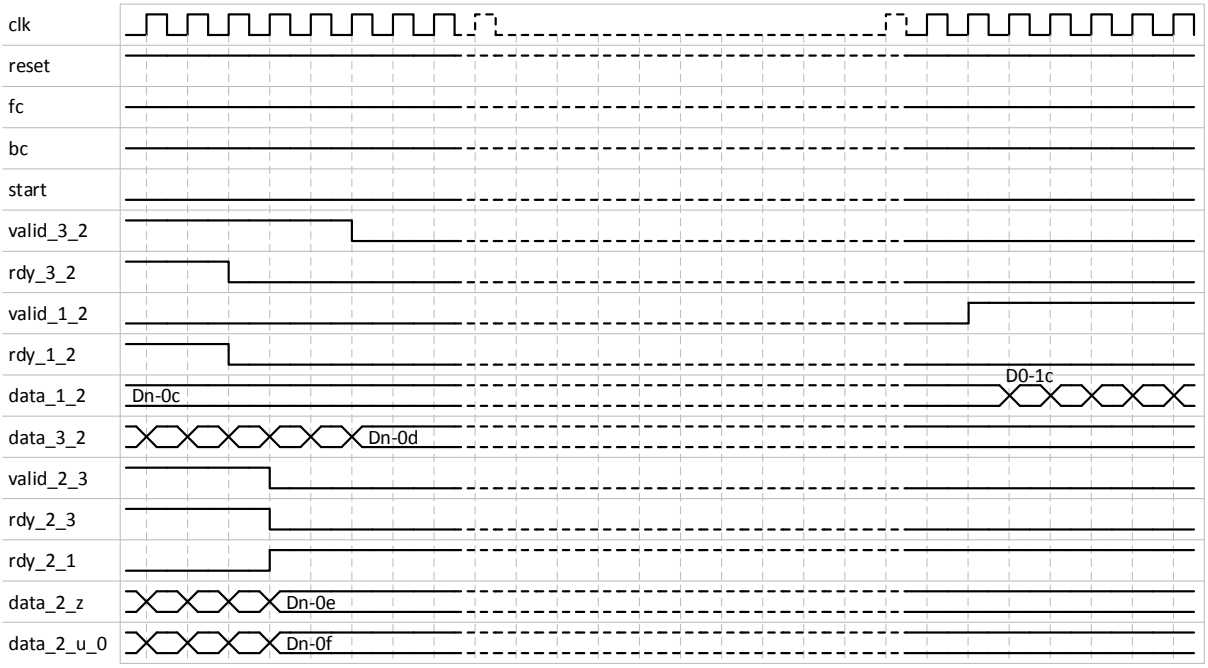
Obr. 16 *Chování rozhraní při přijmu (z cast_3) a vysílání (do cast_1 a cast_3) části dat pro lineární omezení (komponenta cast_2)*

4. Realizace výpočtu bisekce. Tento výpočet je iterační a počet iterací je nastaven pomocí konstanty **poc_iter_HUANG_init**.
5. Vysílání dat pro kvadratické omezení (viz. *Obr. 17*). V této fázi se vybírají data, na základě výsledku podmínky (11), pro výstup na port **data_2_z**. Při splnění podmínky jsou na tomto portu vyslány data rozdílu *uq* a *vq*, při nesplnění podmínky jsou vyslány data z výpočtu bisekce. Z tohoto portu jsou data vysílána do komponent **cast_1** a **cast_3**. Na portu **data_2_u_0** je vysílán rozdíl *uq* a *vq* do komponenty **cast_3**.



Obr. 17 *Chování rozhraní při přijmu (z cast_3) a vysílání (do cast_1 a cast_3) části dat pro kvadratické omezení (komponenta cast_2)*

Po ukončení vysílání dat komponenta čeká na příjem dat z komponenty **cast_1** pro provedení výpočtu v další iteraci algoritmu (viz. *Obr. 18*).



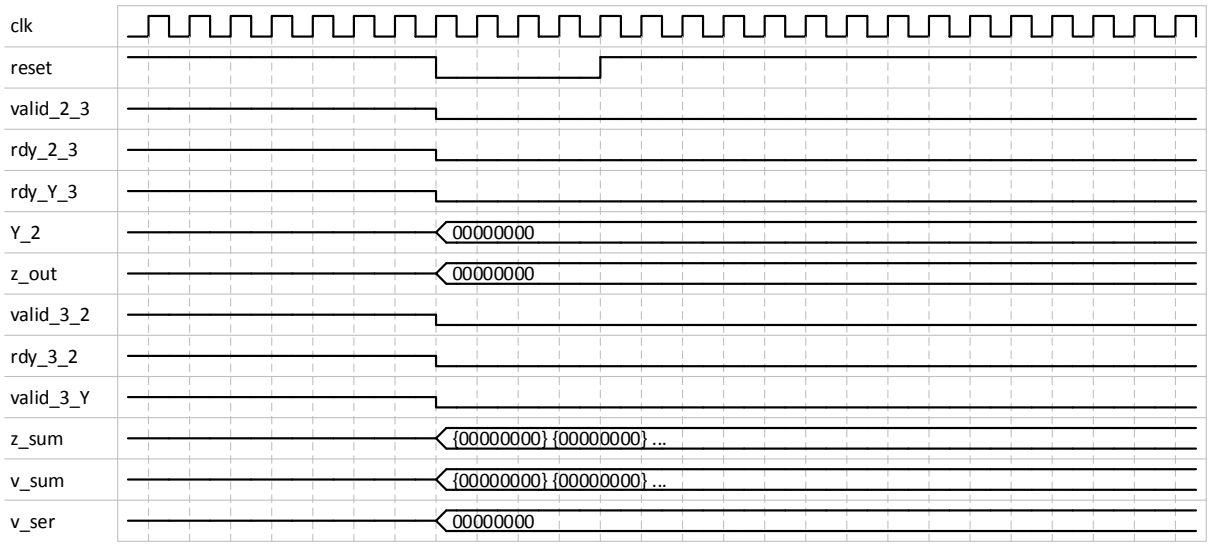
Obr. 18 *Chování rozhraní od vyslání všech dat po příjem dat z cast_1 (komponenta cast_2)*

2.3.4 Pod-komponenta cast_3

Tab. 10 *Rozhraní a generické parametry komponenty cast_3*

Název	Směr (In/Out)	Datový typ	Popis
w_cit_20		positive	Generický parametr inicializován pomocí **B_cit_20**.
w_cit_10		positive	Generický parametr inicializován pomocí **B_cit_10**.
valid_2_3	In	std_logic	Signál z komponenty **cast_2**. Spouští příjem vystavených dat z **cast_2**.
rdy_2_3	In	std_logic	Signál z komponenty **cast_2**. Zastavuje vysílání dat do komponenty **cast_2**.
rdy_Y_3	In	std_logic	Signál z komponenty **cast_Y**. Blokuje vysílání dat do komponenty **cast_Y**.
clk	In	std_logic	Hodinový signál. Reakce na náběžnou hranu.
reset	In	std_logic	Asynchronní reset. Aktivní v logické 0.
Y_2	In	typ_prvek	Sériový vstup dat z komponenty **cast_2**.
z_out	In	typ_prvek	Sériový vstup dat z komponenty **cast_2**.
valid_3_2	Out	std_logic	Signál do komponenty **cast_2**. Označuje vystavení dat z této komponenty.
rdy_3_2	Out	std_logic	Signál do komponenty **cast_2**. Blokuje vysílání dat z komponenty **cast_2**.
valid_3_Y	Out	std_logic	Signál do komponenty **cast_Y**. Označuje vystavení dat z této komponenty.
z_sum	Out	vektor	Paralelní výstup dat do komponenty **cast_Y**.
v_sum	Out	vektor	Paralelní výstup dat do komponenty **cast_Y**.
v_ser	Out	typ_prvek	Sériový výstup dat do komponenty **cast_2**.

