VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ÚLOHY S RŮZNÝM STUPNĚM DŮLEŽITOSTI PŘI ŘÍZENÍ MOTORŮ NA PLATFORMĚ ZYNQ

MIXED CRITICALITIES IN MOTOR CONTROL APPLICATIONS ON ZYNQ PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. David Pamánek

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. Petr Blaha, Ph.D.

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Kybernetika, automatizace a měření**
Ústav automatizace a měřicí techniky

Student: Bc. David Pamánek ***ID:*** 146921
Ročník: 2 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Úlohy s různým stupněm důležitosti při řízení motorů na platformě Zynq

POKYNY PRO VYPRACOVÁNÍ:

1. Nastudujte vektorové řízení synchronních elektrických motorů s permanentními magnety.
2. Seznamte se s SoC Zynq od firmy Xilinx, jeho periferiemi a sběrnicí AXI.
3. Seznamte se s vývojovým prostředím Vivado a SDK od firmy Xilinx pro programování procesorů Zynq.
4. Naprogramujte obsluhu periferií potřebných pro řízení PMS motorů.
5. Realizujte řídicí algoritmus a jednoduché uživatelské rozhraní s ohledem na stupeň důležitosti jednotlivých úloh.
6. Vytvořený software ověřte na vývojovém kitu s PMS motorem.

DOPORUČENÁ LITERATURA:

[1] Sul, S.K.: Control of Electric Machine Drive Systems. February 2011, Wiley-IEEE Press. ISBN: 978-0-4-0-59079-9.

Firemní literatura firmy Xilinx, další dle doporučení vedoucího.

Termín zadání: 8.2.2016 ***Termín odevzdání:*** 16.5.2016

Vedoucí práce: doc. Ing. Petr Blaha, Ph.D.
Konzultant diplomové práce:

doc. Ing. Václav Jirsík, CSc., *předseda oborové rady*

Abstrakt

Tato práce se zabývá problematikou vektorového řízení PMS motorů s využitím vývojové desky ZedBoard od firmy Xilinx, která obsahuje mikročip Zynq-7000. Dále je zde popsána práce s vývojovým prostředím Vivado a jeho součástmi. Ve zbylé části práce je popsána tvorba jednotlivých komponent v prostředí Vivado, které jsou následně spojeny do výsledné aplikace pro demonstraci vektorového řízení malého PMS motoru.

Klíčová slova

PMS motor, Vektorové řízení, Xilinx, Zynq, ZedBoard, Vivado, AXI sběrnice, System on Chip, VHDL.

Abstract

This thesis contains introduction to PMS motor control using development board ZedBoard with Xilinx Zynq-7000 SoC. After that, there is a description of development environment Vivado and other modules. Finally, it contains description or created modules in Vivado environment which were combined together with peripheral drivers to demonstrate field oriented motor control algorithm of small PMS motor.

Keywords

PMS motor, Field Oriented Control, Xilinx, Zynq, ZedBoard, Vivado, AXI bus, System on Chip, VHDL.

Bibliografická citace

PAMÁNEK, D. *Úlohy s různým stupněm důležitosti při řízení motorů na platformě Zynq*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 70 s. Vedoucí diplomové práce doc. Ing. Petr Blaha, Ph.D..

Prohlášení

Prohlašuji, že svou diplomovou práci na téma „Úlohy s různým stupněm důležitosti při řízení motorů na platformě Zynq“ jsem vypracoval 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 uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom 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.

V Brně ……. ……...

(podpis autora)

Poděkování

Především bych tímto rád poděkoval vedoucímu diplomové práce panu doc. Ing. Petru Blahovi Ph.D. za odborné vedení, velmi cenné rady a věnovaný čas během vypracování diplomové práce. Také bych chtěl poděkovat rodině a přátelům za morální podporu.

Obsah

1 ÚVOD

Elektrické pohonné jednotky mají v dnešní době nenahraditelnou pozici. S elektromotory se setkáváme snad ve všech odvětvích průmyslu. Příkladem může být automobilový průmysl, strojní průmysl, ale také komerční sféra. Zatímco dříve se k řízení využívala analogová technika, většina dnešních aplikací těchto pohonů počítá s možností řízení pomocí digitálních obvodů. Digitální řízení přináší nové možnosti využití těchto motorů.

Cílem této práce bude vytvoření softwarového rozhraní mezi PMS motorem a hradlovým polem. Vytvořený software bude obsahovat komponenty, které budou schopny ovládat elementární funkce hardwaru potřebné pro zpětnovazební řízení elektromotorů. Takto vytvořené rozhraní pak půjde snadno využívat pro jednoduché testování a implementaci řídicích algoritmů. Existuje sice celá řada vývojových desek i s řídicím softwarem, ale u každé se najde nějaká nevýhoda, která znesnadňuje práci s ní. Hlavním důvodem vzniku této práce je tedy vytvoření vlastního řešení, které bude možno nadále rozvíjet, rozšiřovat o nové funkce a ladit.

Zadání práce vyžaduje seznámení se s vektorovým řízením elektrických pohonů, je tedy potřeba si nastudovat princip samotného pohonu, všechny hardwarové prostředky potřebné pro jeho ovládání. Této části se věnuje druhá kapitola práce. Ve třetí kapitole je nastíněna problematika systémů, obsahujících řešení více úloh s různým stupněm důležitosti. Čtvrtá kapitola se věnuje vektorovému řízení elektrických pohonů, kde je popsán samotný princip tohoto způsobu řízení a vysvětleny nezbytné transformace souřadnicového systému, které jsou s vektorovým řízením spojeny.

Další částí zadání je seznámení se s SoC Zynq od firmy Xilinx a vývojovou deskou ZedBoard, která tento čip obsahuje. Této části se věnuje začátek páté kapitoly, kde jsou uvedená zařízení popsána včetně AXI sběrnice, se kterou je funkčnost této platformy úzce spjata. Zbytek páté kapitoly se zabývá dalším bodem zadání, kterým je vývojové prostředí Vivado a SDK. Jsou zde popsány všechny využívané součásti tohoto softwaru a ukázána práce s nimi od tvorby projektů, přes práci s editorem vlastních IP bloků až po práci se simulátorem.

Zbytek práce je zaměřen na praktickou část, kde jsou v šesté kapitole popsány jednotlivé komponenty vytvořené v jazyce VHDL, které jsou potřebné pro sestavení aplikace pro ovládání hardwarových součástí elektrického pohonu a samotné vektorové řízení. V sedmé kapitole je popsán postup při testování vytvořených komponent a závěrečná osmá kapitola obsahuje postup při vytváření již zmíněné výsledné aplikace od sestavení projektu, přes řešení nezbytných kroků pro její zprovoznění až po návrh regulátorů.

2 ELEKTRICKÉ POHONY

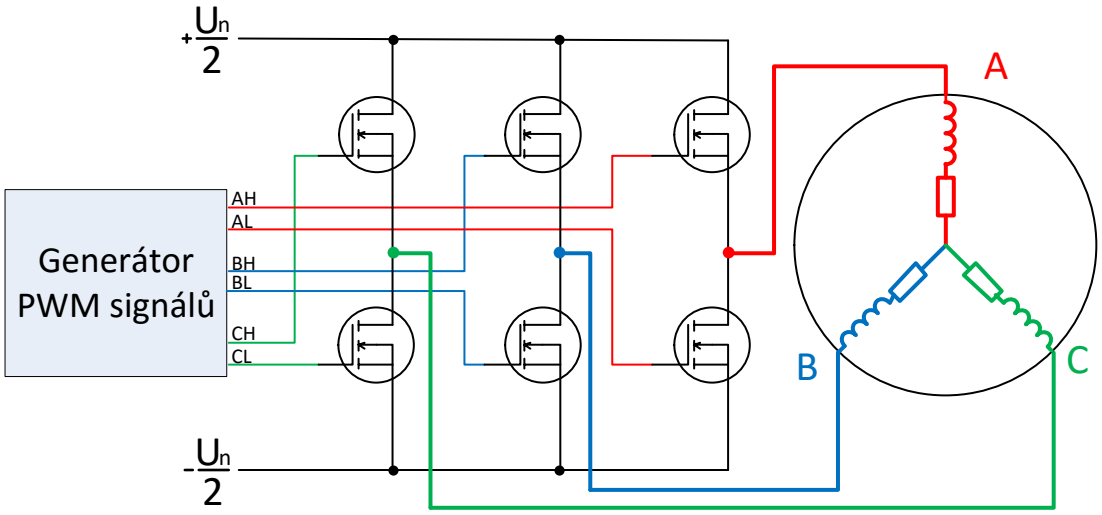
Pod pojmem elektrický pohon rozumíme stroj, který převádí elektrickou energii na mechanickou nebo naopak. V prvním případě tedy pracuje v režimu motoru, ve druhém jako generátor elektrické energie. Základní rozdělení elektrických pohonů podle typu jeho napájení je na stejnosměrné a střídavé. Z historického hlediska jsou stejnosměrné motory starší, postupem času byly ovšem postupně vytlačovány střídavými pohony, které nabízejí celou řadu výhod, které s sebou přináší absence kartáčového komutátoru, jako je například vyšší životnost a spolehlivost, nižší náklady na údržbu a mimo to také menší rozměry a váha při stejném výkonu. Nevýhodou střídavých pohonů je podstatně složitější způsob řízení, který ovšem s rozvojem výkonové elektroniky a teorie řízení hraje čím dál tím menší roli, a proto nacházejí v dnešní době střídavé pohony široké uplatnění téměř ve všech odvětvích průmyslu.

Střídavé pohony se dále dělí na synchronní a asynchronní, podle toho, jestli se rotor otáčí stejnou rychlostí, tedy synchronně, jako točivé magnetické pole, které generujeme ve statoru. Synchronních pohonů existuje celá řada různých typů, které se liší svou konstrukcí a jedním z nich je synchronní motor s permanentními magnety, který se označuje jako PMS, což je zkratka anglických výrazů *Permanent Magnet Synchronous*. Právě PMS motoru se budeme věnovat v této kapitole, kde si popíšeme způsob jeho řízení a k tomu potřebné obvody.[12][13]

2.1 Řízení PMS motorů

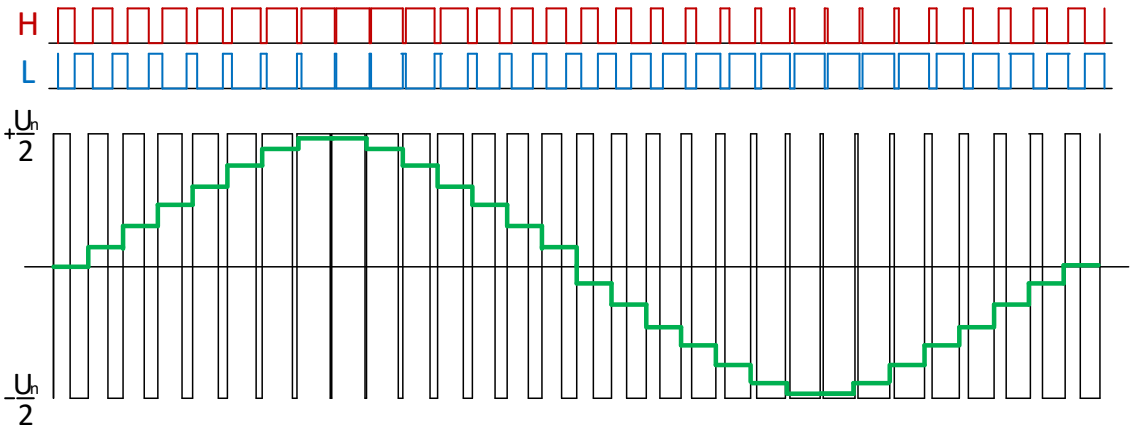
Jak jsme si již naznačili v úvodu této kapitoly, aby došlo k otáčení rotoru, je potřeba pomocí jednotlivých vinutí ve statoru vytvořit točivé magnetické pole. Stator je tvořen třemi cívkami uspořádanými do hvězdy. Pokud tyto cívky budeme budit harmonickými signály, které jsou pro každé vinutí posunuty o 120°, dojde k tomu, že se výsledné magnetické pole otáčet s frekvencí těchto budících signálů. A jelikož je rotor tvořen permanentním magnetem, bude se otáčet tak, aby jeho magnetické pole bylo vždy souhlasně orientované s magnetickým polem statoru.

Abychom mohli nějakým způsobem regulovat takovýto motor, potřebujeme způsob, jak generovat tyto sinusové průběhy a dynamicky měnit jejich amplitudu a frekvenci. K tomu slouží třífázový střídač, který je tvořen dvojicí tranzistorů pro každou fázi motoru, která je připojena ke zdroji stejnosměrného symetrického napětí. Jejich spínání je řízeno dvojicí vůči sobě opačných PWM signálů, které zajistí přepínání mezi kladným a záporným napájecím napětím. Jelikož má vinutí motoru charakter dolní propusti, je brána v potaz pouze stejnosměrná složka tohoto výstupního napětí, která je přímo úměrná střídě řídicích PWM signálů. Schéma třífázového střídače je vidět na obrázku 1.



Obrázek 1: Třífázový střídač

Na následujícím obrázku si můžeme prohlédnout způsob, jakým je vygenerován sinusový průběh. V horní části je vidět dvojice řídicích signálů pro tranzistory, kde červeně je znázorněn průběh signálu pro horní tranzistor a modře pro spodní tranzistor. Pod nimi je vidět průběh výstupního napětí a zeleně vyznačena hodnota stejnosměrné složky tohoto výstupního signálu.



Obrázek 2: Princip generování sinusového signálu

Nutno ještě zmínit, že při přepínání úrovní jednotlivých tranzistorů v jedné větvi nesmí nastat stav, kdy jsou oba tranzistory sepnuté, jelikož by tím došlo ke zkratování napájecího zdroje, čímž by došlo k proudovému přetížení tranzistorů a to by vedlo k jejich zničení. Přestože je podle obrázku 2 aktivní vždy jen jeden z tranzistorů, vlivem rozdílné doby sepnutí a vypnutí tranzistoru, by k tomuto zkratu byť jen na krátkou dobu docházelo. Proto je potřeba zrealizovat mechanizmus, který bude provádět zpoždění nástupné o tzv. ochrannou dobu, která musí být větší než celková doba vypnutí tranzistoru. [10][12]

2.2 Snímání stavových veličin

Pro řízení nejen pohonů, ale i obecně, je důležitou součástí zpětná vazba, pomocí které má řídicí systém přehled o jednotlivých výstupech řízené soustavy. V případě motoru jsou nejdůležitějšími veličinami proudy, které protékají jednotlivými vinutími, úhlová rychlost a aktuální poloha rotoru. V této části se budeme zabývat různými způsoby jejich snímání a popíšeme si základní principy jednotlivých snímačů.

2.2.1 Měření proudu

Nejprve si popíšeme nejpoužívanější metody, které se používají pro snímání proudu. Nejjednodušší způsob, který je možné použít, je zařazení sériového rezistoru s velmi malou hodnotou odporu a následné měření úbytku napětí na něm. Tato metoda je použitelná pro menší proudy, při kterých jsou ztráty dané úbytkem na sériovém rezistoru zanedbatelné. Pro měření větších proudů se používá proudový transformátor, jehož primární vinutí je vřazeno do měřeného obvodu a snímá se proud indukovaný v sekundárním vinutí opět prostřednictvím úbytku napětí na rezistoru vřazeném do sekundární smyčky. Nevýhodou tohoto způsobu je ovšem to, že je použitelný pouze pro snímání střídavých proudů. Posledním způsobem, který si popíšeme je použití snímače na principu Hallova jevu. Zde se využívá skutečnosti, že vodič, kterým protéká elektrický proud, kolem sebe vytváří magnetické pole, které je úměrné velikosti proudu. Pomocí Hallova senzoru jsme tedy schopni přes měření tohoto magnetického pole nepřímo měřit velikost proudu, který vodičem protéká.[13]

2.2.2 Měření otáček a polohy

Jedním ze způsobů jak měřit úhlovou rychlost je použití tachodynama. To je roztáčeno naším regulovaným pohonem, čímž vzniká na jeho výstupu generované napětí úměrné rychlosti jeho otáčení. Pro snímání polohy je možné použít například resolver, ten je tvořen dvěma navzájem kolmými vinutími ve statoru a jedním pohyblivým vinutím, které je spjato s rotorem. Při otáčení se ve statorových vinutích generuje napětí, které je závislé na aktuálním natočení rotorového vinutí.

Zvláštní pozornost bude věnována inkrementálnímu snímači polohy, jelikož právě s tímto typem snímače budeme dále pracovat. Základem tohoto snímače je průsvitný disk, na kterém jsou po obvodu umístěny naopak neprůsvitné značky o definovaném počtu, většinou v řádu stovek až tisíců. Tento disk je umístěn mezi zdroj světla a fotocitlivý snímač. Otáčením tohoto disku dochází k postupnému odkrývání a zastiňování světelného paprsku a tím dostáváme na výstupu obdélníkový signál. Počítáním těchto impulzů jsme schopni určovat relativní polohu, a také můžeme využít toho, že frekvence tohoto výstupního signálu je úměrná úhlové rychlosti.

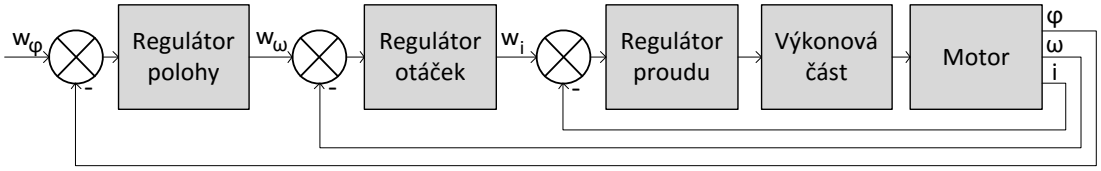
Snímač je zpravidla vybaven ještě druhým výstupem, který je oproti tomu prvnímu posunutý o 90°, díky čemuž jsme schopni získat informaci o směru otáčení. Některé

inkrementální senzory jsou také opatřeny ještě jedním výstupním signálem, pro který je generován jeden impulz během každé celé otáčky. Pokud tedy známe jeho pozici, jsme schopni určovat také absolutní polohu.

Existuje také možnost, jak získat údaje o rychlosti a poloze bez použití snímačů, jelikož jsou tyto veličiny závislé na hodnotách proudů, které protékají jednotlivými vinutími motoru. Pokud tedy známe přesný model konkrétního motoru, jsme schopni navrhnout tzv. stavový rekonstruktor, který zajistí výpočet těchto veličin bez přímého měření.[13]

2.3 Regulace elektrických pohonů

V této části si popíšeme nejčastější způsob regulace elektrických pohonů, kterým je rozvětvený regulační obvod, kdy jsou jednotlivé regulátory zařazeny do kaskády. Typická regulační smyčka je vidět na následujícím obrázku.



Obrázek 3: Uspořádání regulační smyčky

Výhoda tohoto uspořádání spočívá v tom, že regulátor ve vnitřní smyčce může pracovat mnohem rychleji než ten ve vnější a v důsledku dojde k tomu, že celkový systém je při správném nastavení jednotlivých regulátorů mnohem rychlejší, než kdybychom použili pouze jeden regulátor. V našem případě je uvnitř regulátor, který nastavuje požadovanou hodnotu proudu. Žádaná hodnota proudu vystupuje z regulátoru otáček, který je dále nastavován regulátorem polohy, kterou zadá uživatel nebo nadřazený systém. V případě, že je našim cílem řídit pouze otáčky motoru, je možné vypustit regulátor polohy a nastavovat přímo žádanou hodnotu rychlosti.

Jednotlivé regulátory jsou vždy typu PI a jejich nastavování se provádí postupně, kdy nejprve nastavíme nejrychlejší vnitřní smyčku a až po nastavení postupujeme k další smyčce až po tu vnější.

3 VEKTOROVÉ ŘÍZENÍ

V předchozí kapitole jsme si uvedli, že synchronní pohony je nutné budit pomocí sinusového průběhu napětí. Z toho je zřejmé, že jednotlivá fázová napětí a proudy, které jimi protékají, budou časově proměnlivé veličiny, které závisí na aktuální pozici rotoru. Regulace takto časově závislých veličin je velice obtížná, tudíž by bylo výhodnější, kdyby jednotlivé veličiny byly v ustáleném stavu konstantní. Aby tomu tak bylo, nemůžeme se na jednotlivé průběhy dívat z pozice statoru, ale z pozice rotoru. Pokud bychom otáčeli souřadnicovým systémem stejně, jako se otáčí rotor, jevily by se zmíněné veličiny při konstantních otáčkách jako časově nezávislé hodnoty.

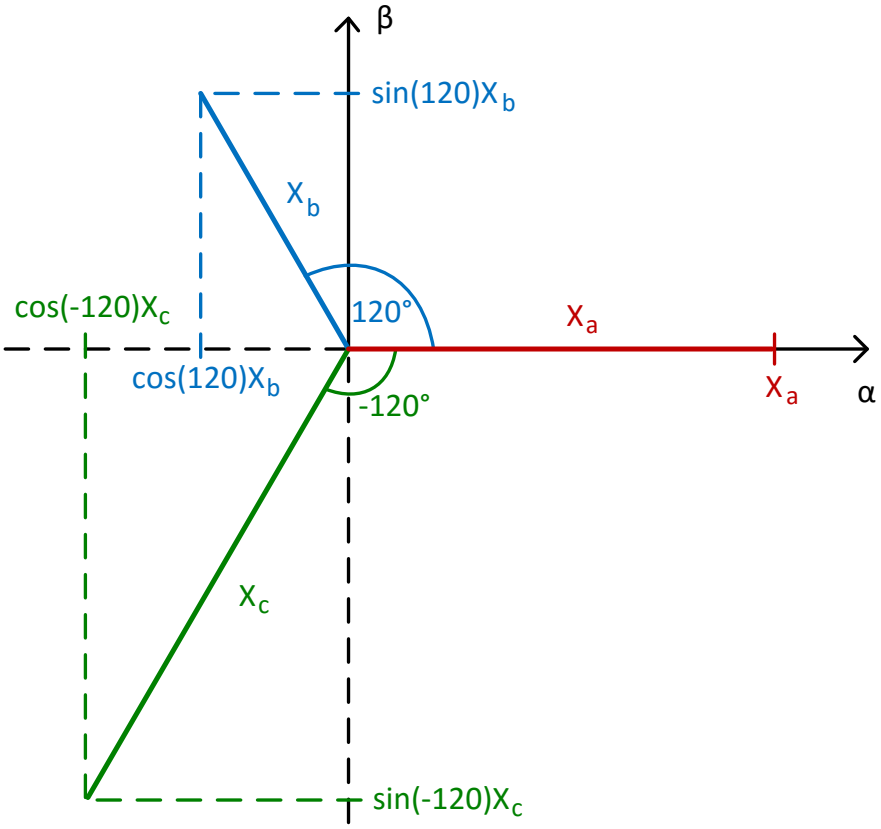
Úlohou vektorového řízení je tedy převedení fázových veličin do takového souřadnicového systému, ve kterém jsou popsány pomocí komplexního vektoru, který charakterizují dvě kolmé složky. První složka vektoru, která reprezentuje jeho reálnou část, je označována jako *d*, z angl. *direct*. Ta je rovnoběžná se směrem magnetického toku rotoru a udává jeho velikost v tomto směru, proto se také označuje jako tokotvorná. Druhá, imaginární, složka, která je vůči ní otočená o 90° v kladném směru otáčení, se označuje *q*, z angl. *quadrature*. Imaginární složka *q* je úměrná točivému momentu, který vyjadřuje působení síly na rotor a tím způsobuje jeho otáčení. Tato složka se proto také nazývá momentotvorná. [4][13]

Budeme tedy potřebovat způsob, jakým převést veličiny jednotlivých fází *a,b* a *c* na odpovídající složky komplexního vektoru *d-q*. Tato transformace se provádí ve dvou krocích. Nejprve je potřeba provést průmět fázových veličin do komplexní roviny označované jako *α-β*, jejíž reálná osa *α* je souhlasně orientovaná s vektorem první fázové veličiny. Tato operace se nazývá Clarkové transformace. Jak jsme si již uvedli, reálná osa v *d-q* souřadnicovém systému má stejný směr jako vektor magnetického toku rotoru, a proto musíme provést druhou transformaci, která se označuje jako Parkova. Ta zajistí to, že vektor v souřadnicích *α-β*, které jsou vázány na jedno vinutí statoru, otočí o úhel, který odpovídá natočení rotoru vůči tomuto statorovému vinutí. Tímto způsobem jsme tedy schopni pozorovat průběhy fázových veličin ze soustavy vztažené k rotoru.

V následujících částech této kapitoly se budeme věnovat matematickému popisu těchto transformací, popíšeme si jejich principy a definujeme si rovnice pro jejich výpočet.

3.1 Clarkové transformace

První transformací, kterou si popíšeme, je Clarkové, jejíž podstatou je zobrazení tří fázových vektorů do komplexní roviny. Výsledný vektor se tedy dá popsat pomocí dvou kolmých složek *α* a *β*. Grafickou interpretaci této operace si můžeme prohlédnout na obrázku 4.



Obrázek 4: Průmět do komplexní roviny

Je tedy zřejmé, že pomocí goniometrických funkcí jsme schopni vypočítat hodnoty jednotlivých složek. Na reálnou osu α se fázové veličiny promítnou jako násobek hodnoty funkce kosinus úhlu, který s osou svírají a na imaginární osu β se promítnou vynásobením hodnotou odpovídající funkci sinus stejného úhlu. Po vyčíslení hodnot goniometrických funkcí dostaneme následující maticový zápis.

$$X_\alpha = X_a - \frac{1}{2}X_b - \frac{1}{2}X_c$$
$$X_\beta = \frac{\sqrt{3}}{2}X_b - \frac{\sqrt{3}}{2}X_c \qquad 3\text{-}1$$

A jelikož víme, že ve třífázové soustavě platí, že součet fázových veličin je roven nule, můžeme z této soustavy vyloučit jednu z proměnných tak, že například za X_c dosadíme následující výraz.

$$X_c = -X_a - X_b \qquad 3\text{-}2$$

Po dosazení tohoto výrazu do rovnice 3-1 a následných úpravách dostaneme zjednodušené rovnice ve tvaru, který můžeme vidět v rovnicích 3-3.

$$X_\alpha = \frac{3}{2} X_a$$
$$X_\beta = \frac{\sqrt{3}}{2} X_a + \sqrt{3} X_b \qquad 3\text{-}3$$

Jak je vidět z první rovnice, touto transformací vzniká navýšení amplitudy signálu a to $\frac{3}{2}$ krát. Aby byla amplituda zachována ve stejném rozsahu, provádí se normalizace vynásobením převrácenou hodnotou tohoto čísla. Výsledná rovnice, která popisuje Clarkové transformaci pro třífázový systém, vypadá tedy následovně.

$$X_\alpha = X_a$$
$$X_\beta = \frac{\sqrt{3}}{3} X_a + \frac{2\sqrt{3}}{3} X_b \qquad 3\text{-}4$$

Ještě nutno dodat, že ve všech rovnicích vystupuje obecná veličina *X*, jelikož tato operace platí pro všechny fázové veličiny jako napětí, proud nebo magnetický tok.

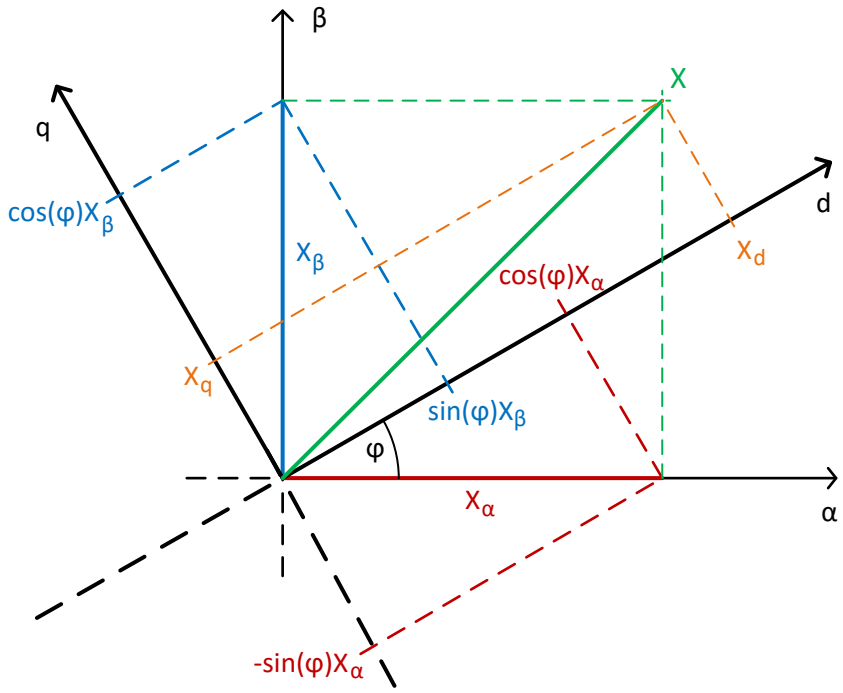
Pro zpětný výpočet fázových veličin z vektoru v souřadnicovém systému *α-β* se využívá inverzní Clarkové transformace, jejíž odvození spočívá v nalezení inverzní transformační matice. Výsledné rovnice, které tuto operaci popisují, vypadají takto.

$$X_a = X_\alpha$$
$$X_b = -\frac{1}{2} X_\alpha + \frac{\sqrt{3}}{2} X_\beta \qquad 3\text{-}5$$
$$X_c = -\frac{1}{2} X_\alpha - \frac{\sqrt{3}}{2} X_\beta$$

Pomocí sady rovnic 3-4 a 3-5 jsme tedy schopni přepočítávat fázové veličiny na kolmé složky komplexního vektoru a zpět.[1][9][13]

3.2 Parkova transformace

V této části se budeme věnovat transformaci komplexního vektoru ze souřadnicového systému *α-β*, který leží v rovině spjaté se statorovou částí pohonu do souřadnicového systému *d-q*, který je naopak svázán s pohybujícím se rotorem. Je tedy zřejmé, že podstatou této operace bude rotace původního vektoru o úhel, který odpovídá natočení rotoru. Jak tato transformace funguje, znázorňuje graficky obrázek 5.



Obrázek 5: Rotace vektoru

Matematické vyjádření této operace, ve které se opět využívá goniometrických funkcí sinus a kosinus pro vytvoření průmětu původního vektoru do natočeného souřadnicového systému, vypadá následovně.

$$\begin{aligned} X_d &= \cos(\varphi)\cdot X_\alpha + \sin(\varphi)\cdot X_\beta \\ X_q &= -\sin(\varphi)\cdot X_\alpha + \cos(\varphi)\cdot X_\beta \end{aligned} \qquad \text{3-6}$$

Je vidět, že v rovnicích vystupuje zmíněný úhel natočení rotoru. Z toho vyplývá, že pokud bychom chtěli provádět tuto transformaci, je nezbytně nutné znát aktuální úhel natočení.

Stejně jako Clarkové transformace má i tato svou inverzní podobu, díky které můžeme převádět vektor z *d-q* roviny zpět do souřadnicového systému *α-β*. Jelikož vektor *d-q* vznikl rotací o úhel *φ*, tak inverzní operace bude spočívat v otočení o úhel opačný, tedy *–φ*. Pokud použijeme následující pravidla, které vycházejí z toho, že funkce kosinus je sudá a funkce sinus je lichá.

$$\begin{aligned} \cos(-\varphi) &= \cos(\varphi) \\ \sin(-\varphi) &= -\sin(\varphi) \end{aligned} \qquad \text{3-7}$$

Dostaneme výsledný tvar rovnic pro provedení inverzní Parkovy transformace, které vypadají následovně.

$$X_\alpha = \cos(\varphi) \cdot X_d - \sin(\varphi) \cdot X_q$$
$$X_\beta = \sin(\varphi) \cdot X_d + \cos(\varphi) \cdot X_q \qquad 3\text{-}8$$

Tímto jsme si definovali veškerý matematický aparát, který je zapotřebí pro nalezení odpovídajících složek vektoru v *d-q* souřadnicích pro fázové veličiny jednotlivých vinutí. Nutno ovšem ještě podotknout, že úhel natočení, který vystupuje ve výše uvedených rovnicích, je úhel elektrický, a pokud pracujeme s motorem, který má více pólových párů, musíme tuto skutečnost při určování elektrického úhlu vzít v potaz. Pokud tedy máme motor se čtyřmi pólovými páry, dojde během jedné fyzické otáčky ke čtyřem otáčkám tohoto elektrického úhlu.[6][9][13]

4 STUPNĚ DŮLEŽITOSTI ÚLOHY

V dnešní době, kdy je díky prudkému nárůstu výkonu řídicích systémů umožněna jejich integrace, vzniká nová řada problémů spojená s touto integrací. Problémy vznikají ve chvíli, kdy jsou kombinovány systémy zpracovávající několik úloh, kde každá z nich má jinou úroveň důležitosti. Příkladem může být systém řídicí jednotky pro moderní automobily, který kombinuje funkce od samotného řízení motoru včetně bezpečnostních mechanismů až po různé zábavní funkce jako palubní počítač. Je zřejmé, že při vzniku poruchy zařízení jsou na tyto jednotlivé úlohy kladeny zcela odlišné požadavky na úroveň bezpečnosti.

Důležitou úlohou při návrhu takového systému, je zajištění, aby celé zařízení přešlo do bezpečného stavu. V tuto chvíli je potřeba vytvořit takový mechanizmus, který zohlední důležitost každé úlohy a odpovídajícím způsobem zajistí prioritu při provádění nezbytných kroků. Musí být tedy zajištěno, aby úlohu s vyšší prioritou nijak nemohla ovlivnit ta s prioritou nižší. Pro úlohy s nejvyšší požadovanou mírou bezpečnosti se také kladou nároky na vysokou spolehlivost mechanismů, které zajišťují zmíněný přechod do bezpečného stavu. Proto jsou tyto prvky většinou zdvojené, aby byla téměř nulová pravděpodobnost jejich selhání.

Ve většině případů se tato problematika řeší na vícejádrových procesorových platformách s operačními systémy reálného času, kde je nutné řešit časové rámce pro zpracování požadavků na přechod do bezpečného stavu u jednotlivých úloh. V tomto směru se nabízí velmi výhodné použití hradlových polí, kde zmíněné rozdělování prostředků není nutné řešit, jelikož veškeré operace probíhají paralelně a nemůžou se žádným způsobem mezi sebou ovlivňovat. Je tedy velice výhodné použít právě hradlového pole pro úlohy s nejvyšším požadavkem na bezpečnost.[5]

5 VÝVOJOVÉ PROSTŘEDKY

V této kapitole, jak již název napovídá, se budeme zabývat vývojovými prostředky, zejména potřebným softwarem, které jsou nezbytné k tvorbě a následnému ladění programů pro SoC Zynq-7000 od firmy Xilinx. Nejprve si popíšeme samotné zařízení, se kterým budeme pracovat a poté se přesuneme k popisu vývojového prostředí, kde si ukážeme některé užitečné nástroje, které budou použity při tvorbě aplikace pro řízení motoru.

5.1 ZedBoard

Jak již bylo uvedeno v úvodu, řídicí aplikace pro PMS motor bude vytvořena pro mikročip Zynq-7000, což je zařízení označované jako SoC, z anglického *System on Chip*. Takto se označují zařízení, kde se na čipu vedle procesoru, případně více jader procesoru, nachází další, dříve často oddělená část, kterou může být blok výkonové elektroniky, nebo programovatelné hradlové pole. Subsystémy jsou sloučeny do jednoho pouzdra a vytváří systém, který je sám schopen vykonávat požadované funkce. Zařízení Zynq-7000 je spojení programovatelného hradlového pole typu FPGA a dvoujádrového procesoru s architekturou ARM Cortex-A9.

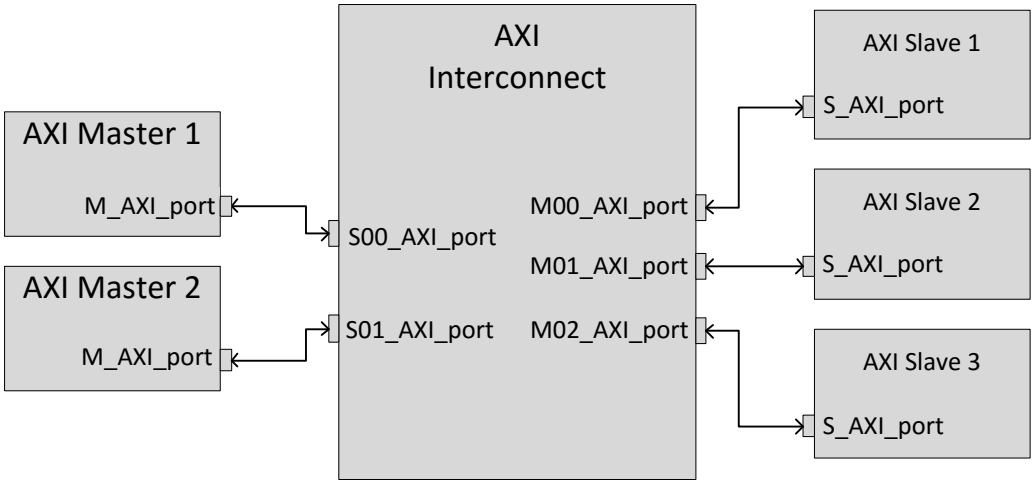
Zynq-7000 je obsažen na vývojové desce s názvem ZedBoard, se kterou budeme dále pracovat. Tato vývojová deska obsahuje kromě uvedeného mikročipu mj. LED diody, přepínače, tlačítka, VGA, HDMI a audio výstupy, na desce se také nachází programovací USB port a debugger. Důležitou součástí je také FMC konektor, který budeme používat pro připojení desky s elektronikou pro řízení motoru.

5.2 AXI sběrnice

Základní stavební jednotkou pro programování moderních hradlových polí je tzv. IP core (*Intellectual Property core*), což je blok dat, který v hradlovém poli vytvoří logický obvod vykonávající požadovanou funkci. Tyto bloky mohou komunikovat mezi sebou nebo s procesorem pomocí AXI sběrnice. Existují tři typy AXI sběrnice a to AXI-Full, AXI-Lite a AXI-Stream. AXI sběrnici je potřeba věnovat pozornost, jelikož se bez ní při programování zařízení Zynq-7000 neobejdeme.

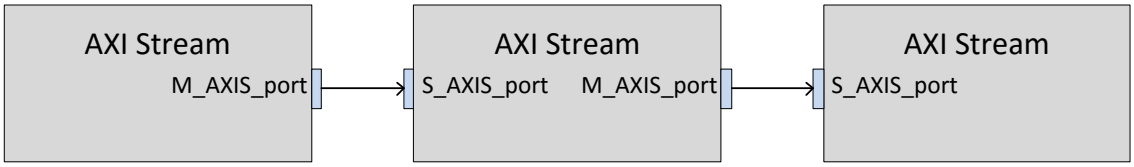
AXI-Full a AXI-Lite představují multi-master, multi-slave sběrnice, které pracují na principu paměťově mapovaných periferií. Rozdíl mezi AXI-Full a AXI-Lite spočívá pouze v tom, že AXI-Full podporuje tzv. *burst* režim. Pokud přenášíme data bez *burst* režimu, tak se při jednom požadavku na čtení nebo zápis přenese jeden blok dat o šířce sběrnice. Při *burst* režimu je možné na jeden požadavek přenést větší objem za sebou umístěných datových bloků v paměti.

O řízení komunikace na sběrnici se stará blok nazvaný AXI Interconnect, pomocí kterého je nutné všechny prvky sběrnice vzájemně propojit. Tento systém se stará o dekódování adres a požadavků, řeší veškeré konflikty na sběrnici a také je schopný propojovat periferie s různými šířkami sběrnice nebo periferie pracující na různých kmitočtech hodinového signálu.



Obrázek 6: AXI Interconnect

AXI-Stream je oproti předchozím dvěma podstatně jednodušší. Pomocí tohoto rozhraní můžeme propojit pouze dvě zařízení, kde jedno z nich je typu master a vysílá proud dat a druhé je typu slave a tento proud dat přijímá. Každá komponenta ovšem může obsahovat vstupní rozhraní typu slave a zároveň i to výstupní typu master, což umožňuje řetězení jednotlivých komponent za sebou. To se typicky využívá například při zpracování videa, kdy každá komponenta provádí danou úpravu snímku a posílá jej dále ke zpracování.[11]



Obrázek 7: Ukázka AXI-Stream IP bloků

5.3 Vivado Design Suite

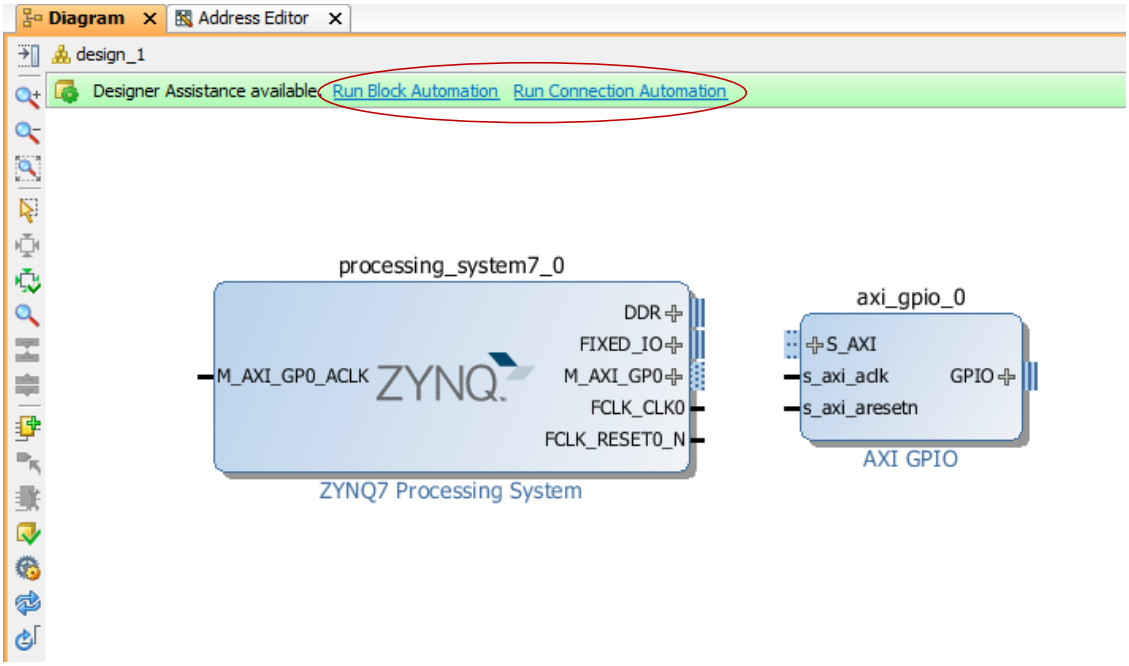
Pro vývoj aplikací pro zařízení z řady Zynq-7000, konkrétně jen pro tu část obsahující hradlové pole, slouží vývojové prostředí *Vivado Design Suite* od firmy Xilinx, s jehož pomocí je možné vytvořit vše potřebné pro naprogramování hradlového pole od napsání programu v jazyce VHDL nebo Verilog, až po samotnou implementaci do zařízení a vytvoření bitstreamu. Jeho součástí je také editor vlastních IP bloků a simulátor pro ověření jejich funkčnosti.

Další součástí balíčku s vývojovým prostředím je také Vivado HLS (*High Level Synthesis*), pomocí kterého je možné vytvářet IP bloky v jazyce *C* nebo *C++*. Výhodou je snadnější tvorba programů než ve VHDL, zejména pokud se jedná o složité matematické operace. Jako cena za usnadnění práce při vytváření aplikací je absence úplné kontroly nad tím, jak bude výsledný obvod hardwarově realizován na hradlovém poli.

5.3.1 Tvorba projektu

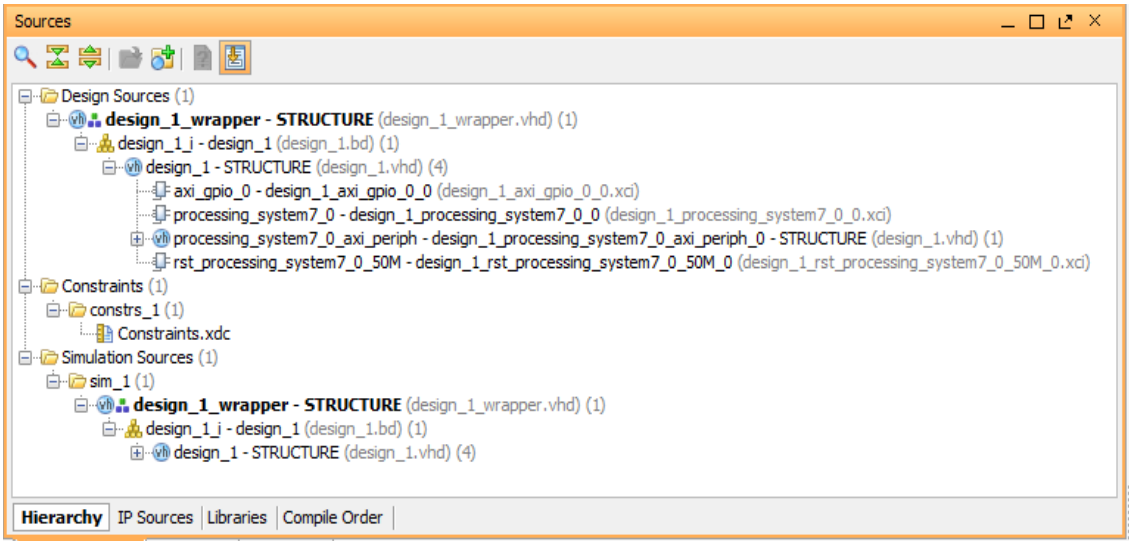
V této části se budeme věnovat vytváření projektu ve vývojovém prostředí *Vivado Design Suite*. Po založení nového projektu a nového blokového designu se otevře podokno v pravé části obrazovky, ve kterém můžeme graficky skládat jednotlivé komponenty. Z vytvořeného schématu poté můžeme nechat automaticky vygenerovat top-level modul nazvaný *HDL wrapper*, který toto schéma popisuje pomocí jazyka VHDL nebo Verilog v závislosti na tom, který z jazyků jsme si vybrali při tvorbě projektu.

Celý princip si ukážeme na příkladu, kdy si vytvoříme jednoduché schéma, ve kterém propojíme procesor s modulem GPIO, na který připojíme LED diody na vývojové desce. Přidáme tedy blok s procesorem, který je označen jako *ZYNQ7 Processing system*, poté nám vývojové prostředí samo nabídne možnost automatického připojení paměti RAM a dalších vstupních a výstupních modulů, které jsou potřebné pro správný chod procesoru. Tato operace se provede po kliknutí na tlačítko *Run Block Automation* v horní části okna viz následující obrázek.



Obrázek 8: Tlačítka pro automatické operace

Poté přidáme modul GPIO a v horní části obrazovky se nám objeví tlačítko *Run Connection Automation*, které také můžeme vidět na obrázku 8. Pokud na něj klikneme, Vivado opět automaticky provede jisté kroky, které jsou potřebné k propojení těchto dvou modulů, a to přidání modulu AXI Interconnect, který je nutný k propojení jednotlivých bloků pomocí AXI sběrnice a také přidá ještě jeden modul označený jako *Processor System Reset*, který je odpovědný za generování reset signálů, které jsou využívány AXI sběrnicí. Nakonec samozřejmě dojde ke správnému propojení odpovídajících signálů rozhraní AXI sběrnice mezi sebou. Poté můžeme opět pomocí tlačítka *Run Connection Automation* využít automatického připojení, tentokrát LED diod k modulu GPIO. Pokud si necháme vygenerovat *HDL wrapper*, uvidíme v levé horní části obrazovky v sekci *Hierarchy* podobnou strukturu jako na následujícím obrázku.