Pro uvedení komponenty do známého stavu je potřeba provést reset (viz. *Obr. 19*). Z obrázku si můžeme všimnout, že tato komponenta po shození resetu (logická 1) nenastavuje žádný signál pro oznámení, že je připravena na příjem dat z předchozí komponenty. Tato komponenta po provedeném resetu blokuje komunikaci do doby než následující komponenta **cast_Y** přijme inicializační data z nadřazeného systému a oznámí, že je připravena na přísun dat z této komponenty.

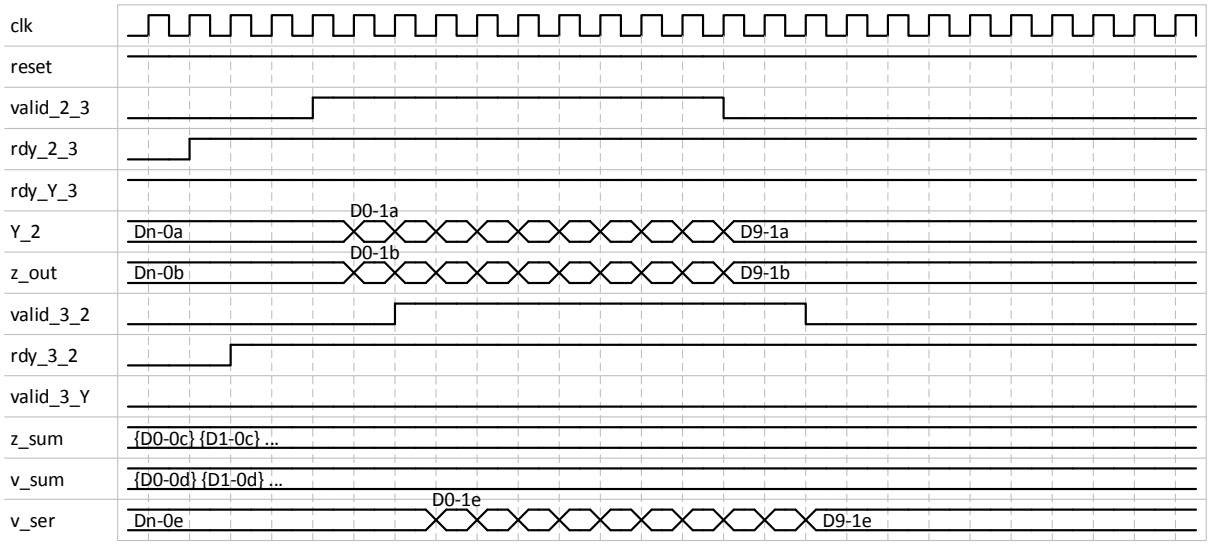


Obr. 19 *Chování rozhraní při resetu (komponenta cast_3)*

Při přijmu dat z předchozí komponenty **cast_2** je prováděno zpracování dat v této komponentě a s malým zpožděním po příjmu prvních dat je výsledek zpracování poslán zpět do komponenty **cast_2** (viz. *Obr. 20* a *Obr. 21*). Data přijata na portu **Y_2** jsou během zpracování vynásobena konstantou **minusJEDNA**, která obsahuje číslo mínus jedna v patřičném formátu čísla s pevnou desetinnou čárkou. Tato úprava vstupních dat zajišťuje, aby mohli být realizovány výpočty z předlohy v programu Matlab:

$$vl = vl + zl - ul \tag{14}$$

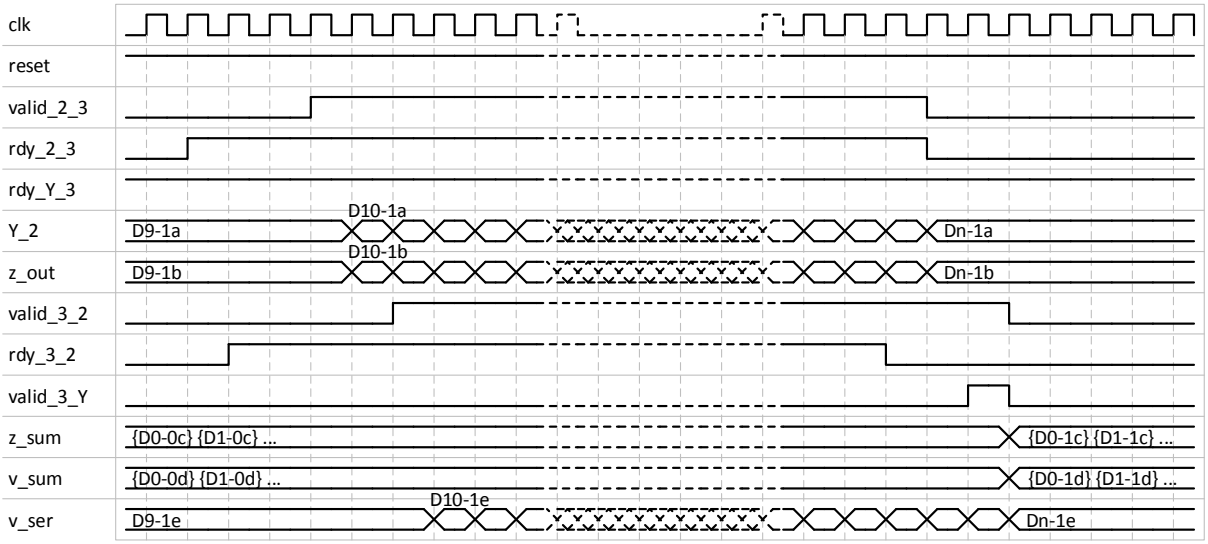
$$vq = vq + zq - uq \tag{15}$$



Obr. 20 *Chování rozhraní při přijmu dat pro lineární omezení z cast_2 a vysílání dat zpět do cast_2 (komponenta cast_3)*

Během vysílání posledního prvku skrze port **v_ser** do komponenty **cast_2** jsou zároveň vyslány data do komponenty **cast_Y** skrze porty **z_sum** a **v_sum** (viz. *Obr. 21*). Data vysílaná portem **z_sum** obsahují sumu všech vektorů přijatých skrze port **z_out**

z komponenty **cast_2**. Komponenta **cast_3** výpočtem sumy vektorů pro výstup na portech vedoucích do komponenty **cast_Y**, také částečně splňuje výpočet (5) z předlohy v programu Matlab.



Obr. 21 *Chování rozhraní při přijmu dat pro kvadratické omezení z cast_2 a vysílání dat do cast_2 a cast_Y (komponenta cast_3)*

3 Simulace a syntéza výsledné implementace

V této pod-kapitole jsou porovnávány výstupy z algoritmu při změnách některých inicializačních konstant. Pro účely simulace byla vytvořena entita, která simuluje výpočty nadřazeného systému. Tato entita může představovat mikroprocesor, který se nachází ve stejném pouzdře jako FPGA. Výpočty této entity jsou prováděny ve stejném formátu čísla s pevnou desetinnou čárkou, proto může docházet k zanášení chyby z této entity již při inicializaci algoritmu.