Obrázek 9: Hierarchie zdrojových souborů

Jako top-level modul zde vidíme soubor *design_1_wrapper.vhd*, který obsahuje definici vnějších vstupů a výstupů námi navržené aplikace, o úroveň níže je soubor *design_1.vhd*, který obsahuje volání všech použitých komponent použitých při návrhu a definuje všechny vnitřní signály, které se zde vyskytují a nakonec jsou jednotlivé signály připojeny ke komponentám pomocí příkazů *port map*. Na nejnižší úrovni se nacházejí *VHDL* popisy jednotlivých komponent.

Další důležitou částí, kterou si popíšeme je soubor obsahující sadu pravidel pro vytvořený projekt. Ten se vkládá do sekce *Constraints* a obsahuje například přiřazení vstupních a výstupních portů k fyzickým pinům hradlového pole nebo definici napěťových úrovní pro tyto piny. Také se zde definují vlastnosti hodinových signálů a další parametry, které souvisí s časováním. V hierarchii se také nachází sekce se soubory pro simulaci, ke kterým se vrátíme později.

5.3.2 Tvorba vlastního IP bloku

Pokud si chceme vytvořit vlastní IP blok, můžeme využít vestavěný modul, který je k tomu určený. Ten vyvoláme tak, že zvolíme v menu v horní části obrazovky záložku *Tools* a v ní *Create and Package IP*. V průvodci zvolíme vytvoření AXI4 periférie, můžeme zvolit název komponenty a poté si můžeme vybrat, v jakém režimu AXI sběrnice má daná komponenta pracovat. Poté můžeme zvolit možnost *Edit IP* a otevře se nám další instance programu *Vivado*, ve které můžeme upravit chování naší nové komponenty.

V sekci *Sources* si můžeme všimnout, že již byly automaticky vytvořeny dva soubory. Jedním z nich je obsluha rozhraní AXI sběrnice, která obsahuje kompletní program pro příjem a odesílání dat a ve většině případů ji nemáme proč měnit. Druhý soubor, který v hierarchii nejvýše prozatím pouze volá výše zmíněnou obsluhu AXI sběrnice. V tomto souboru se provádí definice vstupních a výstupních portů celého IP bloku, dále zde můžeme definovat potřebné signály a vytvořit potřebnou logiku tak, aby blok prováděl požadovanou funkci.

5.3.3 Simulátor

Součástí vývojového prostředí *Vivado Design Suite* je také velice užitečný simulátor, který slouží k ověření funkčnosti jednotlivých komponent, popřípadě celého vytvořeného projektu. Pokud spustíme simulátor, provede se simulace všech vnitřních signálů, které jsou nadefinovány v souboru označeném jako top-level v sekci *Simulation Sources* viz obrázek 9. Naším úkolem je tedy vytvořit tento soubor tak, aby se provedla simulace toho, co zrovna potřebujeme.

Postup, jak spustit simulaci, si opět ukážeme na jednoduchém příkladu. Vyjdeme z toho, že máme vytvořenou primitivní komponentu, například čítače hodinových pulzů. Taková komponenta má dva vstupy, kde první je pro hodinový signál označený jako *clk* a druhý pro asynchronní resetovací signál *ares*. Také obsahuje jeden výstup, označený jako *cnt*, který odpovídá načítané hodnotě. V této komponentě jsme vytvořili vnitřní logiku, která má provést požadovanou operaci, čímž máme vše připraveno pro otestování funkčnosti.

Dalším krokem je vytvoření simulačního zdrojového souboru. To se provede kliknutím na tlačítko *Add Sources* v levé části obrazovky a výběrem možnosti *Add or Create Simulation Sources*. Vytvoříme nový soubor, který pojmenujeme například *counter_tb.vhd*. V tomto souboru je potřeba zavolat naší testovanou komponentu. To znamená, že nejprve musíme definovat, o jakou komponentu se jedná, abychom ji mohli následně v těle programu namapovat. Nyní ještě potřebujeme definovat vnitřní signály, které právě zmíněným mapováním přiřadíme ke vstupům a

výstupům testované komponenty. Za tímto účelem je tedy potřeba vytvořit vnitřní signály *clk*, *ares* a *cnt*.

Posledním krokem, který je nutno udělat, než spustíme simulaci, je definice signálů, které budou vstupovat do testované komponenty. To se dělá tím způsobem, že v části pro popis chování modulu vytvoříme proces s prázdným citlivostním seznamem. Takovýto proces se bude chovat jako nekonečná smyčka. Uvnitř procesu můžeme upravovat hodnoty signálů. Abychom mohli průběhy patřičně časovat, použijeme příkaz *wait* podle následující ukázky, která definuje průběh hodinového signálu *clk*.

Zdrojový kód 1: Definice hodinového signálu pro simulátor

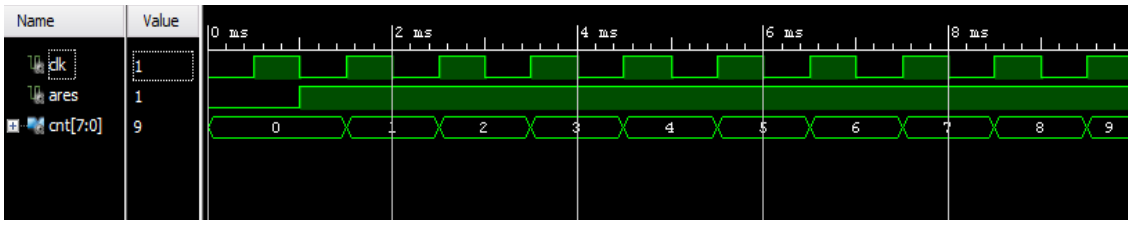
```
process
  begin
    clk <= '0';
    wait for 500us;
    clk <= '1';
    wait for 500us;
  end process;
```

Tímto jsme vytvořili hodinový signál o frekvenci *1 KHz*. Dále musíme ještě definovat druhý vstupní signál, kterým je asynchronní reset. Na tom si ukážeme další využití příkazu *wait* a to tak, že neuvedeme žádný časový údaj. Ten poté bude fungovat tím způsobem, že bude čekat do nekonečna, což je užitečné právě tehdy, když potřebujeme zamezit periodickému opakování procesu při jednorázové změně úrovně signálu.

Zdrojový kód 2: Definice resetovacího signálu pro simulátor

```
process
  begin
    ares <= '0';
    wait for 1 ms;
    ares <= '1';
    wait;
  end process;
```

Tímto zápisem docílíme toho, že první milisekundu bude resetovací signál roven logické nule, tedy aktivní a zbytek simulace zůstane v úrovni logické jedničky. Nyní můžeme spustit simulaci a podívat se na průběhy jednotlivých signálů.



Obrázek 10: Ukázka výstupu simulátoru

Na obrázku 10 si můžeme prohlédnout ukázku toho, jak vypadají simulované průběhy jednotlivých signálů. První dva jsou vstupy, které jsme si nadefinovali, a poslední je právě testovaný výstup komponenty čítače.

5.3.4 Konfigurační parametry

V této části si popíšeme další velice užitečný nástroj, kterým je tvorba konfiguračních parametrů pro vytvořené IP bloky. Jedná se parametry, které je možné editovat přímo z okna pro tvorbu blokového designu bez toho, abychom museli zasahovat do zdrojového kódu dané komponenty. Okno pro editaci těchto parametrů se poté vyvolá stiskem pravé tlačítko myši nad konkrétním IP blokem a následnou volbou možnosti *Customize Block*. Pokud se do tohoto menu podíváme po vytvoření nového IP bloku, budou zde pouze parametry pro nastavení AXI sběrnice. My si nyní ukážeme, jak na toto místo vložit vlastní parametry, které jsou užitečné zejména v případě, že vytváříme nějakou univerzální komponentu.

Pro vytvoření takových parametrů si nejprve musíme v top-level souboru nadefinovat patřičné proměnné v sekci *Generic*. Zde můžeme využít buď datový typ *integer*, pro celočíselné hodnoty, nebo *boolean* pro binární. Po uložení souboru se můžeme přesunout do záložky *Package IP*, kde zvolíme možnost *Customization Parameters*. Zde by se měly automaticky přidat jednotlivé položky pro každou proměnnou vytvořenou v sekci *Generic*. Zde u těchto parametrů lze dále editovat vlastnosti, které ovlivní jejich zobrazení v nastavovacím menu. Zejména jde o popisek, který se u parametru objeví a způsob, jak bude zobrazena hodnota. U celočíselných proměnných můžeme například zvolit, jestli chceme zobrazit pole pro přímou editaci čísla, nebo rozbalovací nabídku jen s některými definovanými hodnotami.

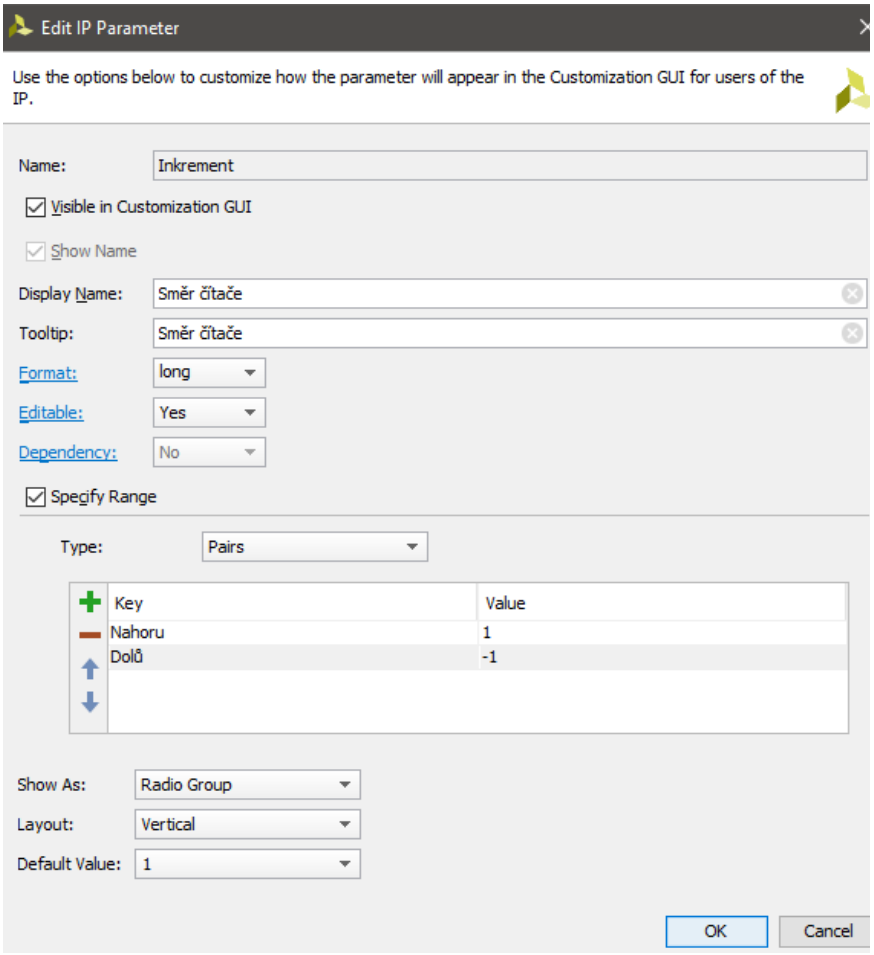
Celý tento proces si opět ukážeme na jednoduchém příkladu. Zůstaneme u našeho čítače z předchozí části a přidáme zde parametr, kterým se bude ovlivňovat jeho směr. Nejprve tedy přidáme celočíselnou proměnnou, kterou si pojmenujeme *Inkrement,* do sekce *Generic*, kde se již nachází automaticky vygenerované parametry AXI sběrnice.

Zdrojový kód 3: Přidání nového parametru

```
generic (
  Inkrement             : integer   := 1;
  C_S00_AXI_DATA_WIDTH  : integer   := 32;
  C_S00_AXI_ADDR_WIDTH  : integer   := 4
);
```

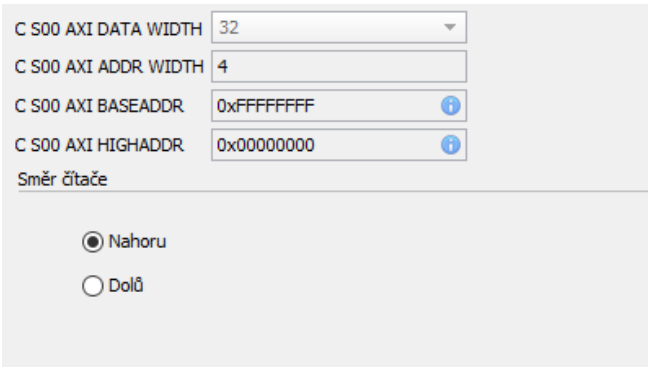
Ta bude nabývat hodnoty 1 nebo -1 a bude použita ve zdrojovém kódu čítače jako číslo, které se bude přičítat k aktuální hodnotě. Nyní se přesuneme k editaci tohoto parametru. Dialogové okno pro editaci můžeme vidět na obrázku 11, kde je také vidět nastavení zobrazovaného názvu jako „Směr čítače“ a definice potřebného rozsahu této proměnné. Zde můžeme také využít toho, že je možné přiřadit různým hodnotám

textový řetězec, který se má zobrazit místo ní. Nakonec je zde vybrán způsob, jak se má parametr zobrazit, tedy skupina výběrových tlačítek, a výchozí hodnota této proměnné.



Obrázek 11: Dialogové okno pro editaci parametru

Nakonec se ještě můžeme přesunout do sekce *Customization GUI*, kde lze provádět úpravy z hlediska rozmístění jednotlivých parametrů v editačním okně. V našem případě není potřeba nic měnit, jelikož zde máme pouze jeden parametr. Výsledek celého procesu si můžeme prohlédnout na obrázku 12.



Obrázek 12: Výsledné zobrazení parametru

5.4 SDK

Druhým programem, který je nezbytně nutný pro vývoj projektu pro SoC Zynq, je prostředí pro tvorbu aplikací, které jsou určeny procesorové části tohoto zařízení. Kromě toho se prostřednictvím tohoto programu provádí nahrávání výsledného bitstreamu, který vznikl v návrhovém prostředí *Vivado*.

Pokud jsme ve fázi, že máme hotový program pro hradlové pole a máme vygenerovaný bitstream, můžeme přistoupit k programování procesoru. Nejprve je ovšem potřeba exportovat nezbytné soubory z programu Vivado, abychom s nimi mohli pracovat. K tomu je potřeba v menu v horní části obrazovky zvolit záložku *File*, poté *Export* a zde vybereme *Export Hardware*. Je nutné zaškrtnout políčko *Include bitstream*. Poté můžeme spustit SDK a to opět z menu v horní části, kde zvolíme *File* a následně *Launch SDK*. Pokud jej spustíme tímto způsobem, dojde k automatickému provázání s exportovanými daty z minulého kroku. Nyní můžeme založit nový projekt, kde si vytvoříme soubor *main.c*, popřípadě *main.cpp*, ve kterém bude hlavní program pro procesor.

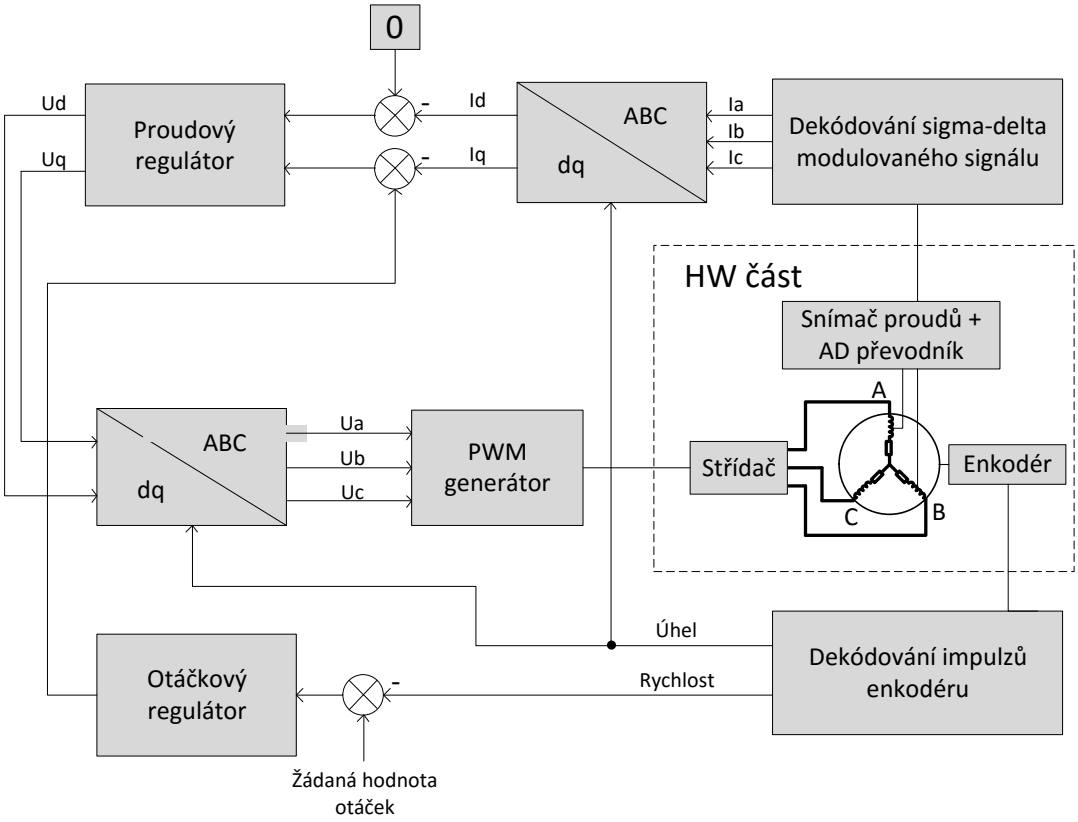
Nyní si projdeme nejdůležitější hlavičkové soubory vygenerované při exportu hardwaru v programu *Vivado*, které je nutné začlenit do nového projektu. Na prvním místě to bude soubor *platform.h*, který obsahuje deklaraci funkce *init_platform*. Ta musí být zavolána na začátku každého programu, jelikož zajišťuje například zapnutí hodinových signálů, které jsou generovány v procesorové části a další nezbytné kroky, které jsou nutné provést pro spuštění programu. Dalším velice užitečným souborem je *xparameters.h*, ve kterém se skrývají definice všech adres, které jsou využívány pro jednotlivé komponenty připojené k AXI sběrnici. Pro samotnou komunikaci přes AXI sběrnici jsou zde připraveny funkce, které jsou zahrnuty ve standardní knihovně *xil_io.h*. Zde se nacházejí například funkce *Xil_In32* nebo *Xil_Out32*. Funkce *Xil_In32*, slouží pro čtení hodnoty z adresy, která je jejím parametrem. Návratovou hodnotou této funkce je hodnota, která se na zvolené adrese nachází. *Xil_Out32* má dva parametry a to požadovanou hodnotu a adresu, kam ji má zapsat. Kromě funkcí pro komunikaci přes AXI sběrnici můžeme využít také funkci *xil_printf*, která odesílá zadaný řetězec přes sériové rozhraní UART.

Pokud máme hotovou aplikaci i pro procesorovou část, můžeme začít s nahráváním hotového programu do zařízení. To se provádí ve dvou krocích, kdy je nejprve potřeba naprogramovat hradlové pole, což se provede stiskem tlačítka *Program FPGA*, které se nachází v menu v horní části pod záložkou *Xilinx Tools*. Druhým krokem je spuštění programu v procesoru, k čemuž dojde po stisku tlačítka *Run* a výběrem možnosti *Launch on Hardware*.

6 REALIZACE KOMPONENT

V této kapitole se budeme zabývat realizací jednotlivých komponent, ze kterých se bude skládat program pro vektorové řízení PMS motorů. Nejprve si popíšeme celkový koncept a všechny nezbytné součásti, které jsou nutné pro regulaci motorů tohoto typu, poté si projdeme detailně jednotlivé součásti a ukážeme si, jak byly naprogramovány v jazyce VHDL.

Celý program bude rozdělený do dílčích bloků, kde každý z nich bude provádět jeden úkon z celkového řetězce zpracování signálů. Rozdělení na tyto části a jejich logické uspořádání je vidět v blokovém schématu na obrázku 13. Naše působení na motor spočívá ve tvarování napěťových signálů, které jsou přiloženy k jednotlivým vinutím statoru a jako zpětnou vazbu na naše působení dostáváme hodnoty proudů, které jimi tečou a aktuální polohu rotoru. Našim úkolem je tedy zpracovat naměřené hodnoty proudů a polohy a z nich vypočítat hodnotu potřebného napěťového signálu. Nejprve je tedy potřeba dekódovat signály ze snímačů, poté provést transformaci trojfázových proudů do d-q roviny, kde je realizována samotná regulace. Vypočítané akční zásahy, které jsou v d-q oblasti je poté potřeba přepočítat zpět do třífázového systému a vygenerovat PWM signály, které vytvoří potřené úrovně napětí na jednotlivých vinutích.

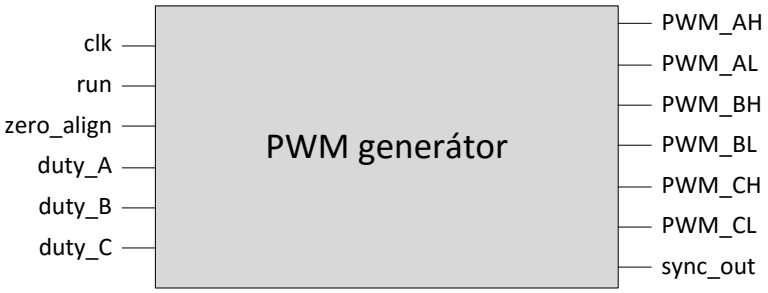


Obrázek 13: Blokové schéma komponent programu

6.1 Generátor PWM

První komponenta, kterou si popíšeme je generátor PWM signálu. Z blokového schématu je zřejmé, že vstupem do tohoto bloku je požadovaná hodnota napětí, která odpovídá střídě, a výstupy jsou samotné modulované signály pro střídač. Ty musí být pro každou fázi motoru dva, jeden normální a druhý negovaný. Kromě těchto signálů jsou potřeba další řídicí signály, které si popíšeme dále v této části.

Realizace generátoru PWM signálu spočívá ve vytvoření čítače a komparátorů, které zajistí přepnutí logické úrovně výstupu při definovaných úrovních. Pro čítač budeme potřebovat další vstup a to hodinový signál. Další nezbytnou funkcí musí být možnost zapnutí a vypnutí tohoto modulu, k čemuž slouží další vstupní signál označený *run*. Jak jsme si uvedli v předchozím textu, kvůli použití inkrementálního snímače musíme být schopni inicializovat polohu rotoru, tak že jej natočíme souhlasně s magnetickým polem jednoho vinutí statoru. Proto potřebujeme další vstupní signál, kterým se bude spouštět toto natočení. Tento signál jsem označil *zero_align*. Posledním signálem je výstup označující konec jedné periody. Po každé periodě je potřeba navzorkovat vstupní signály začít s výpočtem střídy pro další periodu PWM signálu.



Obrázek 14: IP blok s PWM generátorem

Dále si probereme nastavitelné parametry této komponenty. Prvním parametrem je požadovaná frekvence PWM signálu. Je ovšem nutné znát, jaká je frekvence vstupního hodinového signálu, aby bylo možné spočítat dělící poměr mezi periodou PWM a periodou hodin. Také zde lze nastavit jaká je aktivní úroveň PWM signálu pro horní a dolní tranzistor zvlášť, jelikož některé měniče využívají i takovou kombinaci, že jeden ze signálů vyžaduje normální a druhý negovanou logiku. Dalším parametrem je velikost ochranné doby tranzistoru, tedy doby, po kterou nemohou být oba signály v aktivní úrovni zároveň kvůli rozdílné době sepnutí a vypnutí tranzistoru.