Dále jsem vytvořil entitu **new_component**, která obsahuje rozhraní kompatibilní s rozhraním Avalon. Popis rozhraní a chování této komponenty je popsáno níže (viz. *3.1*). Skrze Avalon rozhraní slze mapovat signály do sdílené paměti, sdílené sběrnice atd. Tento typ rozhraní je velmi užitečný a velmi všestranný, je však potřeba držet se specifikace aby bylo naše vlastní rozhraní schopné tento standart splnit. Proto jsem se při jeho návrhu držel oficiální specifikace. [1]

3.1 Komponenta new_component

Cílem této komponenty je vytvoření rozhraní, které je schopné komunikovat s nadřazeným systémem jako „Slave“. Pro dosažení tohoto cíle bylo vytvořeno rozhraní (*Tab. 11*), které splňuje požadavky na příjem/vysílání dat pomocí rozhraní „Avalon Memory-Maped Interface“ (Avalon-MM). Tento typ rozhraní se typicky využívá pro komunikaci s mikroprocesorem, pamětí, DMA, čítačem atd. [1]

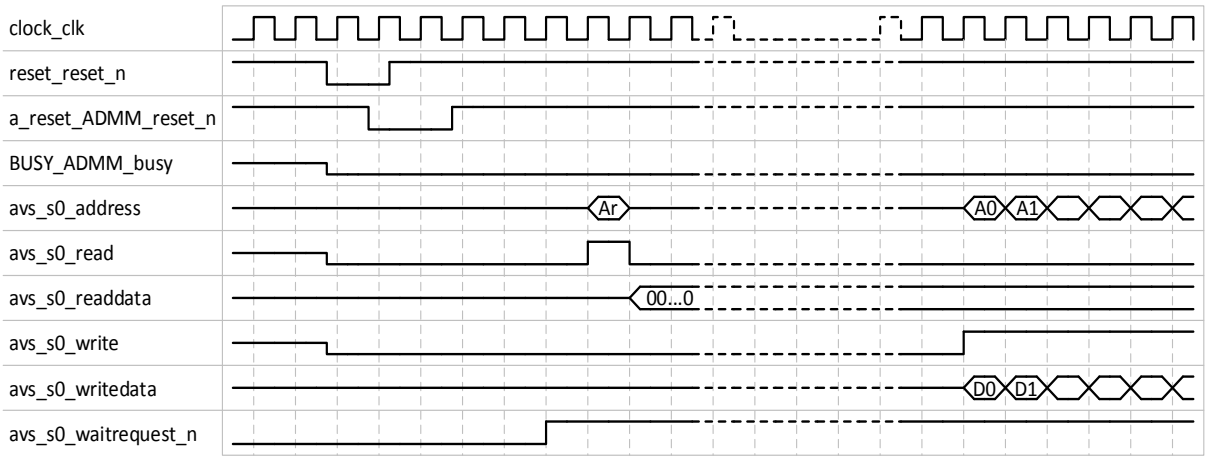
Tab. 11 *Rozhraní a generické parametry komponenty new_component*

Název	Směr (In/Out)	Datový typ	Popis
adres_w		positive	Generický parametr pro nastavení bitové šířky adresy.
bus_w		positive	Generický parametr pro nastavení bitové šířky vstupních/výstupních dat z komponenty.
inter_B_cit_200		positive	Generický parametr pro nastavení parametru **B_cit_200**.
inter_B_cit_145		positive	Generický parametr pro nastavení parametru **B_cit_145**.
inter_B_cit_50		positive	Generický parametr pro nastavení parametru **B_cit_50**.

Název	Směr (In/Out)	Datový typ	Popis
inter_B_cit_20		positive	Generický parametr pro nastavení parametru **B_cit_20.**
inter_B_cit_10		positive	Generický parametr pro nastavení parametru **B_cit_10.**
avs_s0_address	In	typ_adresa[2]	Vstup adresy v rozhraní Avalon.
avs_s0_read	In	std_logic	Vstupní signál pro povolení čtení v rozhraní Avalon.
avs_s0_readdata	Out	typ_data[3]	Výstup dat z rozhraní Avalon.
avs_s0_write	In	std_logic	Vstupní signál pro povolení zápisu v rozhraní Avalon.
avs_s0_writedata	In	typ_data[3]	Vstup dat v rozhraní Avalon.
avs_s0_waitrequest_n	Out	std_logic	Výstupní signál pro pozdržení komunikace v rozhraní Avalon. Aktivní v logické 0.
clock_clk	In	std_logic	Hodinový signál. Reakce na náběžnou hranu.
reset_reset_n	In	std_logic	Asynchronní reset. Aktivní v logické 0.
BUSY_ADMM_busy	Out	std_logic	Výstupní signál pro indikaci zpracovávání dat.
a_reset_ADMM_reset_n	In	std_logic	Asynchronní reset. Aktivní v logické 0.

Tato komponenta obsahuje dva signály pro provedení asynchronního resetu. První z resetovacích signálů **reset_reset_n**, byl zamýšlen pro vykonání resetu při náběhu napájení. Druhý signál **a_reset_ADMM_reset_n**, byl zamýšlen pro ovládání resetu například pomocí tlačítka. Oba tyto signály jsou aktivní v logické 0. Průběhy chování během resetu jsou zobrazeny na obrázku (*Obr. 22*).

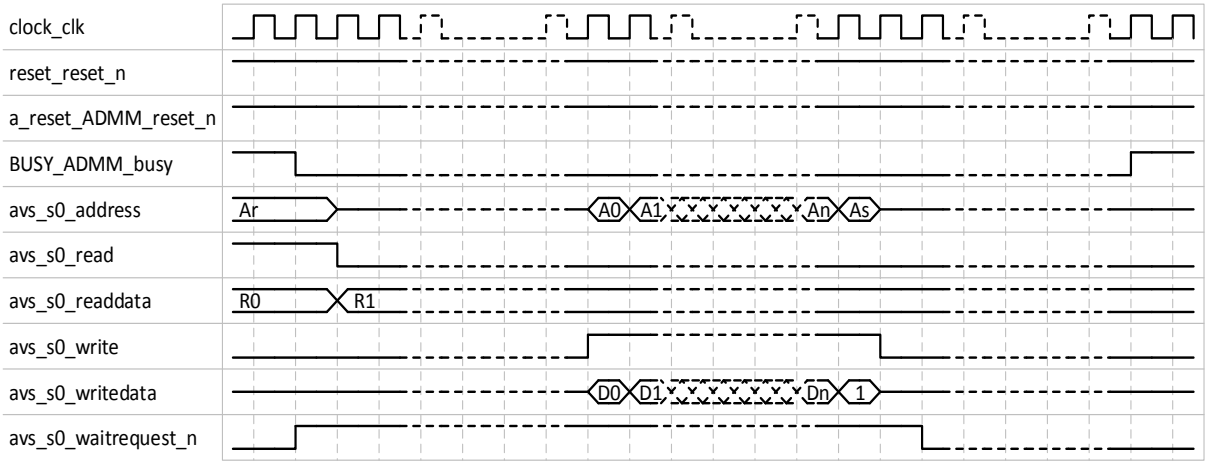
[2] Datový typ std_logic_vector((adres_w – 1) downto 0)
[3] Datový typ std_logic_vector((bus_w – 1) downto 0)



Obr. 22 *Chování během resetu (komponenta new_component)*

Signál **avs_s0_waitrequest_n** označuje, jestli je možné komunikovat s komponentou. Když je tento signál v logické 1, tak je možné z komponenty vyčíst vytavená data a provést inicializaci vstupních dat. Když je tento signál v logické 0, tak to znamená, že algoritmus ještě neskončil zpracovávání dat a je tedy nutné počkat.

V této komponentě je pomocí generického dizajnu implementovaná varianta, kdy je bitová šířka sběrnice pro výstupní a vstupní data větší nebo rovna bitové šířce jednoho prvku (**bus_w >= poc_bitu_init**) a varianta opačná, kdy je bitová šířka sběrnice pro komunikaci menší než bitová šířka jednoho prvku (**bus_w < poc_bitu_init**). Obecný průběh čtení a vysílání dat je zobrazen na obrázku (*Obr. 23*).



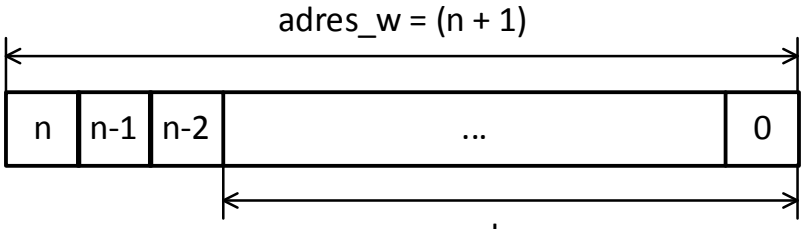
Obr. 23 *Obecný průběh čtení a zápisu dat (komponenta new_component)*

V prvním případě jsou data přenášena celá skrze sběrnici, v případě, že je bitová šířka dat menší než sběrnice, je u výstupních dat rozšířen znaménkový bit, tak aby výsledná data nezměnila svou hodnotu a aby bitová šířka souhlasila s bitovou šířkou sběrnice.

V druhém případě, kdy data mají větší bitovou šířku než má sběrnice, se data vysílají a přijímají po dvou částech. Ve spodní části je přenášen maximální možný počet bitů. V horní

části je přenášen zbytek bitů s případným doplněním bitů do maximálního počtu bitů, jako tomu bylo u varianty, kdy (**bus_w > poc_bitu_init**).

Pomocí adresy **avs_s0_address** se rozhoduje, do jakého registru se má zapisovat případně z něj číst a jaká část dat se má vyslat či přijmout. Způsob adresování je znázorněn na obrázku (*Obr. 24*). Při čtení výstupních dat (*n = 0* a *(n-1) = 0*) musí být adresa rovna 0 (*Ar* v *Obr. 23*). Po zápisu inicializačních dat se pro odstartování algoritmu musí na adresu 1 (při *n = 0* a *(n-1) = 0*) zapsat hodnota 1 (*As* v *Obr. 23*).



A = (bus_w >= poc_bitu_init)

n	n-1	Popis
0	0	Dout; Start
0	1	f_cost
1	0	minus_bc
1	1	fc

A	n-2	Popis
0	0	Součást adresy
0	1	
1	0	Spodní část dat
1	1	Horní část dat

Obr. 24 *Složení adresy*

Signál **BUSY_ADMM_busy** nabývá logické hodnoty 1 při započetí zpracování nových vstupních dat (signál je aktivní). Logické hodnoty 0 nabývá při vystavení výstupních dat z algoritmu (signál není aktivní). Měřením tohoto signálu během jeho aktivní části je tedy možné zjistit dobu od započetí zpracování dat po vystavení dat z algoritmu. Během neaktivní části tohoto signálu dobíhá zpracování poslední iterace algoritmu pod-komponentami **cast_1**, **cast_2** a **cast_3**. Během tohoto času je prostor pro výpočet a inicializaci potřebných vstupních dat pomocí nadřazeného systému. V případě že nadřazený systém nestihne v daném čase zapsat inicializační data a spustit algoritmus a zpracování dat zbylými pod-komponentami je hotové, tak se algoritmus zastaví a čeká na pokyn ke startu algoritmu. V opačném případě, kdy nadřazený systém během neaktivity tohoto signálu dokáže inicializovat a poslat příkaz ke spuštění algoritmu, je signál **avs_s0_waitrequest_n** aktivován (logická 0), ale signál **BUSY_ADMM_busy** stále zůstává neaktivní (logická 0) až do skončení zpracovávání dat z poslední iterace algoritmu zbylými komponentami.

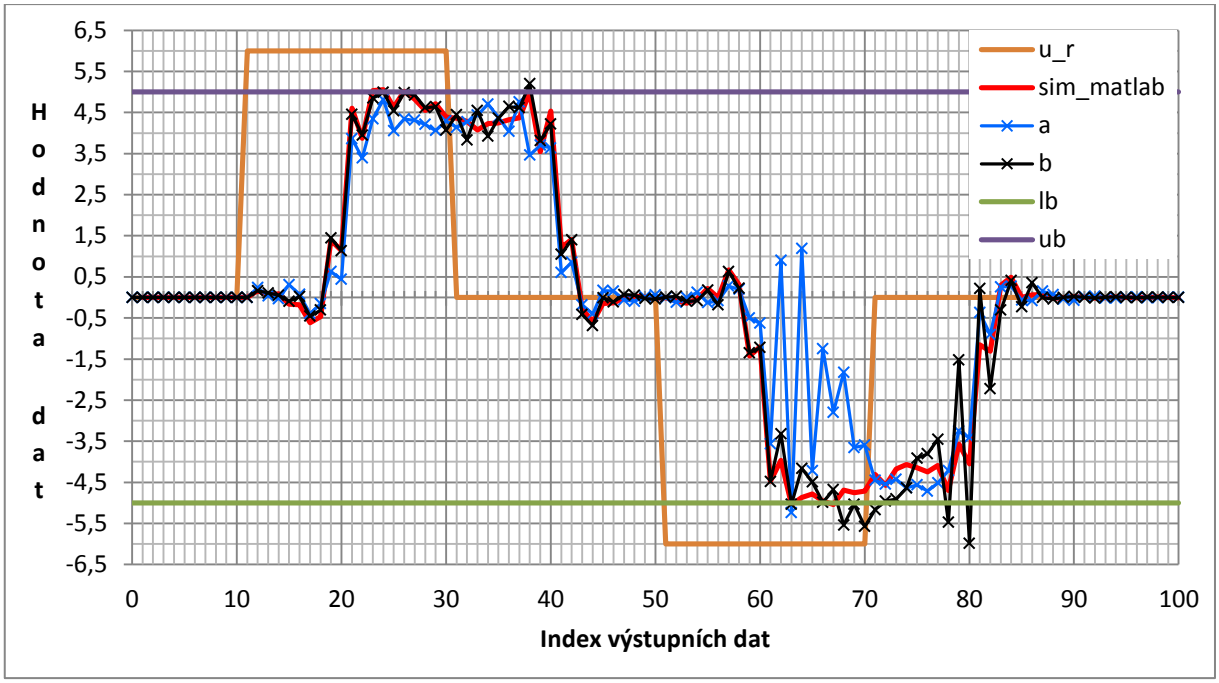
3.2 Porovnání implementace se simulací v programu Matlab

V grafu (*Obr. 25*) je zobrazeno porovnání výsledků simulací při změně konstant pro nastavení frakčních bitů a s tím spojené i bitové šířky prvku (*Tab. 12*). Ostatní generické parametry a hodnoty konstant konfigurované v souboru **pkg_init_positive.vhd** jsou stejné.

Tab. 12 *Hodnoty parametrů pro porovnání při změně počtu frakčních bitů (Q_init)*