Nyní si popíšeme nejdůležitější části zdrojového kódu pro tento modul. Nejprve bylo potřeba zrealizovat čítač hodinových impulzů od nuly do hodnoty, která časově odpovídá jedné periodě PWM signálu. Ta se vypočítá následujícím způsobem:

$$cnt_{\max} = \frac{T_{PWM}}{T_{clk}} \qquad 6\text{-}1$$

Po dosažení této hodnoty je potřeba čítač vynulovat a počítat znovu. Tím vzniká pilovitý signál, který bude potřebný pro další část programu. V následujícím zdrojovém kódu je vidět popis takového čítače. Kromě toho je také nutné jej asynchronně spouštět a zastavovat signálem *run* a při zastavení jej opět vynulovat. Také je zde vidět generace signálu *sync_out*, který signalizuje začátek nové periody. Jeho význam si popíšeme později.

Zdrojový kód 4: Generování pilovitého signálu

```
process (clk,run)
    variable cnt : integer range 0 to period;
begin
    if run='0' then
        cnt := 0;
    elsif (clk'event and clk = '1') then
        if cnt = period then
            cnt := 0;
            sync_out <= '1';
        else
            cnt := cnt + 1;
            sync_out <= '0';
        end if;
    end if;
    counter <= to_unsigned(cnt,32);
end process;
```

Dalším krokem je samotné přepínání logických úrovní výstupních signálů při definovaných úrovních pilovitého signálu z minulého kroku. Jelikož bude tento proces nutné provést pro každou dvojici tranzistorů, je výhodné si vytvořit komponentu, a poté jen přidat potřebný počet instancí podle počtu fází motoru. Je tedy zřejmé, že do komponenty musí vstupovat hodnota čítače, požadovaná střída a výstupem bude dvojice signálů pro horní a dolní tranzistor jedné větve střídače. Nejprve je nutné přepočítat vstupní hodnotu střídy na hodnotu, která bude odpovídat době, po kterou má být signál v aktivní úrovni. Ten dostaneme vynásobením střídy a periody PWM. V následující části kódu vidíme tento přepočet, a také mechanizmus, který novou hodnotu střídy převezme na začátku každé periody.

Zdrojový kód 5: Přepočet střídy

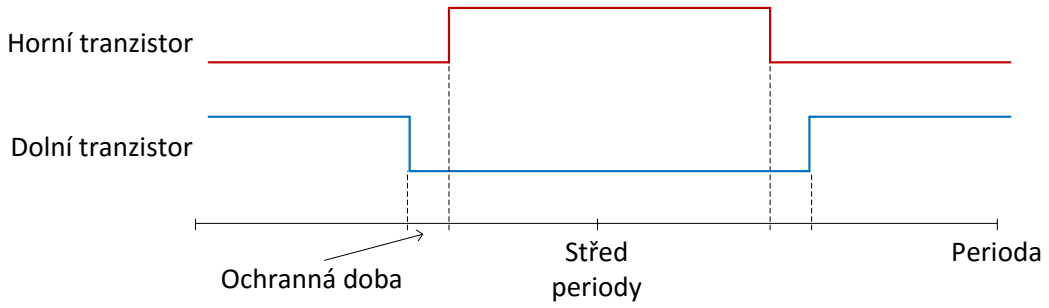
```
process (clk,run) begin
    if (run = '0') then
        half_duty_actual <= (others => '0');
    elsif (clk'event and clk = '1') then
        half_duty_new <= resize(duty * half_period / 65535,32);
        if (counter = 0) then
            half_duty_actual <= half_duty_new;
        end if;
    end if;
end process;
```

Tímto zápisem docílíme toho, že se vytvoří registr, který propustí data na výstup pouze po splnění podmínky, že je hodnota čítače rovna nule. Jelikož budeme generovat PWM signál, který bude zarovnaný na střed periody, je výhodnější pracovat s hodnotou, která odpovídá polovině střídy, protože budou dvě překlápěcí úrovně, jedna polovinu střídy přes středem periody PWM a druhá polovinu střídy za polovinou periody. Tento mechanizmus provádí následující část programu.

Zdrojový kód 6: Tvarování výstupních signálů

```
process (clk,run) begin
  if (run = '0') then
    PWM_H <= not ACTIVE_H;
    PWM_L <= not ACTIVE_L;
  elsif (clk'event and clk = '1') then
   if counter = half_period - half_duty_actual - dead_time then
      PWM_L <= not ACTIVE_L;
   elsif counter = half_period - half_duty_actual then
      PWM_H <= ACTIVE_H;
   elsif counter = half_period + half_duty_actual then
     PWM_H <= not ACTIVE_H;
   elsif counter = half_period + half_duty_actual + dead_time then
     PWM_L <= ACTIVE_L;
   end if;
  end if;
end process;
```

Jak je vidět v kódu, výstupním signálům není přidělována přímo logická úroveň, ale hodnota proměnné, kterou je možné nastavit při konfiguraci toho modulu. Dále si můžete povšimnout, že překlápěcí úrovně pro dolní tranzistor jsou posunuty o ochrannou dobu. Výsledný signál, pokud bychom zvolili jako aktivní úroveň pro oba tranzistory logickou 1, by tedy měl vypadat jako na následujícím obrázku.



Obrázek 15: Průběh PWM signálů

Nakonec byly provedeny úpravy této komponenty, aby díky ní bylo možné řídit vícefázové motory. Byl tedy přidán parametr, kterým je možné vybrat, jestli chceme použít tří, šesti, devíti popřípadě dvanáctifázový motor. Takovéto motory jsou mechanicky provedeny tak, že další trojice fází jsou vždy natočeny o zlomek úhlu, který je dán jejich počtem. Například devítifázový motor má tedy druhou trojici otočenou o 40° vůči té první a třetí trojici o 80°. Nutná úprava tedy spočívá v tom, že řídicí signály

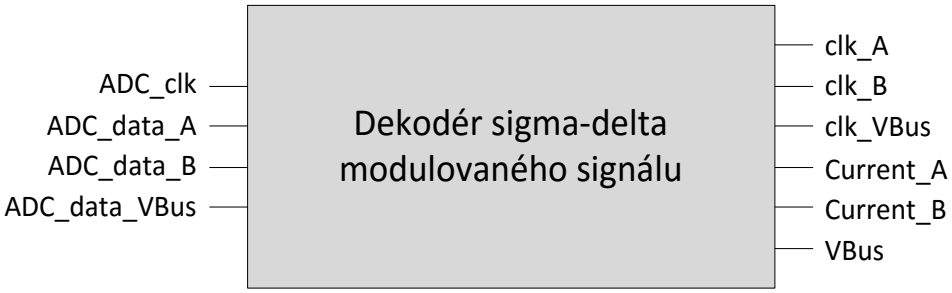
pro jednotlivé trojice fází musí být vůči sobě natočeny o úhel odpovídající jejich fyzickému natočení. Pro realizaci těchto posunů bylo potřeba provést posunutí hodnoty čítače, který řídí překlápění signálů pro tranzistory.

Tímto ovšem vzniká nový problém, který je potřeba vyřešit. Na začátku každé periody PWM signálu dochází ke vzorkování měřených proudů, a pokud dojde ve stejnou chvíli k překlopení úrovně na některém z tranzistorů, bude toto měření zatíženo chybou kvůli přechodovému ději, který při přepínání vznikne. Proto je nutné tyto případy detekovat a v případě, že by k této situaci mělo dojít, je nutné mírně posunout PWM signál, tak aby k překlopení úrovně došlo mimo čas stanovený pro vzorkování.

6.2 Čtení analogových vstupů

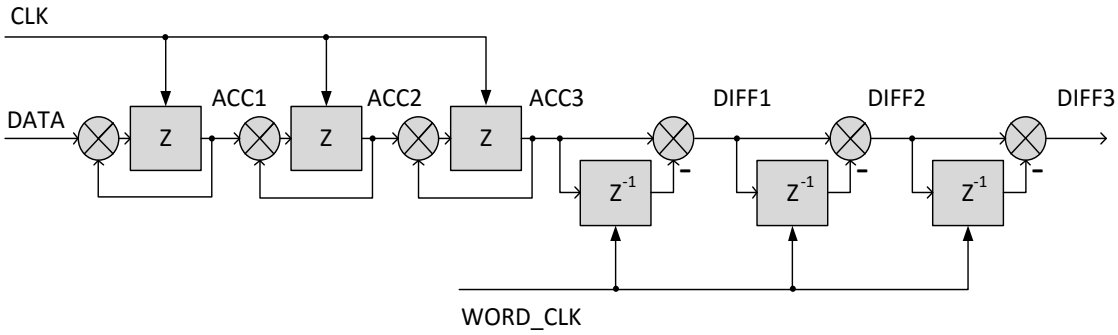
Další součástí, která bude potřeba k řízení PMS motoru, je blok, který dekóduje signály poskytované sigma-delta převodníkem. Výstupem takového modulátoru je PWM signál, jehož střída je úměrná vstupnímu napětí. K dekódování výstupu ze sigma-delta modulátoru se běžně používá digitální filtr $SINC^3$, jehož realizaci si nyní popíšeme.

Potřebné vstupy a výstupy si můžeme prohlédnout na obrázku 16. Je zřejmé, že do komponenty budou vstupovat datové signály z každého převodníku, tedy ze dvou, které měří proud a jednoho, který měří napájecí napětí, dále potřebujeme hodinový signál, se kterým převodník pracuje, jelikož celý algoritmus našeho filtru musí pracovat se stejným hodinovým signálem. Také potřebujeme tento hodinový signál přivést ke každému z převodníků, proto se tento signál nachází i na výstupu z tohoto bloku, odkud bude přiveden na výstupní piny hradlového pole, které jsou k AD převodníku připojeny. Dalšími výstupy jsou filtrované digitální hodnoty odpovídající měřenému napětí.



Obrázek 16: Dekodér sigma-delta modulovaného signálu

Dále si probereme vnitřní strukturu této komponenty. Stejně jako u předchozí komponenty budeme i zde provádět stejnou operaci pro každou měřenou veličinu, proto je opět výhodné si vytvořit samostatnou komponentu, která tuto operaci provede a poté opět vytvořit několik instancí. Vstupem do komponenty, která bude provádět operaci $SINC^3$ filtru bude hodinový signál AD převodníku a datový signál, který z něj vystupuje a na výstupu bude číslo odpovídající měřenému signálu. Na následujícím obrázku je vidět vnitřní struktura této komponenty.



Obrázek 17: Schéma SINC^3 filtru

Následující VHDL popis byl vytvořen na základě zdrojového kódu v katalogovém listu AD převodníku [2]. Nejprve je nutné převést vstupní signál, který reprezentuje jeden bit, na vektor o stejné šířce, jako všechny ostatní signály, se kterými se bude dále počítat. Je to nutné z toho důvodu, abychom toto číslo mohli přičítat k dalšímu vektoru.

Zdrojový kód 7: Převod vstupního signálu na vektor

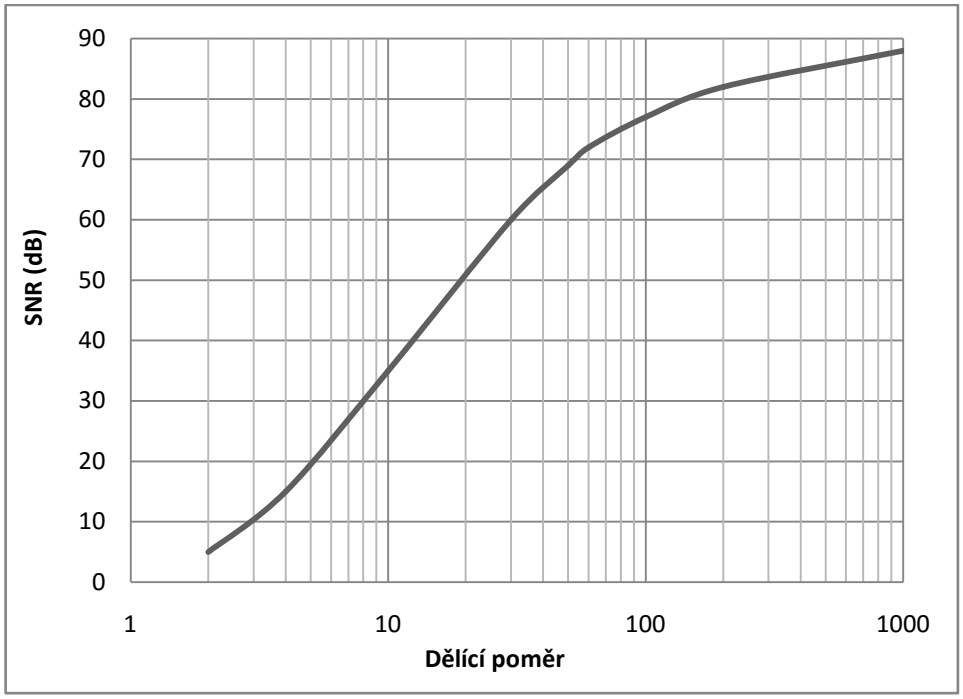
```
process (mdata)
begin
  if (mdata = '0') then
    mdata_unsigned <= (others => '0');
  else
    mdata_unsigned <= to_unsigned(1,mdata_unsigned'length);
  end if;
end process;
```

Dále je potřeba provést operaci tří sumátorů zařazených sériově za sebe, jak je vidět na obrázku 17. To řeší následující proces, který na začátku definuje výchozí stav po resetu a poté provádí samotnou sumaci.

Zdrojový kód 8: Sumační část

```
process (mclk,ares)
begin
  if (ares = '0') then
    acc1 <= (others => '0');
    acc2 <= (others => '0');
    acc3 <= (others => '0');
  elsif (rising_edge(mclk)) then
    acc1 <= acc1 + mdata_unsigned;
    acc2 <= acc2 + acc1;
    acc3 <= acc3 + acc2;
  end if;
end process;
```

Další částí naší komponenty jsou tři diferenční členy, které ovšem pracují na jiném hodinovém kmitočtu, který je dán zvoleným dělícím poměrem. Na následujícím grafu je vidět závislost odstupu signálu od šumu (SNR) na zvoleném dělícím poměru.



Obrázek 18: Závislost SNR na dělícím poměru

Aby byl zajištěn dostatečný odstup signálu od šumu, zvolil jsem dělící poměr 256. Nyní je tedy potřeba vytvořit děličku kmitočtu, která vytvoří 256 krát pomalejší hodinový signál pro další součásti obvodu. Dělička je realizována pomocí dvou procesů, kde v prvním je potřeba stanovit výchozí stav po resetu a realizovat čítač s osmibitovou proměnnou. Její nejvyšší bit je poté použit ve druhém procesu jako dělený hodinový signál.

Zdrojový kód 9: Dělička kmitočtu

```
process (mclk,ares)
begin
  if (ares = '0') then
    word_count <= (others => '0');
  elsif (falling_edge(mclk)) then
    word_count <= word_count + to_unsigned(1,word_count'length);
  end if;
end process;

process (word_count)
begin
  word_clk <= word_count(7);
end process;
```

Nyní, když máme potřebný hodinový signál, stačí realizovat druhou část komponenty, a to již zmíněnou trojici diferenčních členů. Jejich funkčnost zajišťuje následující proces, ve kterém na začátku opět vidíme nulování všech proměnných při resetu, dále je zde provedena samotná diference a nakonec uložení stavů jednotlivých signálů, abychom je měli v příštím průchodu o jeden krok zpožděné.

Zdrojový kód 10: Derivační část

```
process (word_clk,ares)
begin
  if (ares = '0') then
    acc3_delayed <= (others => '0');
    diff1_delayed <= (others => '0');
    diff2_delayed <= (others => '0');
    diff1 <= (others => '0');
    diff2 <= (others => '0');
    diff3 <= (others => '0');
  elsif (rising_edge(word_clk)) then
    diff1 <= acc3 - acc3_delayed;
    diff2 <= diff1 - diff1_delayed;
    diff3 <= diff2 - diff2_delayed;
    acc3_delayed <= acc3;
    diff1_delayed <= diff1;
    diff2_delayed <= diff2;
  end if;
end process;
```

Nakonec už jen překopírujeme horních 16 bitů signálu *diff3* na výstupní port *data* a tím máme funkci digitálního filtru $SINC^3$ hotovou. S výstupní hodnotou musíme ovšem provést malou úpravu. Číslo, které z převodníku dostaneme, je v takovém formátu, že nejmenší číslo odpovídá nejmenšímu měřenému napětí, což je záporné číslo, hodnota uprostřed rozsahu odpovídá nulovému napěťovému signálu a maximální hodnota odpovídá největšímu kladnému napětí. Jelikož pro další výpočty potřebujeme zakódovat ve formátu dvojkového doplňku, aby pro něj platily běžné matematické operace, musíme toto číslo do tohoto formátu převést. K tomu stačí pouze invertovat nejvyšší bit tohoto čísla. V hlavním souboru popisující celý tento blok už jen vytvoříme tři instance tohoto filtru a tímto je celá komponenta hotová.

6.3 Inkrementální snímač polohy

Další velmi důležitou komponentou, kterou si nyní popíšeme, je inkrementální snímač polohy. IP blok, který zajišťuje čtení z tohoto snímače, je opatřen vstupy *A*, *B* a *INDEX*, ke kterým budou připojeny odpovídající signály z inkrementálního snímače, jejichž význam jsme si popsali v kapitole 2. V této kapitole jsme si také popsali, že pokud použijeme inkrementální snímač ke stanovení absolutní polohy, budeme muset zajistit nějaký referenční bod se známým úhlem natočení. V našem případě budeme zjišťovat vzájemné natočení mezi nulovým natočením magnetického pole a polohou, kdy dostaneme impulz na nulovacím vstupu. Abychom tuto kalibraci mohli řídit, musí být komponenta opatřena vstupem, který ji bude spouštět a výstupem, kterým bude indikovat, že byla kalibrace ukončena. Je zřejmé, že požadovaným výstupem z tohoto bloku bude aktuální úhel natočení. Kromě toho budeme také potřebovat i rychlost otáčení, jejíž výpočet byl zařazen do této komponenty, tudíž z ní vystupuje i tato

vypočítaná rychlost. Jak si popíšeme za chvíli, pro výpočet rychlosti budeme potřebovat hodinový signál, který je tedy zařazen mezi vstupy do tohoto bloku.

Obrázek 19: IP blok pro čtení inkrementálního snímače

Aby byla tato komponenta co nejuniverzálnější, je zde možné nastavit několik parametrů. Prvním z nich je počet impulzů na otáčku, které dostaneme z použitého snímače. Také je velmi důležité nastavit počet pólových párů, které obsahuje použitý motor, jelikož nás příliš nezajímá jaký je skutečný úhel natočení motoru, ale hodnota elektrického úhlu. V případě, že bychom tedy měli motor se dvěma pólovými páry, tak elektrický úhel se otočí dvakrát během jedné skutečné otáčky motoru. Také je nutné stanovit pořadí signálů A a B, které bude nutné pro stanovení správného směru otáčení. Posledním parametrem je opět frekvence hodinového signálu, kterou budeme potřebovat při výpočtu rychlosti.

Nyní si popíšeme princip funkčnosti této komponenty. Na začátku je přínosné provést filtraci vstupních signálů, jelikož se může stát, že při přepínání úrovní dojde k nežádoucím zákmitům, což by způsobovalo velkou odchylku při měření úhlu natočení. Zde můžeme s výhodou využít toho, že když se změní jeden signál, tak dokud nedojde ke změně druhého signálu, tak by se ten první již neměl změnit. Proto si zaznamenáme pouze jeho první změnu a až do změny toho druhého budeme ignorovat změny prvního signálu a budeme si jej držet v paměti stejný jako po první změně.

Zdrojový kód 11: Filtr vstupních signálů

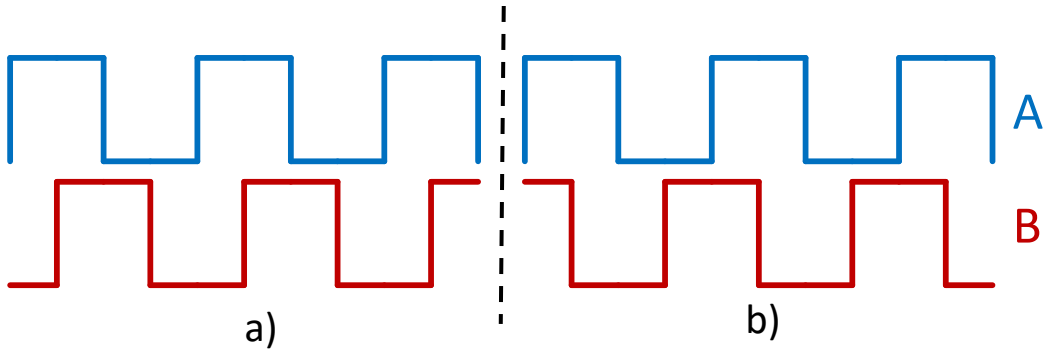
```
sens_in <= sens_b_in & sens_a_in;
case sens_in is
    when "00" => sens_q1 <= '0';
                 sens_q2 <= sens_q2;
    when "01" => sens_q1 <= sens_q1;
                 sens_q2 <= '0';
    when "10" => sens_q1 <= sens_q1;
                 sens_q2 <= '1';
    when "11" => sens_q1 <= '1';
                 sens_q2 <= sens_q2;
    when others => sens_q1 <= sens_q1;
                   sens_q2 <= sens_q2;
```

Abychom mohli použít příkaz *case*, je potřeba vstupní signály sloučit do vektoru, a poté už můžeme provést výše zmíněnou operaci. Dále potřebujeme detekovat hrany jednotlivých signálů. To se provádí tak, že si vytvoříme buffer, který si bude pamatovat hodnotu signálu z předešlého kroku, a poté už ji jen porovnáme s aktuální.

Zdrojový kód 12: Detekce hran

```
process(clk)
begin
  if clk'event and clk='1' then
    delay_sens_q1 <= sens_q1;
    delay_sens_q2 <= sens_q2;
    if sens_q1='1' and delay_sens_q1='0' then
      sens_event <= '1';
      sens_direction <= not sens_q2 xor signal_order;
    else
      sens_event <= '0';
      sens_direction <= sens_direction;
    end if;
  end if;
end process sens_event_detector;
```

V části kódu je vidět, že detekujeme nástupnou hranu prvního signálu a také určujeme směr otáčení. Z obrázku 20 je zřejmé, že při nástupné hraně signálu A můžeme určit směr otáčení podle úrovně signálu B. Je zde také provedena operace *xor* s proměnnou která je nastavena v závislosti na parametru, který definuje pořadí obou signálů.



Obrázek 20: Průběhy signálů A a B při otáčení a) v kladném směru; b) v záporném směru

Obdobným způsobem je potřeba detekovat nástupnou hranu nulovacího signálu. Tímto máme hotový hranový detektor a vyřešený směr otáčení. Další částí bude čítač těchto hran, který musí být schopný měnit směr čítání nahoru nebo dolů v závislosti na směru otáčení. Také by měl obsahovat nulování při příchodu impulzu generovaného referenční značkou, ale to pro nás v této aplikaci není žádoucí kvůli posunutí polohy této značky vůči nulovému natočení magnetického pole. Tudíž je potřeba místo nulování čítač přednastavovat na hodnotu danou tímto posunutím. K nulování by mělo dojít právě při spuštění kalibrace tohoto modulu.