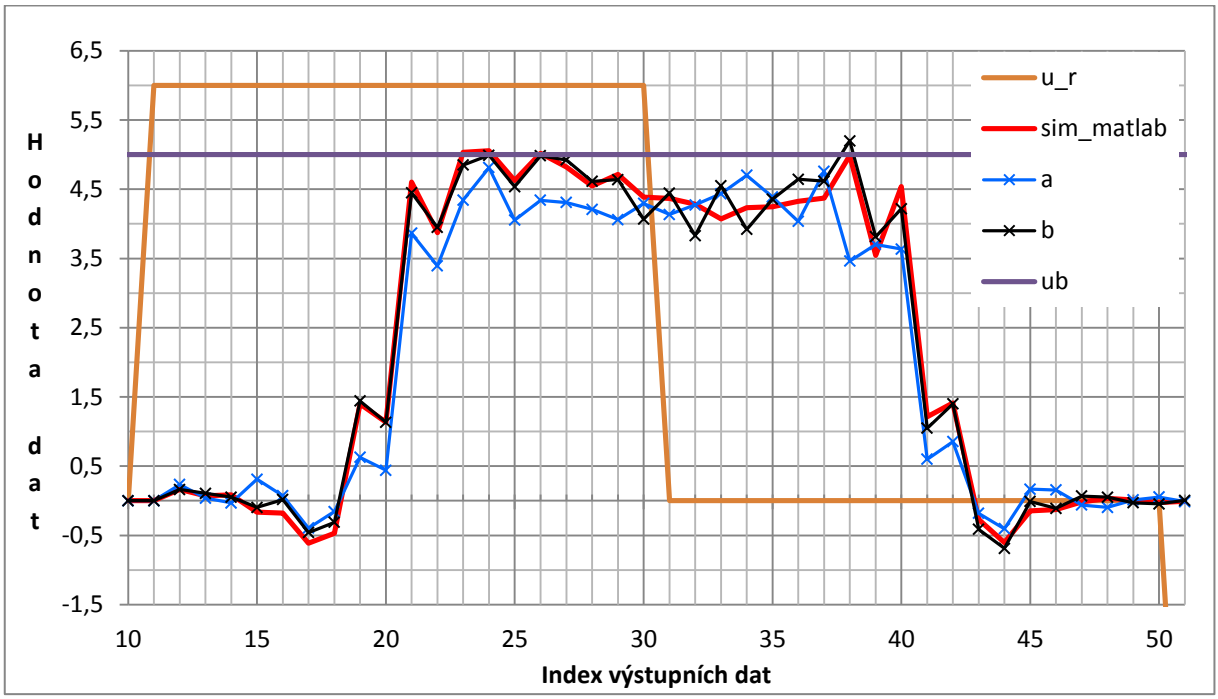
Q_init	poc_bitu_init	Název průběhu
10	32	a
20	42	b

V tomto grafu jsou také zobrazeny hranice lineárního omezení reprezentované průběhy *lb* a *ub*. Dále je v grafu vykreslen použitý vstupní signál *u_r* z hlediska celého systému a pro srovnání je zde průběh výstupních dat ze simulace v programu Matlab (průběh **sim_matlab**). Maximální počet iterací algoritmu byl nastaven na hodnotu 50.



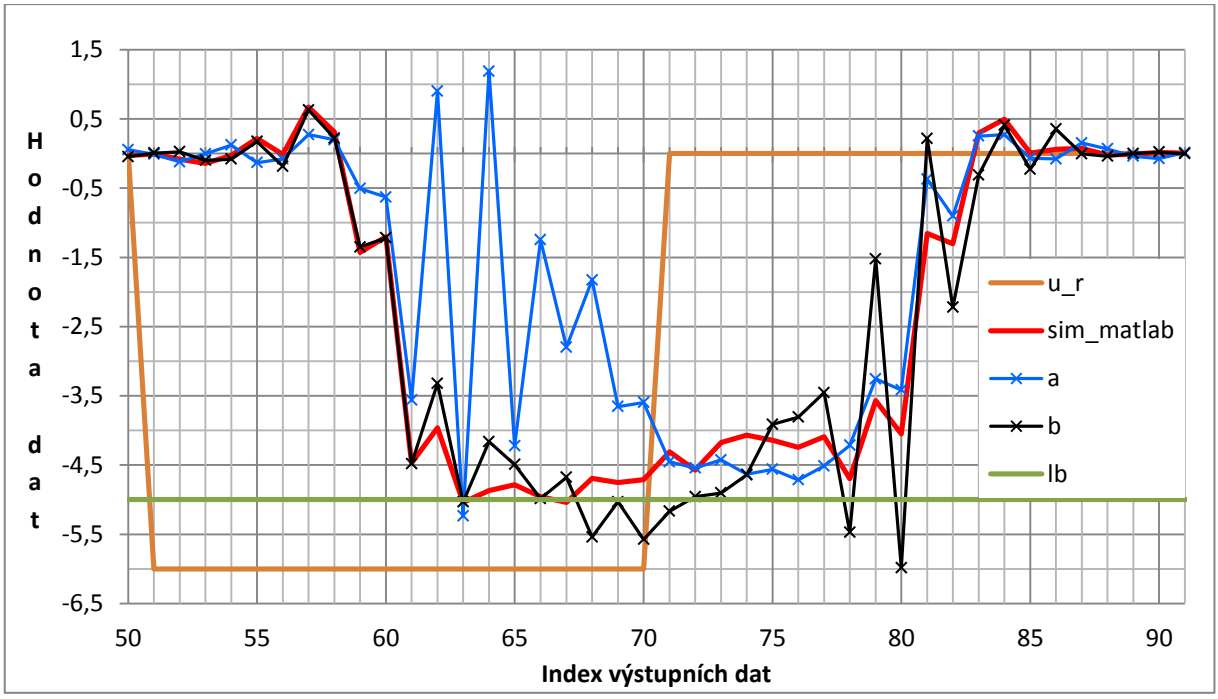
Obr. 25 *Porovnání výstupu dat při změně počtu frakčních bitů*

Z grafu (*Obr. 25*) jde vidět, že počet frakčních bitů má velký dopad na výpočet. V detailním pohledu v grafu (*Obr. 26*) si můžeme všimnout, že průběh **a** dosahuje v průměru nižších hodnot než ostatní průběhy a tím pádem se téměř nepřiblíží k maximální hranici lineárního omezení *ub*.



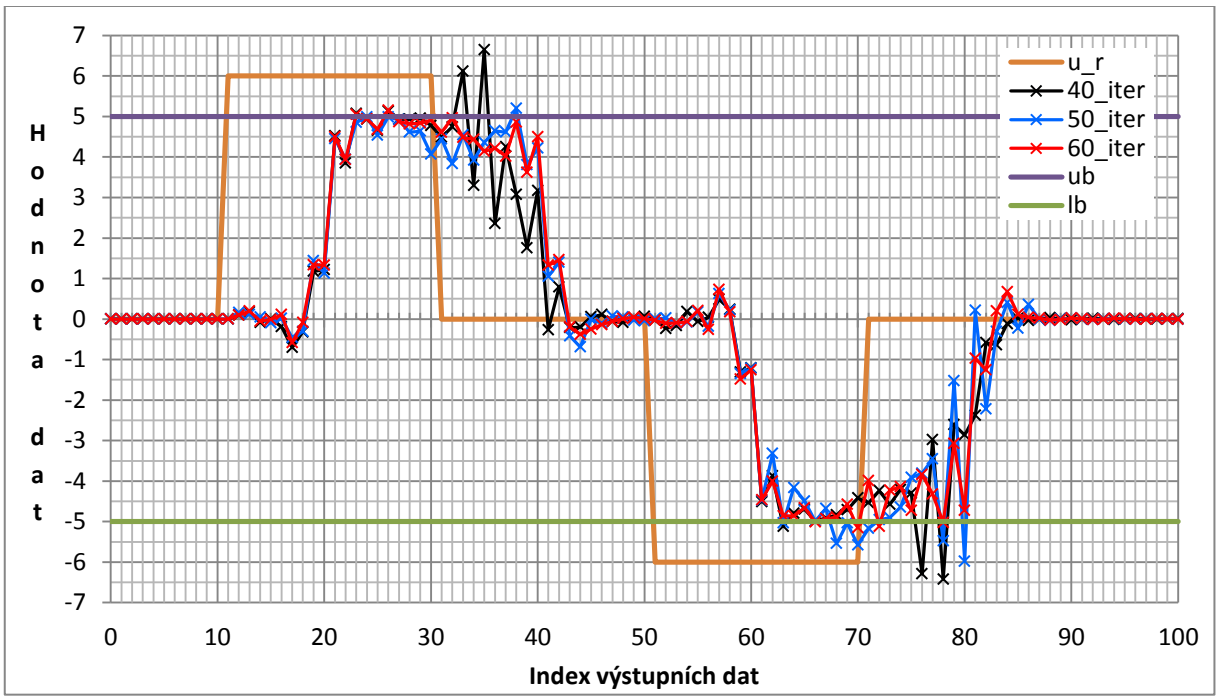
Obr. 26 *Porovnání výstupu dat při změně počtu frakčních bitů (detail na intervalu 10 až 51)*

V grafu (*Obr. 27*) jde vidět, že průběh **a** opět nedosahuje hranice lineárního omezení (v tomto případě minimální hranice *ub*). Z grafu jde také vidět, že průběhy **a** a **b** vykazují známky oscilace, což je způsobeno zmenšenou přesností výpočtu z důvodu zmenšení počtu frakčních bitů. Jak už bylo zmíněno, testovací entita, která provádí výpočty inicializačních dat pro chod algoritmu, také používá stejný formát čísla s pevnou desetinnou čárkou a tím se chyby při výpočtu těchto dat promítnou i do samotného výpočtu algoritmu.