Zdrojový kód 13: Čítač impulzů

```
angle_cnt: PROCESS (clk)
  variable cnt : integer range 0 to pulses;
  variable inc : integer;
begin
  if (sens_direction = '1') then
    inc := 1;
  else
    inc := -1;
  end if;

  if (clk'event and clk = '1') then
    if zero_calib_event='1' then
       cnt := 0;
    else
       if index_event='1' and zero_calib_in='0' then
         cnt := zero_offset;
       else
         if sens_event='1' then
          cnt := cnt + inc;
         end if;
       end if;
    end if;
  end if;
  angle_counter <= cnt;
end process angle_cnt;
```

Další je potřeba vyřešit již zmíněnou kalibraci. Myšlenka je taková, že nadřazený systém zajistí natočení rotoru do nulové polohy a přepne signál *zero_calib* do stavu logické 1. V tu chvíli se musí hodnota čítače vynulovat. Poté se motor začne otáčet a ve chvíli, kdy dostaneme impulz na nulovacím signálu, tak si uložíme hodnotu čítače, ve které se v tu chvíli nachází, a vygenerujeme signál, který signalizuje úspěšné ukončení kalibrace. Poté by měl nadřazený systém vypnout signál *zero_calib* a od té chvíle se bude čítač při každém příchodu nulovacího impulzu přednastavovat na tuhle uloženou hodnotu. Nakonec už jen stačí podle hodnoty čítače vypočítat správnou hodnotu elektrického úhlu podle následujícího vzorce.

$$\varphi = \frac{\left(c \bmod \frac{n}{m}\right) \times \varphi_{\max} \times m}{n} \qquad 6\text{-}2$$

Kde c je hodnota čítače, n je počet impulzů na otáčku, m počet pólových párů a φ_{max} je maximální hodnota výstupní proměnné. Po tomto přepočtu tedy dostaneme hodnotu normalizovanou na plný rozsah proměnné. Pokud tedy použijeme 16-ti bitové číslo, bude hodnota φ_{max} rovna 65535 a dostaneme výsledek v rozsahu 0 – 65535.

Dále si popíšeme princip měření otáček motoru. Úhlová rychlost je definována jako změna úhlu za jednotku času. Změnu polohy získáme opět z inkrementálního snímače a čas můžeme měřit pomocí hodinového signálu. Pro výpočet rychlosti se nám nabízí dvě

varianty a to buď počítat periody hodin mezi dvěma impulzy enkodéru, nebo počítat impulzy z enkodéru během jedné periody hodinového signálu. Pokud ovšem uvažujeme, že hodinový signál bude v řádu desítek MHz, bude jeho perioda v řádu desítek ns. U pulzů z enkodéru je předpokládaná perioda v řádu jednotek μs, pokud uvažujeme, že hodnota otáček bude v řádu tisíců a počet pulzů enkodéru rovněž v tisících. Proto bude v této aplikaci vždy lepší první varianta, tedy počítat periody hodinového signálu. K tomu budeme opět potřebovat čítač, z jehož hodnoty můžeme při detekování impulzu z inkrementálního snímače spočítat rychlost.

Zdrojový kód 14: Čítač pro měření rychlosti

```
process (clk)
  variable   cnt         : integer range 0 to MAX_SPEED_CNT+1;
begin
  if (clk'event and clk = '1') then
    if sens_event='1' then
      cnt := 0;
    else
      if cnt <= max_speed_cnt then
        cnt := cnt + 1;
      end if;
    end if;
  end if;
  speed_counter <= cnt;
end process;
```

Jak je vidět ve zdrojovém kódu, čítač je nutné zastavit, pokud se dostane k maximální hodnotě, aby nedocházelo k jeho přetečení a při detekci impulzu z enkodéru je jej potřeba vynulovat. V tuto chvíli musíme také spočítat rychlost podle následujícího vzorce.

$$\omega = \frac{\Delta\varphi}{\Delta t} = \frac{\frac{1}{n}}{T_{clk}} = \frac{F_{clk}}{n} \qquad 6\text{-}3$$

Kde *n* je opět počet impulzů na jednu otáčku. Takto vypočítaná hodnota odpovídá úhlové rychlosti v otáčkách za sekundu, a jelikož je zvykem ji udávat jako počet otáček za minutu, je potřeba tuto hodnotu ještě vynásobit 60. Po praktických testech snímání rychlosti se ukázalo, že je potřeba rychlost filtrovat, byl přidán ještě buffer, který si pamatuje posledních 5 hodnot, ze kterých je počítán vážený průměr. Také je potřeba detekovat stav, kdy se čítač dostal na horní hranici a nepřišel žádný impulz z enkodéru, což znamená, že se motor netočí a rychlost je tedy nulová.

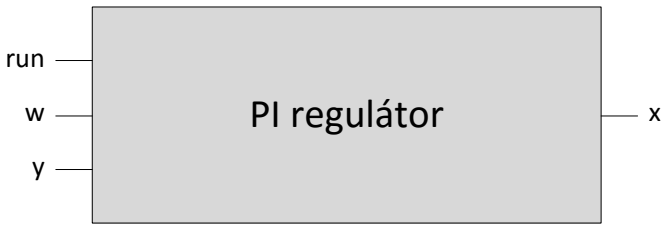
Tímto je tedy funkčnost této komponenty hotová a máme všechny potřebné součásti pro interakci s hardwarovou částí.

6.4 Regulátory

Pro regulaci motoru je potřeba vytvořit komponentu, která bude provádět algoritmus regulátoru. Jak jsme si uvedli v minulé kapitole, budeme používat sadu PI regulátorů zařazenou v kaskádě. Pro výpočet výstupu diskrétního PI regulátoru platí následující diferenční rovnice.

$$x_k = K\left(e_k + \frac{T_{vz}}{T_i}\sum_{i=0}^{k} e_i\right) \qquad 6\text{-}4$$

Kde K a T_i jsou parametry regulátoru, tedy proporcionální zesílení a integrační časová konstanta, e značí regulační odchylku a T_{vz} je perioda vzorkování. Je tedy zřejmé, že do komponenty, která bude provádět tuto operaci, musí vstupovat regulační odchylka nebo jako v našem případě žádaná a měřená hodnota, ze kterých je odchylka dopočítána uvnitř bloku a výstupem bude hodnota akčního zásahu.



Obrázek 21: Komponenta PI regulátoru

Je také velice žádoucí při vypnutí vynulovat obsah integrační složky, aby při příštím spuštění začínala opět od nuly. Proto je zde mezi vstupy zařazen signál *run*, stejný jako ten, který spouští modul pro generování PWM. Dále má tato komponenta několik nastavitelných parametrů jako periodu vzorkování a také jednotlivé parametry regulátoru. Ty jsou zadávány v upraveném tvaru jako zesílení proporcionální složky a zesílení integrační složky. Změna oproti 4-4 je v tom, že místo konstanty T_i zadáváme podíl K/T_i. Tento zápis má jednu praktickou výhodu a to, že jsme schopni pouhým nastavením tohoto parametru na nulu vyřadit integrační složku, čímž dostaneme čistě proporcionální regulátor. Mohlo by také být žádoucí nastavovat tyto parametry za běhu programu nějakým nadřazeným systémem. Proto byl mezi parametry zařazen přepínač mezi výše popsanými nastavenými parametry a parametry které jsou zařazeny mezi vstupy do tohoto bloku. Pokud se tedy zvolí druhá možnost, budou nastavené parametry ignorovány a přibudou další vstupy do tohoto bloku, které tyto parametry nahradí.

Nyní se můžeme přesunout k vnitřnímu popisu komponenty. Jelikož se jedná pouze o matematickou operaci a v jazyce VHDL existují knihovny pro práci se znaménkovým datovým typem, tudíž můžeme bez problému provádět všechny běžné matematické operace. Musíme ovšem brát v potaz fakt, že můžeme pracovat pouze s celočíselným datovým typem, tudíž je potřeba vynásobit všechna čísla zvoleným koeficientem tak, aby horní část odpovídala celé části a dolní část desetinné části. V tomto konkrétním

případě jsou v celém projektu použity vektory o šířce 16 bitů. Pro počítání v tomto bloku jsou všechny proměnné 32 bitové s tím, že horních 16 bitů odpovídá celé části a dolních 16 bitů desetinné. Všechna čísla jsou tedy vynásobena 2^{16}, což odpovídá tomuto bitovému posunu o 16 bitů vlevo. Po provedení všech operací je na výstup vyvedeno pouze horních 16 bitů, tedy pouze celá část. Toto rozšíření o desetinnou část je velice důležité, protože integrační složka se nemusí v jednom kroku navýšit pouze o celé číslo, ale jen o zlomek, který se bude dále navyšovat v každém kroku, dokud se neprojeví navýšením celé části. Pokud bychom počítali pouze s celými čísly, tak by se inkrementace vlivem zaokrouhlování zastavila. Po této úpravě můžeme provést potřebné matematické operace. Nejprve si vypočítáme regulační odchylku jako rozdíl žádané a měřené hodnoty a poté můžeme vypočítat jednotlivé složky regulátoru. Proporcionální složka je jen prosté vynásobení odchylky a proporcionálního zesílení a pro vytvoření integrační složky musíme zrealizovat sumátor. Ten je svou strukturou velmi podobný čítači, rozdíl je pouze v tom, že v každém kroku musíme vypočítat novou hodnotu přírůstku.

Zdrojový kód 15: Integrační složka regulátoru

```
process (clk) is
  variable sum : integer := 0;
  variable inc : integer := 0;
begin
  if (clk'event and clk = '1') then
    if sync_in='1' then
      inc := Kt_norm * to_integer(e_signed);
      if sum >= 0 then
        if inc < 2147483647 - sum then
          sum := sum + inc;
        else
          sum := 2147483647;
        end if;
      else
        if inc > -2147483647 - sum then
          sum := sum + inc;
        else
          sum := -2147483647;
        end if;
      end if;
    end if;
  end if;
end process;
```

Také musíme zajistit, aby nedošlo k přetečení proměnné, proto je tedy nutné přidat podmínku, aby se inkrementace zastavila na maximální hodnotě. Zvolená maximální hodnota zároveň odpovídá maximální hodnotě akčního zásahu, tudíž je tímto vyřešeno i potlačení wind-up jevu. Další důležitou věcí je to, že je počítání synchronizováno hodinovým signálem, ale tato operace nemůžeme proběhnout při každém příchodu hrany hodinového signálu. Celá tato operace musí proběhnout pouze jednou během

periody vzorkování. K zajištění popsané funkčnosti slouží impulz označený jako *sync_in*, který se generuje pouze jednou během této periody a při tomto zápisu slouží jako signál, který povolí nebo zakáže operaci sumátoru. Nakonec už jen stačí obě složky sečíst, ovšem i při tomto sčítání si opět musíme dát pozor, abychom nepřekročili maximální hodnotu a proměnná nám opět nepřetekla.

Takto navržený regulátor můžeme použít jako otáčkový, popřípadě jako polohový, pokud bychom chtěli tuto aplikaci rozšířit o regulaci polohy. Pro použití k regulaci proudu je potřeba ještě několik úprav. Je zřejmé, že budeme muset regulovat obě složky vektoru proudu v *d*-*q* souřadnicovém systému, ale nemůžeme je regulovat jako dvě zcela samostatné veličiny. Problém je v tom, že výsledná hodnota akčního zásahu musí být omezena na polovinu napájecího napětí. Jelikož je naším akčním zásahem střída, která v podstatě udává procentuální část napájecího napětí, je tento problém značně zjednodušený, jelikož se nemusíme potýkat se skutečným fyzikálním rozměrem této veličiny. Stačí se tedy pouze vypořádat se správnou hodnotou střídy, což opět znamená jen to, aby hodnota akčního zásahu zůstala v mezích, které jsou dány velikostí 16-ti bitového čísla. Na tuto hodnotu musí ovšem být omezený modul našeho proudového vektoru, nikoliv jeho složky. Musí tedy platit následující podmínka.

$$\sqrt{U_d^2 + U_q^2} \leq \frac{U_{DC}}{2} \qquad 6\text{-}5$$

A jelikož tedy maximální hodnota napájecího napětí vznikne při střídě 100%, čemuž odpovídá hodnota 2^{16}, tak bude naše limitní hodnota po vydělení dvěma 2^{15}. Prakticky se splnění této podmínky provádí tak, že se jedné složce vektoru, zpravidla *d* složce, dává priorita, která tedy může nabývat plného rozsahu a druhá složka je omezena na hodnotu, která pro ni zbude podle následující rovnice.

$$U_{q,\max} = \sqrt{U_d^2 - \left(\frac{U_{DC}}{2}\right)^2} \qquad 6\text{-}6$$

Je tedy potřeba vytvořit komponentu, která bude obsahovat dva PI regulátory, kde první z nich bude naprosto totožný, jako ten který jsme si popsali před chvílí, a ten druhý bude upravený pouze v tom, že místo konstantní hodnoty, která omezuje maximum integrační složky a výstupní hodnoty akčního zásahu, použijeme proměnnou, jejíž hodnotu vypočítáme podle rovnice 6-6. Pro výpočet odmocniny jsem vytvořil další komponentu, která obsahuje iterační algoritmus, který v podstatě nedělá nic jiného než to, že začne s číslem uprostřed rozsahu, to znamená 1 na pozici nejvyššího bitu, a otestuje, jestli je po umocnění na druhou menší nebo větší než vstupní číslo. Pokud je menší, 1 na této pozici zůstává, jinak se nahradí 0. V dalším kroku se umístí 1 o jeden bit níže a celý postup se opakuje.

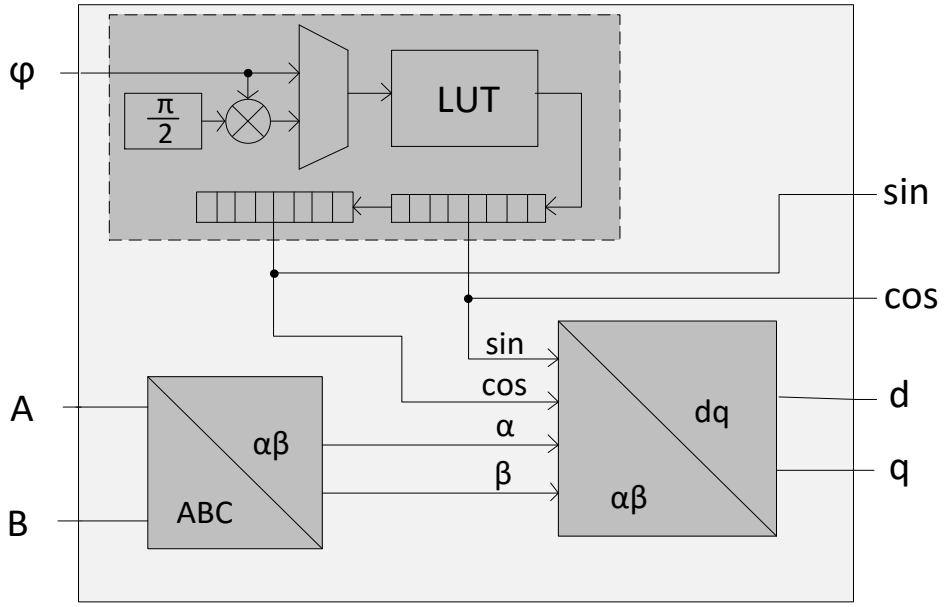
Nyní už jen stačí zajistit časovou posloupnost jednotlivých kroků. Nejprve je totiž potřeba vypočítat hodnotu akčního zásahu pro *d* složku, poté vypočítat omezující hodnotu pro druhý regulátor a nakonec hodnotu akčního zásahu *q* složky. Proto jsem přidal sadu synchronizačních impulzů, které se generují po vypočítání každého kroku, a ten následující je vždy pozastaven, dokud se neaktivuje synchronizační signál z přecházejícího kroku.

6.5 Transformace souřadnic

Pokud chceme provádět regulaci v *d-q* souřadnicích, musíme do tohoto systému převést měřené proudy a akční zásahy zase naopak převádět z *d-q* roviny zpět na napětí pro jednotlivé fáze motoru. K tomu budou sloužit dvě komponenty, které si nyní popíšeme.

6.5.1 Transformace z ABC do d-q

První z transformací, jejíž zjednodušené schéma je na obrázku 22, bude provádět transformaci proudů do *d-q* souřadnicového systému.



Obrázek 22: Blokové schéma modulu pro transformaci z ABC do d-q

Vstupem do této komponenty jsou naměřené proudy z jednotlivých zpracované SINC^3 filtrem a elektrický úhel natočení rotoru. Z těchto hodnot jsou vypočítány odpovídající proudy v *d-q* rovině, které jsou také výstupem tohoto bloku. Kromě nich z komponenty vystupují také vypočítané hodnoty sinu a kosinu. Ty jsou zde zařazeny z důvodu optimalizace, jelikož stejné hodnoty budeme potřebovat i při transformaci v opačném směru a vhledem k tomu, že je jejich výpočet docela náročný z hlediska potřebného hardwaru v hradlovém poli, bylo by zbytečné je počítat znovu.

Jak je vidět na obrázku 22, celá tato komponenta se skládá ze tří dílčích částí. První z nich počítá zmíněný sinus a kosinus, druhá provádí transformaci z *ABC* do *α-β* souřadnic a poslední provádí rotaci tohoto vektoru do *d-q* roviny. Nejprve se tedy podíváme na výpočet hodnot zmíněných goniometrických funkcí. Pro tuto operaci byla zvolena cesta přes vyhledávací tabulku, ve které je uložena navzorkovaná první čtvrtina periody funkce sinus. Jelikož máme k dispozici pouze celá čísla, jsou tyto hodnoty vynásobené koeficientem 2^{15}, čímž dostaneme 15 bitů pro desetinnou část. Nejvyšší 16. bit nese informaci o znaménku. Ukázku vyhledávací tabulky můžeme vidět v následující části kódu.

Zdrojový kód 16: Ukázka LUT

```
Sin_LUT <= to_signed(0,16) when angle_LUT < to_unsigned(16,14) else
           to_signed(51,16) when angle_LUT < to_unsigned(32,14) else
           to_signed(101,16) when angle_LUT < to_unsigned(48,14) else
       …
```

Jak jsem již zmínil, tabulka obsahuje pouze čtvrtinu periody, jelikož nám stačí i pro výpočet úhlu, který se nachází v jiné části, protože zbytek sinusového signálu je vždy nějakým způsobem symetrický s první čtvrtinou. Je tedy potřeba nejprve zjistit v jaké čtvrtperiodě se vstupní úhel nachází. Jelikož je hodnota úhlu normalizovaná na plný rozsah 16-ti bitového čísla neznaménkového typu, je možné podle nejvyšších dvou bitů určit ve které čtvrtině rozsahu úhel leží. Pokud jsou tedy horní dva bity rovny hodnotě „00“, jedná se o první čtvrtperiodu, hodnotě „01“ odpovídá druhá atd. Spodních 14 bitů pak odpovídá hodnotě úhlu, pokud bychom ji počítali od začátku čtvrtperiody, ve které úhel leží. To jsou přesně ty hodnoty, které by měly vstupovat do vyhledávací tabulky. Je tedy potřeba rozdělit vstupní úhel na horní 2 a 14 bitů a na základě horních dvou určit kvadrant. Pokud se úhel nachází v prvním kvadrantu, není potřeba provádět žádné úpravy. Ve druhém kvadrantu musíme úhel odečíst od 90°, aby byla výstupní hodnota z tabulky správná. Totéž platí pro čtvrtý kvadrant.

Zdrojový kód 17: Úprava vstupního úhlu

```
quadrant <= angle(15 downto 14);
angle_LUT <= angle(13 downto 0) when quadrant(0) = '0' else
             to_unsigned(16383,14) - angle(13 downto 0);
```

Další úpravu je potřeba provést s výstupem z tabulky. Pro třetí a čtvrtý kvadrant je potřeba vytvořit opačné číslo vůči tomu, které jsme z tabulky dostali.

Zdrojový kód 18: Úprava výstupní hodnoty

```
sin <= sin_LUT when quadrant(1) = '0' else -sin_LUT;
```

Nyní jsme tedy schopni vypočítat hodnotu funkce sinus. Pro kosinus využijeme tutéž tabulku, jen do ní pošleme úhel o 90° posunutý, jelikož platí následující rovnost.

$$\cos(\varphi) = \sin\left(\varphi + \frac{\pi}{2}\right) \qquad 6\text{-}7$$

Komponenta pro výpočet obou funkcí tedy funguje tak, že při příchodu prvního hodinového signálu pošle do vyhledávací tabulky vstupní hodnotu úhlu a při příchodu druhého hodinového signálu ji přepne na hodnotu posunutou o 90°. Je tedy zřejmé, že stejně jako u regulátoru musíme vytvořit synchronizační signál, kterým budeme zpožďovat jednotlivé kroky výpočtu, jelikož dokud nemáme spočítaný sinus i kosinus, nemůžeme počítat transformaci. Vstup a výstup této komponenty opatřen synchronizačními impulzy, aby bylo možné dodržet časovou posloupnost jednotlivých kroků.

Nyní se přesuneme k jednotlivým transformacím. Pro každou z nich byla vytvořena samostatná komponenta, kde každá z nich neobsahuje nic jiného než matematické operace popsané v kapitole 3. Opět si ale musíme uvědomit, že je potřeba vynásobit desetinná čísla nějakým koeficientem, abychom mohli pracovat i s desetinnou částí jednotlivých operandů. Nejprve se tedy provede Clarkové transformace, která přepočítá vstupní signály *ABC* na výstupní signály *αβ*. Ty se vypočítají podle rovnic 3-4, na základě kterých byl sestaven následující VHDL popis. Konstanty, kterými jsou násobeny signály *A* a *B* jsou ovšem vynásobené 2^{14}. Na výstup je opět odeslána pouze celá část vypočítaného čísla.

Zdrojový kód 19: Clarkové transformace

```
alpha <= a;
mul_temp1 <= to_signed(9459, 16) * a;
mul_temp2 <= to_signed(18919, 16) * b;
add_temp <= mul_temp1 + mul_temp2;
beta <= resize(add_temp(31 downto 14),16);
```

Obdobným způsobem byl vytvořen kód pro druhou část transformace, tedy otočení vektoru z *α-β* souřadnicového systému do roviny *d-q*. Zde použijeme vypočítané hodnoty sinu a kosinu a dosadíme je do rovnic 3-6. Tyto rovnice je opět potřeba převést do VHDL popisu, který můžeme vidět níže. Tentokrát máme vynásobení konstantou již schované v hodnotách sinu a kosinu, které, jak jsme si řekli, jsou vynásobeny 2^{15}. Tuto část musíme opět odebrat od výsledného čísla, abychom dostali pouze celou část.

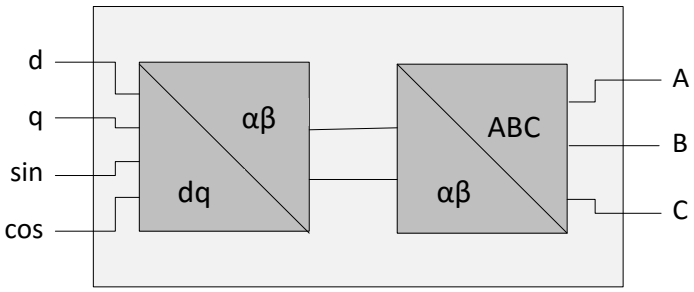
Zdrojový kód 20: Parkova transformace

```
mul_temp1 <= cos * alpha;
mul_temp2 <= sin * beta;
mul_temp3 <= sin * alpha;
mul_temp4 <= cos * beta;
add_temp1 <= mul_temp1 + mul_temp2;
add_temp2 <= mul_temp4 - mul_temp3;
d <= resize(add_temp1(31 downto 15),16);
q <= resize(add_temp2(31 downto 15),16);
```

Na konci kódů pro obě transformace si můžeme všimnout, že se používá funkce *resize*, která mění velikost vektoru, který je jeho parametrem. V našem případě jde o zkrácení vektoru na 16 bitů. Toto zkrácení si můžeme dovolit, jelikož používáme sadu rovnic, která zajišťuje stejnou amplitudu signálů. Je tedy jisté, že stačí, aby byl výstupní vektor ve stejném rozsahu jako ten vstupní.

6.5.2 Transformace z d-q do ABC

Posledním blokem, který je nutný pro řízení motoru, je ten, který zajistí transformaci vypočítaných akčních zásahů regulátoru zpět do třífázového systému. Ta se opět provádí ve dvou krocích, kdy první z nich provádí rotaci vstupního vektoru *d-q* do souřadnicového systému *α-β* a druhý vypočítá odpovídající hodnoty napětí pro jednotlivé fáze.



Obrázek 23: Transformace z d-q do ABC

Pro každou z těchto operací byla opět vytvořena samostatná komponenta. Do první z nich vstupují složky vektoru *d* a *q* a také hodnoty goniometrických funkcí sinus a kosinus pro aktuální úhel natočení rotoru, které jsou vypočítány v komponentě, kterou jsme si popsali v minulé podkapitole. Výstupem jsou složky vektoru v *α-β* souřadnicích, které lze vypočítat pomocí rovnic 3-8.

Nyní už jen stačí převést tyto rovnice do VHDL popisu. Ten je strukturou podobný jako transformace v opačném směru. Opět se zde vstupy násobí úhlem, který je vynásobený 2^{15}, proto nás nakonec spodních 15 bitů, které představují desetinnou část, nezajímá a na výstup se posílají pouze zbylé horní bity. Opět si můžeme dovolit oříznout výstupní vektor na stejnou velikost jako je ten vstupní, tedy 16 bitů.

Zdrojový kód 21: Inverzní Parkova transformace

```
mul_temp1 <= cos * d;
mul_temp2 <= sin * q;
mul_temp3 <= sin * d;
mul_temp4 <= cos * q;
add_temp1 <= mul_temp1 - mul_temp2;
add_temp2 <= mul_temp3 + mul_temp4;
alpha <= resize(add_temp1(31 downto 15),16);
beta <= resize(add_temp2(31 downto 15),16);
```

Vypočítané hodnoty α a β vstupují do druhé komponenty, která z nich vypočítá jednotlivá fázová napětí. K tomu byla použita inverzní Clarkové transformace, kterou jsme si popsali v kapitole 3. V dnešní době je sice v praxi mnohem rozšířenější metoda, která je založená na modulaci prostorového vektoru, z angl. *Space Vector Modulation,* ale pro ověření funkčnosti nám bude stačit použít výše zmíněnou transformaci. Na základě rovnic 3-5 byl tedy vytvořen odpovídající VHDL popis, který opět pracuje s desetinnou částí díky vynásobení hodnotou 2^{15}. Na výstup je potom poslána pouze celá část výsledku, která je stejně jako v předchozích případech zkrácena na požadovanou délku.

Zdrojový kód 22: Inverzní Clarkova transformace

```
mul_temp1 <= -alpha * to_signed(16384, 16);
mul_temp2 <= to_signed(28378, 16) * beta;
add_temp1 <= mul_temp1 + mul_temp2;
add_temp2 <= mul_temp1 - mul_temp2;
a <= alfa;
b <= resize(add_temp1(31 downto 15),16);
c <= resize(add_temp2(31 downto 15),16);
```

Tímto máme hotovou poslední komponentu, která je nutná pro řízení PMS motorů. Spojením všech těchto komponent tedy vznikne aplikace schopná komunikovat se všemi hardwarovými součástmi motoru a také provádět jeho regulaci.

6.6 Ostatní komponenty

V této části si popíšeme další komponenty, které byly vytvořeny pro zajištění správného chodu celé aplikace. Je totiž ještě nutné generovat potřebné řídicí signály a vytvořit rozhraní, pomocí kterého bude možné ovládat motor a nastavovat různé hodnoty.

Nejprve si popíšeme komponentu, která bude řídit kalibraci enkodéru při prvním spuštění aplikace. Vstupy a výstupy tohoto bloku si můžeme prohlédnout na obrázku níže.

Obrázek 24:Řídicí signály

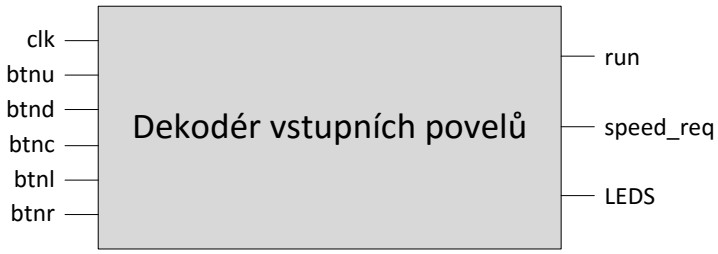
Princip činnosti této komponenty spočívá v tom, že po startu zařízení je potřeba vyslat povel PWM generátoru, aby provedl zarovnání rotoru do nulové polohy. Po časové prodlevě musí vygenerovat signál pro spuštění kalibrace enkodéru.

Zdrojový kód 23: Generování řídicích signálů

```
process (clk)
  begin
    if (clk'event and clk='1') then
      delay_run <= run;
      if delay_run='0' and run='1' and calibrated='0' then
        zero_align <= '1';
        aligning <= '1';
      end if;
      if counter = max_count and aligning = '1' then
        zero_calib <= '1';
        aligning <= '0';
        zero_align <= '0';
      end if;
      if zero_calib_done='1' then
         zero_calib <= '0';
         calibrated <= '1';
      end if;
    end if;
end process;
```

Nejprve je tedy potřeba detekovat nástupnou hranu signálu *run*, což je provedeno ukládáním jeho předchozí hodnoty. Pokud je tato hrana detekována, je přepnut signál *zero_align* do hodnoty logické 1, který dále vyhodnocuje PWM generátor a začne zarovnávat rotor do nulové polohy. Ve stejnou chvíli se spustí čítač, který je obsahem druhého procesu. Ve chvíli kdy čítač dosáhne maximální hodnoty, která lze nastavit pomocí parametru pro tento IP blok, dojde k vypnutí pokynu pro zarovnání a je možné vyslat povel, který v komponentě pro enkodér způsobí nastavení úhlu na nulovou hodnotu. Jakmile obdržíme zpět signál, který oznamuje ukončení kalibrace, je povel pro kalibraci vypnut. Také se nastaví pomocný signál *calibrated*, který zamezí další kalibraci při opětovném zastavování a spouštění motoru.

Další komponenta, kterou si popíšeme, provádí čtení jednotlivých tlačítek a přepínačů, podle kterých nastavuje signál *run* pro zapnutí motoru a požadovanou hodnotu pro regulátor rychlosti. Pro ovládání motoru byla vybrána pětice tlačítek na vývojové desce, kde prostřední tlačítko bude zapínat a vypínat motor, tlačítka vlevo a vpravo budou měnit směr otáčení a tlačítka nahoru a dolů budou zvyšovat nebo snižovat požadovanou rychlost. Do komponenty tedy vstupuje pět tlačítek a vystupuje z něj signál *run* a žádaná hodnota otáček. Dalším výstupem je vektor hodnot pro LED diody, abychom mohli signalizovat nastavení těchto parametrů.



Obrázek 25: Dekodér vstupních povelů

Jelikož máme v úmyslu inkrementovat hodnotu rychlosti v případě, že je stisknuté tlačítko, je potřeba přidat časovou prodlevu, aby přičítání neprobíhalo příliš rychle. Totéž platí pro tlačítko, kterým se bude negovat úroveň signálu *run*.

Zdrojový kód 24: Čtení tlačítek

```
if (clk'event and clk='1') then
  if (cnt = 0) then
    if (btnu='1' and speed_temp<to_signed(3500,13)) then
      speed_temp := speed_temp + to_signed(100,13);
      cnt := 10000000;
    elsif (btnd='1' and speed_temp>to_signed(0,13)) then
      speed_temp := speed_temp - to_signed(100,13);
      cnt := 10000000;
    elsif (btnc='1') then
      run_temp <= not run_temp;
      cnt := 25000000;
    end if;
elsif (cnt > 0) then
  cnt := cnt - 1;
end if;
```

Princip je tedy takový, že pokud dojde ke stisknutí tlačítka, provedou se operace, které po stisku požadujeme, a přednastaví se hodnota čítače, který poté začne čítat směrem dolů. Dokud se nedostane zpět k nule, nedochází k vyhodnocování stavu tlačítek. V další části jsou vyhodnocovány tlačítka vlevo a vpravo.

Zdrojový kód 25: Vyhodnocení směru

```
if (btnl='1') then
  sign := to_signed(1,2);
elsif (btnr='1') then
  sign := to_signed(-1,2);
end if;
```

Pro vyhodnocení směru je do proměnné *sign* nastavena hodnota 1 nebo -1, kterou je následně vynásobena výsledná hodnota požadované rychlosti. Nakonec je zde ještě logika pro rozsvěcování diod, kde první z nich bude svítit, pokud je aktivní signál *run* a zbylých 7 diod se postupně rozsvěcuje s přibývající rychlostí.

7 OVĚŘENÍ FUNKČNOSTI

Obsahem této kapitoly je, jak již název napovídá, ověření funkčnosti jednotlivých komponent, které jsme si popsali v kapitole 6. Jednotlivé IP bloky byly testovány ve dvou krocích. Nejprve s použitím simulátoru, který je součástí vývojového prostředí *Vivado* a následně přímo v hradlovém poli. Jelikož jsou výstupy všech bloků, kromě generátoru PWM, reprezentovány pouze vnitřními signály, nebylo možné si jejich funkčnost ověřit měřením. Pro ověření byla využita AXI sběrnice, díky které je možné číst hodnoty signálů v hradlovém poli pomocí procesorové části, která je dále schopna komunikovat s počítačem přes sériovou sběrnici UART.

7.1 PWM generátor

První testovanou komponentou je PWM generátor. Postup, jak používat simulátor, jsme si popsali v kapitole 5.3.3. Byl tedy vytvořen soubor, který obsahuje definici a volání testované komponenty a tvarování jejích vstupních signálů. Soubor byl následně použit jako top-level pro simulaci. Nyní si popíšeme, jak byly definovány jednotlivé vstupy do testované komponenty. Prvním z nich je hodinový signál, pro který nastavíme frekvenci například 50 MHz.

Zdrojový kód 26: Generování hodinového signálu

```
process begin
  clk <= '0';
  wait for 10 ns;
  clk <= '1';
  wait for 10 ns;
end process;
```

Dále je potřeba definovat jednotlivé řídicí signály, tedy signály pro spuštění této komponenty a signál, který způsobí zarovnání motoru do výchozí pozice. V následujícím procesu je definováno, že ke spuštění má dojít po 200 ns od startu simulace a také, že prvních 500 μs bude komponenta v režimu pro zarovnání.

Zdrojový kód 27: Definice řídicích signálů

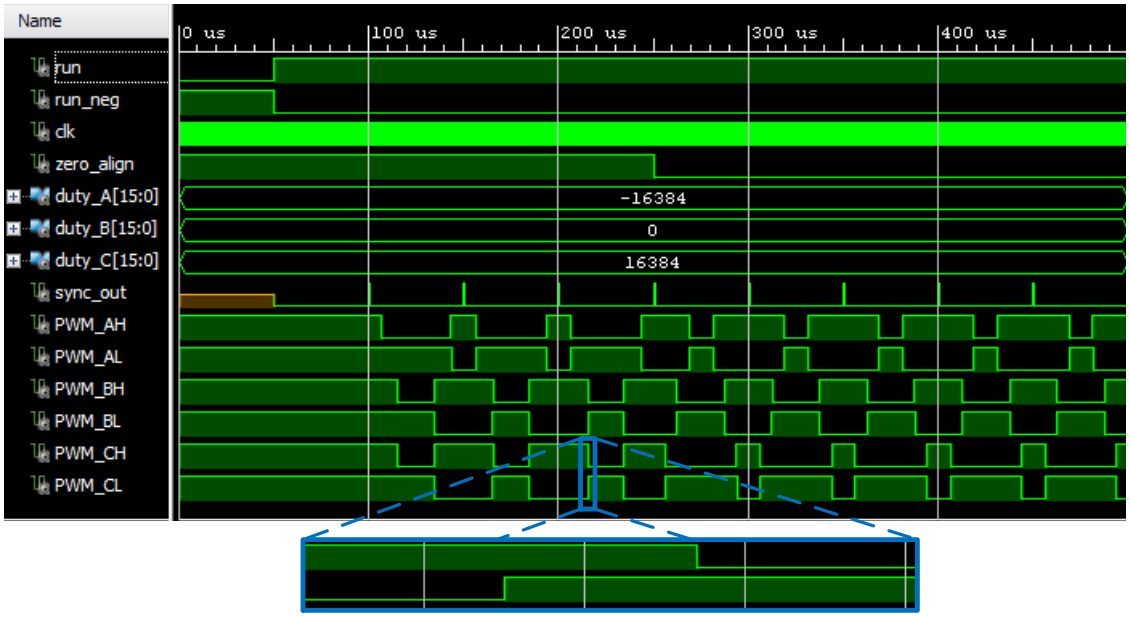
```
process begin
  run <= '0';
  run_neg <= '1';
  zero_align <= '1';
  wait for 200 ns;
  run <= '1';
  run_neg <= '0';
  wait for 500 us;
  zero_align <= '0';
  wait;
end process;
```

Posledními vstupy jsou ty pro jednotlivé střídy, které reprezentují 16. bitové znaménkové číslo. Pro testování nastavíme první z nich jako záporné číslo o hodnotě poloviny rozsahu, která by měla zajistit vygenerování 25% střídy, druhou jako nulu, které odpovídá 50% střída, a třetí nastavíme jako kladné číslo odpovídající 75% střídě.

Zdrojový kód 28: Definice ostatních potřebných signálů

```
process begin
  duty_A <= std_logic_vector(to_signed(16384, duty_A'length));
  duty_B <= std_logic_vector(to_signed(0, duty_B'length));
  duty_C <= std_logic_vector(to_signed(-16384, duty_C'length));
  wait;
end process;
```

Nakonec je potřeba zvolit požadované parametry pro tuto komponentu. Pro připomenutí se jedná o frekvenci hodinového signálu, která je 50 MHz, požadovanou frekvenci PWM, kterou nastavíme například na 20 KHz, dále aktivní úrovně jednotlivých výstupů a hodnota ochranné doby. Abychom otestovali i to, jestli funguje generování PWM v inverzní logice, nastavíme aktivní úrovně jako logickou 0 a pro ověření ochranné doby zvolíme například 100 ns.



Obrázek 26: Výstup PWM generátoru

Výsledek simulace si můžeme prohlédnout na obrázku 26, kde vidíme tři dvojice vzájemně negovaných signálů a signál, který indikuje konec jedné periody. Je vidět, že délka periody vyšla podle očekávání 50 µs, také je zde vidět, že funguje vypnutí modulu a na začátku simulace jsou všechny výstupy drženy v neaktivní úrovni, tedy logické 1. Také vidíme, že po dobu kdy je signál *zero_align* aktivní, jsou ignorovány naše nastavené hodnoty střídy a na výstupu se projeví hodnoty, které zajistí zarovnání rotoru s fází A. Po skončení doby pro zarovnání vidíme, že výstupní signály odpovídají nastaveným střídám, tedy 25% pro signály A, 50% pro B a 75% pro C. Nakonec si

v přiblížené části můžeme prohlédnout vliv ochranné doby, která zajistila, že k přepínání nedochází současně, ale tuto stanovenou dobu jsou oba signály v neaktivní úrovni.

7.2 Čtení analogových signálů

Nyní si popíšeme testovací skript pro druhou komponentu, která zajišťuje čtení analogových signálům, které jsou modulovány sigma-delta převodníkem. Postup pro vytvoření testovacího skriptu je stejný, jako u předchozí komponenty, přeskočím tedy rovnou k definici průběhů jednotlivých signálů.

Stejným způsobem, který jsme si popsali u předchozí simulace, byl vytvořen hodinový signál pro AD převodník o frekvenci 16 MHz.

Zdrojový kód 29: Generace hodinového signálu pro SINC^3 filtr

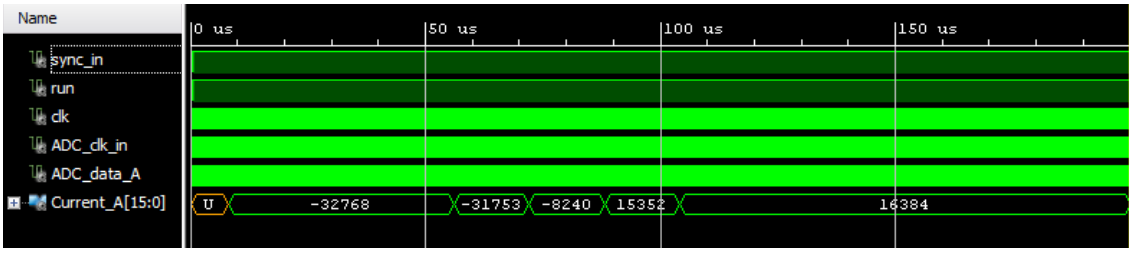
```
process begin
  ADC_clk_in <= '0';
  wait for 31.25 ns;
  ADC_clk_in <= '1';
  wait for 31.25 ns;
end process;
```

Dále je potřeba definovat průběh dat, které by přicházely ze sigma-delta modulátoru. Výstupem tohoto převodníku je PWM signál, jehož střída je úměrná měřenému napětí. Následující část kódu ukazuje vytvoření PWM signálu se střídou 75 %, která by odpovídala hodnotě napětí uprostřed kladné části rozsahu.

Zdrojový kód 30: Simulace výstupu sigma-delta převodníku

```
process begin
  ADC_data_A <= '0';
  wait for 125 ns;
  ADC_data_A <= '1';
  wait for 375 ns;
end process;
```

Výsledek simulace si můžeme prohlédnout na obrázku 27, kde je vidět, že po odeznění přechodového jevu je na výstupu hodnota, která odpovídá předpokladům. Výstupem je totiž 16. bitové znaménkové číslo, jehož kladná část rozsahu je 0 – 32767 a na výstupu simulace vidíme hodnotu 16384, která odpovídá polovině tohoto rozsahu.



Obrázek 27: Výstup SINC^3 filtru

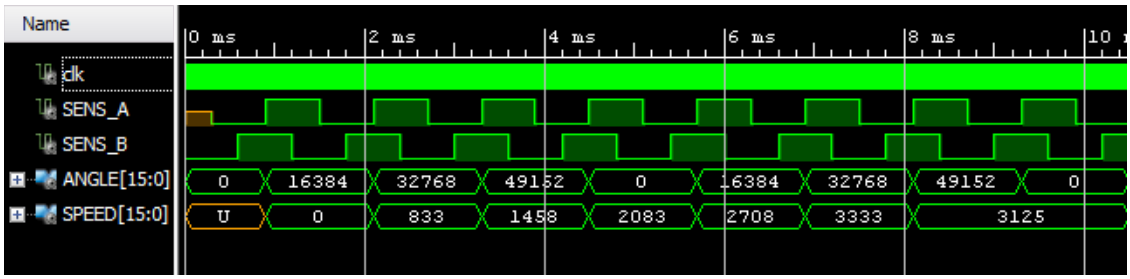
7.3 Inkrementální snímač polohy

Další testovanou komponentou je dekodér signálů z inkrementálního snímače polohy. Ta by tedy měla převést vstupní impulzy ze dvou signálů *A* a *B* na úhel natočení motoru a také by měla umožnit kalibraci nulové polohy. Abychom vše mohli otestovat, musíme si opět vytvořit simulační zdrojový soubor jako v předchozích případech nadefinovat potřebné průběhy vstupních signálů. Nejprve si vytvarujeme průběhy signálů *A* a *B*.

Zdrojový kód 31: Definice průběhů signálů A a B

```
process begin
  SENS_A <= '0';
  wait for 300 us;
  SENS_B <= '0';
  wait for 300 us;
  SENS_A <= '1';
  wait for 300 us;
  SENS_B <= '1';
  wait for 300 us;
end process;
```

Následně si definujeme parametry této komponenty. Prvním parametrem je počet impulzů na otáčku, který pro testovací účely nastavíme na 16, dalším je počet pólových párů, který bude pro tuto simulaci 4. Z toho vyplývá, že hodnota elektrického úhlu, která je výstupem, by měla v průběhu 4 impulzů projít celým svým rozsahem, tedy hodnotami 0 až 2^{16}.



Obrázek 28: Výstup dekodéru inkrementálního snímače

Na obrázku 28 vidíme výsledek naší simulace, ze které je patrné, že hodnoty úhlů se mění podle očekávání. Dále zde můžeme vidět hodnotu vypočítaného úhlu, která také odpovídá předpokládané hodnotě, jelikož ze zdrojového kódu 31 vyplývá, že perioda jednoho impulzu je 1,2 ms, což při 16 impulzech na otáčku odpovídá úhlové rychlosti 3125 ot/min. Správnou hodnotu ovšem vidíme až po šesti krocích, což je dáno jejím filtrováním.

7.4 PI regulátor

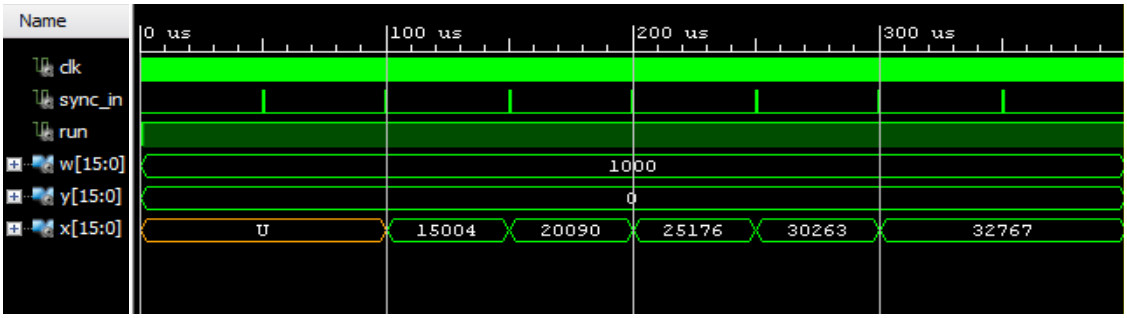
V této části si popíšeme testování další komponenty, kterou je PI regulátor. Nejprve je potřeba nadefinovat vstupní hodinový signál o frekvenci 50 MHz viz zdrojový kód 26.

Dále definujeme synchronizační signál, který je generován jednou během periody PWM signálu.

Zdrojový kód 32: Definice synchronizačního impulzu

```
process begin
  sync_in <= '0';
  wait for 49.980 us;
  sync_in <= '1';
  wait for 20 ns;
end process;
```

Tento impulz se bude generovat jednou za 50 µs, což odpovídá frekvenci 20 kHz. Dále je potřeba nastavit vstupní žádanou a měřenou hodnotu. Abychom se při této simulaci rychle dostali k limitním hodnotám, byly tyto hodnoty nastaveny tak, aby byla výsledná hodnota odchylky natolik vysoká, aby byl velký přírůstek při inkrementaci integrační složky. Žádaná hodnota byla tedy nastavena na 10.000 a měřená hodnota na nulu. Parametry regulátoru byly také nastaveny se stejným úmyslem. Zesílení pro tuto simulaci bylo 10 a hodnota zesílení integrační složky odpovídala časové konstantě 10 µs. Výstupní hodnota by tedy podle kontrolního výpočtu měla začít na 15.000 a s každým krokem se inkrementovat o 5.000.

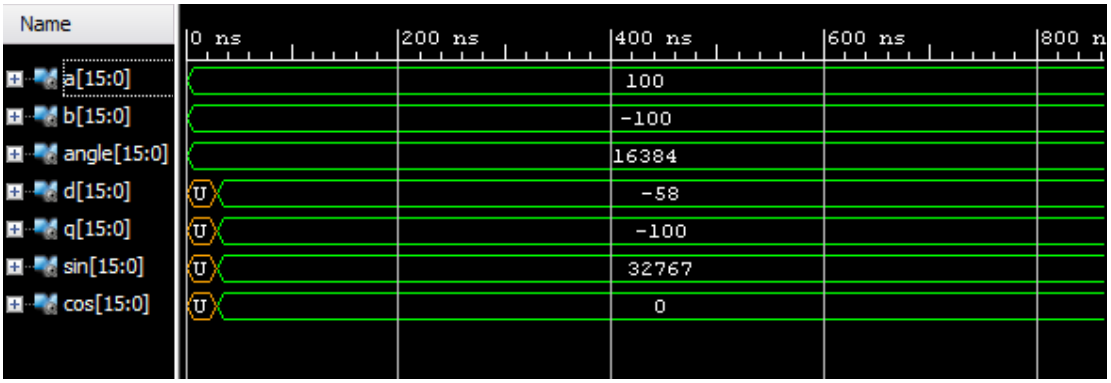


Obrázek 29: Výstup PI regulátoru

Jak je vidět na obrázku 29, výsledek simulace přibližně odpovídá teoretickému předpokladu. Vzniklá odchylka, které si můžeme všimnout, je způsobena celočíselným dělením, které je použito uvnitř komponenty. Také zde vidíme, že k inkrementaci skutečně dochází pouze při příchodu synchronizačního impulzu a výsledný akční zásah nepřekročil stanovenou maximální hodnotu.