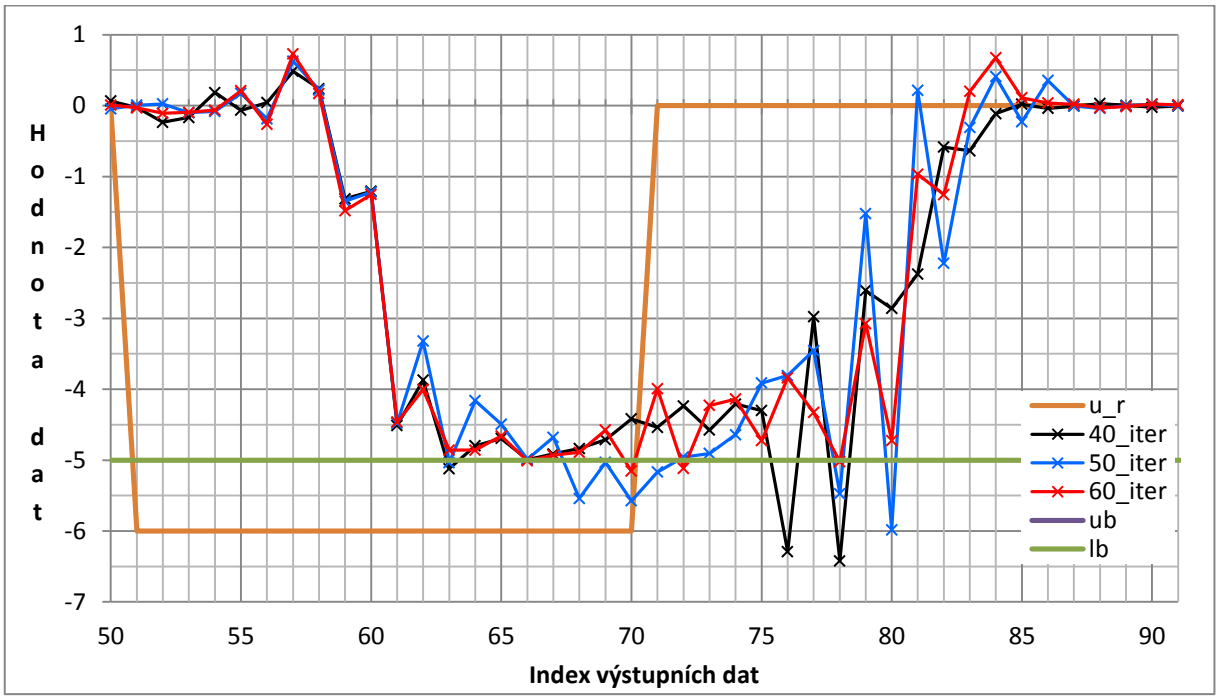
Obr. 27 *Porovnání výstupu dat při změně počtu frakčních bitů (detail na intervalu 50 až 91)*

Pro porovnání výstupních dat při změně počtu iterací algoritmu jsou v grafech (*Obr. 28* a *Obr. 29*) vyobrazeny průběhy výstupních dat se stejnou konfigurací pro konstanty **poc_bitu_init** a **Q_init**. Kdy počet bitů (**poc_bitu_init**) je roven 42b a z toho je 20b pro realizaci frakčních bitů (**Q_init**). Měněným parametrem u těchto průběhů byl maximální počet iterací algoritmu reprezentován konstantou **poc_iter_init** v souboru **pkg_init_positive.vhd**, tento parametr byl inicializován hodnotou 40 (průběh 40_iter), 50 (průběh 50_iter) a nakonec hodnotou 60 (průběh 60_iter).



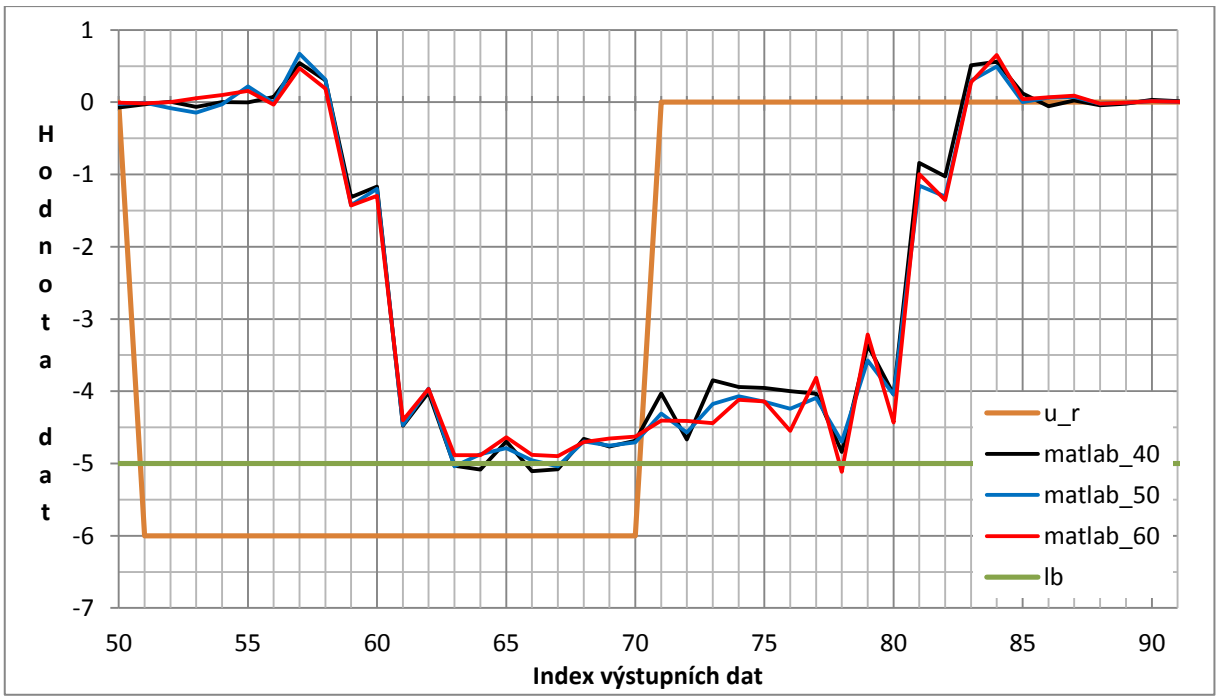
Obr. 28 *Porovnání výstupu dat při změně počtu iterací algoritmu s Q_init = 20*

Při detailnějším pohledu na průběhy (*Obr. 29*) lze vypozorovat, že při menším počtu iterací potřebných pro výstup dat dochází k větším překmitům hranic pro lineární omezení.



Obr. 29 *Porovnání výstupu dat při změně počtu iterací algoritmu s Q_init = 20 (detail na intervalu 50 až 91)*

Pro porovnání jsem provedl simulace pro stejné nastavení iterací algoritmu i v programu Matlab (*Obr. 30*). Výsledky z programu Matlab se při změně iterací od sebe výrazně nelišili jako tomu je u výstupů ze simulace v grafu (*Obr. 29*).



Obr. 30 *Výstupní data simulace v programu Matlab při změně počtu iterací algoritmu (detail na intervalu 50 až 91)*

Z uvedených simulačních dat, můžeme vypozorovat, že například při zvětšování počtu frakčních bitů se výstup z implementovaného algoritmu blíží výsledkům ze simulace

v programu Matlab. Zlepšení těchto průběhů můžeme zařídit tím, že například zvětšíme počet iterací algoritmu pro zpracování dat. Z výsledků provedených simulací je zřejmé, že je důležité dbát na správnou volbu konfiguračních konstant. Protože tyto konstanty silně ovlivňují výsledné zpracování dat. Výsledek lze ovlivnit i jinými konstantami, zde byli předvedeny pouze dva způsoby ovlivnění výsledku.

Je zřejmé, že každá konfigurace má své plusy a mínusy. Například zvětšování počtu bitů potřebných pro zobrazení jednoho prvku má pozitivní vliv na výstupní data, ale také má negativní vliv využití zdrojů při implementaci do FPGA. Například se nám může stát, že pro realizaci algoritmu při nešťastném zvolení konfiguračních konstant budeme potřebovat tolik zdrojů, že nebudeme schopni nalézt zařízení, do kterého bychom tento algoritmus dokázali zaintegrovat. Nebo najdeme zařízení, které dokáže pojmout náš algoritmus, ale kvůli malému místu nebude prostor pro optimální rozmístění a bude nutné například vést spoustu signálů napříč zařízením a kvůli tomu nebudeme moci dosahovat větších rychlostí kvůli správnému dodržení souběhu signálů.

Na délce zpracování dat algoritmem se výrazně promítá nastavení počtu iterací, počtu vektorů atd. Některé varianty pro změny konstant **poc_iter_init** a **poc_iter_HUANG_init** jsou vypsány v tabulce (*Tab. 13*). Výsledky simulací zde již nebudou zobrazovány, protože cílem těchto simulací je porovnání délek výpočtů. Simulace byly provedeny při nastavení **poc_prvku_init** = 10, **poc_vektoru_init** = 20 a **poc_vek_mat_init** = 145. Měněnými konstantami byly **poc_iter_init** a **po_iter_HUANG_init**. Další konstanty nemají vliv na výsledný počet taktů potřebných pro výpočet.

Pro zjištění délky výpočtu byl použit signál **BUSY_ADMM_busy**, kdy se v simulaci při aktivní a neaktivní fázi signálu inkrementovali korespondující čítače vždy při náběžné hraně hodinového signálu. Výsledek čítačů byl ověřen v programu ModelSim pomocí kurzorů.

Tab. 13 *Porovnání délky trvání implementovaného algoritmu*

poc_iter_init	poc_iter_HUANG_init	Počet taktů v části		Počet taktů celkem
		Aktivní	Neaktivní	
40	15	102353	2610	104963
40	20	130628	3335	133963
40	25	158903	4060	162963
50	15	128593	2610	131203
50	20	164118	3335	167453
50	25	199643	4060	203703

poc_iter_init	poc_iter_HUANG_init	Počet taktů v části		Počet taktů celkem
		Aktivní	Neaktivní	
60	15	154833	2610	157443
60	20	197608	3335	200943
60	25	240383	4060	244443

Z výsledků délky trvání algoritmu lze vidět, že délka doby signálu **BUSY_ADMM_busy** v neaktivní části (logická 0) se nemění při změně konstanty **poc_iter_init**, jak už bylo zmíněno, tak během této neaktivní části probíhá čtení a zápis dat z respektive do algoritmu. Je zřejmé, že počty iterací v algoritmu mají velký vliv jak na dobu potřebnou ke zpracování, tak na výstupní data. Pro získání nejlepšího poměru mezi výkonem a časovými nároky je tedy třeba experimentálně zjistit nejlepší nastavení pro konfigurační konstanty.

V této pod-kapitole byly ukázány jen některé konfigurace konstant algoritmu a jejich dopad na jeho chování. V následující kapitole se budou probírat výsledky syntézy při změnách některých konfiguračních konstant a jejich dopad na potřebné zdroje.

3.3 Výsledky syntézy

V této kapitole jsou vypsány výsledky syntéz při změně bitové šířky prvku (konstanta **poc_biru_init**) a změně konfigurace syntézy. Veškeré kompilace byly provedeny se stejným nastavením konfiguračních konstant (*Tab. 14*) vyjma konstant **poc_bitu_init** a **Q_init**, které byli měněným parametrem.

Tab. 14 *Nastavení konfiguračních konstant pro syntézu*

Název	Hodnota
poc_prvku_init	10
poc_vektoru_init	20
poc_iter_init	50
poc_iter_HUANG_init	20
poc_vek_mat_init	145

Pro kompilaci byla využita komponenta **new_component**, která vytváří obálku implementace, bez této komponenty by nebylo možné provést kompilaci pro samostatnou implementaci, protože počet pinů by byl moc velký a kompilace by skončila z důvodu toho, že nemůže najít zařízení s dostatečným počtem pinů. Pro možnost porovnat jednotlivé výstupy syntézy, byly veškeré kompilace prováděny pro zařízení **5CGTFD9E5F35C7N** což

je FPGA z rodiny Cyclone V, toto FPGA můžeme najít například ve vývojové sadě „Cyclone V GT FPGA Development Board“. Výčet vlastností tohoto FPGA je uveden v tabulce (*Tab. 15*), tato tabulka byla převzata z referenčního manuálu pro zmíněnou vývojovou sadu. Případné další informace lze najít v tomto manuálu. [8]

Tab. 15 *Vlastnosti Cyclone V GT FPGA*

Zdroje	**5CGTFD9E5F35C7N**
LE	301
ALM	113560
Registr	454240
Paměť M10K (kb)	12200
Paměť MLAB (kb)	1717
Násobička (18b × 18b)	684
PLL	8
Vysílač (6 Gb/s)	12

Pro první sérii kompilací byla zvolena bitová šířka 32 a počet frakčních bitů 10. Pro tuto konfiguraci byly provedeny kompilace, které byly konfigurovány pro zaměření na balanc, výkon a oblast (viz. *Tab. 16*). U konfigurací se zaměřením na výkon a oblast byla zvolena agresivní strategie, žádné další rozšiřující nastavení nebylo přidáváno.

Tab. 16 *Výstupy syntézy (poc_bitu_init = 32, Q_init = 10)*

Název sledovaného zdroje	**Počet využitých zdrojů**		
	Balanc	**Výkon**	**Oblast**
Registry	113306	115158	113240
ALM	44621	41394	44576
DSP blok	205	205	205
Blok paměti (b)	1717	1717	1717
RAM blok	16	16	16

Druhá série kompilací byla provedena při bitové šířce 42 a počtu frakčních bitů 20. Pro tyto kompilace byly zvoleny stejné konfigurace syntéz jako u první série. Výsledky těchto kompilací jsou v tabulce (*Tab. 17*).