7.5 Transformace souřadnic

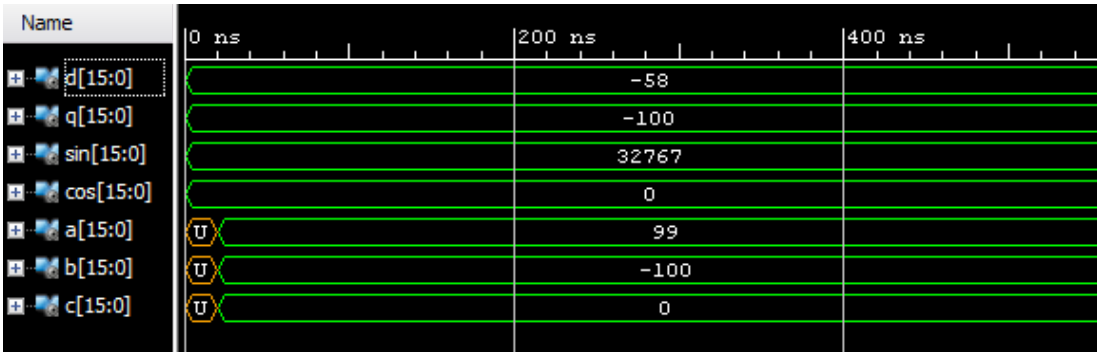
Nyní si ukážeme, jak byla ověřena funkčnost posledních dvou komponent, které zajišťují transformaci z třífázové soustavy do roviny *d*-*q* a zpět. Pro otestování první transformace si nadefinujeme následující hodnoty vstupních signálů. Pro fázový proud *A* zvolíme hodnotu 100, pro *B* hodnotu -100 a úhel nastavíme na hodnotu 16.384, která odpovídá 90°. Podle teoretického výpočtu očekáváme výstupní hodnoty 57,74 pro *d* složku a -100 pro *q* složku.



Obrázek 30: Výstup simulace Clarkové transformace

Na obrázku 30 vidíme, že výstupní hodnoty odpovídají vypočítaným hodnotám. Také zde vidíme vypočítané hodnoty sinu a kosinu, které opět souhlasí, jelikož jsou normalizovány na rozsah znaménkového 16. bitového čísla.

Pro ověření transformace z *d-q* roviny zpět na fázové veličiny využijeme toho, že se jedná o inverzní operaci, tudíž by po zadání vypočítaných hodnot z předchozí simulace, měly vyjít původní zadaná čísla.



Obrázek 31: Výstup simulace Parkovy transformace

Výsledné hodnoty, které můžeme vidět na obrázku 31, odpovídají původním vstupním hodnotám. Odchylka u první fázové veličiny je opět způsobena zaokrouhlováním. Také si můžeme všimnout, že se zde nachází také správně vypočítaná hodnota třetí fázové veličiny, která musí splňovat podmínku, že součet všech fázových veličin je roven nule.

7.6 Testování na vývojové desce

Jak již bylo uvedeno v úvodu této kapitoly, k otestování správného chodu jednotlivých bloků implementovaných do hradlového pole použijeme AXI sběrnici, pomocí které bude procesorová část zařízení vyčítat data z komponent. Následně budou tyto údaje odesílány přes rozhraní UART do konzole v počítači.

Abychom mohli potřebná data vyčítat pomocí procesoru, musí být k dispozici v některém z registrů AXI sběrnice. K tomu využijeme již připravenou komponentu pro

obsluhu AXI sběrnice, která je součástí každého nově vytvořeného IP bloku. Zde vždy stačí rozšířit vstupní signály o ty, které chceme odesílat a následně provést úpravu zdrojového kódu. Úprava spočívá pouze ve výměně výchozí proměnné v části pro odesílání dat za náš odesílaný signál.

V procesorové části vytvoříme aplikaci, která bude číst hodnoty jednotlivých signálů z komponent., k čemuž použijeme funkci *Xil_In32*. Ty následně odešleme do konzole v počítači pomocí funkce *xil_printf*. Samozřejmě musíme realizovat zpoždění mezi jednotlivými požadavky ke čtení, aby k němu docházelo v definovaných intervalech. Jelikož nepotřebujeme definovat přesnou hodnotu tohoto intervalu, bude stačit použít příkaz *for*, který zpomalí hlavní smyčku programu.

Pro testování byla ještě vytvořena pomocná komponenta, která generovala testovací vstupní signály, podle přepínačů na vývojové desce. Tímto byla vždy vytvořena známá vstupní hodnota do testované komponenty. Nejprve byl ověřen generátor PWM signálu s použitím osciloskopu. Následně byly přidávány další komponenty a postupně ověřována správnost jejího výstupu pomocí připraveného programu pro čtení přes AXI sběrnici, dokud nebyl vytvořen kompletní funkční projekt.

8 SESTAVENÍ APLIKACE PRO ŘÍZENÍ

V této části si popíšeme, jak by měl fungovat celý program pro řízení a také si projdeme nastavení jednotlivých parametrů komponent pro konkrétní hardware, který budeme používat. Během toho se pozastavíme nad vzniklými problémy a ukážeme si, jak je vyřešit.

8.1 Funkčnost celého programu

Začneme tedy tím, jak má vlastně celý systém fungovat. Nejprve zde máme generátor PWM, na jehož výstupu je signál o zvolené periodě, která udává periodu vzorkování celého systému, jelikož se nová hodnota střídy načítá právě na začátku každé periody. Se stejnou periodou vzorkování bychom tedy měli vzorkovat také vstupní naměřené signály a také by s touto periodou vzorkování měl pracovat regulátor. Ke vzorkování vstupních signálů využijeme synchronizační impulz, který vystupuje z PWM generátoru a označuje začátek nové periody. Ten tedy musíme přivést do komponenty, která čte data z AD převodníků a také do komponenty zajišťující dekódování impulzů z inkrementálního snímače. Zde musíme přidat ještě část kódu, která zajistí, že se data na výstupu budou aktualizovat pouze s periodou vzorkování.

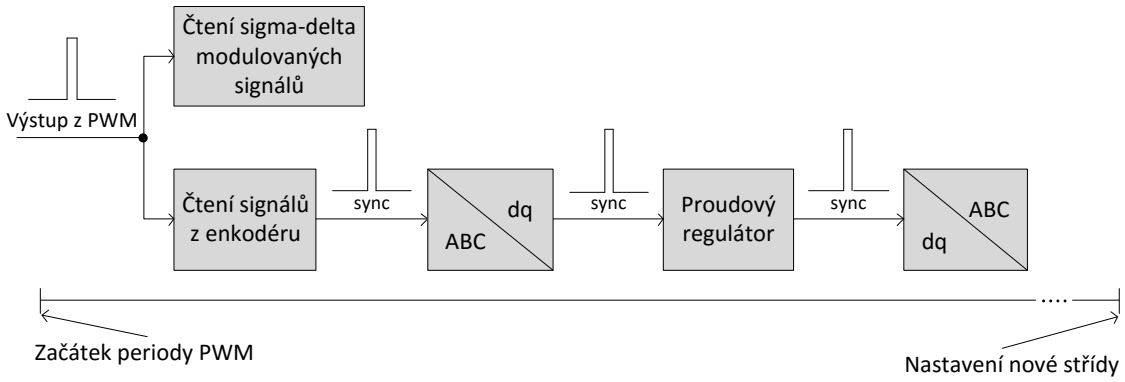
Zdrojový kód 33: Vzorkování vstupních signálů

```
process (clk) begin
    if (clk'event and clk='1') then
        if (sync_in='1') then
            Current_A <= std_logic_vector(A_signal);
            Current_B <= std_logic_vector(B_signal);
            VBus <= std_logic_vector(VBUS_signal);
        end if;
    end if;
end process;
```

Tento zápis zajistí to, že se vytvoří záchytný registr z klopných obvodů mezi signály, které vystupují ze $SINC^3$ filtru a mezi výstupními porty celé komponenty. Obdobným způsobem je potřeba zajistit vzorkování úhlu natočení rotoru a rychlosti na výstupu z komponenty, která tyto veličiny zpracovává. Tímto máme zajištěno, že po celou dobu jedné periody vzorkování bude na výstupu z těchto bloků přidržena stejná hodnota.

Nyní je potřeba si uvědomit, že některé komponenty, které výše zmíněné veličiny zpracovávají, nevypočítají veškeré operace v jednom kroku. Například hodnoty sinu a kosinu jsou vypočítány jedna po druhé nebo také regulátor proudu nejprve počítá d složku akčního zásahu a až poté hodnotu q složky. Proto je nutné zajistit, aby se jednotlivé výpočty prováděly ve správném pořadí. Za tímto účelem byla každá komponenta opatřena vstupem a výstupem pro synchronizační signál, kde příchod

vstupního synchronizačního signálu značí to, že data z předchozí komponenty jsou platná a výstupní synchronizační signál zase oznamuje následujícímu bloku, že dokončil svou operaci a má na svém výstupu správné hodnoty. Tímto způsobem přechází synchronizační impulz od jednoho bloku k druhému, dokud nedojde k poslední komponentě v řetězci zpracování signálů. Časový sled jednotlivých úkonů a přenos synchronizačního signálu je znázorněn na následujícím obrázku.



Obrázek 32: Časový diagram zpracování signálů

Z časového diagramu je zřejmé, že generátor PWM načítá vždy hodnotu o jeden krok zpožděnou, čímž tedy vzniká dopravní zpoždění o délce jedné periody vzorkování. Ta nám ale vůbec nemusí vadit, pokud tuto skutečnost vezmeme v potaz při návrhu parametrů regulátoru. Kromě tohoto dopravního zpoždění se v systému samozřejmě vyskytuje také zpoždění vzniklé kvůli vzorkování, které je rovno polovině periody vzorkování. Proto je tedy nutné při návrhu regulátoru zohlednit obě tyto prodlevy a počítat s celkovým dopravním zpožděním rovným jedné a půl periodě vzorkování.

Po výše uvedených úpravách je zajištěno, že budou jednotlivé bloky pracovat správně při jejich propojení. Můžeme tedy přistoupit k vytvoření celého projektu.

8.2 Tvorba výsledné aplikace

V této části si popíšeme postup při tvorbě aplikace, která bude řídit PMS motor a nestavení jednotlivých bloků pro naše konkrétní použití.

Nejprve si tedy projdeme hardwarové součásti, které budeme mít k dispozici. První komponentou je výkonová a komunikační deska od firmy Analog Devices s označením FMCMOTCON1-EBZ. Výkonová deska obsahuje třífázový střídač s řídícím obvodem IRS2336, operační zesilovače, které upravují úrovně fázových proudů a napájecího napětí pro další zpracování a také obvody pro napájení, nadproudovou ochranu atd. Výkonová část je připojena ke komunikační desce, která slouží jako prostředník mezi ní a řídicím systémem. K propojení se ZedBoardem je deska vybavena FMC konektorem, dále se zde nachází sigma-delta modulátory AD7401 pro snímání výše zmíněných signálů a konektor pro připojení enkodéru popřípadě Hallových sond pro snímání

polohy. Sem bude připojený inkrementální snímač ENC-A2I-1250-197-H, který má, jak již název napovídá, 1250 impulzů na otáčku. A nakonec je potřeba zmínit použitý motor, který nese označení BLY171D-24V-4000. Jedná se o třífázový PMS motor se čtyřmi pólovými páry.

Nyní se můžeme přesunout k tvorbě samotné aplikace. Po založení nového projektu a vytvoření blokového designu můžeme začít vkládat jednotlivé komponenty. Jako první je potřeba přidat řídicí systém Zynq-7000, kterému je potřeba nastavit požadované parametry. Dvojklikem na komponentu se otevře konfigurační menu, kde v záložce *Clock Configuration* rozklikneme sekci *PL Fabric Clocks*, kde máme možnost nastavení až čtyř různých hodinových signálů, které poté můžeme využít v hradlovém poli. První z nich o frekvenci 100 MHz bude použit pro AXI sběrnici. Pro funkčnost naší aplikace budeme potřebovat další hodinový signál, kterému nastavíme frekvenci například 50 MHz, ta nám zajistí dostatečné rozlišení PWM signálu. Další hodinový signál budeme potřebovat pro sigma-delta modulátory. Frekvence tohoto signálu musí být podle katalogového listu [2] v rozsahu 5 až 20 MHz, zvolíme tedy maximální možnou hodnotu. Další parametry není nutné měnit, můžeme tedy změny potvrdit a pokračovat k další komponentě.

Jako další blok přidáme PWM generátor, kterému nastavíme hodnotu frekvence vstupního hodinového signálu na 50 MHz a kmitočet PWM na 20 KHz. Dále je potřeba nastavit aktivní úroveň logiky řízení tranzistorů. Z katalogového listu [8] pro řídicí obvod IRS2336 lze vyčíst, že pro spínání obou tranzistorů platí inverzní logika, proto je potřeba nastavit aktivní úroveň jako logickou 0. Poslední parametr, který definuje velikost ochranné doby tranzistorů, můžeme ponechat nulový, jelikož je tato ochrana implementována přímo do výše zmíněného řídicího obvodu.

Dále přidáme blok pro čtení signálů z AD převodníků a enkodéru, kterému je opět potřeba nastavit frekvenci hodinového signálu 50 MHz, dále počet impulzů na otáčku na 1250 a počet pólových párů, který je v našem případě roven čtyřem. Komponenty pro transformace signálů z *ABC* do *d-q* a zpět nevyžadují nastavení žádných parametrů, tudíž je stačí pouze přidat. Dále přidáme proudový a otáčkový regulátor, kterým je potřeba nastavit periodu vzorkování a parametry jednotlivých složek regulátoru a nakonec vložíme blok s řídicími signály.

Vše propojíme podle schématu na obrázku 13 a synchronizační signály podle obrázku 32. Jako žádanou hodnotu otáčkového regulátoru připojíme výstupní vektor z bloku s řídicími signály a jeho další výstupy (povel k natočení rotoru a k zahájení kalibrace) spojíme s odpovídajícími porty PWM generátoru a enkodéru.

Nakonec je potřeba zajistit propojení některých portů s fyzickými piny hradlového pole. Jedná se o PWM signály, datové a hodinové signály AD převodníků, impulzy z enkodéru, tlačítka a LED diody na vývojové desce. To se provede tak, že se nejprve

označí požadovaný pin a pravého tlačítka myši otevřeme nabídku, ze které vybereme možnost *Make External*. Dále potřebujeme zjistit, na které externí piny hradlového pole musíme připojit. Je tedy nutné projít schéma zapojení komunikační desky a najít odpovídající pozice na FMC konektoru, ke kterým je ještě potřeba přiřadit jednotlivé piny hradlového pole podle schématu k desce ZedBoard. Dohledaná propojení jsou zobrazena v následující tabulce.

Tabulka 1: Mapa externích signálů

Signál	Pin	Signál	Pin
PWM_AH	A16	btnu	T18
PWM_AL	A17	btnd	R16
PWM_BH	C15	btnc	P16
PWM_BL	B15	btnl	N15
PWM_CH	A21	btnr	R18
PWM_CL	A22	LEDS[0]	T22
ADC_clk_A	P17	LEDS[1]	T21
ADC_clk_B	P18	LEDS[2]	U22
ADC_clk_VBUS	M22	LEDS[3]	U21
ADC_data_A	T16	LEDS[4]	V22
ADC_data_B	T17	LEDS[5]	W22
ADC_data_VBUS	N18	LEDS[6]	U19
SENS_A	J16	LEDS[7]	U14
SENS_B	J17	stop	F22
SENS_INDEX	G15	stop_inv	G22

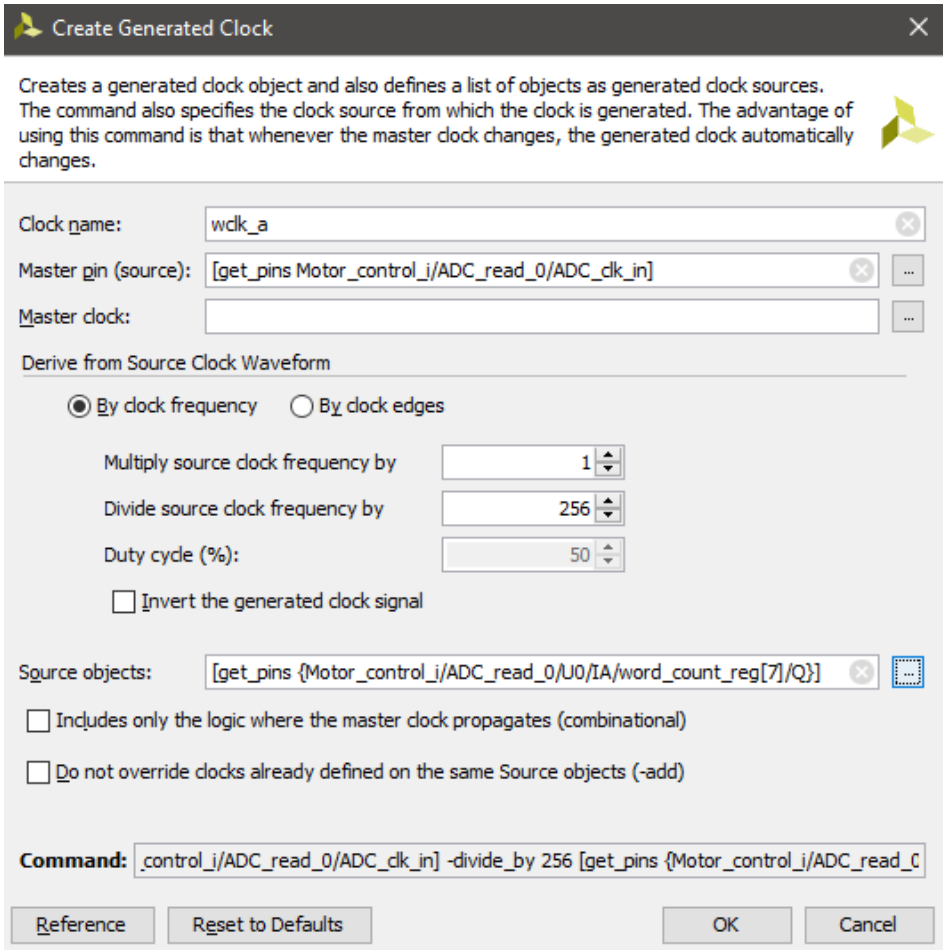
Samotné přiřazení se provádí pomocí *.xdc* souboru v sekci *Constraints*. Zde je potřeba vytvořit nový soubor a vyplnit podle tabulky. V následující části kódu je uveden příklad pro první signál.

Zdrojový kód 34: Ukázka definice externích portů

```
set_property PACKAGE_PIN A16 [get_ports PWM_AH]
set_property IOSTANDARD LVCMOS25 [get_ports PWM_AH]
```

Zde první řádek provede samotné propojení a druhý řádek definuje napěťovou úroveň logického signálu, která bude v našem případě 2,5 V. Nyní můžeme spustit syntézu našeho projektu. Po skončení této operace se v levém panelu odemkne položka *Synthesized Design*, ze které vybereme možnost *Edit Timing Constraints*.

Jelikož jsme v komponentě obsahující SINC^3 filtr vytvořili nový hodinový signál, musí se tato skutečnost nadefinovat ještě v této sekci, jinak by se při implementaci tento signál nevytvořil. Pro připomenutí se jedná o hodinový signál, který je ve zdrojovém kódu 9 označný jako *word_clk* a vznikl vydělením frekvence vstupního hodinového signálu AD převodníku hodnotou 256. Pokud chceme takovýto signál nadefinovat, je potřeba otevřít záložku *Create Generated Clocks,* kde přidáme novou položku. Nejprve musíme signál nějak unikátně pojmenovat, poté zvolit původní zdroj hodinového signálu, kterým je zmíněný hodinový signál pro AD převodník. Název signálu je potřeba zadat s absolutní cestou ve struktuře projektu, s čímž pomůže vyhledávací nástroj, který otevřeme stiskem tlačítka vpravo se třemi tečkami. Dále zvolíme dělící poměr 256 a nakonec vyplníme pin, který bude zdrojem našeho nového hodinového signálu. Tím je nejvyšší, sedmý, bit registru označeného jako *word_count*, který je realizován klopným obvodem typu D, jehož výstupem je pin Q. Opět k němu musíme zadat absolutní cestu. Dialogové okno se všemi nastavenými parametry můžeme vidět na následujícím obrázku.



Obrázek 33: Vytvoření hodinového signálu

Po potvrzení všech změn přibude další položka do našeho *.xdc* souboru, která vypadá následovně.

Zdrojový kód 35: Příkaz k vytvoření hodinového signálu

```
create_generated_clock -name wclk_a -source [get_pins
Motor_control_i/ADC_read_0/ADC_clk_in] -divide_by 256 [get_pins
{Motor_control_i/ADC_read_0/U0/IA/word_count_reg[7]/Q}]
```

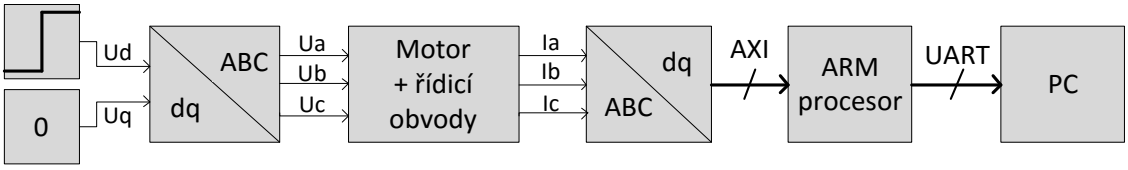
Nyní je vše připraveno pro vytvoření celého programu pro hradlové pole a můžeme projít všemi kroky od syntézy přes implementaci až k vygenerování bitstreamu, který tímto bude připraven pro stažení do zařízení. Kompletní program je k dispozici v příloze B.

8.3 Návrh regulátorů

Aby vytvořená aplikace mohla efektivně řídit motor, je potřeba navrhnout optimální parametry jednotlivých regulátorů. K tomu je nutné nejprve provést identifikaci regulovaného systému. V této části se budeme zabývat právě touto identifikací, která bude založená na analýze přechodových charakteristik.

V kapitole 2.3 jsme si uvedli, že všechny dílčí regulátory, které je potřeba navrhnout jsou typu PI a při návrhu jejich parametrů se postupuje od vnitřní regulační smyčky, tedy proudové, směrem vně, v našem případě pouze po otáčkovou smyčku.

Nejprve tedy provedeme návrh dvojice proudových regulátorů. Jelikož regulace probíhá v *d*-*q* oblasti, je potřeba i při identifikaci pracovat se složkami komplexního vektoru proudu *d* a *q*. V našem případě nemusíme provádět návrh obou proudových regulátorů, jelikož motor, se kterým pracujeme, má stejnou hodnotu indukčností L_d a L_q, a proto je dynamické chování v obou těchto složkách stejné. Stačí tedy navrhnout pouze jeden z regulátorů a druhý nastavit pomocí stejných parametrů. Pro samotné měření bude použito následující zapojení.



Obrázek 34: Schéma pro měření přechodové charakteristiky proudu

Abychom mohli tímto způsobem naměřit potřebná data, musíme nastavit odesílání potřebných dat z hradlového pole do procesoru, který je schopen dále komunikovat s počítačem. Na straně hradlového pole je tedy nutné nastavit, aby byla aktuální hodnota *d* složky přístupná ke čtení z procesoru. Jak jsme si uvedli v kapitole 5, je při vytvoření nové komponenty automaticky vygenerován soubor pro obsluhu AXI sběrnice, ve kterém nastavíme, aby se při čtení jednoho z registrů odeslala hodnota naší požadované proměnné. Na straně procesoru můžeme opět využít připravenou funkci pro čtení. Problém je ovšem v tom, že jelikož potřebujeme analyzovat dynamické vlastnosti systému, musíme data vyčítat v přesně stanovených časových intervalech. K tomu

využijeme přerušení časovače. Na základě [14] byla vytvořena aplikace pro procesor, která s periodou vzorkování vyčítá námi požadovanou hodnotu *d* složky proudu a po ukončení stanovené doby měření je odesílá do PC za použití sériového rozhraní UART.