Tab. 17 *Výstupy syntézy (poc_bitu_init = 42, Q_init = 20)*

Název sledovaného zdroje	**Počet využitých zdrojů**		
	Balanc	**Výkon**	**Oblast**
Registry	175760	178290	175676
ALM	66836	62500	66722
DSP blok	272	272	272
Blok paměti (b)	2463	2463	2463
RAM blok	20	20	20

Z uvedených dat lze vidět značné rozdíly ve využití zdrojů při změně konfiguračních konstant, dále může pozorovat dopad konfigurace kompilace na výsledné využití zdrojů zařízení. V případě kompilace se zaměřením na výkon můžeme pozorovat výrazné zvýšení využití registrů na úkor ALM oproti kompilaci, která se zaměřuje na zmenšení využité oblasti. Toto zvýšení využití registrů má za následek zlepšení časování u souběhu signálů a kvůli tomu je pak možné používat vyšší frekvence pro hodinový signál. Samozřejmě tím pádem náš dizajn zabírá větší plochu, protože tyto registry jsou součástí ALM a mi díky tomu některé ALM bloky už nemůžeme naplno využít. Naproti tomu při zaměření na využití co nejmenší oblasti se snažíme o co nevětší využití jednotlivých bloků ALM na co nejmenší ploše. [9]

Závěr

Základní princip prediktivních algoritmů pro řízení a příklad jejich dělení je naznačen v první části (viz. Kapitola 1) této práce. Jelikož pro plné porozumění jednotlivým prediktivním algoritmů je potřeba mít hluboké znalosti pro řešení optimalizačních problémů, což je matematicky velmi složitá oblast, kde je v současné době prováděn značný výzkum. Tak z tohoto důvodu v této části byly jen zmíněny některé algoritmy, které se mohou použít pro řízení motorů, a byl zde spíše kladen důraz na přiblížení principu, na kterém pracuje algoritmus ADMM.

Cílem této práce bylo vytvoření návrhu implementace algoritmu ADMM v obvodech programovatelné logiky při respektování moderních trendů a nástrojů pro návrh číslicových systémů a výsledná implementace měla být použita k řízení pohonu v reálné aplikaci. Díky značným problémům při návrhu implementace a jejím ladění, nebylo možné z časových důvodů provést integraci výsledné implementace do FPGA. Nicméně byl vytvořen prototyp, který je popisován v kapitole (2). Tato kapitola popisuje chování jednotlivých pod-komponent ze kterých se výsledný návrh prototypu skládá aby bylo možné tento návrh použít a případně modifikovat při následných experimentech.

Pro účely spojení navržené implementace v FPGA s integrovaným procesorem na jednom čipu byla vytvořena komponenta „new_component“, která díky implementovanému rozhraní Avalo-MM toto spojení dokáže zprostředkovat. Spojení FPGA a integrovaného procesoru je moderním způsobem při návrhu číslicových systémů. Popisu chování této komponenty je věnována pod-kapitola (viz. 3.1). [1]

V této práci byly dále porovnány výsledky simulací návrhu se simulacemi z programu Matlab (viz. 3.2), který byl předlohou pro samotný návrh implementace (viz. Příloha I. a II.). V této práci byly také předvedeny dopady změn některých konstant na výslednou délku zpracování dat a výsledky syntézy implementace (viz. 3.2 a 3.3).

Z tabulky (Tab. 13) jde vidět, že při použité konfiguraci implementace, je prakticky nemožné tuto implementaci použít při reálném řízení pohonu, protože potřebná frekvence pro provedení zpracování dat by musela být příliš velká. Jedním z řešení tohoto problému může být zvolení jiné konfigurace, nebo v rámci experimentů uzpůsobit vzorkovací frekvenci řízené veličiny maximální době zpracování dat implementací. Dalším řešením je optimalizace implementovaného návrhu, kde hlavním kandidátem pro tuto optimalizaci je komponenta cast_2. U této komponenty nebyla provedena dostatečná atomizace při jejím návrhu, díky tomu se stala málo čitelnou a z toho důvodu byla velkým zdrojem chyb při vývoji.

Použitá literatura

[1] INTEL CORPORATION. *Avalon Interface Specifications: Updated for Intel Quartus Prime Desing Suite: 17.1* [online]. MNL-AVABUSREF | 2018.03.22. [cit. 2018-04-05]. Dostupné z: https://www.altera.com/documentation/nik1412467993397.html

[2] LINDER, A. et al. *Model-Based Predictive Control of Electric Drivers* [online]. München: Cuvillier Verlag, 2010 [cit. 2018-05-10]. ISBN 3869553987. Dostupné z: http://www.cuvillier.ch/flycms/de/html/30/-UickI3zKPS73dU4=/Buchdetails.html

[3] HUBÁČEK, J. *Prediktivní řízení s omezením vstupních a výstupních veličin*. Zlín: 2008. Diplomová práce. Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, Ústav řízení procesů [cit. 2018-01-13]. Dostupné z: http://digilib.k.utb.cz/bitstream/handle/10563/6645/hub%C3%A1%C4%8Dek_2008_dp.pdf?sequence=1&isAllowed=y

[4] JEREZ, J. L. et al. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transaction on: Automatic Control* [online]. IEEE, 2014, **59** (12), 3238-51 [cit. 2017-12-02]. ISSN 1558-2523. Dostupné z: https://ieeexplore.ieee.org/document/6882832/

[5] MYNÁŘ, B. Z. *Algoritmy prediktivního řízení elektrických pohonů*. Brno: 2013. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky [cit. 2018-01-13]. Dostupné z: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=82649

[6] MÁCHA, V. V. ŠMÍDL a Z. PEROUTKA. Implementation of Predictive Spectrum Control of a LC filter using ADMM. *IEEE International Symposium on: Predictive Control of Electrical Drivers and Power Elektronics …* [online]. Pilsen: IEEE, 2017 [cit. 2018-04-20]. ISBN 979-1-5386-0507-3. Dostupné z: https://ieeexplore.ieee.org/abstract/document/8071272/

[7] PINKER, J. a M. POUPA. *Číslicové systémy a jazyk VHDL*. Praha: BEN-technická literatura, 2006. ISBN 80-7300-198-5.

[8] ALTERA CORPORATION. *Cyclone V GT FPGA Development Board: Reference Manual* [online]. MNL-01078-1.3. San Jose: 2017 [cit. 2018-05-20]. Dostupné z: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_cvgt_fpga_dev_board.pdf

[9] ALTERA CORPORATION. *White Paper: FPGA Architecture* [online]. WP-01003-1.0. San Jose: 2006 [cit. 2018-05-18]. Dostupné z: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01003.pdf

Příloha I. Úryvek ze simulace v programu Matlab (ADMM)

```
% ADMM solver
 for j = 1:iter_max
 % u update
  tmp = rho*(zl + vl) + sum(rho*(zq + vq),2);
  u = A_cost_inv*(f_cost + tmp);

 % Linear constraints
  ul = alp*u + (1-alp)*zl;                    % over relaxation
  zl = min(ub, max(lb,ul - vl));              % z_l update
  vl = vl + zl - ul;                          % v_l update

 % Quadratic constraints
  uq = alp*u*ones(1,m*nq) + (1-alp)*zq;       % over relaxation
  for k = 1:nq                                % z_q update
     ind = k:k+m-1;
     zq(:,k) = Huang(uq(:,ind),vq(:,ind),Ac{k},fc{k},-bc{k},...
                  E_vec{k},EI_vec{k},ED_val{k},E_val{k});
  end
  vq = vq + zq - uq;                          % v_q update
end
```

Příloha II. Úryvek ze simulace v programu Matlab (Huang)

```
%% Huang - bisection
function z_out = Huang(u,v,A,f,b,E_vec,EI_vec,ED_val,E_val)
   u_0 = u - v;
   if u_0'*A*u_0 -2*f'*u_0 < b
      z_out = u_0;
   else
    % Transformation to coordinates system given by eigen vectors
      z_out = u;
      len = length(E_val);
      u_0 = EI_vec*u_0(1:len);
      f = EI_vec*f(1:len);

    % Bisection
      mu_low = -1/max(E_val);
      mu_high = 100;
      for i =1:20
         mu = (mu_low + mu_high)/2;
         z = (u_0 + mu*f)./(1 + mu*E_val);
         if z'*ED_val*z - 2*f'*z - b > 0
            mu_low = mu;
         else
            mu_high = mu;
         end
      end
      z_out(1:len) = E_vec*z;
   end
end
```