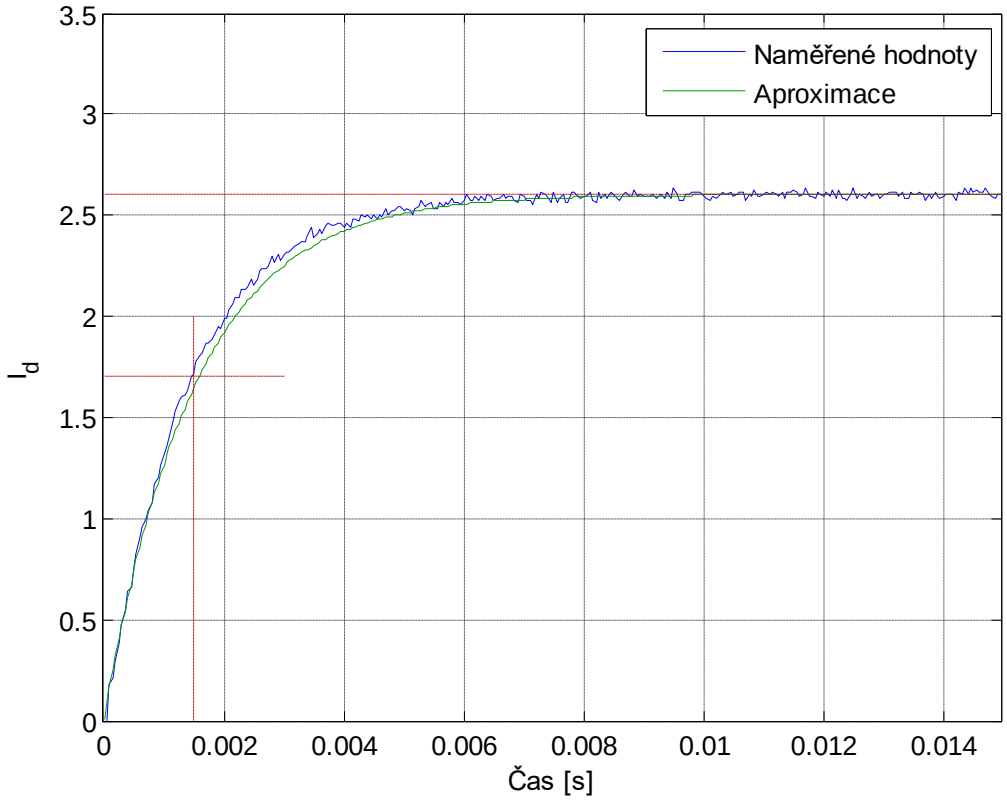
Pro další zpracování byla data importována do programu MATLAB, kde bylo provedeno další zpracování. Jelikož víme, že přenos proudu vinutím motoru by měl mít charakter setrvačného článku prvního řádu, bude mu také odpovídat naše aproximace. Obecný setrvačný článek prvního řádu je možné popsat přenosem 8-1.

$$F(p) = \frac{K_s}{Tp+1} \qquad 8\text{-}1$$

Hodnotu statického zesílení K_s určíme jako hodnotu, na které se ustálí přechodová charakteristika a časovou konstantu T jako čas, kdy hodnota přechodové charakteristiky nabude 63,2 % z maximální hodnoty. Výsledný přenos vypadá následovně.

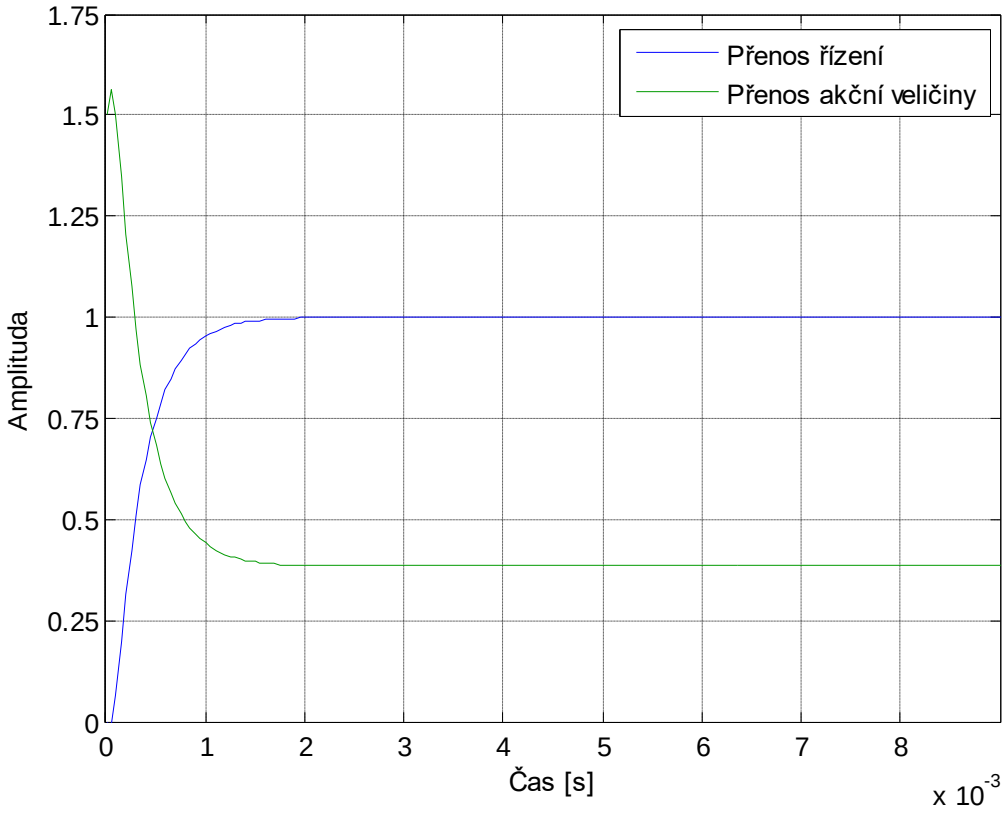
$$F_{aprox}(p) = \frac{2{,}6}{0{,}0015p+1} \qquad 8\text{-}2$$

Na následujícím grafu si můžeme prohlédnout srovnání naměřené přechodové charakteristiky a přechodové charakteristiky aproximovaného systému. V grafu je také červeně vyznačeno statické zesílení a časová konstanta.



Obrázek 35: Přechodová charakteristika d složky proudu

K získanému přenosu soustavy je potřeba ještě přidat dopravní zpoždění o velikosti 1,5 násobku periody vzorkování. K tomuto dopravnímu zpoždění dochází vlivem samotného vzorkování a tím, že se akční zásah projeví vždy o jednu periodu vzorkování později. Po přidání tohoto zpoždění se můžeme přesunout k návrhu regulátoru. Návrh PI regulátoru spočívá v nalezení dvou parametrů, kterými je proporcionální zesílení a poloha jedné nuly. Nula regulátoru byla nastavena tak, aby vykompenzovala pól soustavy. Následně byla empiricky nastavena hodnota proporcionálního zesílení tak, aby se hodnoty akčního zásahu na výstupu tohoto regulátoru pohybovaly v realizovatelných mezích. Průběhy přenosu řízení a akčního zásahu v uzavřené smyčce s navrženým regulátorem si můžeme prohlédnout na obrázku 36.

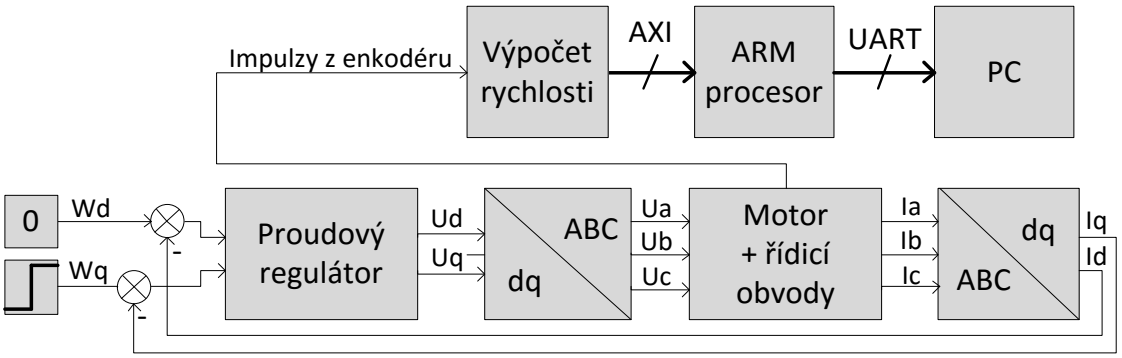


Obrázek 36: Přenosy řízení a akční veličiny v uzavřené proudové smyčce

Výsledný přenosu regulátoru vypadá následujícím způsobem.

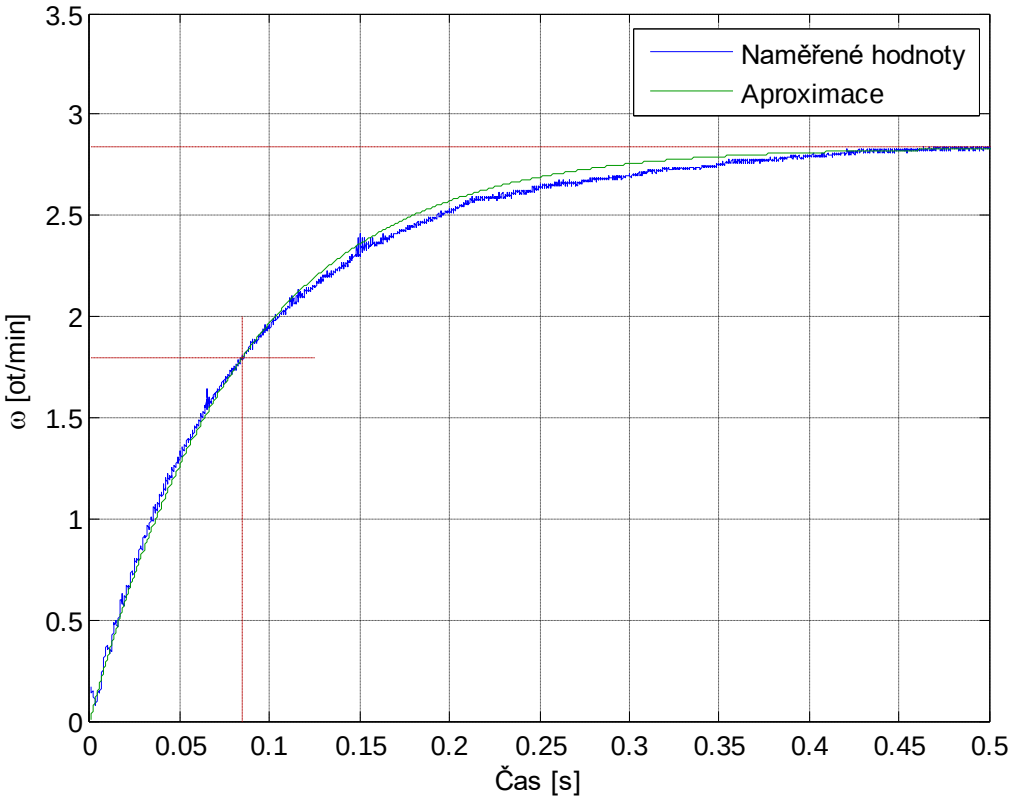
$$F_{Rd}(p) = \frac{1{,}5 \cdot (0{,}0015p + 1)}{p} \qquad 8\text{-}3$$

Teď když máme navrženu proudovou smyčku, můžeme se přesunout k návrhu regulátoru otáček. Je tedy potřeba provést identifikaci, tentokrát celé vnitřní smyčky. Pro změření přechodové charakteristiky byl opět vytvořen program pro hradlové pole, tentokrát podle schématu na obrázku 37.



Obrázek 37: Schéma pro měření přechodové charakteristiky otáček

Do procesoru byl opět nahrán program pro vyčítání dat přes AXI sběrnici, který jsme si popsali u předchozího měření a naměřená data byla dále zpracována pomocí programu MATLAB. Změřenou přechodovou charakteristiku včetně přenosu soustavy, která představuje její aproximaci je vidět na obrázku 38.



Obrázek 38: Přechodová charakteristika otáček

Soustava byla opět aproximována setrvačným článkem prvního řádu, jehož parametry, tedy statické zesílení a časová konstanta jsou opět vyznačeny v grafu s přechodovou charakteristikou. Přenos aproximace této soustavy je následující.

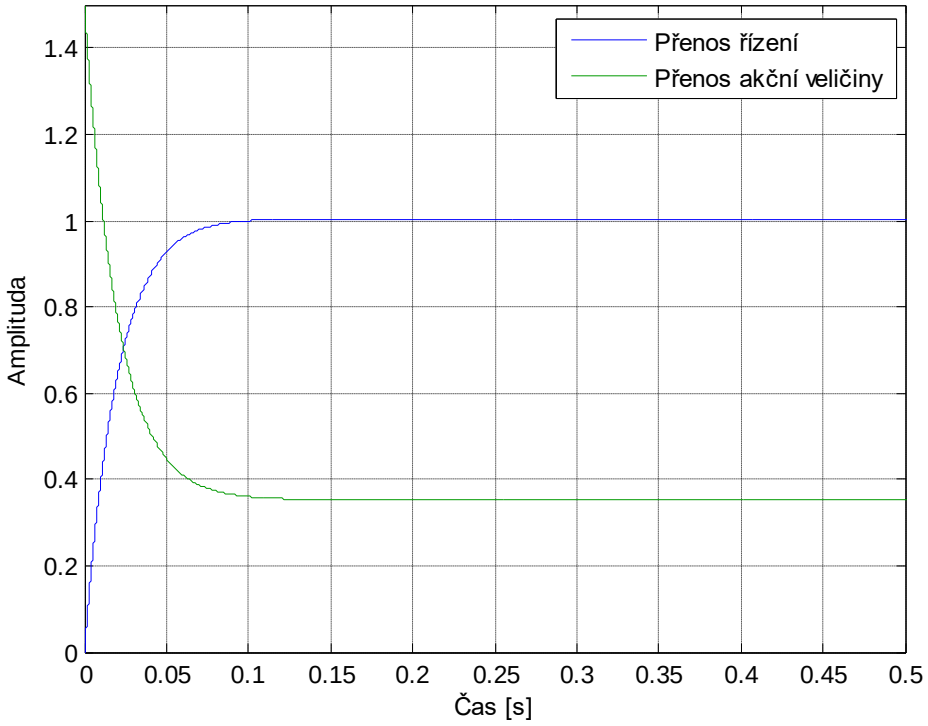
$$F_{aprox}(p) = \frac{2,8}{0,08p + 1} \qquad 8\text{-}4$$

K tomuto přenosu by mělo opět být přidáno dopravní zpoždění, stejně jako v předchozím případě, ale jelikož je jeho hodnota o tři řády nižší než časová konstanta této soustavy, můžeme si dovolit toto dopravní zpoždění zanedbat.

K navržení regulátoru byl zvolen stejný postup jako při návrhu proudových regulátorů. Nejprve byla navržena nula PI regulátoru, aby opět došlo ke kompenzaci pólu regulované soustavy. Poté bylo experimentálním způsobem upravováno zesílení tak, aby se hodnoty akčního zásahu opět pohybovaly v realizovatelných mezích. Parametry výsledného otáčkového regulátoru můžeme vyčíst z přenosu 8-5.

$$F_{R\omega}(p) = \frac{1,5 \cdot (0,08p + 1)}{p} \qquad 8\text{-}5$$

Na obrázku 39 si můžeme prohlédnout přenosy v uzavřené smyčce pro řízení a akční veličinu s tímto navrženým regulátorem.



Obrázek 39: Přenosy řízení a akční veličiny v uzavřené otáčkové smyčce

Tímto máme navrženy všechny potřebné parametry pro jednotlivé regulátory. Oba projekty pro hradlové pole jsou k dispozici v přílohách, konkrétně ten pro měření přechodové charakteristiky přenosu proudu v příloze C a pro měření přechodové charakteristiky přenosu otáček v příloze D. Skripty pro samotný výpočet aproximačních přenosů a parametrů regulátorů včetně naměřených dat se nacházejí v příloze E.

8.4 Přechod do bezpečného stavu

Posledním úkolem bude zajistit přechod celého zařízení do bezpečného stavu. Nejprve je potřeba si nadefinovat, co tím bezpečným stavem bude. Pro tuto práci bylo určeno, že v případě poruchy musí dojít k zastavení motoru. Je tedy potřeba vytvořit mechanizmus, který toto zastavení zajistí. Při jeho tvorbě budeme předpokládat, že máme k dispozici signál, který nás informuje o tom, že někde vznikla porucha. Tento signál je z důvodu vyšší spolehlivosti zdvojený a jeden z nich pracuje s inverzní logikou.

Aby došlo k zastavení motoru, stačí vypnout generování PWM signálu. Komponenta obsahující PWM generátor je již vybavena vstupem *run*, který toto generování spouští nebo zastavuje. Právě toho využijeme a náš bezpečnostní mechanizmus vložíme do cesty tomuto signálu, a pokud budeme detekovat aktivní úroveň na některém z poruchových signálů, musí být signál *run* uveden do stavu logické 0. Abychom ovšem zachovali zdvojené vedení poruchového signálu, musíme přidat také druhý signál *run*, který bude pracovat s inverzní logikou stejně jako druhý poruchový signál. K provedení této operace použijeme následující blok kombinační logiky.

Zdrojový kód 36: Logika pro přechod do bezpečného stavu

```
run_out <= run_in and not stop and stop_inv;
run_out_inv <= not run_in or stop or not stop_inv;
```

Nakonec je ovšem potřeba upravit blok PWM generátoru tak, aby zohlednil invertovaný signál pro spouštění. Ve zdrojovém kódu 6, tedy upravíme podmínku pro zastavení následujícím způsobem.

Zdrojový kód 37: Úprava generátoru PWM

```
if (run = '0' or run_neg = '1') then
  PWM_H <= not ACTIVE_H;
  PWM_L <= not ACTIVE_L;
elsif …
```

Tímto máme vyřešený přechod do bezpečného stavu, který jsme si nadefinovali. Pro testovací účely byly signály, které indikují poruchu vyvedeny na přepínače umístěné na vývojové desce, abychom jimi mohli simulovat vznik chyby v zařízení.

9 ZÁVĚR

Cílem této práce bylo vytvoření softwarového rozhraní mezi PMS motorem a vývojovou deskou *ZedBoard*, které bude zajišťovat obsluhu periferií a vektorové řízení motoru. Nejprve byl tedy nastudován princip činnosti synchronních pohonů a veškerých obvodů spjatých s jeho řízením. Dále bylo nutné seznámit se s problematikou vektorového řízení a s potřebnými transformacemi souřadnicového systému. V další části práce bylo provedeno seznámení se s platformou *Zynq*, s vývojovou deskou *ZedBoard* a s vývojovým prostředím *Vivado*, ve kterém se toto zařízení programuje. To obnáší také seznámení se s AXI sběrnicí, která zajišťuje veškerou komunikaci mezi periferiemi uvnitř zařízení *Zynq*.

Poté následovala praktická část ve, které byly vytvořeny elementární komponenty potřebné pro řízení PMS motoru. První součástí je PWM generátor ovládající měnič, který budí vinutí jednotlivých fází motoru. Dále bylo potřeba obsluhovat měřící část řídící desky motoru, která obsahuje sigma-delta modulátory pro snímání proudů jednotlivými vinutími a napájecího napětí. Výstup z těchto převodníků bylo potřeba patřičně dekódovat pomocí $SINC^3$ filtru. Také bylo potřeba dekódovat výstup z inkrementálního snímače za účelem zjišťování polohy rotoru a rychlosti jeho otáčení. Nakonec byla vytvořena regulační část, která se skládá z dvojice proudových a jednoho otáčkového regulátoru a jednotlivých transformací souřadnic, které převádějí měřené fázové proudy do *d*-*q* roviny a akční zásahy regulátorů v podobě dvou složek komplexního vektoru napětí *d* a *q* zpět na odpovídající hodnoty pro jednotlivé fáze.

Výše popsané komponenty byly řádně otestovány a odladěny nejprve s použitím simulátoru, který je součástí vývojového prostředí *Vivado*, a poté přímo v hradlovém poli. Jednotlivé komponenty byly spojeny do aplikace, která je schopna regulovat PMS motor. Výsledný program byl obohacen o rozhraní, díky kterému je možné ovládat motor pomocí tlačítek a přepínačů umístěných na vývojové desce *ZedBoard*. Také byl přidán mechanismus, který zajistí uvedení zařízení do definovaného bezpečného stavu při detekci poruchového signálu. Projekt navíc obsahuje jednoduchý program pro procesorovou část, díky které je možné pomocí terminálu v počítači sledovat vybrané veličiny. Nakonec byly navrženy parametry jednotlivých regulátorů. Výsledkem této práce je plně funkční aplikace schopna vektorového řízení PMS motoru.

SEZNAM LITERATURY

[1] Alpha–beta transformation. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2008 [cit. 2016-05-09]. Dostupné z: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_transformation

[2] ANALOG DEVICES. *AD7401A: Isolated Sigma-Delta Modulator* [online]. Rev. C. 2011 [cit. 2016-01-08]. Dostupné z: http://www.analog.com/media/en/technical-documentation/data-sheets/AD7401A.pdf

[3] ANALOG DEVICES. *AD-FMCMOTCON1-EBZ User Guide* [online]. 2014 [cit. 2016-01-08]. Dostupné z: https://wiki.analog.com/resources/eval/user-guides/ad-fmcmotcon1-ebz

[4] BLAHA, Petr a Pavel VÁCLAVEK. Bezsnímačové řízení asynchronních motorů. *Automa*. 2003, **2003**(3), 19-22.

[5] BURNS, Alan a Robert I. DAVIS. *Mixed Criticality Systems - A Review* [online]. Seventh edition. University of York, 2016 [cit. 2016-05-06]. Dostupné z: https://www-users.cs.york.ac.uk/burns/review.pdf

[6] Direct–quadrature–zero transformation. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2008 [cit. 2016-05-09]. Dostupné z: https://en.wikipedia.org/wiki/Direct%E2%80%93quadrature%E2%80%93zero_transformation

[7] INTERNATIONAL RECTIFIER. *IRFB/S/SL3806PbF: HEXFET Power MOSFET*[online]. 2008 [cit. 2016-01-08]. Dostupné z: http://www.irf.com/product-info/datasheets/data/irfs3806pbf.pdf

[8] INTERNATIONAL RECTIFIER. *IRS2336(D) IRS23364D High Voltage 3 Phase Gate Driver IC* [online]. 2011 [cit. 2016-05-13]. Dostupné z: http://www.irf.com/product-info/datasheets/data/irs2336.pdf

[9] MICROSEMI. *Clarke and Inverse Clarke Transformations Hardware Implementation User Guide* [online]. 2013, , 20 [cit. 2016-05-09]. Dostupné z: http://www.microsemi.com/document-portal/doc_view/132801-clarke-and-inverse-clarke-transformations-hardware-implementation-user-guide

[10] PATOČKA, Miroslav.: *Vybrané statě z výkonové elektroniky*, sv. 2, skriptum FEKT, VUT Brno

[11] SADRI, Mohammadsadegh. *ZYNQ Training* [online]. 2016 [cit. 2016-01-09]. Dostupné z: http://www.googoolia.com/wp/category/zynq-training/

[12] SKALICKÝ, Jiří. *Elektrické regulované pohony* [online]. Brno, 2007 [cit. 2016-01-08]. Dostupné z: https://www.vutbr.cz/www_base/priloha.php?dpid=18964

[13] SUL, Seung-Ki. *Control of electric machine drive system*. Hoboken, N.J.: Wiley-IEEE, 2011. ISBN 978-047-0590-799.

[14] TAYLOR, Adam. How to Use Interrupts on the Zynq Soc. *Xcell Journal*. 2014, **2014**(87), 38-43.

[15] XILINX. Vivado AXI Reference Guide [online]. V3.0. 2015 [cit. 2016-01-08]. Dostupné z: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

SEZNAM PŘÍLOH